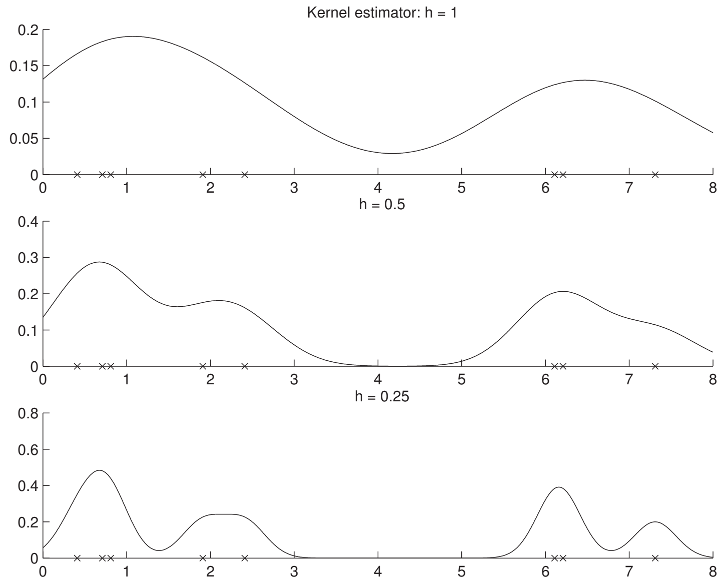


Figure 8.3 Kernel estimate for various bin lengths.

overlap of the kernels and we get a smoother estimate (see figure 8.3). If $K(\cdot)$ is everywhere nonnegative and integrates to 1, namely, if it is a legitimate density function, so will $\hat{p}(\cdot)$ be. Furthermore, $\hat{p}(\cdot)$ will inherit all the continuity and differentiability properties of the kernel $K(\cdot)$, so that, for example, if $K(\cdot)$ is Gaussian, then $\hat{p}(\cdot)$ will be smooth having all the derivatives.

One problem is that the window width is fixed across the entire input space. Various adaptive methods have been proposed to tailor h as a function of the density around x.

8.2.3 *k*-Nearest Neighbor Estimator

The nearest neighbor class of estimators adapts the amount of smoothing to the *local* density of data. The degree of smoothing is controlled by k, the number of neighbors taken into account, which is much smaller than N, the sample size. Let us define a distance between a and b, for example, $|a-b|$, and for each x, we define

$$d_1(x) \le d_2(x) \le \cdots \le d_N(x)$$

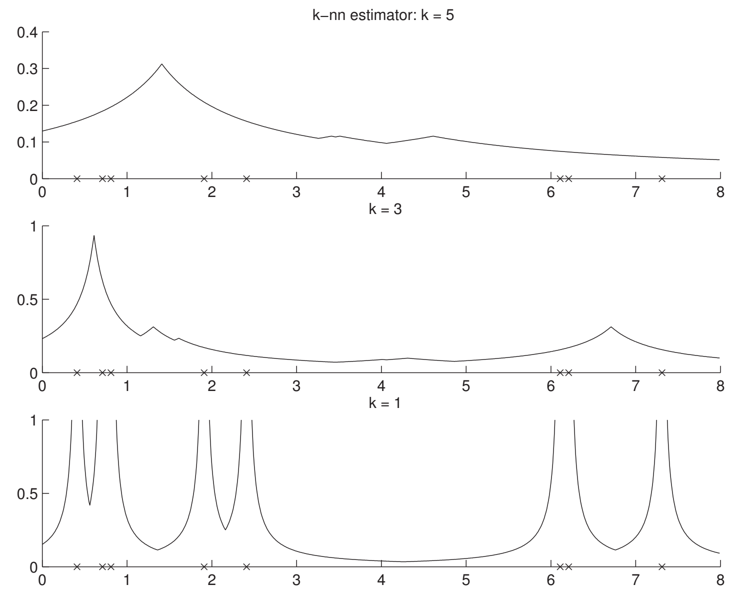


Figure 8.4 *k*-nearest neighbor estimate for various *k* values.

to be the distances arranged in ascending order, from x to the points in the sample: $d_1(x)$ is the distance to the nearest sample, $d_2(x)$ is the distance to the next nearest, and so on. If x^t are the data points, then we define $d_1(x) = \min_t |x - x^t|$, and if i is the index of the closest sample, namely, $i = \arg\min_t |x - x^t|$, then $d_2(x) = \min_{j \neq i} |x - x^j|$, and so forth.

k-NEAREST NEIGHBOR ESTIMATE

The *k-nearest neighbor* (*k*-nn) density estimate is

$$\hat{p}(x) = \frac{k}{2Nd_k(x)} \tag{8.8}$$

This is like a naive estimator with $h = 2d_k(x)$, the difference being that instead of fixing h and checking how many samples fall in the bin, we fix k, the number of observations to fall in the bin, and compute the bin size. Where density is high, bins are small, and where density is low, bins are larger (see figure 8.4).

The k-nn estimator is not continuous; its derivative has a discontinuity at all $\frac{1}{2}(x^{(j)} + x^{(j+k)})$ where $x^{(j)}$ are the order statistics of the sample. The k-nn is not a probability density function since it integrates to ∞, not 1.

To get a smoother estimate, we can use a kernel function whose effect decreases with increasing distance

$$\hat{p}(x) = \frac{1}{N d_k(x)} \sum_{t=1}^{N} K\left(\frac{x - x^t}{d_k(x)}\right) \tag{8.9}$$

This is like a kernel estimator with adaptive smoothing parameter $h = d_k(x)$. $K(\cdot)$ is typically taken to be the Gaussian kernel.

8.3 Generalization to Multivariate Data

Given a sample of d-dimensional observations $\mathcal{X} = \{\boldsymbol{x}^t\}_{t=1}^N$, the multivariate kernel density estimator is

$$\hat{p}(\boldsymbol{x}) = \frac{1}{N h^d} \sum_{t=1}^{N} K\left(\frac{\boldsymbol{x} - \boldsymbol{x}^t}{h}\right) \tag{8.10}$$

with the requirement that

$$\int_{\Re^d} K(\boldsymbol{x}) d\boldsymbol{x} = 1$$

The obvious candidate is the multivariate Gaussian kernel:

$$K(\boldsymbol{u}) = \left(\frac{1}{\sqrt{2\pi}}\right)^d \exp\left[-\frac{\|\boldsymbol{u}\|^2}{2}\right] \tag{8.11}$$

CURSE OF DIMENSIONALITY

However, care should be applied to using nonparametric estimates in high-dimensional spaces because of the *curse of dimensionality*: Let us say $\boldsymbol{x}$ is eight-dimensional, and we use a histogram with ten bins per dimension, then there are 10^8 bins, and unless we have lots of data, most of these bins will be empty and the estimates in there will be 0. In high dimensions, the concept of "close" also becomes blurry so we should be careful in choosing h.

For example, the use of the Euclidean norm in equation 8.11 implies that the kernel is scaled equally on all dimensions. If the inputs are on different scales, they should be normalized to have the same variance. Still, this does not take correlations into account, and better results are achieved when the kernel has the same form as the underlying distribution

$$K(\boldsymbol{u}) = \frac{1}{(2\pi)^{d/2} |\mathbf{S}|^{1/2}} \exp\left[-\frac{1}{2}\boldsymbol{u}^T \mathbf{S}^{-1} \boldsymbol{u}\right] \tag{8.12}$$

where $\mathbf{S}$ is the sample covariance matrix. This corresponds to using Mahalanobis distance instead of the Euclidean distance.

8.4 Nonparametric Classification

When used for classification, we use the nonparametric approach to estimate the class-conditional densities, $p(\boldsymbol{x}|C_i)$. The kernel estimator of the class-conditional density is given as

$$\hat{p}(\boldsymbol{x}|C_i) = \frac{1}{N_i h^d} \sum_{t=1}^{N} K\left(\frac{\boldsymbol{x} - \boldsymbol{x}^t}{h}\right) r_i^t \tag{8.13}$$

where r_i^t is 1 if $\boldsymbol{x}^t \in C_i$ and 0 otherwise. N_i is the number of labeled instances belonging to C_i: $N_i = \sum_t r_i^t$. The MLE of the prior density is $\hat{P}(C_i) = N_i/N$. Then, the discriminant can be written as

$$\begin{aligned} g_i(\boldsymbol{x}) &= \hat{p}(\boldsymbol{x}|C_i)\hat{P}(C_i) \\ &= \frac{1}{Nh^d} \sum_{t=1}^{N} K\left(\frac{\boldsymbol{x} - \boldsymbol{x}^t}{h}\right) r_i^t \end{aligned} \tag{8.14}$$

and $\boldsymbol{x}$ is assigned to the class for which the discriminant takes its maximum. The common factor $1/(Nh^d)$ can be ignored. So each training instance votes for its class and has no effect on other classes; the weight of vote is given by the kernel function $K(\cdot)$, typically giving more weight to closer instances.

For the special case of k-nn estimator, we have

$$\hat{p}(\boldsymbol{x}|C_i) = \frac{k_i}{N_i V^k(\boldsymbol{x})} \tag{8.15}$$

where k_i is the number of neighbors out of the k nearest that belong to C_i and $V^k(\boldsymbol{x})$ is the volume of the d-dimensional hypersphere centered at $\boldsymbol{x}$, with radius $r = \|\boldsymbol{x} - \boldsymbol{x}_{(k)}\|$ where $\boldsymbol{x}_{(k)}$ is the k-th nearest observation to $\boldsymbol{x}$ (among all neighbors from all classes of $\boldsymbol{x}$): $V^k = r^d c_d$ with c_d as the volume of the unit sphere in d dimensions, for example, $c_1 = 2, c_2 = \pi, c_3 = 4\pi/3$, and so forth. Then

$$\hat{P}(C_i|\boldsymbol{x}) = \frac{\hat{p}(\boldsymbol{x}|C_i)\hat{P}(C_i)}{\hat{p}(\boldsymbol{x})} = \frac{k_i}{k} \tag{8.16}$$

k-NN CLASSIFIER

The *k-nn classifier* assigns the input to the class having most examples among the k neighbors of the input. All neighbors have equal vote, and the class having the maximum number of voters among the k neighbors is chosen. Ties are broken arbitrarily or a weighted vote is taken. k is generally taken to be an odd number to minimize ties: Confusion is

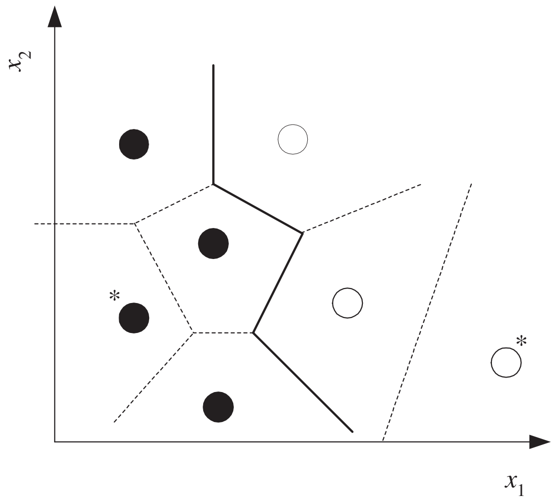


Figure 8.5 Dotted lines are the Voronoi tesselation and the straight line is the class discriminant. In condensed nearest neighbor, those instances that do not participate in defining the discriminant (marked by '*') can be removed without increasing the training error.

NEAREST NEIGHBOR CLASSIFIER

VORONOI TESSELATION

generally between two neighboring classes. A special case of k-nn is the *nearest neighbor classifier* where $k = 1$ and the input is assigned to the class of the nearest pattern. This divides the space in the form of a *Voronoi tesselation* (see figure 8.5).

8.5 Condensed Nearest Neighbor

Time and space complexity of nonparametric methods are proportional to the size of the training set, and *condensing* methods have been proposed to decrease the number of stored instances without degrading performance. The idea is to select the smallest subset $\mathcal{Z}$ of $\mathcal{X}$ such that when $\mathcal{Z}$ is used in place of $\mathcal{X}$, error does not increase (Dasarathy 1991).

CONDENSED NEAREST NEIGHBOR

The best-known and earliest method is *condensed nearest neighbor* where 1-nn is used as the nonparametric estimator for classification (Hart 1968). 1-nn approximates the discriminant in a piecewise linear manner, and only the instances that define the discriminant need be kept; an in-

```
$\mathcal{Z} \leftarrow \emptyset$
Repeat
    For all $\boldsymbol{x} \in \mathcal{X}$ (in random order)
        Find $\boldsymbol{x}' \in \mathcal{Z}$ such that $\|\boldsymbol{x} - \boldsymbol{x}'\| = \min_{\boldsymbol{x}^j \in \mathcal{Z}} \|\boldsymbol{x} - \boldsymbol{x}^j\|$
        If class($\boldsymbol{x}$)$\neq$class($\boldsymbol{x}'$) add $\boldsymbol{x}$ to $\mathcal{Z}$
Until $\mathcal{Z}$ does not change
```

Figure 8.6 Condensed nearest neighbor algorithm.

stance inside the class regions need not be stored as *its* nearest neighbor is of the same class and its absence does not cause any error (on the training set) (figure 8.5). Such a subset is called a consistent subset, and we would like to find the minimal consistent subset.

Hart proposed a greedy algorithm to find $\mathcal{Z}$ (figure 8.6). The algorithm starts with an empty $\mathcal{Z}$ and passing over the instances in $\mathcal{X}$ one by one in a random order, checks whether they can be classified correctly by 1-nn using the instances already stored in $\mathcal{Z}$. If an instance is misclassified, it is added to $\mathcal{Z}$; if it is correctly classified, $\mathcal{Z}$ is unchanged. We should pass over the training set a few times until no further instances are added. The algorithm does a local search and depending on the order in which the training instances are seen, different subsets may be found, which may have different accuracies on the validation data. Thus it does not guarantee finding the minimal consistent subset, which is known to be NP-complete (Wilfong 1992).

Condensed nearest neighbor is a greedy algorithm that aims to minimize training error and complexity, measured by the size of the stored subset. We can write an augmented error function

$$E'(\mathcal{Z}|\mathcal{X}) = E(\mathcal{X}|\mathcal{Z}) + \lambda|\mathcal{Z}| \tag{8.17}$$

where $E(\mathcal{X}|\mathcal{Z})$ is the error on $\mathcal{X}$ storing $\mathcal{Z}$. $|\mathcal{Z}|$ is the cardinality of $\mathcal{Z}$, and the second term penalizes complexity. As in any regularization scheme, λ represents the trade-off between the error and complexity such that for small λ, error becomes more important, and as λ gets larger, complex models are penalized more. Condensed nearest neighbor is one method to minimize equation 8.17, but other algorithms to optimize it can also be devised.

8.6 Distance-Based Classification

The k-nearest neighbor classifier assigns an instance to the class most heavily represented among its neighbors. It is based on the idea that the more similar the instances, the more likely it is that they belong to the same class. We can use the same approach for classification as long as we have a reasonable similarity or distance measure (Chen et al. 2009).

Most classification algorithms can be recast as a distance-based classifier. For example, in section 5.5, we saw the parametric approach with Gaussian classes, and there, we talked about the *nearest mean classifier* where we choose C_i if

$$\mathcal{D}(\boldsymbol{x}, \boldsymbol{m}_i) = \min_{j=1}^{K} \mathcal{D}(\boldsymbol{x}, \boldsymbol{m}_j) \tag{8.18}$$

In the case of hyperspheric Gaussians where dimensions are independent and all are in the same scale, the distance measure is the Euclidean:

$$\mathcal{D}(\boldsymbol{x}, \boldsymbol{m}_i) = \|\boldsymbol{x} - \boldsymbol{m}_i\|$$

Otherwise it is the Mahalanobis distance:

$$\mathcal{D}(\boldsymbol{x}, \boldsymbol{m}_i) = (\boldsymbol{x} - \boldsymbol{m}_i)^T \mathbf{S}_i^{-1} (\boldsymbol{x} - \boldsymbol{m}_i)$$

where $\mathbf{S}_i$ is the covariance matrix of C_i.

In the semiparametric approach where each class is written as a mixture of Gaussians, we can say roughly speaking that we choose C_i if among all cluster centers of all classes, one that belongs to C_i is the closest:

$$\min_{l=1}^{k_i} \mathcal{D}(\boldsymbol{x}, \boldsymbol{m}_{il}) = \min_{j=1}^{K} \min_{l=1}^{k_j} \mathcal{D}(\boldsymbol{x}, \boldsymbol{m}_{jl}) \tag{8.19}$$

where k_j is the number of clusters of C_j and $\boldsymbol{m}_{jl}$ denotes the center of cluster l of C_j. Again, the distance used is the Euclidean or Mahalanobis depending on the shape of the clusters.

The nonparametric case can be even more flexible: Instead of having a distance measure per class or per cluster, we can have a different one for each neighborhood, that is, for each small region in the input space. In other words, we can define *locally adaptive distance functions* that we can then use in classification, for example, with k-nn (Hastie and Tibshirani 1996; Domeniconi, Peng, and Gunopulos 2002; Ramanan and Baker 2011).

DISTANCE LEARNING

The idea of *distance learning* is to parameterize $\mathcal{D}(\boldsymbol{x}, \boldsymbol{x}^t|\theta)$, learn θ from a labeled sample in a supervised manner, and then use it with k-nn (Bellet, Habrard, and Sebban 2013). The most common approach is to use the Mahalanobis distance:

$$\mathcal{D}(\boldsymbol{x}, \boldsymbol{x}^t|\mathbf{M}) = (\boldsymbol{x} - \boldsymbol{x}^t)^T\mathbf{M}(\boldsymbol{x} - \boldsymbol{x}^t) \tag{8.20}$$

LARGE MARGIN NEAREST NEIGHBOR

where the parameter is the positive definite matrix $\mathbf{M}$. An example is the *large margin nearest neighbor* algorithm (Weinberger and Saul 2009) where $\mathbf{M}$ is estimated so that for all instances in the training set, the distance to a neighbor with the same label is always less than the distance to a neighbor with a different label—we discuss this algorithm in detail in section 13.13.

When the input dimensionality is high, to avoid overfitting, one approach is to add sparsity constraints on $\mathbf{M}$. The other approach is to use a low-rank approximation where we factor $\mathbf{M}$ as $\mathbf{L}^T\mathbf{L}$ and $\mathbf{L}$ is $k \times d$ with $k < d$. In this case:

$$\begin{aligned}\mathcal{D}(\boldsymbol{x}, \boldsymbol{x}^t|\mathbf{M}) &= (\boldsymbol{x} - \boldsymbol{x}^t)^T\mathbf{M}(\boldsymbol{x} - \boldsymbol{x}^t) = (\boldsymbol{x} - \boldsymbol{x}^t)^T\mathbf{L}^T\mathbf{L}(\boldsymbol{x} - \boldsymbol{x}^t)\\ &= (\mathbf{L}(\boldsymbol{x} - \boldsymbol{x}^t))^T(\mathbf{L}(\boldsymbol{x} - \boldsymbol{x}^t)) = (\mathbf{L}\boldsymbol{x} - \mathbf{L}\boldsymbol{x}^t)^T(\mathbf{L}\boldsymbol{x} - \mathbf{L}\boldsymbol{x}^t))\\ &= (\boldsymbol{z} - \boldsymbol{z}^t)^T(\boldsymbol{z} - \boldsymbol{z}^t) = \|\boldsymbol{z} - \boldsymbol{z}^t\|^2 \end{aligned} \tag{8.21}$$

where $\boldsymbol{z} = \mathbf{L}\boldsymbol{x}$ is the k-dimensional projection of $\boldsymbol{x}$, and we learn $\mathbf{L}$ instead of $\mathbf{M}$. We see that the Mahalanobis distance in the original d-dimensional $\boldsymbol{x}$ space corresponds to the (squared) Euclidean distance in the new k-dimensional space. This implies the three-way relationship between distance estimation, dimensionality reduction, and feature extraction: The ideal distance measure is defined as the Euclidean distance in a new space whose (fewest) dimensions are extracted from the original inputs in the best possible way. This is demonstrated in figure 8.7.

HAMMING DISTANCE

With discrete data, *Hamming distance* that counts the number of nonmatching attributes can be used:

$$HD(\boldsymbol{x}, \boldsymbol{x}^t) = \sum_{j=1}^{d} 1(x_j \neq x_j^t) \tag{8.22}$$

where

$$1(a) = \begin{cases} 1 & \text{if } a \text{ is true} \\ 0 & \text{otherwise} \end{cases}$$

This framework can be used with application-dependent similarity or distance measures as well. We may have specialized similarity/distance

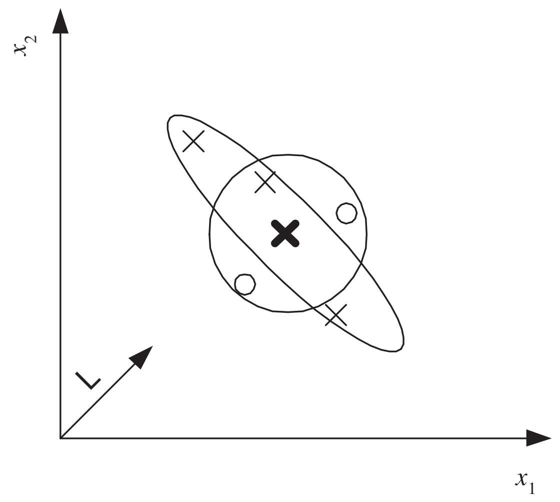


Figure 8.7 The use of Mahalanobis vs. Euclidean distance in k-nearest neighbor classification. There are two classes indicated by '$\circ$' and '$\times$'. The bold '$\times$' is the test instance and $k = 3$. Points that are of equal Euclidean distance define a circle that here leads to misclassification. We see that there is a certain correlation structure that can be captured by the Mahalanobis distance; it defines an ellipse and leads to correct classification. We also see that if the data is projected on the direction showed by L, we can do correct classification in that reduced one-dimensional space.

scores for matching image parts in vision, sequence alignment scores in bioinformatics, and document similarity measures in natural language processing; these can all be used without explicitly needing to represent those entities as vectors and using a general-purpose distance such as the Euclidean distance. In chapter 13, we will talk about kernel functions that have a similar role.

As long as we have a similarity score function between two instances $S(\boldsymbol{x}, \boldsymbol{x}^t)$, we can define a *similarity-based representation* $\boldsymbol{x}'$ of instance $\boldsymbol{x}$ as the N-dimensional vector of scores with all the training instances, $\boldsymbol{x}^t, t = 1, \ldots, N$:

$$\boldsymbol{x}' = [s(\boldsymbol{x}, \boldsymbol{x}^1), s(\boldsymbol{x}, \boldsymbol{x}^2), \ldots, s(\boldsymbol{x}, \boldsymbol{x}^N)]^T$$

This can then be used as a vector to be handled by any learner (Pekalska and Duin 2002); in the context of kernel machines, we will call this the *empirical kernel map* (section 13.7).

8.7 Outlier Detection

An *outlier*, *novelty*, or *anomaly* is an instance that is very much different from other instances in the sample. An outlier may indicate an abnormal behavior of the system; for example, in a dataset of credit card transactions, it may indicate fraud; in an image, outliers may indicate anomalies, for example, tumors; in a dataset of network traffic, outliers may be intrusion attempts; in a health-care scenario, an outlier indicates a significant deviation from patient's normal behavior. Outliers may also be recording errors—for example, due to faulty sensors—that should be detected and discarded to get reliable statistics.

OUTLIER DETECTION

Outlier detection is not generally cast as a supervised, two-class classification problem of seperating typical instances and outliers, because generally there are very few instances that can be labeled as outliers and they do not fit a consistent pattern that can be easily captured by a two-class classifier. Instead, it is the typical instances that are modeled; this is sometimes called *one-class classification*. Once we model the typical instances, any instance that does not fit the model (and this may occur in many different ways) is an anomaly. Another problem that generally occurs is that the data used to train the outlier detector is unlabeled and may contain outliers mixed with typical instances.

ONE-CLASS CLASSIFICATION

Outlier detection basically implies spotting what does not normally happen; that is, it is density estimation followed by checking for instances with too small probability under the estimated density. As usual, the fitted model can be parametric, semiparametric, or nonparametric. In the parametric case (section 5.4), for example, we can fit a Gaussian to the whole data and any instance having a low probability, or equally, with high Mahalanobis distance to the mean, is a candidate for being an outlier. In the semiparametric case (section 7.2), we fit, for example, a mixture of Gaussians and check whether an instance has small probability; this would be an instance that is far from its nearest cluster center or one that forms a cluster by itself.

Still when the data that is used for fitting the model itself includes outliers, it makes more sense to use a nonparametric density estimator, because the more parametric a model is, the less robust it will be to the

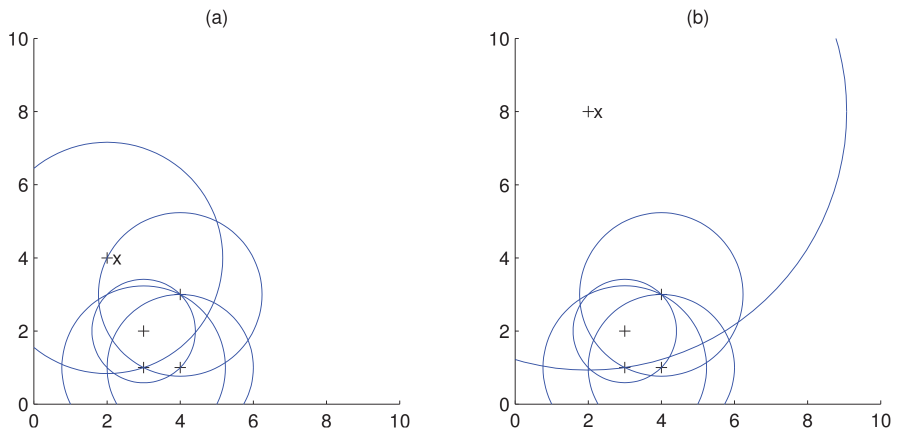


Figure 8.8 Training instances are shown by '+', '×' is the query, and the radius of the circle centered at an instance is equal to the distance to the third nearest neighbor. (a) LOF of '×' is close to 1 and it is not an outlier. (b) LOF of '×' is much larger than 1 and it is likely to be an outlier.

presence of outliers—for example, a single outlier may seriously corrupt the estimated mean and covariance of a Gaussian.

LOCAL OUTLIER FACTOR

In nonparametric density estimation, as we discussed in the preceding sections, the estimated probability is high where there are many training instances nearby and the probability decreases as the neighborhood becomes more sparse. One example is the *local outlier factor* that compares the denseness of the neighborhood of an instance with the average denseness of the neighborhoods of its neighbors (Breunig et al. 2000). Let us define $d_k(\boldsymbol{x})$ as the distance between instance $\boldsymbol{x}$ and its k-th nearest neighbor. Let us define $\mathcal{N}(\boldsymbol{x})$ as the set of training instances that are in the neighborhood of $\boldsymbol{x}$, for example, its k nearest neighbors. Consider $d_k(\boldsymbol{s})$ for $\boldsymbol{s} \in \mathcal{N}(\boldsymbol{x})$. We compare $d_k(\boldsymbol{x})$ with the average of $d_k(\boldsymbol{s})$ for such $\boldsymbol{s}$:

$$\text{LOF}(\boldsymbol{x}) = \frac{d_k(\boldsymbol{x})}{\sum_{\boldsymbol{s}\in\mathcal{N}(\boldsymbol{x})} d_k(\boldsymbol{s})/|\mathcal{N}(\boldsymbol{x})|} \qquad (8.23)$$

If LOF$(\boldsymbol{x})$ is close to 1, $\boldsymbol{x}$ is not an outlier; as it gets larger, the probability that it is an outlier increases (see figure 8.8).

8.8 Nonparametric Regression: Smoothing Models

In regression, given the training set $\mathcal{X} = \{x^t, r^t\}$ where $r^t \in \Re$, we assume

$$r^t = g(x^t) + \epsilon$$

In parametric regression, we assume a polynomial of a certain order and compute its coefficients that minimize the sum of squared error on the training set. Nonparametric regression is used when no such polynomial can be assumed; we only assume that close x have close $g(x)$ values. As in nonparametric density estimation, given x, our approach is to find the neighborhood of x and average the r values in the neighborhood to calculate $\hat{g}(x)$. The nonparametric regression estimator is also called a *smoother* and the estimate is called a *smooth* (Härdle 1990). There are various methods for defining the neighborhood and averaging in the neighborhood, similar to methods in density estimation. We discuss the methods for the univariate x; they can be generalized to the multivariate case in a straightforward manner using multivariate kernels, as in density estimation.

SMOOTHER

8.8.1 Running Mean Smoother

If we define an origin and a bin width and average the r values in the bin as in the histogram, we get a *regressogram* (see figure 8.9)

REGRESSOGRAM

$$\hat{g}(x) = \frac{\sum_{t=1}^{N} b(x, x^t) r^t}{\sum_{t=1}^{N} b(x, x^t)} \tag{8.24}$$

where

$$b(x, x^t) = \begin{cases} 1 & \text{if } x^t \text{ is the same bin with } x \\ 0 & \text{otherwise} \end{cases}$$

Having discontinuities at bin boundaries is disturbing as is the need to fix an origin. As in the naive estimator, in the *running mean smoother*, we define a bin symmetric around x and average in there (figure 8.10).

RUNNING MEAN SMOOTHER

$$\hat{g}(x) = \frac{\sum_{t=1}^{N} w\left(\frac{x - x^t}{h}\right) r^t}{\sum_{t=1}^{N} w\left(\frac{x - x^t}{h}\right)} \tag{8.25}$$

where

$$w(u) = \begin{cases} 1 & \text{if } |u| < 1 \\ 0 & \text{otherwise} \end{cases}$$

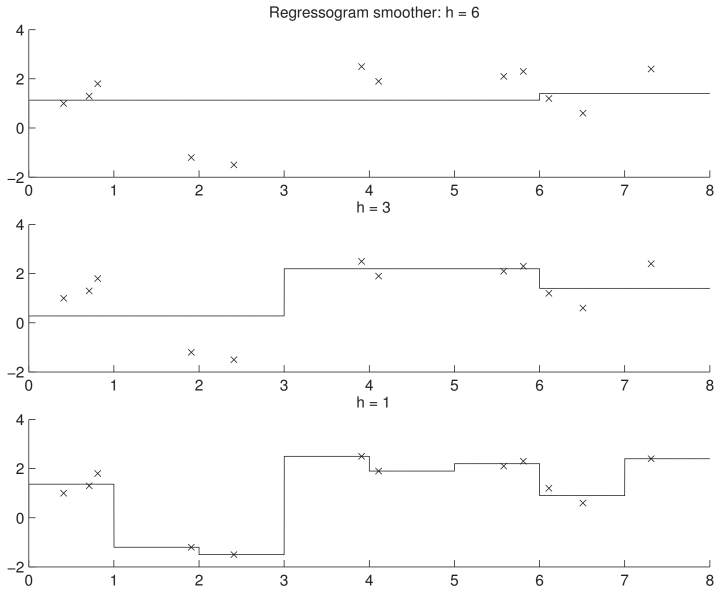


Figure 8.9 Regressograms for various bin lengths. '×' denote data points.

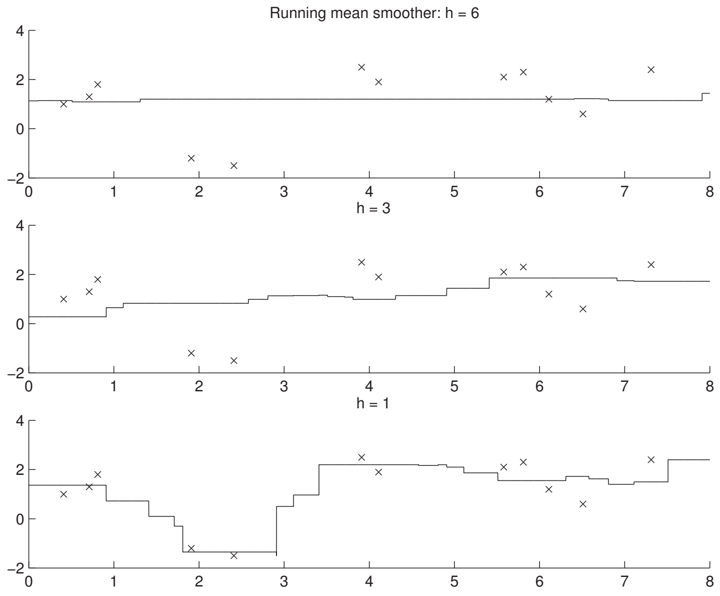


Figure 8.10 Running mean smooth for various bin lengths.

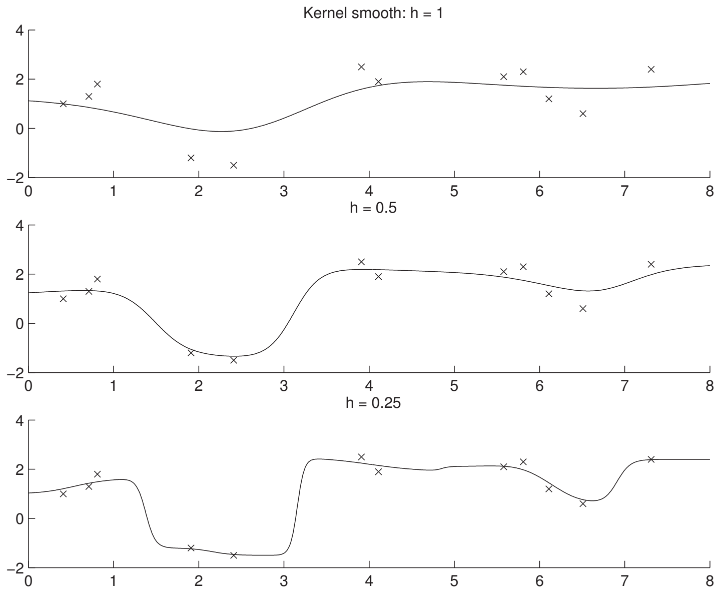


Figure 8.11 Kernel smooth for various bin lengths.

This method is especially popular with evenly spaced data, such as time series. In applications where there is noise, we can use the median of the r^t in the bin instead of their mean.

8.8.2 Kernel Smoother

KERNEL SMOOTHER

As in the kernel estimator, we can use a kernel giving less weight to further points, and we get the *kernel smoother* (see figure 8.11):

$$\hat{g}(x) = \frac{\sum_t K\left(\frac{x-x^t}{h}\right) r^t}{\sum_t K\left(\frac{x-x^t}{h}\right)} \tag{8.26}$$

k-NN SMOOTHER

Typically a Gaussian kernel $K(\cdot)$ is used. Instead of fixing h, we can fix k, the number of neighbors, adapting the estimate to the density around x, and get the *k-nn smoother*.

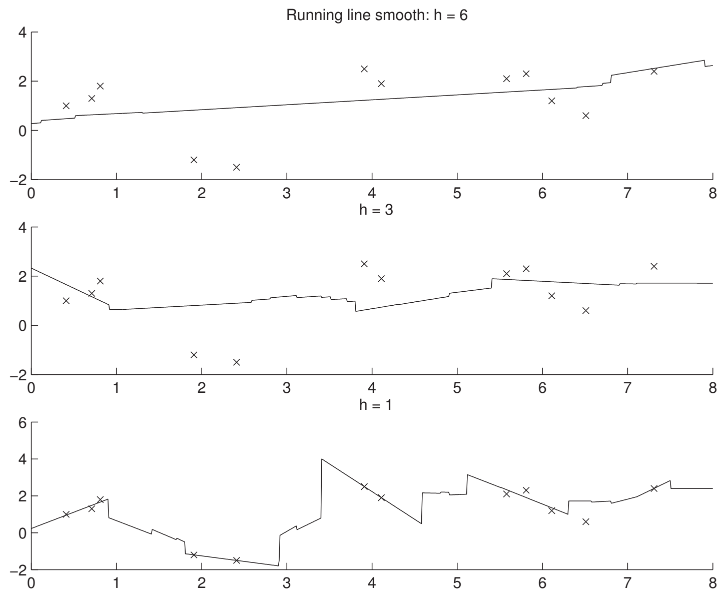


Figure 8.12 Running line smooth for various bin lengths.

8.8.3 Running Line Smoother

Instead of taking an average and giving a constant fit at a point, we can take into account one more term in the Taylor expansion and calculate a linear fit. In the *running line smoother*, we can use the data points in the neighborhood, as defined by h or k, and fit a local regression line (see figure 8.12).

RUNNING LINE SMOOTHER

In the *locally weighted running line smoother*, known as *loess*, instead of a hard definition of neighborhoods, we use kernel weighting such that distant points have less effect on error.

LOCALLY WEIGHTED RUNNING LINE SMOOTHER

8.9 How to Choose the Smoothing Parameter

In nonparametric methods, for density estimation or regression, the critical parameter is the smoothing parameter as used in bin width or kernel spread h, or the number of neighbors k. The aim is to have an estimate that is less variable than the data points. As we have discussed previously, one source of variability in the data is noise and the other is the variability in the unknown underlying function. We should smooth just enough to get rid of the effect of noise—not less, not more. With too large h or k, many instances contribute to the estimate at a point and we also smooth the variability due to the function (oversmoothing); with too small h or k, single instances have a large effect and we do not even smooth over the noise (undersmoothing). In other words, small h or k leads to small bias but large variance. Larger h or k decreases variance but increases bias. Geman, Bienenstock, and Doursat (1992) discuss bias and variance for nonparametric estimators.

This requirement is explicitly coded in a regularized cost function as used in *smoothing splines*:

SMOOTHING SPLINES

$$\sum_t \left[r^t - \hat{g}(x^t)\right]^2 + \lambda \int_a^b [\hat{g}''(x)]^2 dx \tag{8.27}$$

The first term is the error of fit. $[a, b]$ is the input range; $\hat{g}''(\cdot)$ is the *curvature* of the estimated function $\hat{g}(\cdot)$ and as such measures the variability. Thus the second term penalizes fast-varying estimates. λ trades off variability and error where, for example, with large λ, we get smoother estimates.

Cross-validation is used to tune h, k, or λ. In density estimation, we choose the parameter value that maximizes the likelihood of the validation set. In a supervised setting, trying a set of candidates on the training set (see figure 8.13), we choose the parameter value that minimizes the error on the validation set.

8.10 Notes

k-nearest neighbor and kernel-based estimation were proposed sixty years ago, but because of the need for large memory and computation, the approach was not popular for a long time (Aha, Kibler, and Albert 1991). With advances in parallel processing and with memory and computation getting cheaper, such methods have recently become more widely used. Textbooks on nonparametric estimation are Silverman 1986 and Scott 1992. Dasarathy 1991 is a collection of many papers on k-nn and editing/condensing rules; Aha 1997 is another collection.

The nonparametric methods are very easy to parallelize on a Single Instruction Multiple Data (SIMD) machine; each processor stores one training instance in its local memory and in parallel computes the kernel

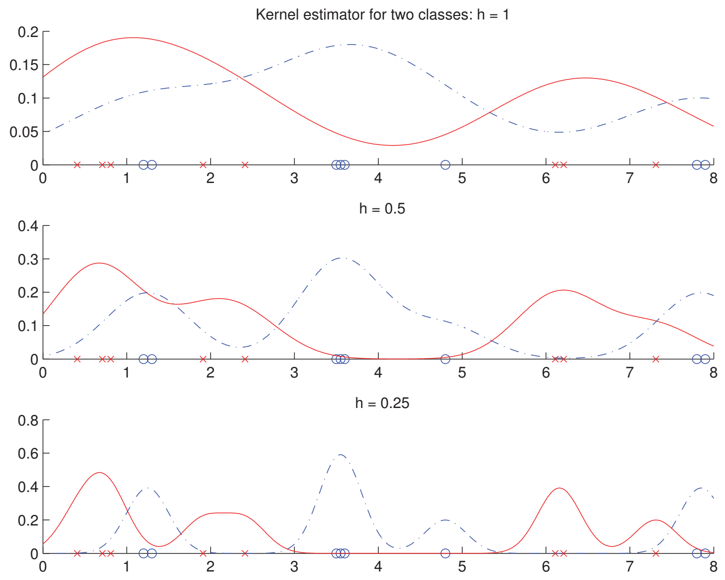


Figure 8.13 Kernel estimate for various bin lengths for a two-class problem. Plotted are the conditional densities, $p(x|C_i)$. It seems that the top one oversmooths and the bottom undersmooths, but whichever is the best will depend on where the validation data points are.

function value for that instance (Stanfill and Waltz 1986). Multiplying with a kernel function can be seen as a convolution, and we can use Fourier transformation to calculate the estimate more efficiently (Silverman 1986). It has also been shown that spline smoothing is equivalent to kernel smoothing.

CASE-BASED REASONING

In artificial intelligence, the nonparametric approach is called *case-based reasoning*. The output is found by interpolating from known similar past "cases." This also allows for some knowledge extraction: The given output can be justified by listing these similar past cases.

Due to its simplicity, k-nn is the most widely used nonparametric classification method and is quite successful in practice in a variety of applications. One nice property is that they can be used even with very few labeled instances; for example, in a forensic application, we may have only one face image per person.

It has been shown (Cover and Hart 1967; reviewed in Duda, Hart, and

Stork 2001) that in the large sample case when $N \to \infty$, the risk of nearest neighbor ($k = 1$) is never worse than twice the Bayes' risk (which is the best that can be achieved), and, in that respect, it is said that "half of the available information in an infinite collection of classified samples is contained in the nearest neighbor" (Cover and Hart 1967, 21). In the case of k-nn, it has been shown that the risk asymptotes to the Bayes' risk as k goes to infinity.

The most critical factor in nonparametric estimation is the distance metric used. With discrete attributes, we can simply use the Hamming distance where we just sum up the number of nonmatching attributes. More sophisticated distance functions are discussed in Wettschereck, Aha, and Mohri 1997 and Webb 1999.

Distance estimation or metric learning is a popular research area; see Bellet, Habrard, and Sebban 2013 for a comprehensive recent survey. The different ways similarity measures can be used in classification are discussed by Chen et al. (2009); examples of local distance methods in computer vision are given in Ramanan and Baker 2011.

Outlier/anomaly/novelty detection arises as an interesting problem in various contexts, from faults to frauds, and in detecting significant deviations from the past data, for example, churning customers. It is a very popular research area, and two comprehensive surveys include those by Hodge and Austin (2004) and Chandola, Banerjee, and Kumar (2009).

ADDITIVE MODELS

Nonparametric regression is discussed in detail in Härdle 1990. Hastie and Tibshirani (1990) discuss smoothing models and propose *additive models* where a multivariate function is written as a sum of univariate estimates. Locally weighted regression is discussed in Atkeson, Moore, and Schaal 1997. These models bear much similarity to radial basis functions and mixture of experts that we discuss in chapter 12.

In the condensed nearest neighbor algorithm, we saw that we can keep only a subset of the training instances, those that are close to the boundary, and we can define the discriminant using them only. This idea bears much similarity to the *support vector machines* that we discuss in chapter 13. There we also discuss various kernel functions to measure similarity between instances and how we can choose the best. Writing the prediction as a sum of the combined effects of training instances also underlies *Gaussian processes* (chapter 16), where a kernel function is called a *covariance function*.

8.11 Exercises

1. How can we have a smooth histogram?

   SOLUTION: We can interpolate between the two nearest bin centers. We can consider the bin centers as x^t, consider the histogram values as r^t, and use any interpolation scheme, linear or kernel-based.

2. Show equation 8.16.

   SOLUTION: Given that

   $$\hat{p}(\boldsymbol{x}|C_i) = \frac{k_i}{N_i V^k(\boldsymbol{x})} \quad \text{and} \quad \hat{P}(C_i) = \frac{N_i}{N}$$

   we can write

   $$\begin{aligned} \hat{P}(C_i|\boldsymbol{x}) &= \frac{\hat{p}(\boldsymbol{x}|C_i)\hat{P}(C_i)}{\sum_j \hat{p}(\boldsymbol{x}|C_j)\hat{P}(C_j)} = \frac{\frac{k_i}{N_i V^k(\boldsymbol{x})}\frac{N_i}{N}}{\sum_j \frac{k_j}{N_j V^k(\boldsymbol{x})}\frac{N_j}{N}} \\ &= \frac{k_i}{\sum_j k_j} = \frac{k_i}{k} \end{aligned}$$

3. Parametric regression (section 5.8) assumes Gaussian noise and hence is not robust to outliers; how can we make it more robust ?

4. How can we detect outliers after hierarchical clustering (section 7.8) ?

5. How does condensed nearest neighbor behave if $k > 1$?

   SOLUTION: When $k > 1$, to get full accuracy without any misclassification, it may be necessary to store an instance multiple times so that the correct class gets the majority of the votes. For example, if $k = 3$ and $\boldsymbol{x}$ has two neighbors both belonging to a different class, we need to store $\boldsymbol{x}$ twice (i.e., it gets added in two epochs), so that if $\boldsymbol{x}$ is seen during test, the majority (two in this case) out of three neighbors belong to the correct class.

6. In condensed nearest neighbor, an instance previously added to $\mathcal{Z}$ may no longer be necessary after a later addition. How can we find such instances that are no longer necessary?

7. In a regressogram, instead of averaging in a bin and doing a constant fit, we can use the instances falling in a bin and do a linear fit (see figure 8.14). Write the code and compare this with the regressogram proper.

8. Write the error function for loess discussed in section 8.8.3.

   SOLUTION: The output is calculated using a linear model $g(x) = ax+b$, where, in the running line smoother, we minimize

   $$E(a,b|x,\mathcal{X}) = \sum_t w\left(\frac{x-x^t}{h}\right)[r^t - (ax^t + b)]^2$$

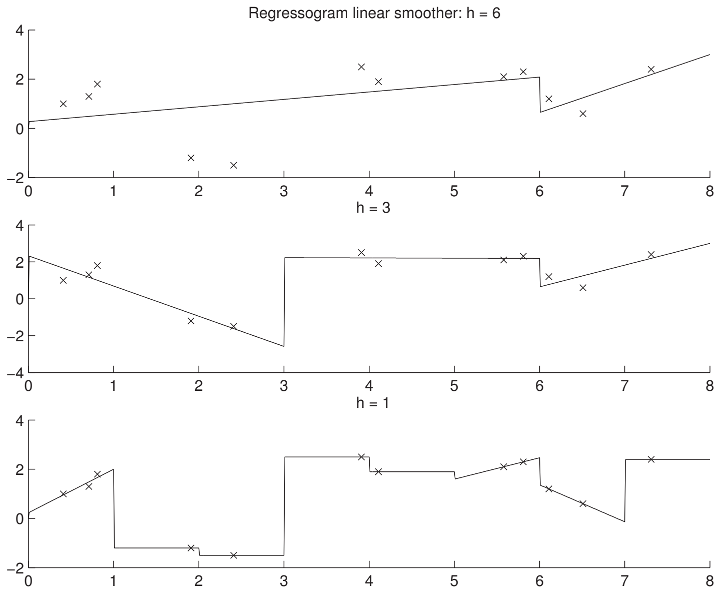


Figure 8.14 Regressograms with linear fits in bins for various bin lengths.

and

$$w(u) = \begin{cases} 1 & \text{if } |u| < 1 \\ 0 & \text{otherwise} \end{cases}$$

Note that we do not have one error function but rather, for each test input x, we have another error function taking into account only the data closest to x, which is minimized to fit a line in that neighborhood.

Loess is the weighted version of running line smoother where a kernel function $K(\cdot) \in (0, 1)$ replaces the $w(\cdot) \in \{0, 1\}$:

$$E(a, b|x, \mathcal{X}) = \sum_t K\left(\frac{x - x^t}{h}\right) [r^t - (ax^t + b)]^2$$

9. Propose an incremental version of the running mean estimator, which, like the condensed nearest neighbor, stores instances only when necessary.
10. Generalize kernel smoother to multivariate data.
11. In the running smoother, we can fit a constant, a line, or a higher-degree polynomial at a test point. How can we choose among them?

    SOLUTION: By cross-validation.

12. In the running mean smoother, besides giving an estimate, can we also calculate a confidence interval indicating the variance (uncertainty) around the estimate at that point?

8.12 References

Aha, D. W., ed. 1997. Special Issue on Lazy Learning. *Artificial Intelligence Review* 11 (1–5): 7–423.

Aha, D. W., D. Kibler, and M. K. Albert. 1991. “Instance-Based Learning Algorithm.” *Machine Learning* 6:37–66.

Atkeson, C. G., A. W. Moore, and S. Schaal. 1997. “Locally Weighted Learning.” *Artificial Intelligence Review* 11:11–73.

Bellet, A., A. Habrard, and M. Sebban. 2013. “A Survey on Metric Learning for Feature Vectors and Structured Data.” *arXiv:1306.6709v2*.

Breunig, M. M., H.-P. Kriegel, R. T. Ng, and J. Sander. 2000. “LOF: Identifying Density-Based Local Outliers.” In *ACM SIGMOD International Conference on Management of Data*, 93–104. New York: ACM Press.

Chandola, V., A. Banerjee, and V. Kumar. 2009. “Anomaly Detection: A Survey.” *ACM Computing Surveys* 41 (3): 15:1–15:58.

Chen, Y., E. K. Garcia, M. R. Gupta, A. Rahimi, and L. Cazzanti. 2009. “Similarity-Based Classification: Concepts and Algorithms.” *Journal of Machine Learning Research* 11:747–776.

Cover, T. M., and P. E. Hart. 1967. “Nearest Neighbor Pattern Classification.” *IEEE Transactions on Information Theory* 13:21–27.

Dasarathy, B. V. 1991. *Nearest Neighbor Norms: NN Pattern Classification Techniques.* Los Alamitos, CA: IEEE Computer Society Press.

Domeniconi, C., J. Peng, and D. Gunopulos. 2002. “Locally Adaptive Metric Nearest-Neighbor Classification.” *IEEE Transactions on Pattern Analysis and Machine Intelligence* 24:1281–1285.

Duda, R. O., P. E. Hart, and D. G. Stork. 2001. *Pattern Classification*, 2nd ed. New York: Wiley.

Geman, S., E. Bienenstock, and R. Doursat. 1992. “Neural Networks and the Bias/Variance Dilemma.” *Neural Computation* 4:1–58.

Härdle, W. 1990. *Applied Nonparametric Regression.* Cambridge, UK: Cambridge University Press.

Hart, P. E. 1968. “The Condensed Nearest Neighbor Rule.” *IEEE Transactions on Information Theory* 14:515–516.

Hastie, T. J., and R. J. Tibshirani. 1990. *Generalized Additive Models.* London: Chapman and Hall.

Hastie, T. J., and R. J. Tibshirani. 1996. “Discriminant Adaptive Nearest Neighbor Classification.” *IEEE Transactions on Pattern Analysis and Machine Intelligence* 18:607–616.

Hodge, V. J., and J. Austin. 2004. “A Survey of Outlier Detection Methodologies.” *Artificial Intelligence Review* 22:85–126.

Pekalska, E., and R. P. W. Duin. 2002. “Dissimilarity Representations Allow for Building Good Classifiers.” *Pattern Recognition Letters* 23:943–956.

Ramanan, D., and S. Baker. 2011. “Local Distance Functions: A Taxonomy, New Algorithms, and an Evaluation.” *IEEE Transactions on Pattern Analysis and Machine Intelligence* 33:794–806.

Scott, D. W. 1992. *Multivariate Density Estimation.* New York: Wiley.

Silverman, B. W. 1986. *Density Estimation in Statistics and Data Analysis.* London: Chapman and Hall.

Stanfill, C., and D. Waltz. 1986. “Toward Memory-Based Reasoning.” *Communications of the ACM* 29:1213–1228.

Webb, A. 1999. *Statistical Pattern Recognition.* London: Arnold.

Weinberger, K. Q., and L. K. Saul. 2009. “Distance Metric Learning for Large Margin Classification.” *Journal of Machine Learning Research* 10:207–244.

Wettschereck, D., D. W. Aha, and T. Mohri. 1997. “A Review and Empirical Evaluation of Feature Weighting Methods for a Class of Lazy Learning Algorithms.” *Artificial Intelligence Review* 11:273–314.

Wilfong, G. 1992. “Nearest Neighbor Problems.” *International Journal on Computational Geometry and Applications* 2:383–416.

9 *Decision Trees*

A decision tree is a hierarchical data structure implementing the divide-and-conquer strategy. It is an efficient nonparametric method, which can be used for both classification and regression. We discuss learning algorithms that build the tree from a given labeled training sample, as well as how the tree can be converted to a set of simple rules that are easy to understand. Another possibility is to learn a rule base directly.

9.1 Introduction

In parametric estimation, we define a model over the whole input space and learn its parameters from all of the training data. Then we use the same model and the same parameter set for any test input. In nonparametric estimation, we divide the input space into local regions, defined by a distance measure like the Euclidean norm, and for each input, the corresponding local model computed from the training data in that region is used. In the instance-based models we discussed in chapter 8, given an input, identifying the local data defining the local model is costly; it requires calculating the distances from the given input to all of the training instances, which is $\mathcal{O}(N)$.

DECISION TREE

A *decision tree* is a hierarchical model for supervised learning whereby the local region is identified in a sequence of recursive splits in a smaller number of steps. A decision tree is composed of internal decision nodes and terminal leaves (see figure 9.1). Each *decision node m* implements a test function $f_m(\boldsymbol{x})$ with discrete outcomes labeling the branches. Given an input, at each node, a test is applied and one of the branches is taken depending on the outcome. This process starts at the root and is repeated

DECISION NODE

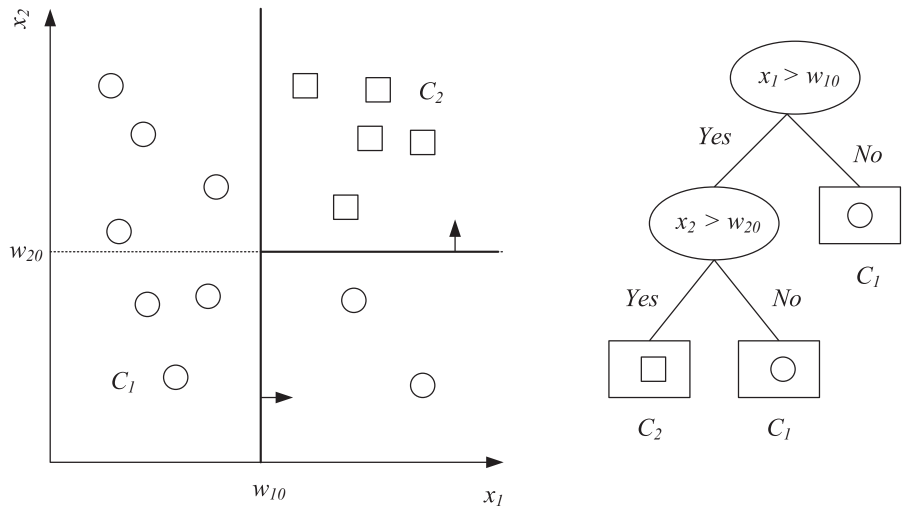


Figure 9.1 Example of a dataset and the corresponding decision tree. Oval nodes are the decision nodes and rectangles are leaf nodes. The univariate decision node splits along one axis, and successive splits are orthogonal to each other. After the first split, $\{\boldsymbol{x} | x_1 < w_{10}\}$ is pure and is not split further.

LEAF NODE recursively until a *leaf node* is hit, at which point the value written in the leaf constitutes the output.

A decision tree is also a nonparametric model in the sense that we do not assume any parametric form for the class densities and the tree structure is not fixed a priori but the tree grows, branches and leaves are added, during learning depending on the complexity of the problem inherent in the data.

Each $f_m(\boldsymbol{x})$ defines a discriminant in the d-dimensional input space dividing it into smaller regions that are further subdivided as we take a path from the root down. $f_m(\cdot)$ is a simple function and when written down as a tree, a complex function is broken down into a series of simple decisions. Different decision tree methods assume different models for $f_m(\cdot)$, and the model class defines the shape of the discriminant and the shape of regions. Each leaf node has an output label, which in the case of classification is the class code and in regression is a numeric value. A leaf node defines a localized region in the input space where instances falling in this region have the same labels (in classification), or very similar numeric outputs (in regression). The boundaries of the regions are defined by the discriminants that are coded in the internal nodes on the path from the root to the leaf node.

The hierarchical placement of decisions allows a fast localization of the region covering an input. For example, if the decisions are binary, then in the best case, each decision eliminates half of the cases. If there are b regions, then in the best case, the correct region can be found in $\log_2 b$ decisions. Another advantage of the decision tree is interpretability. As we will see shortly, the tree can be converted to a set of *IF-THEN rules* that are easily understandable. For this reason, decision trees are very popular and sometimes preferred over more accurate but less interpretable methods.

We start with univariate trees where the test in a decision node uses only one input variable and we see how such trees can be constructed for classification and regression. We later generalize this to multivariate trees where all inputs can be used in an internal node.

9.2 Univariate Trees

UNIVARIATE TREE

In a *univariate tree*, in each internal node, the test uses only one of the input dimensions. If the used input dimension, x_j, is discrete, taking one of n possible values, the decision node checks the value of x_j and takes the corresponding branch, implementing an n-way split. For example, if an attribute is color $\in$ {red, blue, green}, then a node on that attribute has three branches, each one corresponding to one of the three possible values of the attribute.

A decision node has discrete branches and a numeric input should be discretized. If x_j is numeric (ordered), the test is a comparison

$$f_m(\boldsymbol{x}) : x_j > w_{m0} \tag{9.1}$$

BINARY SPLIT

where w_{m0} is a suitably chosen threshold value. The decision node divides the input space into two: $L_m = \{\boldsymbol{x} | x_j > w_{m0}\}$ and $R_m = \{\boldsymbol{x} | x_j \le w_{m0}\}$; this is called a *binary split*. Successive decision nodes on a path from the root to a leaf further divide these into two using other attributes and generating splits orthogonal to each other. The leaf nodes define hyperrectangles in the input space (see figure 9.1).

Tree induction is the construction of the tree given a training sample. For a given training set, there exists many trees that code it with no error, and, for simplicity, we are interested in finding the smallest among them, where tree size is measured as the number of nodes in the tree and the complexity of the decision nodes. Finding the smallest tree is NP-complete (Quinlan 1986), and we are forced to use local search procedures based on heuristics that give reasonable trees in reasonable time.

Tree learning algorithms are greedy and, at each step, starting at the root with the complete training data, we look for the best split. This splits the training data into two or n, depending on whether the chosen attribute is numeric or discrete. We then continue splitting recursively with the corresponding subset until we do not need to split anymore, at which point a leaf node is created and labeled.

9.2.1 Classification Trees

CLASSIFICATION TREE

IMPURITY MEASURE

In the case of a decision tree for classification, namely, a *classification tree*, the goodness of a split is quantified by an *impurity measure*. A split is pure if after the split, for all branches, all the instances choosing a branch belong to the same class. Let us say for node m, N_m is the number of training instances reaching node m. For the root node, it is N. N_m^i of N_m belong to class C_i, with $\sum_i N_m^i = N_m$. Given that an instance reaches node m, the estimate for the probability of class C_i is

$$\hat{P}(C_i|\boldsymbol{x}, m) \equiv p_m^i = \frac{N_m^i}{N_m} \tag{9.2}$$

Node m is pure if p_m^i for all i are either 0 or 1. It is 0 when none of the instances reaching node m are of class C_i, and it is 1 if all such instances are of C_i. If the split is pure, we do not need to split any further and can add a leaf node labeled with the class for which p_m^i is 1. One possible function to measure impurity is *entropy* (Quinlan 1986) (see figure 9.2):

ENTROPY

$$\mathcal{I}_m = -\sum_{i=1}^{K} p_m^i \log_2 p_m^i \tag{9.3}$$

where $0 \log 0 \equiv 0$. Entropy in information theory specifies the minimum number of bits needed to encode the class code of an instance. In a two-class problem, if $p^1 = 1$ and $p^2 = 0$, all examples are of C^1, and we do not need to send anything, and the entropy is 0. If $p^1 = p^2 = 0.5$, we need to send a bit to signal one of the two cases, and the entropy is 1. In between these two extremes, we can devise codes and use less than a bit per message by having shorter codes for the more likely class and

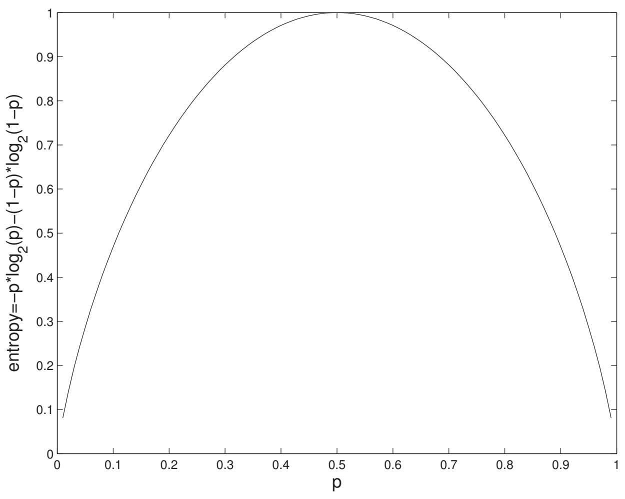


Figure 9.2 Entropy function for a two-class problem.

longer codes for the less likely. When there are $K > 2$ classes, the same discussion holds and the largest entropy is $\log_2 K$ when $p^i = 1/K$.

But entropy is not the only possible measure. For a two-class problem where $p^1 \equiv p$ and $p^2 = 1 - p$, $\phi(p, 1-p)$ is a nonnegative function measuring the impurity of a split if it satisfies the following properties (Devroye, Györfi, and Lugosi 1996):

- $\phi(1/2, 1/2) \geq \phi(p, 1-p)$, for any $p \in [0, 1]$.
- $\phi(0, 1) = \phi(1, 0) = 0$.
- $\phi(p, 1-p)$ is increasing in p on $[0, 1/2]$ and decreasing in p on $[1/2, 1]$.

Examples are

1. Entropy

(9.4) $$\phi(p, 1-p) = -p \log_2 p - (1-p) \log_2 (1-p)$$

Equation 9.3 is the generalization to $K > 2$ classes.

Gini index

2. *Gini index* (Breiman et al. 1984)

(9.5) $$\phi(p, 1-p) = 2p(1-p)$$

3. Misclassification error

$$\phi(p, 1-p) = 1 - \max(p, 1-p) \tag{9.6}$$

These can be generalized to $K > 2$ classes, and the misclassification error can be generalized to minimum risk given a loss function (exercise 1). Research has shown that there is not a significant difference between these three measures.

If node m is not pure, then the instances should be split to decrease impurity, and there are multiple possible attributes on which we can split. For a numeric attribute, multiple split positions are possible. Among all, we look for the split that minimizes impurity after the split because we want to generate the smallest tree. If the subsets after the split are closer to pure, fewer splits (if any) will be needed afterward. Of course this is locally optimal, and we have no guarantee of finding the smallest decision tree.

Let us say at node m, N_{mj} of N_m take branch j; these are $\boldsymbol{x}^t$ for which the test $f_m(\boldsymbol{x}^t)$ returns outcome j. For a discrete attribute with n values, there are n outcomes, and for a numeric attribute, there are two outcomes ($n = 2$), in either case satisfying $\sum_{j=1}^{n} N_{mj} = N_m$. N_{mj}^i of N_{mj} belong to class C_i: $\sum_{i=1}^{K} N_{mj}^i = N_{mj}$. Similarly, $\sum_{j=1}^{n} N_{mj}^i = N_m^i$.

Then given that at node m, the test returns outcome j, the estimate for the probability of class C_i is

$$\hat{P}(C_i|\boldsymbol{x}, m, j) \equiv p_{mj}^i = \frac{N_{mj}^i}{N_{mj}} \tag{9.7}$$

and the total impurity after the split is given as

$$\mathcal{I}'_m = -\sum_{j=1}^{n} \frac{N_{mj}}{N_m} \sum_{i=1}^{K} p_{mj}^i \log_2 p_{mj}^i \tag{9.8}$$

In the case of a numeric attribute, to be able to calculate p_{mj}^i using equation 9.1, we also need to know w_{m0} for that node. There are $N_m - 1$ possible w_{m0} between N_m data points: We do not need to test for all (possibly infinite) points; it is enough to test, for example, at halfway between points. Note also that the best split is always between adjacent points belonging to different classes. So we try them, and the best in terms of purity is taken for the purity of the attribute. In the case of a discrete attribute, no such iteration is necessary.

```
GenerateTree($\mathcal{X}$)
    If NodeEntropy($\mathcal{X}$)< $\theta_I$ /* equation 9.3 */
      Create leaf labelled by majority class in $\mathcal{X}$
      Return
    $i$ ← SplitAttribute($\mathcal{X}$)
    For each branch of $\boldsymbol{x}_i$
      Find $\mathcal{X}_i$ falling in branch
      GenerateTree($\mathcal{X}_i$)

SplitAttribute($\mathcal{X}$)
    MinEnt← MAX
    For all attributes $i = 1, \ldots, d$
        If $\boldsymbol{x}_i$ is discrete with $n$ values
          Split $\mathcal{X}$ into $\mathcal{X}_1, \ldots, \mathcal{X}_n$ by $\boldsymbol{x}_i$
          e ← SplitEntropy($\mathcal{X}_1, \ldots, \mathcal{X}_n$) /* equation 9.8 */
          If e<MinEnt MinEnt ← e; bestf ← i
        Else /* $\boldsymbol{x}_i$ is numeric */
          For all possible splits
              Split $\mathcal{X}$ into $\mathcal{X}_1, \mathcal{X}_2$ on $\boldsymbol{x}_i$
              e←SplitEntropy($\mathcal{X}_1, \mathcal{X}_2$)
              If e<MinEnt MinEnt ← e; bestf ← i
    Return bestf
```

Figure 9.3 Classification tree construction.

So for all attributes, discrete and numeric, and for a numeric attribute for all split positions, we calculate the impurity and choose the one that has the minimum entropy, for example, as measured by equation 9.8. Then tree construction continues recursively and in parallel for all the branches that are not pure, until all are pure. This is the basis of the *classification and regression tree* (CART) algorithm (Breiman et al. 1984), *ID3* algorithm (Quinlan 1986), and its extension *C4.5* (Quinlan 1993). The pseudocode of the algorithm is given in figure 9.3.

CLASSIFICATION AND REGRESSION TREE
ID3
C4.5

It can also be said that at each step during tree construction, we choose the split that causes the largest decrease in impurity, which is the difference between the impurity of data reaching node m (equation 9.3) and the total entropy of data reaching its branches after the split (equation 9.8).

One problem is that such splitting favors attributes with many values. When there are many values, there are many branches, and the impurity can be much less. For example, if we take training index t as an attribute, the impurity measure will choose that because then the impurity of each branch is 0, although it is not a reasonable feature. Nodes with many branches are complex and go against our idea of splitting class discriminants into simple decisions. Methods have been proposed to penalize such attributes and to balance the impurity drop and the branching factor.

When there is noise, growing the tree until it is purest, we may grow a very large tree and it overfits; for example, consider the case of a mislabeled instance amid a group of correctly labeled instances. To alleviate such overfitting, tree construction ends when nodes become pure enough, namely, a subset of data is not split further if $\mathcal{I} < \theta_I$. This implies that we do not require that p^i_{mj} be exactly 0 or 1 but close enough, with a threshold θ_p. In such a case, a leaf node is created and is labeled with the class having the highest p^i_{mj}.

θ_I (or θ_p) is the complexity parameter, like h or k of nonparametric estimation. When they are small, the variance is high and the tree grows large to reflect the training set accurately, and when they are large, variance is lower and a smaller tree roughly represents the training set and may have large bias. The ideal value depends on the cost of misclassification, as well as the costs of memory and computation.

It is generally advised that in a leaf, one stores the posterior probabilities of classes, instead of labeling the leaf with the class having the highest posterior. These probabilities may be required in later steps, for example, in calculating risks. Note that we do not need to store the instances reaching the node or the exact counts; just ratios suffice.

9.2.2 Regression Trees

REGRESSION TREE

A *regression tree* is constructed in almost the same manner as a classification tree, except that the impurity measure that is appropriate for classification is replaced by a measure appropriate for regression. Let us say for node m, $\mathcal{X}_m$ is the subset of $\mathcal{X}$ reaching node m; namely, it is the set of all $\boldsymbol{x} \in \mathcal{X}$ satisfying all the conditions in the decision nodes on the path from the root until node m. We define

$$b_m(\boldsymbol{x}) = \begin{cases} 1 & \text{if } \boldsymbol{x} \in \mathcal{X}_m\text{: } \boldsymbol{x} \text{ reaches node } m \\ 0 & \text{otherwise} \end{cases} \tag{9.9}$$

In regression, the goodness of a split is measured by the mean square error from the estimated value. Let us say g_m is the estimated value in node m.

$$E_m = \frac{1}{N_m} \sum_t (r^t - g_m)^2 b_m(\boldsymbol{x}^t) \tag{9.10}$$

where $N_m = |\mathcal{X}_m| = \sum_t b_m(\boldsymbol{x}^t)$.

In a node, we use the mean (median if there is too much noise) of the required outputs of instances reaching the node

$$g_m = \frac{\sum_t b_m(\boldsymbol{x}^t) r^t}{\sum_t b_m(\boldsymbol{x}^t)} \tag{9.11}$$

Then equation 9.10 corresponds to the variance at m. If at a node, the error is acceptable, that is, $E_m < \theta_r$, then a leaf node is created and it stores the g_m value. Just like the regressogram of chapter 8, this creates a piecewise constant approximation with discontinuities at leaf boundaries.

If the error is not acceptable, data reaching node m is split further such that the sum of the errors in the branches is minimum. As in classification, at each node, we look for the attribute (and split threshold for a numeric attribute) that minimizes the error, and then we continue recursively.

Let us define $\mathcal{X}_{mj}$ as the subset of $\mathcal{X}_m$ taking branch j: $\cup_{j=1}^n \mathcal{X}_{mj} = \mathcal{X}_m$. We define

$$b_{mj}(\boldsymbol{x}) = \begin{cases} 1 & \text{if } \boldsymbol{x} \in \mathcal{X}_{mj}\text{: } \boldsymbol{x} \text{ reaches node } m \text{ and takes branch } j \\ 0 & \text{otherwise} \end{cases} \tag{9.12}$$

g_{mj} is the estimated value in branch j of node m.

$$g_{mj} = \frac{\sum_t b_{mj}(\boldsymbol{x}^t) r^t}{\sum_t b_{mj}(\boldsymbol{x}^t)} \tag{9.13}$$

and the error after the split is

$$E'_m = \frac{1}{N_m} \sum_j \sum_t (r^t - g_{mj})^2 b_{mj}(\boldsymbol{x}^t) \tag{9.14}$$

The drop in error for any split is given as the difference between equation 9.10 and equation 9.14. We look for the split such that this drop is maximum or, equivalently, where equation 9.14 takes its minimum. The code given in figure 9.3 can be adapted to training a regression tree by replacing entropy calculations with mean square error and class labels with averages.

Mean square error is one possible error function; another is worst possible error

$$E_m = \max_j \max_t |r^t - g_{mj}| b_{mj}(\boldsymbol{x}^t) \tag{9.15}$$

and using this, we can guarantee that the error for any instance is never larger than a given threshold.

The acceptable error threshold is the complexity parameter; when it is small, we generate large trees and risk overfitting; when it is large, we underfit and smooth too much (see figures 9.4 and 9.5).

Similar to going from running mean to running line in nonparametric regression, instead of taking an average at a leaf that implements a constant fit, we can also do a linear regression fit over the instances choosing the leaf:

$$g_m(\boldsymbol{x}) = \boldsymbol{w}_m^T \boldsymbol{x} + w_{m0} \tag{9.16}$$

This makes the estimate in a leaf dependent on $\boldsymbol{x}$ and generates smaller trees, but there is the expense of extra computation at a leaf node.

9.3 Pruning

Frequently, a node is not split further if the number of training instances reaching a node is smaller than a certain percentage of the training set—for example, 5 percent—regardless of the impurity or error. The idea is that any decision based on too few instances causes variance and thus generalization error. Stopping tree construction early on before it is full is called *prepruning* the tree.

PREPRUNING

POSTPRUNING

Another possibility to get simpler trees is *postpruning*, which in practice works better than prepruning. We saw before that tree growing is greedy and at each step, we make a decision, namely, generate a decision node, and continue further on, never backtracking and trying out an alternative. The only exception is postpruning where we try to find and prune unnecessary subtrees.

PRUNING SET

In postpruning, we grow the tree full until all leaves are pure and we have no training error. We then find subtrees that cause overfitting and we prune them. From the initial labeled set, we set aside a *pruning set*, unused during training. For each subtree, we replace it with a leaf node

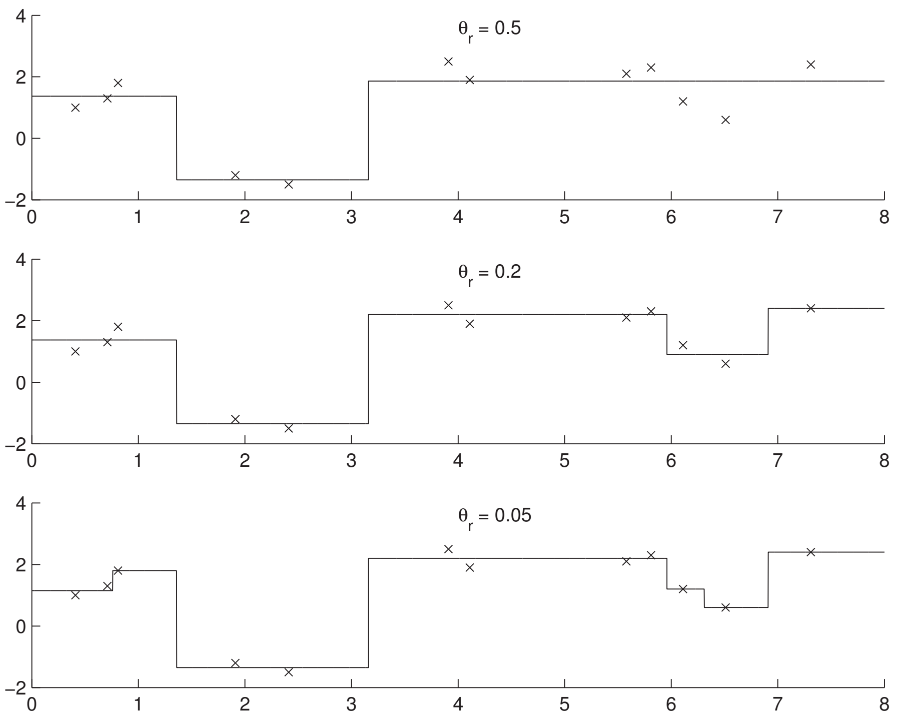


Figure 9.4 Regression tree smooths for various values of θ_r. The corresponding trees are given in figure 9.5.

labeled with the training instances covered by the subtree (appropriately for classification or regression). If the leaf node does not perform worse than the subtree on the pruning set, we prune the subtree and keep the leaf node because the additional complexity of the subtree is not justified; otherwise, we keep the subtree.

For example, in the third tree of figure 9.5, there is a subtree starting with condition $x < 6.31$. This subtree can be replaced by a leaf node of $y = 0.9$ (as in the second tree) if the error on the pruning set does not increase during the substitution. Note that the pruning set should not be confused with (and is distinct from) the validation set.

Comparing prepruning and postpruning, we can say that prepruning is faster but postpruning generally leads to more accurate trees.

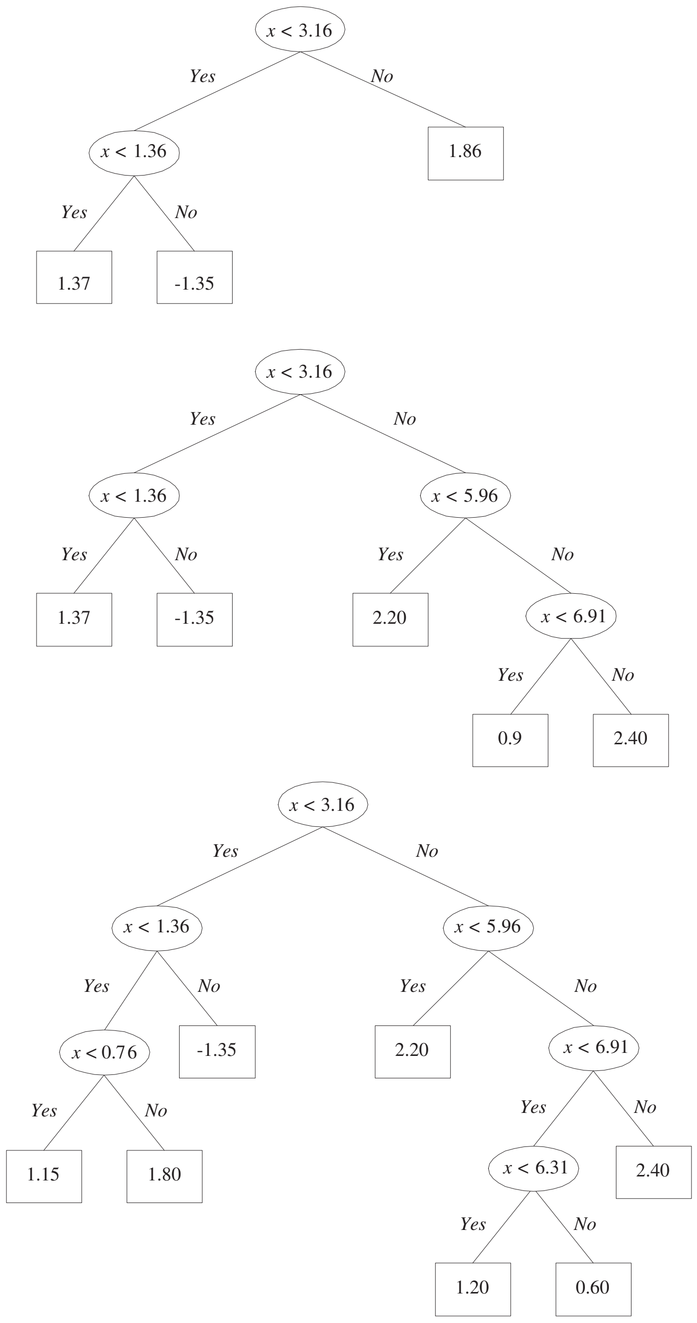


Figure 9.5 Regression trees implementing the smooths of figure 9.4 for various values of θ_r.

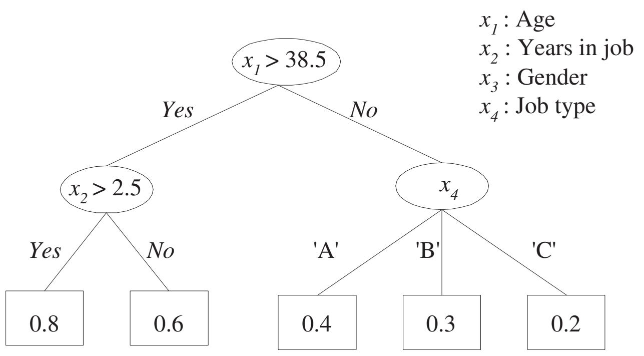


Figure 9.6 Example of a (hypothetical) decision tree. Each path from the root to a leaf can be written down as a conjunctive rule, composed of conditions defined by the decision nodes on the path.

9.4 Rule Extraction from Trees

A decision tree does its own feature extraction. The univariate tree only uses the necessary variables, and after the tree is built, certain features may not be used at all. We can also say that features closer to the root are more important globally. For example, the decision tree given in figure 9.6 uses x_1, x_2, and x_4, but not x_3. It is possible to use a decision tree for feature extraction: we build a tree and then take only those features used by the tree as inputs to another learning method.

INTERPRETABILITY

Another main advantage of decision trees is *interpretability*: The decision nodes carry conditions that are simple to understand. Each path from the root to a leaf corresponds to one conjunction of tests, as all those conditions should be satisfied to reach to the leaf. These paths together can be written down as a set of *IF-THEN rules*, called a *rule base*. One such method is *C4.5Rules* (Quinlan 1993).

IF-THEN RULES

For example, the decision tree of figure 9.6 can be written down as the following set of rules:

R1: IF (age $>$ 38.5) AND (years-in-job $>$ 2.5) THEN $y = 0.8$
R2: IF (age $>$ 38.5) AND (years-in-job $\leq$ 2.5) THEN $y = 0.6$
R3: IF (age $\leq$ 38.5) AND (job-type = ‘A’) THEN $y = 0.4$
R4: IF (age $\leq$ 38.5) AND (job-type = ‘B’) THEN $y = 0.3$
R5: IF (age $\leq$ 38.5) AND (job-type = ‘C’) THEN $y = 0.2$

KNOWLEDGE EXTRACTION

Such a rule base allows *knowledge extraction*; it can be easily understood and allows experts to verify the model learned from data. For each rule, one can also calculate the percentage of training data covered by the rule, namely, *rule support*. The rules reflect the main characteristics of the dataset: They show the important features and split positions. For instance, in this (hypothetical) example, we see that in terms of our purpose (y), people who are thirty-eight years old or less are different from people who are thirty-nine or more years old. And among this latter group, it is the job type that makes them different, whereas in the former group, it is the number of years in a job that is the best discriminating characteristic.

RULE SUPPORT

In the case of a classification tree, there may be more than one leaf labeled with the same class. In such a case, these multiple conjunctive expressions corresponding to different paths can be combined as a disjunction (OR). The class region then corresponds to a union of these multiple patches, each patch corresponding to the region defined by one leaf. For example, class C_1 of figure 9.1 is written as

IF $(x \le w_{10})$ OR $((x_1 > w_{10})$ AND $(x_2 \le w_{20}))$ THEN C_1

PRUNING RULES

Pruning rules is possible for simplification. Pruning a subtree corresponds to pruning terms from a number of rules at the same time. It may be possible to prune a term from one rule without touching other rules. For example, in the previous rule set, for R3, if we see that all whose `job-type='A'` have outcomes close to 0.4, regardless of `age`, $R3$ can be pruned as

$R3'$: IF (job-type='A') THEN y =0.4

Note that after the rules are pruned, it may not be possible to write them back as a tree anymore.

9.5 Learning Rules from Data

RULE INDUCTION

As we have just seen, one way to get IF-THEN rules is to train a decision tree and convert it to rules. Another is to learn the rules directly. *Rule induction* works similar to tree induction except that rule induction does a depth-first search and generates one path (rule) at a time, whereas tree induction goes breadth-first and generates all paths simultaneously.

Rules are learned one at a time. Each rule is a conjunction of conditions on discrete or numeric attributes (as in decision trees) and these conditions are added one at a time, to optimize some criterion, for example, minimize entropy. A rule is said to *cover* an example if the example satisfies all the conditions of the rule. Once a rule is grown and pruned, it is added to the rule base and all the training examples covered by the rule are removed from the training set, and the process continues until enough rules are added. This is called *sequential covering*. There is an outer loop of adding one rule at a time to the rule base and an inner loop of adding one condition at a time to the current rule. These steps are both greedy and do not guarantee optimality. Both loops have a pruning step for better generalization.

SEQUENTIAL COVERING

RIPPER

IREP

One example of a rule induction algorithm is *Ripper* (Cohen 1995), based on an earlier algorithm *Irep* (Fürnkranz and Widmer 1994). We start with the case of two classes where we talk of positive and negative examples, then later generalize to $K > 2$ classes. Rules are added to explain positive examples such that if an instance is not covered by any rule, then it is classified as negative. So a rule when it matches is either correct (true positive), or it causes a false positive. The pseudocode of the outer loop of Ripper is given in figure 9.7.

FOIL

In Ripper, conditions are added to the rule to maximize an information gain measure used in Quinlan's (1990) *Foil* algorithm. Let us say we have rule R and R' is the candidate rule after adding a condition. Change in gain is defined as

$$Gain(R', R) = s \cdot \left(\log_2 \frac{N'_+}{N'} - \log_2 \frac{N_+}{N} \right) \tag{9.17}$$

where N is the number of instances that are covered by R and N_+ is the number of true positives in them. N' and N'_+ are similarly defined for R'. s is the number of true positives in R, which are still true positives in R', after adding the condition. In terms of information theory, the change in gain measures the reduction in bits to encode a positive instance.

RULE VALUE METRIC

Conditions are added to a rule until it covers no negative example. Once a rule is grown, it is pruned back by deleting conditions in reverse order, to find the rule that maximizes the *rule value metric*

$$rvm(R) = \frac{p - n}{p + n} \tag{9.18}$$

where p and n are the number of true and false positives, respectively, on the pruning set, which is one-third of the data, having used two-thirds as the growing set.

```
Ripper(Pos,Neg,$k$)
    RuleSet ← LearnRuleSet(Pos,Neg)
    For $k$ times
        RuleSet ← OptimizeRuleSet(RuleSet,Pos,Neg)
LearnRuleSet(Pos,Neg)
    RuleSet ← ∅
    DL ← DescLen(RuleSet,Pos,Neg)
    Repeat
        Rule ← LearnRule(Pos,Neg)
        Add Rule to RuleSet
        DL' ← DescLen(RuleSet,Pos,Neg)
        If DL'>DL+64
          PruneRuleSet(RuleSet,Pos,Neg)
          Return RuleSet
        If DL'<DL DL ← DL'
          Delete instances covered by Rule from Pos and Neg
    Until Pos = ∅
    Return RuleSet
PruneRuleSet(RuleSet,Pos,Neg)
    For each Rule ∈ RuleSet in reverse order
        DL ← DescLen(RuleSet,Pos,Neg)
        DL' ← DescLen(RuleSet-Rule,Pos,Neg)
        IF DL'<DL Delete Rule from RuleSet
    Return RuleSet
OptimizeRuleSet(RuleSet,Pos,Neg)
    For each Rule ∈ RuleSet
        DL0 ← DescLen(RuleSet,Pos,Neg)
        DL1 ← DescLen(RuleSet-Rule+
                ReplaceRule(RuleSet,Pos,Neg),Pos,Neg)
        DL2 ← DescLen(RuleSet-Rule+
                ReviseRule(RuleSet,Rule,Pos,Neg),Pos,Neg)
        If DL1=min(DL0,DL1,DL2)
                Delete Rule from RuleSet and
                        add ReplaceRule(RuleSet,Pos,Neg)
        Else If DL2=min(DL0,DL1,DL2)
                Delete Rule from RuleSet and
                        add ReviseRule(RuleSet,Rule,Pos,Neg)
    Return RuleSet
```

Figure 9.7 Ripper algorithm for learning rules. Only the outer loop is given; the inner loop is similar to adding nodes in a decision tree.

Once a rule is grown and pruned, all positive and negative training examples covered by the rule are removed from the training set. If there are remaining positive examples, rule induction continues. In the case of noise, we may stop early, namely, when a rule does not explain enough number of examples. To measure the worth of a rule, minimum description length (section 4.8) is used (Quinlan 1995). Typically, we stop if the description of the rule is not shorter than the description of instances it explains. The description length of a rule base is the sum of the description lengths of all the rules in the rule base, plus the description of instances not covered by the rule base. Ripper stops adding rules when the description length of the rule base is more than 64 bits larger than the best description length so far. Once the rule base is learned, we pass over the rules in reverse order to see if they can be removed without increasing the description length.

Rules in the rule base are also optimized after they are learned. Ripper considers two alternatives to a rule: One, called the replacement rule, starts from an empty rule, is grown, and is then pruned. The second, called the revision rule, starts with the rule as it is, is grown, and is then pruned. These two are compared with the original rule, and the shortest of three is added to the rule base. This optimization of the rule base can be done k times, typically twice.

When there are $K > 2$ classes, they are ordered in terms of their prior probabilities such that C_1 has the lowest prior probability and C_K has the highest. Then a sequence of two-class problems are defined such that, first, instances belonging to C_1 are taken as positive examples and instances of all other classes are taken as negative examples. Then, having learned C_1 and all its instances removed, it learns to separate C_2 from $C_3, \ldots, C_K$. This process is repeated until only C_K remains. The empty default rule is then labeled C_K, so that if an instance is not covered by any rule, it will be assigned to C_K.

PROPOSITIONAL RULES
FIRST-ORDER RULES

For a training set of size N, Ripper's complexity is $\mathcal{O}(N \log^2 N)$ and is an algorithm that can be used on very large training sets (Dietterich 1997). The rules we learn are *propositional rules*. More expressive, *first-order rules* have variables in conditions, called *predicates*. A *predicate* is a function that returns true or false depending on the value of its argument. Predicates therefore allow defining relations between the values of attributes, which cannot be done by propositions (Mitchell 1997):

IF Father(y, x) AND Female(y) THEN Daughter(x, y)

INDUCTIVE LOGIC PROGRAMMING

Such rules can be seen as programs in a logic programming language, such as Prolog, and learning them from data is called *inductive logic programming*. One such algorithm is Foil (Quinlan 1990).

BINDING

Assigning a value to a variable is called *binding*. A rule matches if there is a set of bindings to the variables existing in the training set. Learning first-order rules is similar to learning propositional rules with an outer loop of adding rules, and an inner loop of adding conditions to a rule, with prunings at the end of each loop. The difference is in the inner loop, where at each step we consider one predicate to add (instead of a proposition) and check the increase in the performance of the rule (Mitchell 1997). To calculate the performance of a rule, we consider all possible bindings of the variables, count the number of positive and negative bindings in the training set, and use, for example, equation 9.17. In this first-order case, we have predicates instead of propositions, so they should be previously defined, and the training set is a set of predicates known to be true.

9.6 Multivariate Trees

MULTIVARIATE TREE

In the case of a univariate tree, only one input dimension is used at a split. In a *multivariate tree*, at a decision node, all input dimensions can be used and thus it is more general. When all inputs are numeric, a binary linear multivariate node is defined as

$$f_m(\boldsymbol{x}) : \boldsymbol{w}_m^T \boldsymbol{x} + w_{m0} > 0 \tag{9.19}$$

Because the linear multivariate node takes a weighted sum, discrete attributes should be represented by 0/1 dummy numeric variables. Equation 9.19 defines a hyperplane with arbitrary orientation (see figure 9.8). Successive nodes on a path from the root to a leaf further divide these, and leaf nodes define polyhedra in the input space. The univariate node with a numeric feature is a special case when all but one of w_{mj} are 0. Thus the univariate numeric node of equation 9.1 also defines a linear discriminant but one that is orthogonal to axis x_j, intersecting it at w_{m0} and parallel to all other x_i. We therefore see that in a univariate node there are d possible orientations ($\boldsymbol{w}_m$) and $N_m - 1$ possible thresholds ($-w_{m0}$), making an exhaustive search possible. In a multivariate node, there are $2^d \binom{N_m}{d}$ possible hyperplanes (Murthy, Kasif, and Salzberg 1994) and an exhaustive search is no longer practical.

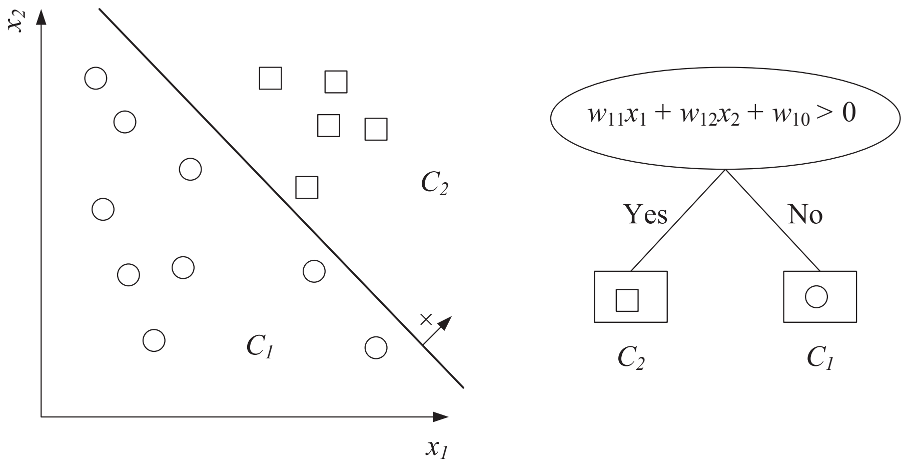


Figure 9.8 Example of a linear multivariate decision tree. The linear multivariate node can place an arbitrary hyperplane and thus is more general, whereas the univariate node is restricted to axis-aligned splits.

When we go from a univariate node to a linear multivariate node, the node becomes more flexible. It is possible to make it even more flexible by using a nonlinear multivariate node. For example, with a quadratic, we have

(9.20) $$f_m(\boldsymbol{x}) : \boldsymbol{x}^T \mathbf{W}_m \boldsymbol{x} + \boldsymbol{w}_m^T \boldsymbol{x} + w_{m0} > 0$$

Guo and Gelfand (1992) propose to use a multilayer perceptron (chapter 11) that is a linear sum of nonlinear basis functions, and this is another way of having nonlinear decision nodes. Another possibility is a

SPHERE NODE

sphere node (Devroye, Györfi, and Lugosi 1996)

(9.21) $$f_m(\boldsymbol{x}) : \|\boldsymbol{x} - \boldsymbol{c}_m\| \le \alpha_m$$

where $\boldsymbol{c}_m$ is the center and α_m is the radius.

There are a number of algorithms proposed for learning multivariate decision trees for classification: The earliest is the multivariate version of the CART algorithm (Breiman et al. 1984), which fine-tunes the weights w_{mj} one by one to decrease impurity. CART also has a preprocessing stage to decrease dimensionality through subset selection (chapter 6) and reduce the complexity of the node. An algorithm with some extensions

OC1

to CART is the *OC1* algorithm (Murthy, Kasif, and Salzberg 1994). One possibility (Loh and Vanichsetakul 1988) is to assume that all classes are Gaussian with a common covariance matrix, thereby having linear discriminants separating each class from the others (chapter 5). In such a case, with K classes, each node has K branches and each branch carries the discriminant separating one class from the others. Brodley and Utgoff (1995) propose a method where the linear discriminants are trained to minimize classification error (chapter 10). Guo and Gelfand (1992) propose a heuristic to group $K > 2$ classes into two supergroups, and then binary multivariate trees can be learned. Loh and Shih (1997) use 2-means clustering (chapter 7) to group data into two. Yıldız and Alpaydın (2000) use LDA (chapter 6) to find the discriminant once the classes are grouped into two.

Any classifier approximates the real (unknown) discriminant choosing one hypothesis from its hypothesis class. When we use univariate nodes, our approximation uses piecewise, axis-aligned hyperplanes. With linear multivariate nodes, we can use arbitrary hyperplanes and do a better approximation using fewer nodes. If the underlying discriminant is curved, nonlinear nodes work better. The branching factor has a similar effect in that it specifies the number of discriminants that a node defines. A binary decision node with two branches defines one discriminant separating the input space into two. An n-way node separates into n. Thus, there is a dependency among the complexity of a node, the branching factor, and tree size. With simple nodes and low branching factors, one may grow large trees, but such trees, for example, with univariate binary nodes, are more interpretable. Linear multivariate nodes are more difficult to interpret. More complex nodes also require more data and are prone to overfitting as we get down the tree and have less and less data. If the nodes are complex and the tree is small, we also lose the main idea of the tree, which is that of dividing the problem into a set of simple problems. After all, we can have a very complex classifier in the root that separates all classes from each other, but then this will not be a tree!

9.7 Notes

Divide-and-conquer is a frequently used heuristic that has been used since the days of Caesar to break a complex problem, for example, Gaul, into a group of simpler problems. Trees are frequently used in computer science to decrease complexity from linear to log time. Decision trees were made popular in statistics in Breiman et al. 1984 and in machine learning in Quinlan 1986 and Quinlan 1993. Multivariate tree induction methods became popular more recently; a review and comparison on many datasets are given in Yıldız and Alpaydın 2000. Many researchers (e.g., Guo and Gelfand 1992), proposed to combine the simplicity of trees with the accuracy of multilayer perceptrons (chapter 11). Many studies, however, have concluded that the univariate trees are quite accurate and interpretable, and the additional complexity brought by linear (or nonlinear) multivariate nodes is hardly justified. A recent survey is given by Rokach and Maimon (2005).

OMNIVARIATE DECISION TREE

The *omnivariate decision tree* (Yıldız and Alpaydın 2001) is a hybrid tree architecture where the tree may have univariate, linear multivariate, or nonlinear multivariate nodes. The idea is that during construction, at each decision node, which corresponds to a different subproblem defined by the subset of the training data reaching that node, a different model may be appropriate and the appropriate one should be found and used. Using the same type of nodes everywhere corresponds to assuming that the same inductive bias is good in all parts of the input space. In an omnivariate tree, at each node, candidate nodes of different types are trained and compared using a statistical test (chapter 19) on a validation set to determine which one generalizes the best. The simpler one is chosen unless a more complex one is shown to have significantly higher accuracy. Results show that more complex nodes are used early in the tree, closer to the root, and as we go down the tree, simple univariate nodes suffice. As we get closer to the leaves, we have simpler problems and, at the same time, we have less data. In such a case, complex nodes overfit and are rejected by the statistical test. The number of nodes increases exponentially as we go down the tree; therefore, a large majority of the nodes are univariate and the overall complexity does not increase much.

Decision trees are used more frequently for classification than for regression. They are very popular: They learn and respond quickly, and are accurate in many domains (Murthy 1998). It is even the case that a decision tree is preferred over more accurate methods, because it is interpretable. When written down as a set of IF-THEN rules, the tree can be understood and the rules can be validated by human experts who have knowledge of the application domain.

It is generally recommended that a decision tree be tested and its accuracy be taken as a benchmark before more complicated algorithms are employed. Analysis of the tree also allows an understanding of the important features, and the univariate tree does its own automatic feature extraction. Another big advantage of the univariate tree is that it can use numeric and discrete features together, without needing to convert one type into the other.

The decision tree is a nonparametric method, similar to the instance-based methods discussed in chapter 8, but there are a number of differences:

- Each leaf node corresponds to a "bin," except that the bins need not be the same size (as in Parzen windows) or contain an equal number of training instances (as in *k*-nearest neighbor).
- The bin divisions are not done based only on similarity in the input space, but supervised output information through entropy or mean square error is also used.
- Another advantage of the decision tree is that, thanks to the tree structure, the leaf ("bin") is found much faster with smaller number of comparisons.
- The decision tree, once it is constructed, does not store all the training set but only the structure of the tree, the parameters of the decision nodes, and the output values in leaves; this implies that the space complexity is also much less, as opposed to instance-based nonparametric methods that store all training examples.

With a decision tree, a class need not have a single description to which all instances should match. It may have a number of possible descriptions that can even be disjoint in the input space.

The decision tree we discussed until now have *hard* decision nodes; that is, we take one of the branches depending on the test. We start from the root and follow a single path and stop at a leaf where we output the response value stored in that leaf. In a *soft decision tree*, however, we take *all* the branches but with different probabilities, and we follow in parallel all the paths and reach all the leaves, but with different probabilities. The output is the weighted average of all the outputs in all the leaves where the weights correspond to the probabilities accumulated over the paths; we will discuss this in section 12.9.

SOFT DECISION TREE

In chapter 17, we talk about combining multiple learners; one of the most popular models combined is a decision tree, and an ensemble of decision trees is called a *decision forest*. We will see that if we train not

DECISION FOREST

one but many decision trees, each on a random subset of training set or a random subset of the input features, and combine their predictions, overall accuracy can be significantly increased. This is the idea behind the *random forest* method.

RANDOM FOREST

The tree is different from the statistical models discussed in previous chapters. The tree codes directly the discriminants separating class instances without caring much for how those instances are distributed in the regions. The decision tree is *discriminant-based*, whereas the statistical methods are *likelihood-based* in that they explicitly estimate $p(\boldsymbol{x}|C_i)$ before using Bayes' rule and calculating the discriminant. Discriminant-based methods directly estimate the discriminants, bypassing the estimation of class densities. We further discuss such discriminant-based methods in the chapters ahead.

9.8 Exercises

1. Generalize the Gini index (equation 9.5) and the misclassification error (equation 9.6) for $K > 2$ classes. Generalize misclassification error to risk, taking a loss function into account.

   SOLUTION:

   - Gini index with $K > 2$ classes: $\phi(p_1, p_2, \ldots, p_K) = \sum_{i=1}^K \sum_{j<i} p_i p_j$
   - Misclassification error: $\phi(p_1, p_2, \ldots, p_K) = 1 - \max_{i=1}^K p_i$
   - Risk: $\phi_\Lambda(p_1, p_2, \ldots, p_K) = \min_{i=1}^K \sum_{k=1}^K \lambda_{ik} p_k$ where Λ is the $K \times K$ loss matrix.

2. For a numeric input, instead of a binary split, one can use a ternary split with two thresholds and three branches as

   $x_j < w_{ma},\ w_{ma} \le x_j < w_{mb},\ x_j \ge w_{mb}$

   Propose a modification of the tree induction method to learn the two thresholds, w_{ma}, w_{mb}. What are the advantages and the disadvantages of such a node over a binary node?

   SOLUTION: For the numeric attributes, instead of one split threshold, we need to try all possible pairs of split thresholds and choose the best. When there are two splits, there are three children, and in calculating the entropy after the splits, we need to sum up over the three sets corresponding to the instances taking the three branches.

   The complexity of finding the best pair is $\mathcal{O}(N_m^2)$ instead of $\mathcal{O}(N_m)$ and each node stores two thresholds instead of one and has three branches instead of two. The advantage is that one ternary node splits an input into three, whereas this requires two successive binary nodes. Which one is better depends on the data at hand; if we have hypotheses that require bounded intervals (e.g., rectangles), a ternary node may be advantageous.

3. Propose a tree induction algorithm with backtracking.

4. In generating a univariate tree, a discrete attribute with n possible values can be represented by n 0/1 dummy variables and then treated as n separate numeric attributes. What are the advantages and disadvantages of this approach?

5. Derive a learning algorithm for sphere trees (equation 9.21). Generalize to ellipsoid trees.

6. In a regression tree, we discussed that in a leaf node, instead of calculating the mean, we can do a linear regression fit and make the response at the leaf dependent on the input. Propose a similar method for classification trees.

   SOLUTION: This implies that at each leaf, we will have a linear classifier trained with instances reaching there. That linear classifier will generate posterior probabilities for the different classes, and those probabilities will be used in the entropy calculation. That is, it is not necessary for a leaf to be pure, that is, to contain instances of only one class; it is enough that the classifier in that leaf generates posterior probabilities close to 0 or 1.

7. Propose a rule induction algorithm for regression.

8. In regression trees, how can we get rid of discontinuities at the leaf boundaries?

9. Let us say that for a classification problem, we already have a trained decision tree. How can we use it in addition to the training set in constructing a k-nearest neighbor classifier?

   SOLUTION: The decision tree does feature selection, and we can use only the features used by the tree. The average number of instances per leaf also gives us information about a good k value.

10. In a multivariate tree, very probably, at each internal node, we will not be needing all the input variables. How can we decrease dimensionality at a node?

    SOLUTION: Each subtree handles a local region in the input space that can be explained by a small number of features. We can do feature selection or extraction using only the subset of the instances reaching that node. Ideally, as we go down the tree, we would expect to need fewer features.

9.9 References

Breiman, L., J. H. Friedman, R. A. Olshen, and C. J. Stone. 1984. *Classification and Regression Trees*. Belmont, CA: Wadsworth International Group.

Brodley, C. E., and P. E. Utgoff. 1995. "Multivariate Decision Trees." *Machine Learning* 19:45–77.

Cohen, W. 1995. "Fast Effective Rule Induction." In *Twelfth International Conference on Machine Learning*, ed. A. Prieditis and S. J. Russell, 115–123. San Mateo, CA: Morgan Kaufmann.

Devroye, L., L. Györfi, and G. Lugosi. 1996. *A Probabilistic Theory of Pattern Recognition*. New York: Springer.

Dietterich, T. G. 1997. "Machine Learning Research: Four Current Directions." *AI Magazine* 18:97–136.

Fürnkranz, J., and G. Widmer. 1994. "Incremental Reduced Error Pruning." In *Eleventh International Conference on Machine Learning*, ed. W. Cohen and H. Hirsh, 70–77. San Mateo, CA: Morgan Kaufmann.

Guo, H., and S. B. Gelfand. 1992. "Classification Trees with Neural Network Feature Extraction." *IEEE Transactions on Neural Networks* 3:923–933.

Loh, W.-Y., and Y. S. Shih. 1997. "Split Selection Methods for Classification Trees." *Statistica Sinica* 7:815–840.

Loh, W.-Y., and N. Vanichsetakul. 1988. "Tree-Structured Classification via Generalized Discriminant Analysis." *Journal of the American Statistical Association* 83:715–725.

Mitchell, T. 1997. *Machine Learning*. New York: McGraw-Hill.

Murthy, S. K. 1998. "Automatic Construction of Decision Trees from Data: A Multi-Disciplinary Survey." *Data Mining and Knowledge Discovery* 4:345–389.

Murthy, S. K., S. Kasif, and S. Salzberg. 1994. "A System for Induction of Oblique Decision Trees." *Journal of Artificial Intelligence Research* 2:1–32.

Quinlan, J. R. 1986. "Induction of Decision Trees." *Machine Learning* 1:81–106.

Quinlan, J. R. 1990. "Learning Logical Definitions from Relations." *Machine Learning* 5:239–266.

Quinlan, J. R. 1993. *C4.5: Programs for Machine Learning*. San Mateo, CA: Morgan Kaufmann.

Quinlan, J. R. 1995. "MDL and Categorical Theories (continued)." In *Twelfth International Conference on Machine Learning*, ed. A. Prieditis and S. J. Russell, 467–470. San Mateo, CA: Morgan Kaufmann.

Rokach, L., and O. Maimon. 2005. “Top-Down Induction of Decision Trees Classifiers—A Survey.” *IEEE Transactions on Systems, Man, and Cybernetics–Part C* 35:476–487.

Yıldız, O. T., and E. Alpaydın. 2000. “Linear Discriminant Trees.” In *Seventeenth International Conference on Machine Learning*, ed. P. Langley, 1175–1182. San Francisco: Morgan Kaufmann.

Yıldız, O. T., and E. Alpaydın. 2001. “Omnivariate Decision Trees.” *IEEE Transactions on Neural Networks* 12:1539–1546.

10 *Linear Discrimination*

In linear discrimination, we assume that instances of a class are linearly separable from instances of other classes. This is a discriminant-based approach that estimates the parameters of the linear discriminant directly from a given labeled sample.

10.1 Introduction

We remember from the previous chapters that in classification we define a set of discriminant functions $g_j(\boldsymbol{x}), j = 1, \ldots, K$, and then we

choose C_i if $g_i(\boldsymbol{x}) = \max_{j=1}^{K} g_j(\boldsymbol{x})$

Previously, when we discussed methods for classification, we first estimated the prior probabilities, $\hat{P}(C_i)$, and the class likelihoods, $\hat{p}(\boldsymbol{x}|C_i)$, then used Bayes' rule to calculate the posterior densities. We then defined the discriminant functions in terms of the posterior, for example,

$g_i(\boldsymbol{x}) = \log \hat{P}(C_i|\boldsymbol{x})$

LIKELIHOOD-BASED CLASSIFICATION

This is called *likelihood-based classification*, and we have previously discussed the parametric (chapter 5), semiparametric (chapter 7), and nonparametric (chapter 8) approaches to estimating the class likelihoods, $p(\boldsymbol{x}|C_i)$.

DISCRIMINANT-BASED CLASSIFICATION

We are now going to discuss *discriminant-based classification* where we assume a model directly for the discriminant, bypassing the estimation of likelihoods or posteriors. The discriminant-based approach, as we also saw for the case of decision trees in chapter 9, makes an assumption on the form of the discriminant between the classes and makes no assumption about, or requires no knowledge of the densities—for example, whether they are Gaussian, or whether the inputs are correlated, and so forth.

We define a model for the discriminant

$$g_i(\boldsymbol{x}|\Phi_i)$$

explicitly parameterized with the set of parameters Φ_i, as opposed to a likelihood-based scheme that has implicit parameters in defining the likelihood densities. This is a different inductive bias: Instead of making an assumption on the form of the class densities, we make an assumption on the form of the boundaries separating classes.

Learning is the optimization of the model parameters $\boldsymbol{\Phi}_i$ to maximize the quality of the separation, that is, the classification accuracy on a given labeled training set. This differs from the likelihood-based methods that search for the parameters that maximize sample likelihoods, separately for each class.

In the discriminant-based approach, we do not care about correctly estimating the densities inside class regions; all we care about is the correct estimation of the *boundaries* between the class regions. Those who advocate the discriminant-based approach (e.g., Vapnik 1995) state that estimating the class densities is a harder problem than estimating the class discriminants, and it does not make sense to solve a hard problem to solve an easier problem. This is of course true only when the discriminant can be approximated by a simple function.

In this chapter, we concern ourselves with the simplest case where the discriminant functions are linear in $\boldsymbol{x}$:

$$g_i(\boldsymbol{x}|\boldsymbol{w}_i, w_{i0}) = \boldsymbol{w}_i^T\boldsymbol{x} + w_{i0} = \sum_{j=1}^{d} w_{ij}x_j + w_{i0} \tag{10.1}$$

LINEAR DISCRIMINANT

The *linear discriminant* is used frequently mainly due to its simplicity: Both the space and time complexities are $\mathcal{O}(d)$. The linear model is easy to understand: the final output is a weighted sum of the input attributes x_j. The magnitude of the weight w_j shows the importance of x_j and its sign indicates if the effect is positive or negative. Most functions are additive in that the output is the sum of the effects of several attributes where the weights may be positive (enforcing) or negative (inhibiting). For example, when a customer applies for credit, financial institutions calculate the applicant's credit score that is generally written as a sum of the effects of various attributes; for example, yearly income has a positive effect (higher incomes increase the score).

In many applications, the linear discriminant is also quite accurate. We know, for example, that when classes are Gaussian with a shared covariance matrix, the optimal discriminant is linear. The linear discriminant, however, can be used even when this assumption does not hold, and the model parameters can be calculated without making any assumptions on the class densities. We should always use the linear discriminant before trying a more complicated model to make sure that the additional complexity is justified.

As always, we formulate the problem of finding a linear discriminant function as a search for the parameter values that minimize an error function. In particular, we concentrate on *gradient* methods for optimizing a criterion function.

10.2 Generalizing the Linear Model

QUADRATIC DISCRIMINANT

When a linear model is not flexible enough, we can use the *quadratic discriminant* function and increase complexity

(10.2) $$g_i(\boldsymbol{x}|\mathbf{W}_i, \boldsymbol{w}_i, w_{i0}) = \boldsymbol{x}^T\mathbf{W}_i\boldsymbol{x} + \boldsymbol{w}_i\boldsymbol{x} + w_{i0}$$

but this approach is $O(d^2)$ and we again have the bias/variance dilemma: The quadratic model, though is more general, requires much larger training sets and may overfit on small samples.

HIGHER-ORDER TERMS

PRODUCT TERMS

An equivalent way is to preprocess the input by adding *higher-order terms*, also called *product terms*. For example, with two inputs x_1 and x_2, we can define new variables

$$z_1 = x_1, z_2 = x_2, z_3 = x_1^2, z_4 = x_2^2, z_5 = x_1x_2$$

and take $\boldsymbol{z} = [z_1, z_2, z_3, z_4, z_5]^T$ as the input. The linear function defined in the five-dimensional $\boldsymbol{z}$ space corresponds to a nonlinear function in the two-dimensional $\boldsymbol{x}$ space. Instead of defining a nonlinear function (discriminant or regression) in the original space, what we do is to define a suitable nonlinear transformation to a new space where the function can be written in a linear form.

We write the discriminant as

(10.3) $$g_i(\boldsymbol{x}) = \sum_{j=1}^{k} w_j \phi_{ij}(\boldsymbol{x})$$

BASIS FUNCTION

where $\phi_{ij}(\boldsymbol{x})$ are *basis functions*. Higher-order terms are only one set of possible basis functions; other examples are

- $\sin(x_1)$
- $\exp(-(x_1 - m)^2/c)$
- $\exp(-\|\boldsymbol{x} - \boldsymbol{m}\|^2/c)$
- $\log(x_2)$
- $1(x_1 > c)$
- $1(ax_1 + bx_2 > c)$

where m, a, b, c are scalars, $\boldsymbol{m}$ is a d-dimensional vector, and $1(b)$ returns 1 if b is true and returns 0 otherwise. The idea of writing a nonlinear function as a linear sum of nonlinear basis functions is an old idea and was originally called *potential functions* (Aizerman, Braverman, and Rozonoer 1964). Multilayer perceptrons (chapter 11) and radial basis functions (chapter 12) have the advantage that the parameters of the basis functions can be fine-tuned to the data during learning. In chapter 13, we discuss support vector machines that use kernel functions built from such basis functions.

POTENTIAL FUNCTION

10.3 Geometry of the Linear Discriminant

10.3.1 Two Classes

Let us start with the simpler case of two classes. In such a case, one discriminant function is sufficient:

$$\begin{aligned} g(\boldsymbol{x}) &= g_1(\boldsymbol{x}) - g_2(\boldsymbol{x}) \\ &= (\boldsymbol{w}_1^T\boldsymbol{x} + w_{10}) - (\boldsymbol{w}_2^T\boldsymbol{x} + w_{20}) \\ &= (\boldsymbol{w}_1 - \boldsymbol{w}_2)^T\boldsymbol{x} + (w_{10} - w_{20}) \\ &= \boldsymbol{w}^T\boldsymbol{x} + w_0 \end{aligned}$$

and we

$$\text{choose} \begin{cases} C_1 & \text{if } g(\boldsymbol{x}) > 0 \\ C_2 & \text{otherwise} \end{cases}$$

WEIGHT VECTOR

THRESHOLD

This defines a hyperplane where $\boldsymbol{w}$ is the *weight vector* and w_0 is the *threshold*. This latter name comes from the fact that the decision rule can be rewritten as follows: Choose C_1 if $\boldsymbol{w}^T\boldsymbol{x} > -w_0$, and choose C_2

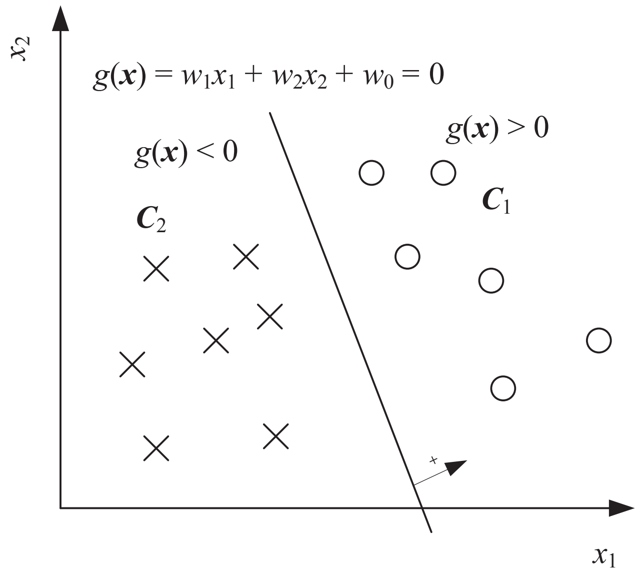


Figure 10.1 In the two-dimensional case, the linear discriminant is a line that separates the examples from two classes.

otherwise. The hyperplane divides the input space into two half-spaces: the decision region $\mathcal{R}_1$ for C_1 and $\mathcal{R}_2$ for C_2. Any $\boldsymbol{x}$ in $\mathcal{R}_1$ is on the *positive* side of the hyperplane and any $\boldsymbol{x}$ in $\mathcal{R}_2$ is on its *negative* side. When $\boldsymbol{x}$ is $\mathbf{0}$, $g(\boldsymbol{x}) = w_0$ and we see that if $w_0 > 0$, the origin is on the positive side of the hyperplane, and if $w_0 < 0$, the origin is on the negative side, and if $w_0 = 0$, the hyperplane passes through the origin (see figure 10.1).

Take two points $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ both on the decision surface; that is, $g(\boldsymbol{x}_1) = g(\boldsymbol{x}_2) = 0$, then

$$\begin{aligned} \boldsymbol{w}^T\boldsymbol{x}_1 + w_0 &= \boldsymbol{w}^T\boldsymbol{x}_2 + w_0 \\ \boldsymbol{w}^T(\boldsymbol{x}_1 - \boldsymbol{x}_2) &= 0 \end{aligned}$$

and we see that $\boldsymbol{w}$ is normal to any vector lying on the hyperplane. Let us rewrite $\boldsymbol{x}$ as (Duda, Hart, and Stork 2001)

$$\boldsymbol{x} = \boldsymbol{x}_p + r\frac{\boldsymbol{w}}{\|\boldsymbol{w}\|}$$

where $\boldsymbol{x}_p$ is the normal projection of $\boldsymbol{x}$ onto the hyperplane and r gives us the distance from $\boldsymbol{x}$ to the hyperplane, negative if $\boldsymbol{x}$ is on the negative

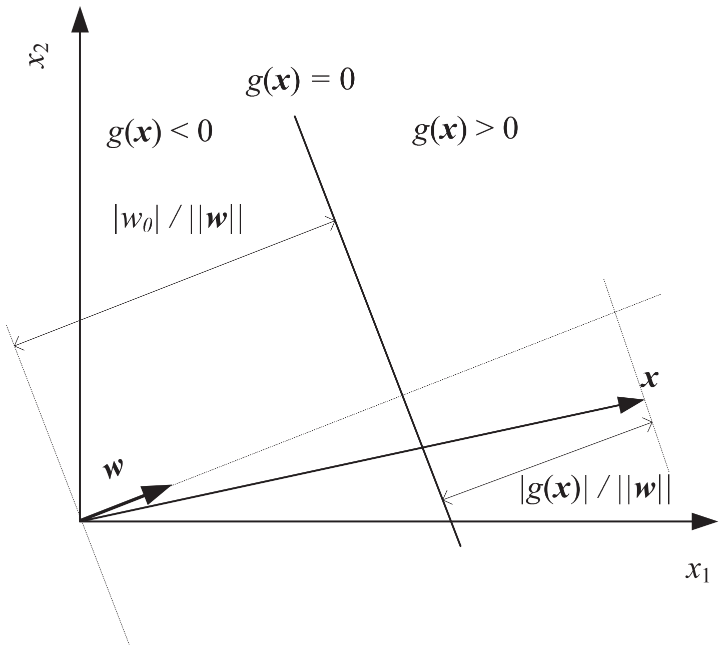


Figure 10.2 The geometric interpretation of the linear discriminant.

side, and positive if $\boldsymbol{x}$ is on the positive side (see figure 10.2). Calculating $g(\boldsymbol{x})$ and noting that $g(\boldsymbol{x}_p) = 0$, we have

$$r = \frac{g(\boldsymbol{x})}{\|\boldsymbol{w}\|} \tag{10.4}$$

We see then that the distance to origin is

$$r_0 = \frac{w_0}{\|\boldsymbol{w}\|} \tag{10.5}$$

Thus w_0 determines the location of the hyperplane with respect to the origin, and $\boldsymbol{w}$ determines its orientation.

10.3.2 Multiple Classes

When there are $K > 2$ classes, there are K discriminant functions. When they are linear, we have

$$g_i(\boldsymbol{x}|\boldsymbol{w}_i, w_{i0}) = \boldsymbol{w}_i^T \boldsymbol{x} + w_{i0} \tag{10.6}$$

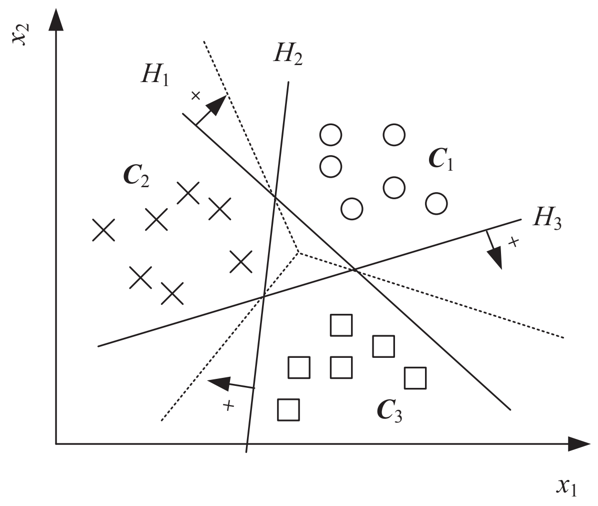


Figure 10.3 In linear classification, each hyperplane H_i separates the examples of C_i from the examples of all other classes. Thus for it to work, the classes should be linearly separable. Dotted lines are the induced boundaries of the linear classifier.

We are going to talk about learning later on but for now, we assume that the parameters, $\boldsymbol{w}_i, w_{i0}$, are computed so as to have

(10.7) $$g_i(\boldsymbol{x}|\boldsymbol{w}_i, w_{i0}) = \begin{cases} > 0 & \text{if } \boldsymbol{x} \in C_i \\ \leq 0 & \text{otherwise} \end{cases}$$

LINEARLY SEPARABLE CLASSES

for all $\boldsymbol{x}$ in the training set. Using such discriminant functions corresponds to assuming that all classes are *linearly separable*; that is, for each class C_i, there exists a hyperplane H_i such that all $\boldsymbol{x} \in C_i$ lie on its positive side and all $\boldsymbol{x} \in C_j, j \neq i$ lie on its negative side (see figure 10.3).

During testing, given $\boldsymbol{x}$, ideally, we should have only one $g_j(\boldsymbol{x}), j = 1, \ldots, K$ greater than 0 and all others should be less than 0, but this is not always the case: The positive half-spaces of the hyperplanes may overlap, or, we may have a case where all $g_j(\boldsymbol{x}) < 0$. These may be taken as *reject* cases, but the usual approach is to assign $\boldsymbol{x}$ to the class having the highest discriminant:

(10.8) $$\text{Choose } C_i \text{ if } g_i(\boldsymbol{x}) = \max_{j=1}^{K} g_j(\boldsymbol{x})$$

Remembering that $|g_i(\boldsymbol{x})|/\|\boldsymbol{w}_i\|$ is the distance from the input point to the hyperplane, assuming that all $\boldsymbol{w}_i$ have similar length, this assigns the

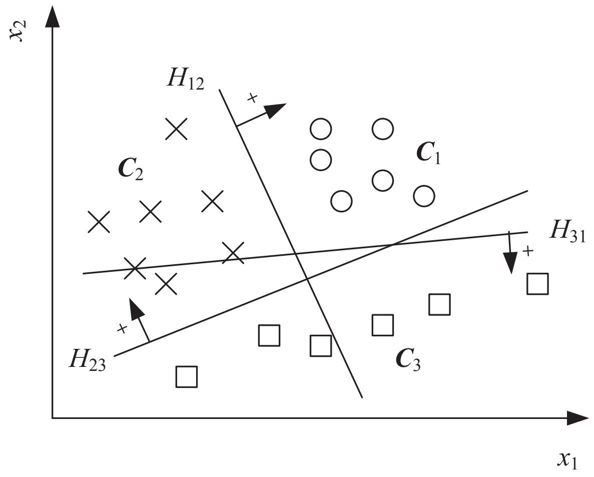


Figure 10.4 In pairwise linear separation, there is a separate hyperplane for each pair of classes. For an input to be assigned to C_1, it should be on the positive side of H_{12} and H_{13} (which is the negative side of H_{31}); we do not care about the value of H_{23}. In this case, C_1 is not linearly separable from other classes but is pairwise linearly separable.

point to the class (among all $g_j(\boldsymbol{x}) > 0$) to whose hyperplane the point is most distant. This is called a *linear classifier*, and geometrically it divides the feature space into K convex decision regions $\mathcal{R}_i$ (see figure 10.3).

LINEAR CLASSIFIER

10.4 Pairwise Separation

PAIRWISE SEPARATION

If the classes are not linearly separable, one approach is to divide it into a set of linear problems. One possibility is *pairwise separation* of classes (Duda, Hart, and Stork 2001). It uses $K(K-1)/2$ linear discriminants, $g_{ij}(\boldsymbol{x})$, one for every pair of distinct classes (see figure 10.4):

$$g_{ij}(\boldsymbol{x}|\boldsymbol{w}_{ij}, w_{ij0}) = \boldsymbol{w}_{ij}^T\boldsymbol{x} + w_{ij0}$$

The parameters $\boldsymbol{w}_{ij}, j \neq i$ are computed during training so as to have

$$g_{ij}(\boldsymbol{x}) = \begin{cases} > 0 & \text{if } \boldsymbol{x} \in C_i \\ \leq 0 & \text{if } \boldsymbol{x} \in C_j \\ \text{don't care} & \text{otherwise} \end{cases} \quad i, j = 1, \ldots, K \text{ and } i \neq j \tag{10.9}$$

that is, if $\boldsymbol{x}^t \in C_k$ where $k \neq i, k \neq j$, then $\boldsymbol{x}^t$ is not used during training of $g_{ij}(\boldsymbol{x})$.

During testing, we

choose C_i if $\forall j \neq i, g_{ij}(\boldsymbol{x}) > 0$

In many cases, this may not be true for any i and if we do not want to reject such cases, we can relax the conjunction by using a summation and choosing the maximum of

$$g_i(\boldsymbol{x}) = \sum_{j \neq i} g_{ij}(\boldsymbol{x}) \tag{10.10}$$

Even if the classes are not linearly separable, if the classes are pairwise linearly separable—which is much more likely—pairwise separation can be used, leading to nonlinear separation of classes (see figure 10.4). This is another example of breaking down a complex (e.g., nonlinear) problem, into a set of simpler (e.g., linear) problems. We have already seen decision trees (chapter 9) that use this idea, and we will see more examples of this in chapter 17 on combining multiple models, for example, error-correcting output codes, and mixture of experts, where the number of linear models is less than $\mathcal{O}(K^2)$.

10.5 Parametric Discrimination Revisited

In chapter 5, we saw that if the class densities, $p(\boldsymbol{x}|C_i)$, are Gaussian and share a common covariance matrix, the discriminant function is linear

$$g_i(\boldsymbol{x}) = \boldsymbol{w}_i^T \boldsymbol{x} + w_{i0} \tag{10.11}$$

where the parameters can be analytically calculated as

$$\begin{aligned} \boldsymbol{w}_i &= \boldsymbol{\Sigma}^{-1}\boldsymbol{\mu}_i \\ w_{i0} &= -\frac{1}{2}\boldsymbol{\mu}_i^T \boldsymbol{\Sigma}^{-1}\boldsymbol{\mu}_i + \log P(C_i) \end{aligned} \tag{10.12}$$

Given a dataset, we first calculate the estimates for $\boldsymbol{\mu}_i$ and $\boldsymbol{\Sigma}$ and then plug the estimates, $\boldsymbol{m}_i$, $\mathbf{S}$, in equation 10.12 and calculate the parameters of the linear discriminant.

Let us again see the special case where there are two classes. We define $y \equiv P(C_1|\boldsymbol{x})$ and $P(C_2|\boldsymbol{x}) = 1 - y$. Then in classification, we

$$\text{choose } C_1 \text{ if } \begin{cases} y > 0.5 \\ \frac{y}{1-y} > 1 \\ \log \frac{y}{1-y} > 0 \end{cases} \quad \text{and } C_2 \text{ otherwise}$$

LOGIT
LOG ODDS

$\log y/(1-y)$ is known as the *logit* transformation or *log odds* of y. In the case of two normal classes sharing a common covariance matrix, the log odds is linear:

$$\begin{aligned}\text{logit}(P(C_1|\boldsymbol{x})) = \log \frac{P(C_1|\boldsymbol{x})}{1-P(C_1|\boldsymbol{x})} &= \log \frac{P(C_1|\boldsymbol{x})}{P(C_2|\boldsymbol{x})} \\ &= \log \frac{p(\boldsymbol{x}|C_1)}{p(\boldsymbol{x}|C_2)} + \log \frac{P(C_1)}{P(C_2)} \\ &= \log \frac{(2\pi)^{-d/2}|\boldsymbol{\Sigma}|^{-1/2}\exp[-(1/2)(\boldsymbol{x}-\boldsymbol{\mu}_1)^T\boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu}_1)]}{(2\pi)^{-d/2}|\boldsymbol{\Sigma}|^{-1/2}\exp[-(1/2)(\boldsymbol{x}-\boldsymbol{\mu}_2)^T\boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu}_2)]} + \log \frac{P(C_1)}{P(C_2)} \\ &= \boldsymbol{w}^T\boldsymbol{x} + w_0 \end{aligned} \tag{10.13}$$

where

$$\begin{aligned} \boldsymbol{w} &= \boldsymbol{\Sigma}^{-1}(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2) \\ w_0 &= -\frac{1}{2}(\boldsymbol{\mu}_1 + \boldsymbol{\mu}_2)^T\boldsymbol{\Sigma}^{-1}(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2) + \log\frac{P(C_1)}{P(C_2)} \end{aligned} \tag{10.14}$$

The inverse of logit

$$\log \frac{P(C_1|\boldsymbol{x})}{1-P(C_1|\boldsymbol{x})} = \boldsymbol{w}^T\boldsymbol{x} + w_0$$

LOGISTIC
SIGMOID

is the *logistic* function, also called the *sigmoid* function (see figure 10.5):

$$P(C_1|\boldsymbol{x}) = \text{sigmoid}(\boldsymbol{w}^T\boldsymbol{x} + w_0) = \frac{1}{1+\exp\left[-(\boldsymbol{w}^T\boldsymbol{x} + w_0)\right]} \tag{10.15}$$

During training, we estimate $\boldsymbol{m}_1, \boldsymbol{m}_2, \mathbf{S}$ and plug these estimates in equation 10.14 to calculate the discriminant parameters. During testing, given $\boldsymbol{x}$, we can either

1. calculate $g(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x} + w_0$ and choose C_1 if $g(\boldsymbol{x}) > 0$, or
2. calculate $y = \text{sigmoid}(\boldsymbol{w}^T\boldsymbol{x} + w_0)$ and choose C_1 if $y > 0.5$,

because sigmoid$(0) = 0.5$. In this latter case, sigmoid transforms the discriminant value to a posterior probability. This is valid when there are two classes and one discriminant; we see in section 10.7 how we can estimate posterior probabilities for $K > 2$.

10.6 Gradient Descent

In likelihood-based classification, the parameters were the sufficient statistics of $p(\boldsymbol{x}|C_i)$ and $P(C_i)$, and the method we used to estimate the parameters is maximum likelihood. In the discriminant-based approach,

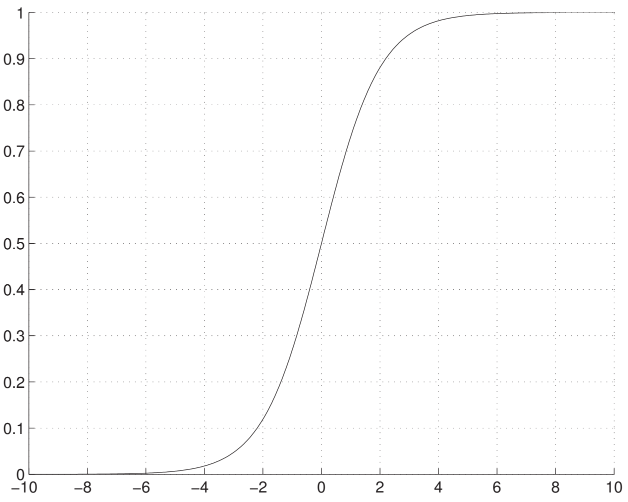


Figure 10.5 The logistic, or sigmoid, function.

the parameters are those of the discriminants, and they are optimized to minimize the classification error on the training set. When $\boldsymbol{w}$ denotes the set of parameters and $E(\boldsymbol{w}|\mathcal{X})$ is the error with parameters $\boldsymbol{w}$ on the given training set $\mathcal{X}$, we look for

$$\boldsymbol{w}^* = \arg\min_{\boldsymbol{w}} E(\boldsymbol{w}|\mathcal{X})$$

In many cases, some of which we will see shortly, there is no analytical solution and we need to resort to iterative optimization methods, the most commonly employed being that of *gradient descent*. When $E(\boldsymbol{w})$ is a differentiable function of a vector of variables, we have the *gradient vector* composed of the partial derivatives

GRADIENT DESCENT

GRADIENT VECTOR

$$\nabla_w E = \left[\frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, \ldots, \frac{\partial E}{\partial w_d}\right]^T$$

and the *gradient descent* procedure to minimize E starts from a random $\boldsymbol{w}$, and at each step, updates $\boldsymbol{w}$, in the opposite direction of the gradient

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i}, \forall i \qquad (10.16)$$

$$w_i = w_i + \Delta w_i \qquad (10.17)$$

where η is called the *stepsize*, or *learning factor*, and determines how much to move in that direction. Gradient ascent is used to maximize a function and goes in the direction of the gradient. When we get to a minimum (or maximum), the derivative is 0 and the procedure terminates. This indicates that the procedure finds the nearest minimum that can be a local minimum, and there is no guarantee of finding the global minimum unless the function has only one minimum. The use of a good value for η is also critical; if it is too small, the convergence may be too slow, and a large value may cause oscillations and even divergence.

Throughout this book, we use gradient methods that are simple and quite effective. We keep in mind, however, that once a suitable model and an error function is defined, the optimization of the model parameters to minimize the error function can be done by using one of many possible techniques. There are second-order methods and conjugate gradient that converge faster, at the expense of more memory and computation. More costly methods like simulated annealing and genetic algorithms allow a more thorough search of the parameter space and do not depend as much on the initial point.

10.7 Logistic Discrimination

10.7.1 Two Classes

LOGISTIC DISCRIMINATION

In *logistic discrimination*, we do not model the class-conditional densities, $p(\boldsymbol{x}|C_i)$, but rather their ratio. Let us again start with two classes and assume that the log likelihood ratio is linear:

$$\log \frac{p(\boldsymbol{x}|C_1)}{p(\boldsymbol{x}|C_2)} = \boldsymbol{w}^T\boldsymbol{x} + w_0^o \tag{10.18}$$

This indeed holds when the class-conditional densities are normal (equation 10.13). But logistic discrimination has a wider scope of applicability; for example, $\boldsymbol{x}$ may be composed of discrete attributes or may be a mixture of continuous and discrete attributes.

Using Bayes' rule, we have

$$\begin{aligned}
\text{logit}(P(C_1|\boldsymbol{x})) &= \log \frac{P(C_1|\boldsymbol{x})}{1 - P(C_1|\boldsymbol{x})} \\
&= \log \frac{p(\boldsymbol{x}|C_1)}{p(\boldsymbol{x}|C_2)} + \log \frac{P(C_1)}{P(C_2)} \\
&= \boldsymbol{w}^T\boldsymbol{x} + w_0 \qquad (10.19)
\end{aligned}$$

where

$$w_0 = w_0^o + \log \frac{P(C_1)}{P(C_2)} \tag{10.20}$$

Rearranging terms, we get the sigmoid function again:

$$y = \hat{P}(C_1|\boldsymbol{x}) = \frac{1}{1 + \exp[-(\boldsymbol{w}^T\boldsymbol{x} + w_0)]} \tag{10.21}$$

as our estimator of $P(C_1|\boldsymbol{x})$.

Let us see how we can learn $\boldsymbol{w}$ and w_0. We are given a sample of two classes, $\mathcal{X} = \{\boldsymbol{x}^t, r^t\}$, where $r^t = 1$ if $\boldsymbol{x} \in C_1$ and $r^t = 0$ if $\boldsymbol{x} \in C_2$. We assume r^t, given $\boldsymbol{x}^t$, is Bernoulli with probability $y^t \equiv P(C_1|\boldsymbol{x}^t)$ as calculated in equation 10.21:

$$r^t|\boldsymbol{x}^t \sim \text{Bernoulli}(y^t)$$

Here, we see the difference from the likelihood-based methods where we modeled $p(\boldsymbol{x}|C_i)$; in the discriminant-based approach, we model directly $r|\boldsymbol{x}$. The sample likelihood is

$$l(\boldsymbol{w}, w_0|\mathcal{X}) = \prod_t (y^t)^{(r^t)}(1-y^t)^{(1-r^t)} \tag{10.22}$$

We know that when we have a likelihood function to maximize, we can always turn it into an error function to be minimized as $E = -\log l$, and in our case, we have *cross-entropy*:

CROSS-ENTROPY

$$E(\boldsymbol{w}, w_0|\mathcal{X}) = -\sum_t r^t \log y^t + (1-r^t)\log(1-y^t) \tag{10.23}$$

Because of the nonlinearity of the sigmoid function, we cannot solve directly and we use gradient descent to minimize cross-entropy, equivalent to maximizing the likelihood or the log likelihood. If $y = \text{sigmoid}(a) = 1/(1+\exp(-a))$, its derivative is given as

$$\frac{dy}{da} = y(1-y)$$

and we get the following update equations:

$$\begin{aligned}
\Delta w_j &= -\eta\frac{\partial E}{\partial w_j} = \eta\sum_t\left(\frac{r^t}{y^t} - \frac{1-r^t}{1-y^t}\right)y^t(1-y^t)x_j^t \\
&= \eta\sum_t(r^t - y^t)x_j^t, j = 1,\ldots,d \\
\Delta w_0 &= -\eta\frac{\partial E}{\partial w_0} = \eta\sum_t(r^t-y^t) \qquad (10.24)
\end{aligned}$$

```
For $j = 0, \dots, d$
    $w_j \leftarrow \text{rand}(-0.01, 0.01)$
Repeat
    For $j = 0, \dots, d$
        $\Delta w_j \leftarrow 0$
    For $t = 1, \dots, N$
        $o \leftarrow 0$
        For $j = 0, \dots, d$
            $o \leftarrow o + w_j x_j^t$
        $y \leftarrow \text{sigmoid}(o)$
        For $j = 0, \dots, d$
            $\Delta w_j \leftarrow \Delta w_j + (r^t - y) x_j^t$
    For $j = 0, \dots, d$
        $w_j \leftarrow w_j + \eta \Delta w_j$
Until convergence
```

Figure 10.6 Logistic discrimination algorithm implementing gradient descent for the single output case with two classes. For w_0, we assume that there is an extra input x_0, which is always $+1$: $x_0^t \equiv +1, \forall t$.

It is best to initialize w_j with random values close to 0; generally they are drawn uniformly from the interval $[-0.01, 0.01]$. The reason for this is that if the initial w_j are large in magnitude, the weighted sum may also be large and may saturate the sigmoid. We see from figure 10.5 that if the initial weights are close to 0, the sum will stay in the middle region where the derivative is nonzero and an update can take place. If the weighted sum is large in magnitude (smaller than -5 or larger than $+5$), the derivative of the sigmoid will be almost 0 and weights will not be updated.

Pseudocode is given in figure 10.6. We see an example in figure 10.7 where the input is one-dimensional. Both the line $wx + w_0$ and its value after the sigmoid are shown as a function of learning iterations. We see that to get outputs of 0 and 1, the sigmoid hardens, which is achieved by increasing the magnitude of w, or $\|\boldsymbol{w}\|$ in the multivariate case.

Once training is complete and we have the final $\boldsymbol{w}$ and w_0, during testing, given $\boldsymbol{x}^t$, we calculate $y^t = \text{sigmoid}(\boldsymbol{w}^T \boldsymbol{x}^t + w_0)$ and we choose C_1 if $y^t > 0.5$ and choose C_2 otherwise. This implies that to minimize the number of misclassifications, we do not need to continue learning un-

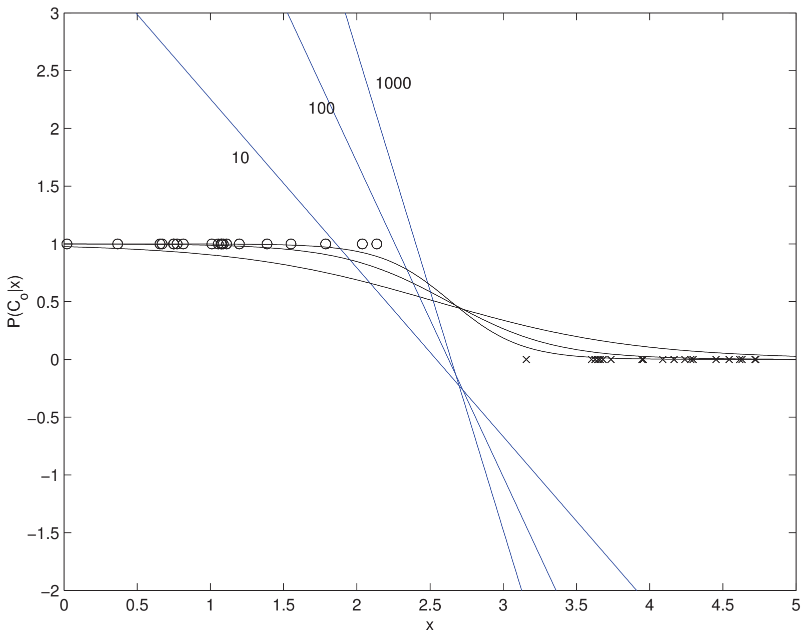


Figure 10.7 For a univariate two-class problem (shown with '$\circ$' and '$\times$'), the evolution of the line $wx + w_0$ and the sigmoid output after 10, 100, and 1,000 iterations over the sample.

til all y^t are 0 or 1, but only until y^t are less than or greater than 0.5, that is, on the correct side of the decision boundary. If we do continue training beyond this point, cross-entropy will continue decreasing ($|w_j|$ will continue increasing to harden the sigmoid), but the number of misclassifications will not decrease. Generally, we continue training until the number of misclassifications does not decrease (which will be 0 if the classes are linearly separable). Actually *stopping early* before we have 0 training error is a form of regularization. Because we start with weights almost 0 and they move away as training continues, stopping early corresponds to a model with more weights close to 0 and effectively fewer parameters.

EARLY STOPPING

Note that though we assumed the log ratio of the class densities are linear to derive the discriminant, we estimate directly the posterior and never explicitly estimate $p(\boldsymbol{x}|C_i)$ or $P(C_i)$.

10.7.2 Multiple Classes

Let us now generalize to $K > 2$ classes. We take one of the classes, for example, C_K, as the reference class and assume that

$$\log \frac{p(\boldsymbol{x}|C_i)}{p(\boldsymbol{x}|C_K)} = \boldsymbol{w}_i^T \boldsymbol{x} + w_{i0}^o \tag{10.25}$$

Then we have

$$\frac{P(C_i|\boldsymbol{x})}{P(C_K|\boldsymbol{x})} = \exp[\boldsymbol{w}_i^T \boldsymbol{x} + w_{i0}] \tag{10.26}$$

with $w_{i0} = w_{i0}^o + \log P(C_i)/P(C_K)$.

We see that

$$\begin{aligned} \sum_{i=1}^{K-1} \frac{P(C_i|\boldsymbol{x})}{P(C_K|\boldsymbol{x})} &= \frac{1 - P(C_K|\boldsymbol{x})}{P(C_K|\boldsymbol{x})} = \sum_{i=1}^{K-1} \exp[\boldsymbol{w}_i^T \boldsymbol{x} + w_{i0}] \\ &\Rightarrow P(C_K|\boldsymbol{x}) = \frac{1}{1 + \sum_{i=1}^{K-1} \exp[\boldsymbol{w}_i^T \boldsymbol{x} + w_{i0}]} \end{aligned} \tag{10.27}$$

and also that

$$\begin{aligned} \frac{P(C_i|\boldsymbol{x})}{P(C_K|\boldsymbol{x})} &= \exp[\boldsymbol{w}_i^T \boldsymbol{x} + w_{i0}] \\ &\Rightarrow P(C_i|\boldsymbol{x}) = \frac{\exp[\boldsymbol{w}_i^T \boldsymbol{x} + w_{i0}]}{1 + \sum_{j=1}^{K-1} \exp[\boldsymbol{w}_j^T \boldsymbol{x} + w_{j0}]},\ i = 1, \dots, K-1 \end{aligned} \tag{10.28}$$

To treat all classes uniformly, we can write

$$y_i = \hat{P}(C_i|\boldsymbol{x}) = \frac{\exp[\boldsymbol{w}_i^T \boldsymbol{x} + w_{i0}]}{\sum_{j=1}^{K} \exp[\boldsymbol{w}_j^T \boldsymbol{x} + w_{j0}]},\ i = 1, \dots, K \tag{10.29}$$

SOFTMAX

which is called the *softmax* function (Bridle 1990). If the weighted sum for one class is sufficiently larger than for the others, after it is boosted through exponentiation and normalization, its corresponding y_i will be close to 1 and the others will be close to 0. Thus it works like taking a maximum, except that it is differentiable; hence the name softmax. Softmax also guarantees that $\sum_i y_i = 1$.

Let us see how we can learn the parameters. In this case of $K > 2$ classes, each sample point is a multinomial trial with one draw; that is, $\boldsymbol{r}^t|\boldsymbol{x}^t \sim \text{Mult}_K(1, \boldsymbol{y}^t)$, where $y_i^t \equiv P(C_i|\boldsymbol{x}^t)$. The sample likelihood is

$$l(\{\boldsymbol{w}_i, w_{i0}\}_i|\mathcal{X}) = \prod_t \prod_i (y_i^t)^{r_i^t} \tag{10.30}$$

and the error function is again cross-entropy:

$$E(\{\boldsymbol{w}_i, w_{i0}\}_i|\mathcal{X}) = -\sum_t \sum_i r_i^t \log y_i^t \tag{10.31}$$

We again use gradient descent. If $y_i = \exp(a_i)/\sum_j \exp(a_j)$, we have

$$\frac{\partial y_i}{\partial a_j} = y_i(\delta_{ij} - y_j) \tag{10.32}$$

where δ_{ij} is the Kronecker delta, which is 1 if $i = j$ and 0 if $i \neq j$ (exercise 3). Given that $\sum_i r_i^t = 1$, we have the following update equations, for $j = 1, \ldots, K$

$$\begin{aligned}
\Delta \boldsymbol{w}_j &= \eta \sum_t \sum_i \frac{r_i^t}{y_i^t} y_i^t (\delta_{ij} - y_j^t)\boldsymbol{x}^t \\
&= \eta \sum_t \sum_i r_i^t (\delta_{ij} - y_j^t)\boldsymbol{x}^t \\
&= \eta \sum_t \left[\sum_i r_i^t \delta_{ij} - y_j^t \sum_i r_i^t\right] \boldsymbol{x}^t \\
&= \eta \sum_t (r_j^t - y_j^t)\boldsymbol{x}^t
\end{aligned}$$

$$\Delta w_{j0} = \eta \sum_t (r_j^t - y_j^t) \tag{10.33}$$

Note that because of the normalization in softmax, $\boldsymbol{w}_j$ and w_{j0} are affected not only by $\boldsymbol{x}^t \in C_j$ but also by $\boldsymbol{x}^t \in C_i, i \neq j$. The discriminants are updated so that the correct class has the highest weighted sum after softmax, and the other classes have their weighted sums as low as possible. Pseudocode is given in figure 10.8. For a two-dimensional example with three classes, the contour plot is given in figure 10.9, and the discriminants and the posterior probabilities in figure 10.10.

During testing, we calculate all $y_k, k = 1, \ldots, K$ and choose C_i if $y_i = \max_k y_k$. Again we do not need to continue training to minimize cross-entropy as much as possible; we train only until the correct class has the highest weighted sum, and therefore we can stop training earlier by checking the number of misclassifications.

When data are normally distributed, the logistic discriminant has a comparable error rate to the parametric, normal-based linear discriminant (McLachlan 1992). Logistic discrimination can still be used when the class-conditional densities are nonnormal or when they are not unimodal, as long as classes are linearly separable.

```
For $i = 1, \dots, K$
    For $j = 0, \dots, d$
        $w_{ij} \leftarrow$ rand$(-0.01, 0.01)$
Repeat
    For $i = 1, \dots, K$
        For $j = 0, \dots, d$
            $\Delta w_{ij} \leftarrow 0$
    For $t = 1, \dots, N$
        For $i = 1, \dots, K$
            $o_i \leftarrow 0$
            For $j = 0, \dots, d$
                $o_i \leftarrow o_i + w_{ij} x_j^t$
        For $i = 1, \dots, K$
            $y_i \leftarrow \exp(o_i) / \sum_k \exp(o_k)$
        For $i = 1, \dots, K$
            For $j = 0, \dots, d$
                $\Delta w_{ij} \leftarrow \Delta w_{ij} + (r_i^t - y_i) x_j^t$
    For $i = 1, \dots, K$
        For $j = 0, \dots, d$
            $w_{ij} \leftarrow w_{ij} + \eta \Delta w_{ij}$
Until convergence
```

Figure 10.8 Logistic discrimination algorithm implementing gradient descent for the case with $K > 2$ classes. For generality, we take $x_0^t \equiv 1, \forall t$.

The ratio of class-conditional densities is of course not restricted to be linear (Anderson 1982; McLachlan 1992). Assuming a quadratic discriminant, we have

$$\log \frac{p(\boldsymbol{x}|C_i)}{p(\boldsymbol{x}|C_K)} = \boldsymbol{x}^T \mathbf{W}_i \boldsymbol{x} + \boldsymbol{w}_i^T \boldsymbol{x} + w_{i0} \tag{10.34}$$

corresponding to and generalizing parametric discrimination with multivariate normal class-conditionals having different covariance matrices. When d is large, just as we can simplify (regularize) $\mathbf{\Sigma}_i$, we can equally do it on $\mathbf{W}_i$ by taking only its leading eigenvectors into account.

As discussed in section 10.2, any specified function of the basic variables can be included as x-variates. One can, for example, write the dis-

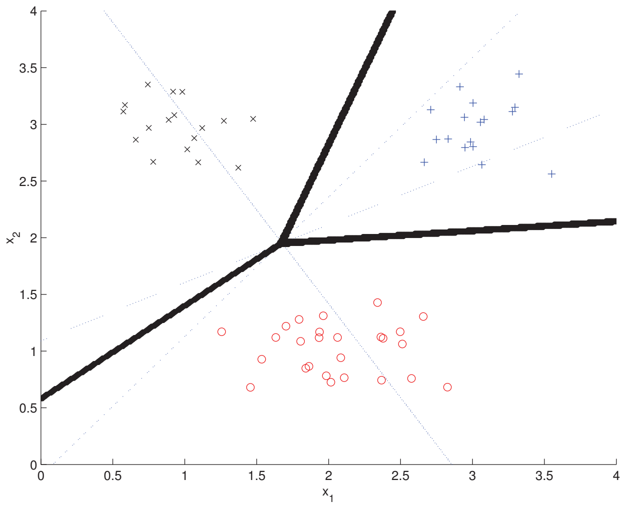


Figure 10.9 For a two-dimensional problem with three classes, the solution found by logistic discrimination. Thin lines are where $g_i(\boldsymbol{x}) = 0$, and the thick line is the boundary induced by the linear classifier choosing the maximum.

criminant as a linear sum of nonlinear basis functions

$$\log \frac{p(\boldsymbol{x}|C_i)}{p(\boldsymbol{x}|C_K)} = \boldsymbol{w}_i^T \boldsymbol{\phi}(\boldsymbol{x}) + w_{i0} \tag{10.35}$$

where $\boldsymbol{\phi}(\cdot)$ are the basis functions, which can be viewed as transformed variables. In neural network terminology, this is called a *multilayer perceptron* (chapter 11), and sigmoid is the most popular basis function. When a Gaussian basis function is used, the model is called *radial basis functions* (chapter 12). We can even use a completely nonparametric approach, for example, Parzen windows (chapter 8).

10.8 Discrimination by Regression

In regression, the probabilistic model is

$$r^t = y^t + \epsilon \tag{10.36}$$

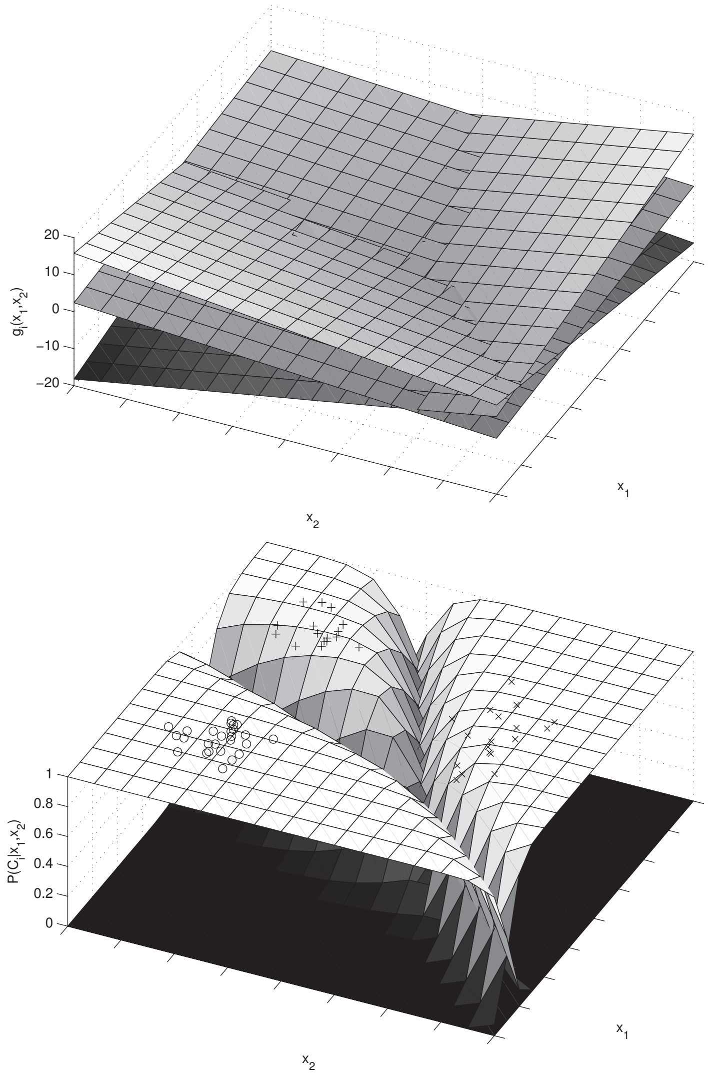


Figure 10.10 For the same example in figure 10.9, the linear discriminants (top), and the posterior probabilities after the softmax (bottom).

where $\epsilon \sim \mathcal{N}(0, \sigma^2)$. If $r^t \in \{0, 1\}$, y^t can be constrained to lie in this range using the sigmoid function. Assuming a linear model and two classes, we have

$$y^t = \text{sigmoid}(\boldsymbol{w}^T \boldsymbol{x}^t + w_0) = \frac{1}{1 + \exp[-(\boldsymbol{w}^T \boldsymbol{x}^t + w_0)]} \tag{10.37}$$

Then the sample likelihood in regression, assuming $r|\boldsymbol{x} \sim \mathcal{N}(y, \sigma^2)$, is

$$l(\boldsymbol{w}, w_0|\mathcal{X}) = \prod_t \frac{1}{\sqrt{2\pi}\sigma} \exp\left[-\frac{(r^t - y^t)^2}{2\sigma^2}\right] \tag{10.38}$$

Maximizing the log likelihood is minimizing the sum of square errors:

$$E(\boldsymbol{w}, w_0|\mathcal{X}) = \frac{1}{2}\sum_t (r^t - y^t)^2 \tag{10.39}$$

Using gradient descent, we get

$$\begin{aligned} \Delta \boldsymbol{w} &= \eta \sum_t (r^t - y^t) y^t (1 - y^t) \boldsymbol{x}^t \\ \Delta w_0 &= \eta \sum_t (r^t - y^t) y^t (1 - y^t) \end{aligned} \tag{10.40}$$

This method can also be used when there are $K > 2$ classes. The probabilistic model is

$$\boldsymbol{r}^t = \boldsymbol{y}^t + \boldsymbol{\epsilon} \tag{10.41}$$

where $\boldsymbol{\epsilon} \sim \mathcal{N}_K(0, \sigma^2 \mathbf{I}_K)$. Assuming a linear model for each class, we have

$$y_i^t = \text{sigmoid}(\boldsymbol{w}_i^T \boldsymbol{x}^t + w_{i0}) = \frac{1}{1 + \exp[-(\boldsymbol{w}_i^T \boldsymbol{x}^t + w_{i0})]} \tag{10.42}$$

Then the sample likelihood is

$$l(\{\boldsymbol{w}_i, w_{i0}\}_i|\mathcal{X}) = \prod_t \frac{1}{(2\pi)^{K/2}|\Sigma|^{1/2}} \exp\left[-\frac{\|\boldsymbol{r}^t - \boldsymbol{y}^t\|^2}{2\sigma^2}\right] \tag{10.43}$$

and the error function is

$$E(\{\boldsymbol{w}_i, w_{i0}\}_i|\mathcal{X}) = \frac{1}{2}\sum_t \|\boldsymbol{r}^t - \boldsymbol{y}^t\|^2 = \frac{1}{2}\sum_t \sum_i (r_i^t - y_i^t)^2 \tag{10.44}$$

The update equations for $i = 1, \ldots, K$, are

$$\begin{aligned} \Delta \boldsymbol{w}_i &= \eta \sum_t (r_i^t - y_i^t) y_i^t (1 - y_i^t) \boldsymbol{x}^t \\ \Delta w_{i0} &= \eta \sum_t (r_i^t - y_i^t) y_i^t (1 - y_i^t) \end{aligned} \tag{10.45}$$

But note that in doing so, we do not make use of the information that only one of y_i needs to be 1 and all others are 0, or that $\sum_i y_i = 1$. The softmax function of equation 10.29 allows us to incorporate this extra information we have due to the outputs' estimating class posterior probabilities. Using sigmoid outputs in $K > 2$ case, we treat y_i as if they are independent functions.

Note also that for a given class, if we use the regression approach, there will be updates until the right output is 1 and all others are 0. This is not in fact necessary because during testing, we are just going to choose the maximum anyway; it is enough to train only until the right output is larger than others, which is exactly what the softmax function does.

So this approach with multiple sigmoid outputs is more appropriate when the classes are *not* mutually exclusive and exhaustive. That is, for an $\boldsymbol{x}^t$, all r_i^t may be 0; namely, $\boldsymbol{x}^t$ does not belong to any of the classes, or more than one r_i^t may be 1, when classes overlap.

10.9 Learning to Rank

RANKING

Ranking is an application area of machine learning that is different from classification and regression, and is sort of between the two. Unlike classification and regression where there is an input $\boldsymbol{x}^t$ and a desired output r^t, in ranking we are asked to put two or more instances in the correct order (Liu 2011).

For example, let us say $\boldsymbol{x}^u$ and $\boldsymbol{x}^v$ represent two movies, and let us say that a user has enjoyed u more than v (in this case, we need to give higher rank to movies similar to u). This is labeled as $r^u \prec r^v$. What we learn is not a discriminant or a regression function but a *score function* $g(\boldsymbol{x}|\theta)$, and what is important are not the absolute values of $g(\boldsymbol{x}^u|\theta)$ and $g(\boldsymbol{x}^v|\theta)$, but that we need to give a higher score to $\boldsymbol{x}^u$ than $\boldsymbol{x}^v$; that is, $g(\boldsymbol{x}^u|\theta) > g(\boldsymbol{x}^v|\theta)$ should be satisfied for all such pairs of u and v.

As usual, we assume a certain model $g(\cdot)$ and we optimize its parameters θ so that all rank constraints are satisfied. Then, for example, to make a recommendation among the movies that the user has not yet seen, we choose the one with the highest score:

$$\text{Choose } u \text{ if } g(\boldsymbol{x}^u|\theta) = \max_t g(\boldsymbol{x}^t|\theta)$$

Sometimes, instead of only the topmost, we may want a list of the highest k.

We can note here the advantage and difference of a ranker. If users rate the movies they have seen as "enjoyed/not enjoyed," this will be a two-class classification problem and a classifier can be used, but taste is nuanced and a binary rating is very hard. On the other hand, if people rate their enjoyment of a movie on a scale of, say, 1 to 10, this will be a regression problem, but such absolute values are difficult to assign. It is more natural and easier for people to say that of the two movies they have watched, they like one more than the other, instead of a yes/no decision or a numeric value.

Ranking has many applications. In search engines, for example, given a query, we want to retrieve the most relevant documents. If we retrieve and display the current top ten candidates and then the user clicks the third one skipping the first two, we understand that the third should have been ranked higher than the first and the second. Such click logs are used to train rankers.

Sometimes reranking is used to improve the output of a ranker with additional information. For example, in speech recognition, an acoustic model can first be used to generate an ordered list of possible sentences, and then the N-best candidates can then be reranked using features from a language model; this can improve accuracy significantly (Shen and Joshi 2005).

A ranker can be trained in a number of different ways. For all (u, v) pairs where $r^u \prec r^v$ is defined, we have an error if $g(\boldsymbol{x}^v|\theta) > g(\boldsymbol{x}^u|\theta)$. Generally, we do not have a full ordering of all N^2 pairs but a subset, thereby defining a partial order. The sum of differences make up the error:

$$E(\boldsymbol{w}|\{r^u, r^v\}) = \sum_{r^u \prec r^v} \left[g(\boldsymbol{x}^v|\theta) - g(\boldsymbol{x}^u|\theta)\right]_+ \tag{10.46}$$

where a_+ is equal to a if $a \geq 0$ and 0 otherwise.

Let us use a linear model, as we do throughout this chapter:

$$g(\boldsymbol{x}|\boldsymbol{w}) = \boldsymbol{w}^T\boldsymbol{x} \tag{10.47}$$

Because we do not care about the absolute values, we do not need w_0. The error in equation 10.46 becomes

$$E(\boldsymbol{w}|\{r^u, r^v\}) = \sum_{r^u \prec r^v} \boldsymbol{w}^T(\boldsymbol{x}^v - \boldsymbol{x}^u)_+ \tag{10.48}$$

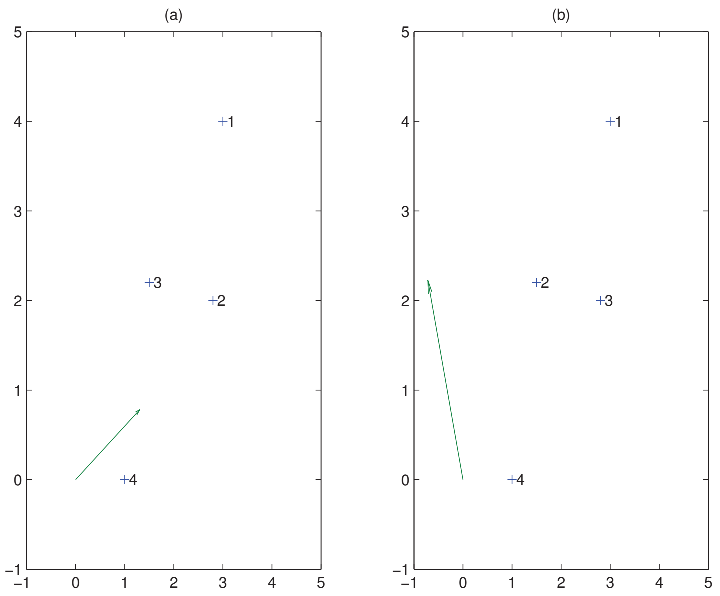


Figure 10.11 Sample ranking problems and solutions. Data points are indicated by '+' and the numbers next to them indicate the rank where 1 is the highest. We have a full ordering here. The arrow indicate the learned $\boldsymbol{w}$. In (a) and (b), we see two different ranking problems and the two corresponding solutions.

We can do an online update of $\boldsymbol{w}$ using gradient descent. For each $r^u \prec r^v$ where $g(\boldsymbol{x}^v|\theta) > g(\boldsymbol{x}^u|\theta)$, we do a small update:

$$\Delta w_j = -\eta \frac{\partial E}{\partial w_j} = -\eta(\boldsymbol{x}_j^v - \boldsymbol{x}_j^u), j = 1, \ldots, d \tag{10.49}$$

$\boldsymbol{w}$ is chosen so that when the instances are projected onto $\boldsymbol{w}$, the correct orderings are obtained. In figure 10.11, we see example data and the projection directions learned. We see there that a slight change in the ranks may cause a big change in $\boldsymbol{w}$.

For error functions and gradient descent approaches in ranking and their use in practice, see Burges et al. 2005 and Shin and Josh 2005. Sometimes for confident decision, when $r^u \prec r^v$, we require that the output be not only larger, but larger with a margin, for example, $g(\boldsymbol{x}^u|\theta) > 1 + g(\boldsymbol{x}^u|\theta)$; we will see an example of this when we talk about learning to rank using kernel machines in section 13.11.

10.10 Notes

The linear discriminant, due to its simplicity, is the classifier most used in pattern recognition (Duda, Hart, and Stork 2001; McLachlan 1992). We discussed the case of Gaussian distributions with a common covariance matrix in chapter 4 and Fisher's linear discriminant in chapter 6, and in this chapter we discuss the logistic discriminant. In chapter 11, we discuss the perceptron that is the neural network implementation of the linear discriminant. In chapter 13, we discuss support vector machines, another type of linear discriminant.

Logistic discrimination is covered in more detail in Anderson 1982 and in McLachlan 1992. Logistic (sigmoid) is the inverse of logit, which is the *canonical link* in case of Bernoulli samples. Softmax is its generalization to multinomial samples. More information on such *generalized linear models* is given in McCullogh and Nelder 1989.

GENERALIZED LINEAR MODELS

Ranking has recently become a major application area of machine learning because of its use in search engines, information retrieval, and natural language processing. An extensive review of both important applications and machine learning algorithms is given in Liu 2011. The model we discussed here is a linear model; in section 13.11, we discuss how to learn a ranker using kernel machines where we get a nonlinear model with kernels that allow the integration of different measures of similarity.

Generalizing linear models by using nonlinear basis functions is a very old idea. We will discuss multilayer perceptrons (chapter 11) and radial basis functions (chapter 12) where the parameters of the basis functions can also be learned from data while learning the discriminant. Support vector machines (chapter 13) use kernel functions built from such basis functions.

10.11 Exercises

1. For each of the following basis functions, describe where it is nonzero:

   a. $\sin(x_1)$

   b. $\exp(-(x_1 - a)^2/c)$

   c. $\exp(-\|\boldsymbol{x} - \boldsymbol{a}\|^2/c)$

   d. $\log(x_2)$

   e. $1(x_1 > c)$

f. $1(ax_1 + bx_2 > c)$

2. For the two-dimensional case of figure 10.2, show equations 10.4 and 10.5.

3. Show that the derivative of the softmax, $y_i = \exp(a_i)/\sum_j \exp(a_j)$, is $\partial y_i/\partial a_j = y_i(\delta_{ij} - y_j)$, where δ_{ij} is 1 if $i = j$ and 0 otherwise.

4. With $K = 2$, show that using two softmax outputs is equal to using one sigmoid output.

   SOLUTION:

$$\begin{aligned} y_1 &= \frac{\exp o_1}{\exp o_1 + \exp o_2} = \frac{1}{1 + \exp(o_2 - o_1)} = \frac{1}{1 + \exp(-(o_1 - o_2))} \\ &= \text{sigmoid}(o_1 - o_2) \end{aligned}$$

   For example, if we have $o_1 = \boldsymbol{w}_1^T \boldsymbol{x}$, we have

$$y_1 = \frac{\exp \boldsymbol{w}_1^T \boldsymbol{x}}{\exp \boldsymbol{w}_1^T \boldsymbol{x} + \exp \boldsymbol{w}_2^T \boldsymbol{x}} = \text{sigmoid}(\boldsymbol{w}_1^T \boldsymbol{x} - \boldsymbol{w}_2^T \boldsymbol{x}) = \text{sigmoid}(\boldsymbol{w}^T \boldsymbol{x})$$

   where $\boldsymbol{w} \equiv \boldsymbol{w}_1 - \boldsymbol{w}_2$ and $y_2 = 1 - y_1$.

5. How can we learn $\mathbf{W}_i$ in equation 10.34?

   SOLUTION: For example, if we have two inputs x_1 and x_2, we have

$$\begin{aligned} \log \frac{p(x_1, x_2|C_i)}{p(x_1, x_2|C_K)} &= \mathbf{W}_{i11}x_1^2 + \mathbf{W}_{i12}x_1x_2 + \mathbf{W}_{i21}x_2x_1 + \mathbf{W}_{i22}x_2^2 \\ &+ w_{i1}x_1 + w_{i2}x_2 + w_{i0} \end{aligned}$$

   Then we can use gradient descent and take derivative with respect to any $\mathbf{W}_{jkl}$ to calculate an update rule:

$$\Delta \mathbf{W}_{jkl} = \eta \sum_t (r_j^t - y_j^t) x_k^t x_l^t$$

6. In using quadratic (or higher-order) discriminants as in equation 10.34, how can we keep variance under control?

7. What is the implication of the use of a single η for all x_j in gradient descent?

   SOLUTION: Using a single η for all x_j implies doing updates in the same scale, which in turn implies that all x_j are in the same scale. If they are not, it is a good idea to normalize all x_j, for example, by z-normalization, before training. Note that we need to save the scaling parameters for all inputs, so that the same scaling can also be done later to the test instances.

8. In the univariate case for classification as in figure 10.7, what do w and w_0 correspond to?

   SOLUTION: The slope and the intercept of the line, which are then fed to the sigmoid.

9. Let us say for univariate x, $x \in (2, 4)$ belong to C_1 and $x < 2$ or $x > 4$ belong to C_2. How can we separate the two classes using a linear discriminant?

   SOLUTION: We define an extra variable $z \equiv x^2$ and use the linear discriminant $w_2 z + w_1 x + w_0$ in the (z, x) space, which corresponds to a quadratic discriminant in the x space. For example, we can manually write

$$\text{Choose } \begin{cases} C_1 & \text{if } (x-3)^2 - 1 \le 0 \\ C_2 & \text{otherwise} \end{cases}$$

   or rewrite it using a sigmoid (see figure 10.12):

$$\text{Choose } \begin{cases} C_1 & \text{if sigmoid}((x-3)^2 - 1) \le 0.5 \\ C_2 & \text{otherwise} \end{cases}$$

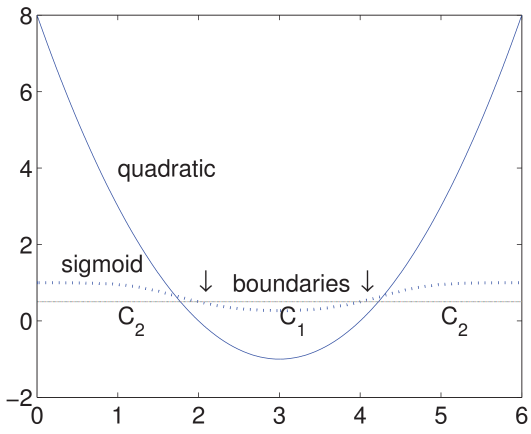


Figure 10.12 The quadratic discriminant, before and after the sigmoid. The boundaries are where the discriminant is 0 or where the sigmoid is 0.5.

   Or, we can use *two* linear discriminants in the x space, one separating at 2 and the other separating at 4, and then we can OR them. Such layered linear discriminants are discussed in chapter 11.

10. For the sample data in figure 10.11, define ranks such that a linear model would not be able to learn them. Explain how the model can be generalized so that they can be learned.

10.12 References

Aizerman, M. A., E. M. Braverman, and L. I. Rozonoer. 1964. “Theoretical Foundations of the Potential Function Method in Pattern Recognition Learning.” *Automation and Remote Control* 25:821–837.

Anderson, J. A. 1982. “Logistic Discrimination.” In *Handbook of Statistics*, Vol. 2, *Classification, Pattern Recognition and Reduction of Dimensionality*, ed. P. R. Krishnaiah and L. N. Kanal, 169–191. Amsterdam: North Holland.

Bridle, J. S. 1990. “Probabilistic Interpretation of Feedforward Classification Network Outputs with Relationships to Statistical Pattern Recognition.” In *Neurocomputing: Algorithms, Architectures and Applications*, ed. F. Fogelman-Soulie and J. Herault, 227–236. Berlin: Springer.

Burges, C., T. Shaked, E. Renshaw, A. Lazier, M. Deeds, N. Hamilton, and G. Hullender. 2005. “Learning to Rank using Gradient Descent.” In *22nd International Conference on Machine Learning*, 89–96, New York: ACM Press.

Duda, R. O., P. E. Hart, and D. G. Stork. 2001. *Pattern Classification*, 2nd ed. New York: Wiley.

Liu, T.-Y. 2011. *Learning to Rank for Information Retrieval*. Heidelberg: Springer.

McCullagh, P., and J. A. Nelder. 1989. *Generalized Linear Models*. London: Chapman and Hall.

McLachlan, G. J. 1992. *Discriminant Analysis and Statistical Pattern Recognition*. New York: Wiley.

Shen, L., and A. K. Joshi. 2005. “Ranking and Reranking with Perceptron.” *Machine Learning* 60:73–96.

Vapnik, V. 1995. *The Nature of Statistical Learning Theory*. New York: Springer.

11 *Multilayer Perceptrons*

The multilayer perceptron is an artificial neural network structure and is a nonparametric estimator that can be used for classification and regression. We discuss the backpropagation algorithm to train a multilayer perceptron for a variety of applications.

11.1 Introduction

ARTIFICIAL NEURAL network models, one of which is the *perceptron* we discuss in this chapter, take their inspiration from the brain. There are cognitive scientists and neuroscientists whose aim is to understand the functioning of the brain (Posner 1989; Thagard 2005), and toward this aim, build models of the natural neural networks in the brain and make simulation studies.

ARTIFICIAL NEURAL NETWORKS

However, in engineering, our aim is not to understand the brain per se, but to build useful machines. We are interested in *artificial neural networks* because we believe that they may help us build better computer systems. The brain is an information processing device that has some incredible abilities and surpasses current engineering products in many domains—for example, vision, speech recognition, and learning, to name three. These applications have evident economic utility if implemented on machines. If we can understand how the brain performs these functions, we can define solutions to these tasks as formal algorithms and implement them on computers.

NEURONS

The human brain is quite different from a computer. Whereas a computer generally has one processor, the brain is composed of a very large (10^{11}) number of processing units, namely, *neurons*, operating in parallel. Though the details are not known, the processing units are believed to be much simpler and slower than a processor in a computer. What also makes the brain different, and is believed to provide its computational power, is the large connectivity. Neurons in the brain have connections, called *synapses*, to around 10^4 other neurons, all operating in parallel. In a computer, the processor is active and the memory is separate and passive, but it is believed that in the brain, both the processing and memory are distributed together over the network; processing is done by the neurons, and the memory is in the synapses between the neurons.

SYNAPSES

11.1.1 Understanding the Brain

According to Marr (1982), understanding an information processing system has three levels, called the *levels of analysis*:

LEVELS OF ANALYSIS

1. *Computational theory* corresponds to the goal of computation and an abstract definition of the task.

2. *Representation and algorithm* is about how the input and the output are represented and about the specification of the algorithm for the transformation from the input to the output.

3. *Hardware implementation* is the actual physical realization of the system.

One example is sorting: The computational theory is to order a given set of elements. The representation may use integers, and the algorithm may be Quicksort. After compilation, the executable code for a particular processor sorting integers represented in binary is one hardware implementation.

The idea is that for the same computational theory, there may be multiple representations and algorithms manipulating symbols in that representation. Similarly, for any given representation and algorithm, there may be multiple hardware implementations. We can use one of various sorting algorithms, and even the same algorithm can be compiled on computers with different processors and lead to different hardware implementations.

To take another example, ‘6’, ‘VI’, and ‘110’ are three different representations of the number six. There is a different algorithm for addition depending on the representation used. Digital computers use binary representation and have circuitry to add in this representation, which is one particular hardware implementation. Numbers are represented differently, and addition corresponds to a different set of instructions on an abacus, which is another hardware implementation. When we add two numbers in our head, we use another representation and an algorithm suitable to that representation, which is implemented by the neurons. But all these different hardware implementations—for example, us, abacus, digital computer—implement the same computational theory, addition.

The classic example is the difference between natural and artificial flying machines. A sparrow flaps its wings; a commercial airplane does not flap its wings but uses jet engines. The sparrow and the airplane are two hardware implementations built for different purposes, satisfying different constraints. But they both implement the same theory, which is aerodynamics.

The brain is one hardware implementation for learning or pattern recognition. If from this particular implementation, we can do reverse engineering and extract the representation and the algorithm used, and if from that in turn, we can get the computational theory, we can then use another representation and algorithm, and in turn a hardware implementation more suited to the means and constraints we have. One hopes our implementation will be cheaper, faster, and more accurate.

Just as the initial attempts to build flying machines looked very much like birds until we discovered aerodynamics, it is also expected that the first attempts to build structures possessing brain's abilities will look like the brain with networks of large numbers of processing units, until we discover the computational theory of intelligence. So it can be said that in understanding the brain, when we are working on artificial neural networks, we are at the representation and algorithm level.

Just as the feathers are irrelevant to flying, in time we may discover that neurons and synapses are irrelevant to intelligence. But until that time there is one other reason why we are interested in understanding the functioning of the brain, and that is related to parallel processing.

11.1.2 Neural Networks as a Paradigm for Parallel Processing

Since the 1980s, computer systems with thousands of processors have been commercially available. The software for such parallel architectures, however, has not advanced as quickly as hardware. The reason for this is that almost all our theory of computation up to that point was based on serial, one-processor machines. We are not able to use the parallel machines we have efficiently because we cannot program them efficiently.

PARALLEL PROCESSING

There are mainly two paradigms for *parallel processing*: In single instruction, multiple data (SIMD) machines, all processors execute the same instruction but on different pieces of data. In multiple instruction, multiple data (MIMD) machines, different processors may execute different instructions on different data. SIMD machines are easier to program because there is only one program to write. However, problems rarely have such a regular structure that they can be parallelized over a SIMD machine. MIMD machines are more general, but it is not an easy task to write separate programs for all the individual processors; additional problems are related to synchronization, data transfer between processors, and so forth. SIMD machines are also easier to build, and machines with more processors can be constructed if they are SIMD. In MIMD machines, processors are more complex, and a more complex communication network should be constructed for the processors to exchange data arbitrarily.

Assume now that we can have machines where processors are a little bit more complex than SIMD processors but not as complex as MIMD processors. Assume we have simple processors with a small amount of local memory where some parameters can be stored. Each processor implements a fixed function and executes the same instructions as SIMD processors; but by loading different values into the local memory, they can be doing different things and the whole operation can be distributed over such processors. We will then have what we can call neural instruction, multiple data (NIMD) machines, where each processor corresponds to a neuron, local parameters correspond to its synaptic weights, and the whole structure is a neural network. If the function implemented in each processor is simple and if the local memory is small, then many such processors can be fit on a single chip.

The problem now is to distribute a task over a network of such processors and to determine the local parameter values. This is where learning comes into play: We do not need to program such machines and determine the parameter values ourselves if such machines can learn from examples.

Thus, artificial neural networks are a way to make use of the parallel hardware we can build with current technology and—thanks to learning—they need not be programmed. Therefore, we also save ourselves the effort of programming them.

In this chapter, we discuss such structures and how they are trained.

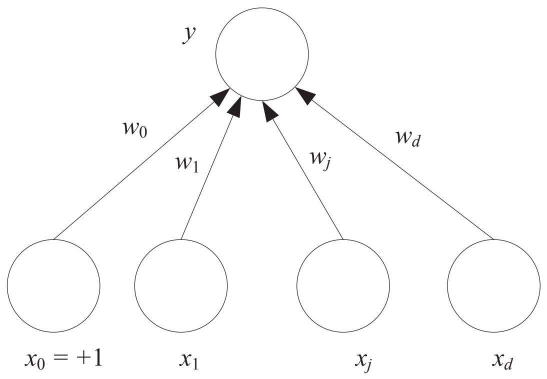


Figure 11.1 Simple perceptron. $x_j, j = 1, \ldots, d$ are the input units. x_0 is the bias unit that always has the value 1. y is the output unit. w_j is the weight of the directed connection from input x_j to the output.

Keep in mind that the operation of an artificial neural network is a mathematical function that can be implemented on a serial computer—as it generally is—and training the network is not much different from statistical techniques that we have discussed in the previous chapters. Thinking of this operation as being carried out on a network of simple processing units is meaningful only if we have the parallel hardware, and only if the network is so large that it cannot be simulated fast enough on a serial computer.

11.2 The Perceptron

PERCEPTRON The *perceptron* is the basic processing element. It has inputs that may come from the environment or may be the outputs of other perceptrons. CONNECTION WEIGHT Associated with each input, $x_j \in \Re, j = 1, \ldots, d$, is a *connection weight*, SYNAPTIC WEIGHT or *synaptic weight* $w_j \in \Re$, and the output, y, in the simplest case is a weighted sum of the inputs (see figure 11.1):

$$y = \sum_{j=1}^{d} w_j x_j + w_0 \qquad (11.1)$$

w_0 is the intercept value to make the model more general; it is generally BIAS UNIT modeled as the weight coming from an extra *bias unit*, x_0, which is always

$+1$. We can write the output of the perceptron as a dot product

$$y = \boldsymbol{w}^T \boldsymbol{x} \tag{11.2}$$

where $\boldsymbol{w} = [w_0, w_1, \ldots, w_d]^T$ and $\boldsymbol{x} = [1, x_1, \ldots, x_d]^T$ are *augmented* vectors to include also the bias weight and input.

During testing, with given weights, $\boldsymbol{w}$, for input $\boldsymbol{x}$, we compute the output y. To implement a given task, we need to *learn* the weights $\boldsymbol{w}$, the parameters of the system, such that correct outputs are generated given the inputs.

When $d = 1$ and x is fed from the environment through an input unit, we have

$$y = wx + w_0$$

which is the equation of a line with w as the slope and w_0 as the intercept. Thus this perceptron with one input and one output can be used to implement a linear fit. With more than one input, the line becomes a (hyper)plane, and the perceptron with more than one input can be used to implement multivariate linear fit. Given a sample, the parameters w_j can be found by regression (see section 5.8).

The perceptron as defined in equation 11.1 defines a hyperplane and as such can be used to divide the input space into two: the half-space where it is positive and the half-space where it is negative (see chapter 10). By using it to implement a linear discriminant function, the perceptron can separate two classes by checking the sign of the output. If we define $s(\cdot)$ as the *threshold function*

THRESHOLD FUNCTION

$$s(a) = \begin{cases} 1 & \text{if } a > 0 \\ 0 & \text{otherwise} \end{cases} \tag{11.3}$$

then we can

$$\text{choose} \begin{cases} C_1 & \text{if } s(\boldsymbol{w}^T \boldsymbol{x}) > 0 \\ C_2 & \text{otherwise} \end{cases}$$

Remember that using a linear discriminant assumes that classes are linearly separable. That is to say, it is assumed that a hyperplane $\boldsymbol{w}^T \boldsymbol{x} = 0$ can be found that separates $\boldsymbol{x}^t \in C_1$ and $\boldsymbol{x}^t \in C_2$. If at a later stage we need the posterior probability—for example, to calculate risk—we need to use the sigmoid function at the output as

$$\begin{aligned} o &= \boldsymbol{w}^T \boldsymbol{x} \\ y &= \text{sigmoid}(o) = \frac{1}{1 + \exp[-\boldsymbol{w}^T \boldsymbol{x}]} \end{aligned} \tag{11.4}$$

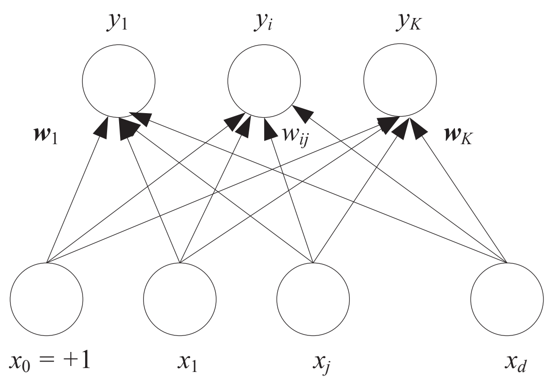


Figure 11.2 K parallel perceptrons. $x_j, j = 0, \ldots, d$ are the inputs and $y_i, i = 1, \ldots, K$ are the outputs. w_{ij} is the weight of the connection from input x_j to output y_i. Each output is a weighted sum of the inputs. When used for K-class classification problem, there is a postprocessing to choose the maximum, or softmax if we need the posterior probabilities.

When there are $K > 2$ outputs, there are K perceptrons, each of which has a weight vector $\boldsymbol{w}_i$ (see figure 11.2)

$$\begin{aligned} y_i &= \sum_{j=1}^{d} w_{ij}x_j + w_{i0} = \boldsymbol{w}_i^T \boldsymbol{x} \\ \boldsymbol{y} &= \mathbf{W}\boldsymbol{x} \end{aligned} \tag{11.5}$$

where w_{ij} is the weight from input x_j to output y_i. $\mathbf{W}$ is the $K \times (d+1)$ weight matrix of w_{ij} whose rows are the weight vectors of the K perceptrons. When used for classification, during testing, we

$$\text{choose } C_i \text{ if } y_i = \max_k y_k$$

Each perceptron is a *local* function of its inputs and synaptic weights. In classification, if we need the posterior probabilities (instead of just the code of the winner class) and use the softmax, we need the values of all outputs. Implementing this as a neural network results in a two-stage process, where the first calculates the weighted sums, and the second calculates the softmax values; but we denote this as a single layer:

$$\begin{aligned} o_i &= \boldsymbol{w}_i^T \boldsymbol{x} \\ y_i &= \frac{\exp o_i}{\sum_k \exp o_k} \end{aligned} \tag{11.6}$$

Remember that by defining auxiliary inputs, the linear model can also be used for polynomial approximation; for example, define $x_3 = x_1^2, x_4 = x_2^2, x_5 = x_1 x_2$ (section 10.2). The same can also be used with perceptrons (Durbin and Rumelhart 1989). In section 11.5, we see multilayer perceptrons where such nonlinear functions are learned from data in a "hidden" layer instead of being assumed a priori.

Any of the methods discussed in chapter 10 on linear discrimination can be used to calculate $\boldsymbol{w}_i, i = 1, \ldots, K$ offline and then plugged into the network. These include parametric approach with a common covariance matrix, logistic discrimination, discrimination by regression, and support vector machines. In some cases, we do not have the whole sample at hand when training starts, and we need to iteratively update parameters as new examples arrive; we discuss this case of *online* learning in section 11.3.

Equation 11.5 defines a linear transformation from a d-dimensional space to a K-dimensional space and can also be used for dimensionality reduction if $K < d$. One can use any of the methods of chapter 6 to calculate $\mathbf{W}$ offline and then use the perceptrons to implement the transformation, for example, PCA. In such a case, we have a two-layer network where the first layer of perceptrons implements the linear transformation and the second layer implements the linear regression or classification in the new space. We note that because both are linear transformations, they can be combined and written down as a single layer. We will see the more interesting case where the first layer implements *nonlinear* dimensionality reduction in section 11.5.

11.3 Training a Perceptron

The perceptron defines a hyperplane, and the neural network perceptron is just a way of *implementing* the hyperplane. Given a data sample, the weight values can be calculated *offline* and then when they are plugged in, the perceptron can be used to calculate the output values.

In training neural networks, we generally use online learning where we are not given the whole sample, but we are given instances one by one and would like the network to update its parameters after each instance, adapting itself slowly in time. Such an approach is interesting for a number of reasons:

1. It saves us the cost of storing the training sample in an external memory and storing the intermediate results during optimization. An ap-

proach like support vector machines (chapter 13) may be quite costly with large samples, and in some applications, we may prefer a simpler approach where we do not need to store the whole sample and solve a complex optimization problem on it.

2. The problem may be changing in time, which means that the sample distribution is not fixed, and a training set cannot be chosen a priori. For example, we may be implementing a speech recognition system that adapts itself to its user.

3. There may be physical changes in the system. For example, in a robotic system, the components of the system may wear out, or sensors may degrade.

ONLINE LEARNING

In *online learning*, we do not write the error function over the whole sample but on individual instances. Starting from random initial weights, at each iteration we adjust the parameters a little bit to minimize the error, without forgetting what we have previously learned. If this error function is differentiable, we can use gradient descent.

For example, in regression the error on the single instance pair with index t, (x^t, r^t), is

$$E^t(\boldsymbol{w}|\boldsymbol{x}^t, r^t) = \frac{1}{2}(r^t - y^t)^2 = \frac{1}{2}[r^t - (\boldsymbol{w}^T\boldsymbol{x}^t)]^2$$

and for $j = 0, \ldots, d$, the online update is

$$\Delta w_j^t = \eta(r^t - y^t)x_j^t \tag{11.7}$$

where η is the learning factor, which is gradually decreased in time for convergence. This is known as *stochastic gradient descent*.

STOCHASTIC GRADIENT DESCENT

Similarly, update rules can be derived for classification problems using logistic discrimination where updates are done after each pattern, instead of summing them and doing the update after a complete pass over the training set. With two classes, for the single instance $(\boldsymbol{x}^t, \boldsymbol{r}^t)$ where $r_i^t = 1$ if $\boldsymbol{x}^t \in C_1$ and $r_i^t = 0$ if $\boldsymbol{x}^t \in C_2$, the single output is

$$y^t = \text{sigmoid}(\boldsymbol{w}^T\boldsymbol{x}^t)$$

and the cross-entropy is

$$E^t(\boldsymbol{w}|\boldsymbol{x}^t, r^t) = -r^t \log y^t - (1 - r^t)\log(1 - y^t)$$

Using gradient descent, we get the following online update rule for $j = 0, \ldots, d$:

$$\Delta w_j^t = \eta(r^t - y^t)x_j^t \tag{11.8}$$

When there are $K > 2$ classes, for the single instance $(\boldsymbol{x}^t, \boldsymbol{r}^t)$ where $r_i^t = 1$ if $\boldsymbol{x}^t \in C_i$ and 0 otherwise, the outputs are

$$y_i^t = \frac{\exp \boldsymbol{w}_i^T \boldsymbol{x}^t}{\sum_k \exp \boldsymbol{w}_k^T \boldsymbol{x}^t}$$

and the cross-entropy is

$$E^t(\{\boldsymbol{w}_i\}_i | \boldsymbol{x}^t, \boldsymbol{r}^t) = -\sum_i r_i^t \log y_i^t$$

Using gradient descent, we get the following online update rule, for $i = 1, \ldots, K,\ j = 0, \ldots, d$:

$$\Delta w_{ij}^t = \eta(r_i^t - y_i^t)x_j^t \tag{11.9}$$

which is the same as the equations we saw in section 10.7 except that we do not sum over all of the instances but update after a single instance. The pseudocode of the algorithm is given in figure 11.3, which is the online version of figure 10.8.

Both equations 11.7 and 11.9 have the form

$$\text{Update} = \text{LearningFactor} \cdot (\text{DesiredOutput} - \text{ActualOutput}) \cdot \text{Input} \tag{11.10}$$

Let us try to get some insight into what this does. First, if the actual output is equal to the desired output, no update is done. When it is done, the magnitude of the update increases as the difference between the desired output and the actual output increases. We also see that if the actual output is less than the desired output, update is positive if the input is positive and negative if the input is negative. This has the effect of increasing the actual output and decreasing the difference. If the actual output is greater than the desired output, update is negative if the input is positive and positive if the input is negative; this decreases the actual output and makes it closer to the desired output.

When an update is done, its magnitude depends also on the input. If the input is close to 0, its effect on the actual output is small and therefore its weight is also updated by a small amount. The greater an input, the greater the update of its weight.

```
For $i = 1, \ldots, K$
    For $j = 0, \ldots, d$
        $w_{ij} \leftarrow$ rand$(-0.01, 0.01)$
Repeat
    For all $(\boldsymbol{x}^t, r^t) \in \mathcal{X}$ in random order
        For $i = 1, \ldots, K$
            $o_i \leftarrow 0$
            For $j = 0, \ldots, d$
                $o_i \leftarrow o_i + w_{ij} x_j^t$
        For $i = 1, \ldots, K$
            $y_i \leftarrow \exp(o_i) / \sum_k \exp(o_k)$
        For $i = 1, \ldots, K$
            For $j = 0, \ldots, d$
                $w_{ij} \leftarrow w_{ij} + \eta(r_i^t - y_i) x_j^t$
Until convergence
```

Figure 11.3 Perceptron training algorithm implementing stochastic online gradient descent for the case with $K > 2$ classes. This is the online version of the algorithm given in figure 10.8.

Finally, the magnitude of the update depends on the learning factor, η. If it is too large, updates depend too much on recent instances; it is as if the system has a very short memory. If this factor is small, many updates may be needed for convergence. In section 11.8.1, we discuss methods to speed up convergence.

11.4 Learning Boolean Functions

In a Boolean function, the inputs are binary and the output is 1 if the corresponding function value is true and 0 otherwise. Therefore, it can be seen as a two-class classification problem. As an example, for learning to AND two inputs, the table of inputs and required outputs is given in table 11.1. An example of a perceptron that implements AND and its geometric interpretation in two dimensions is given in figure 11.4. The discriminant is

$$y = s(x_1 + x_2 - 1.5)$$

Table 11.1 Input and output for the AND function

x_1	x_2	r
0	0	0
0	1	0
1	0	0
1	1	1

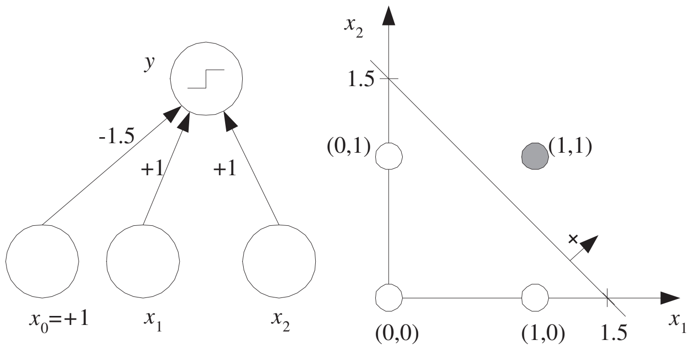


Figure 11.4 The perceptron that implements AND and its geometric interpretation.

that is, $\boldsymbol{x} = [1, x_1, x_2]^T$ and $\boldsymbol{w} = [-1.5, 1, 1]^T$. Note that $y = s(x_1 + x_2 - 1.5)$ satisfies the four constraints given by the definition of AND function in table 11.1, for example, for $x_1 = 1, x_2 = 0$, $y = s(-0.5) = 0$. Similarly it can be shown that $y = s(x_1 + x_2 - 0.5)$ implements OR.

Though Boolean functions like AND and OR are linearly separable and are solvable using the perceptron, certain functions like XOR are not. The table of inputs and required outputs for XOR is given in table 11.2. As can be seen in figure 11.5, the problem is not linearly separable. This can also be proved by noting that there are no w_0, w_1, and w_2 values that satisfy the following set of inequalities:

$$\begin{array}{rrrl} & & w_0 & \leq 0 \\ & w_2 + & w_0 & > 0 \\ w_1 + & & w_0 & > 0 \\ w_1 + & w_2 + & w_0 & \leq 0 \end{array}$$

Table 11.2 Input and output for the XOR function

x_1	x_2	r
0	0	0
0	1	1
1	0	1
1	1	0

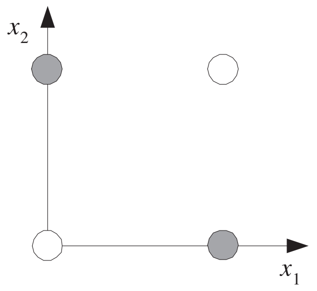


Figure 11.5 XOR problem is not linearly separable. We cannot draw a line where the empty circles are on one side and the filled circles on the other side.

This result should not be very surprising to us since the VC dimension of a line (in two dimensions) is three. With two binary inputs there are four cases, and thus we know that there exist problems with two inputs that are not solvable using a line; XOR is one of them.

11.5 Multilayer Perceptrons

A perceptron that has a single layer of weights can only approximate linear functions of the input and cannot solve problems like the XOR, where the discrimininant to be estimated is nonlinear. Similarly, a perceptron cannot be used for nonlinear regression. This limitation does not apply to feedforward networks with intermediate or *hidden layers* between the input and the output layers. If used for classification, such *multilayer perceptrons* (MLP) can implement nonlinear discriminants and, if used for regression, can approximate nonlinear functions of the input.

HIDDEN LAYERS

MULTILAYER PERCEPTRONS

Input $\boldsymbol{x}$ is fed to the input layer (including the bias), the "activation" propagates in the forward direction, and the values of the hidden units z_h are calculated (see figure 11.6). Each hidden unit is a perceptron by itself and applies the nonlinear sigmoid function to its weighted sum:

$$z_h = \text{sigmoid}(\boldsymbol{w}_h^T \boldsymbol{x}) = \frac{1}{1 + \exp\left[-\left(\sum_{j=1}^{d} w_{hj}x_j + w_{h0}\right)\right]}, \; h = 1, \ldots, H \tag{11.11}$$

The output y_i are perceptrons in the second layer taking the hidden units as their inputs

$$y_i = \boldsymbol{v}_i^T \boldsymbol{z} = \sum_{h=1}^{H} v_{ih}z_h + v_{i0} \tag{11.12}$$

where there is also a bias unit in the hidden layer, which we denote by z_0, and v_{i0} are the bias weights. The input layer of x_j is not counted since no computation is done there and when there is a hidden layer, this is a two-layer network.

As usual, in a regression problem, there is no nonlinearity in the output layer in calculating y. In a two-class discrimination task, there is one sigmoid output unit and when there are $K > 2$ classes, there are K outputs with softmax as the output nonlinearity.

If the hidden units' outputs were linear, the hidden layer would be of no use: Linear combination of linear combinations is another linear combination. Sigmoid is the continuous, differentiable version of thresholding. We need differentiability because the learning equations we will see are gradient-based. Another sigmoid (S-shaped) nonlinear basis function that can be used is the hyperbolic tangent function, tanh, which ranges from -1 to $+1$, instead of 0 to $+1$. In practice, there is no difference between using the sigmoid and the tanh. Still another possibility is the Gaussian, which uses Euclidean distance instead of the dot product for similarity; we discuss such radial basis function networks in chapter 12.

The output is a linear combination of the nonlinear basis function values computed by the hidden units. It can be said that the hidden units make a nonlinear transformation from the d-dimensional input space to the H-dimensional space spanned by the hidden units, and, in this space, the second output layer implements a linear function.

One is not limited to having one hidden layer, and more hidden layers with their own incoming weights can be placed after the first hidden layer with sigmoid hidden units, thus calculating nonlinear functions of the

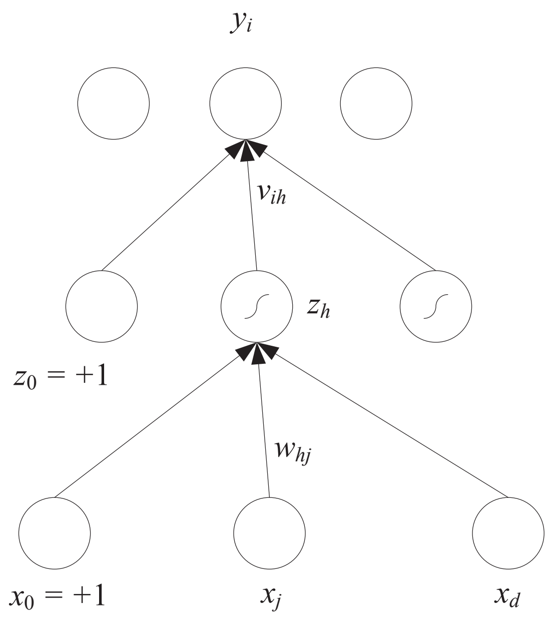


Figure 11.6 The structure of a multilayer perceptron. $x_j, j = 0, \ldots, d$ are the inputs and $z_h, h = 1, \ldots, H$ are the hidden units where H is the dimensionality of this hidden space. z_0 is the bias of the hidden layer. $y_i, i = 1, \ldots, K$ are the output units. w_{hj} are weights in the first layer, and v_{ih} are the weights in the second layer.

first layer of hidden units and implementing more complex functions of the inputs. In practice, people rarely go beyond one hidden layer since analyzing a network with many hidden layers is quite complicated; but sometimes when the hidden layer contains too many hidden units, it may be sensible to go to multiple hidden layers, preferring “long and narrow” networks to “short and fat” networks.

11.6 MLP as a Universal Approximator

We can represent any Boolean function as a disjunction of conjunctions, and such a Boolean expression can be implemented by a multilayer perceptron with one hidden layer. Each conjunction is implemented by one

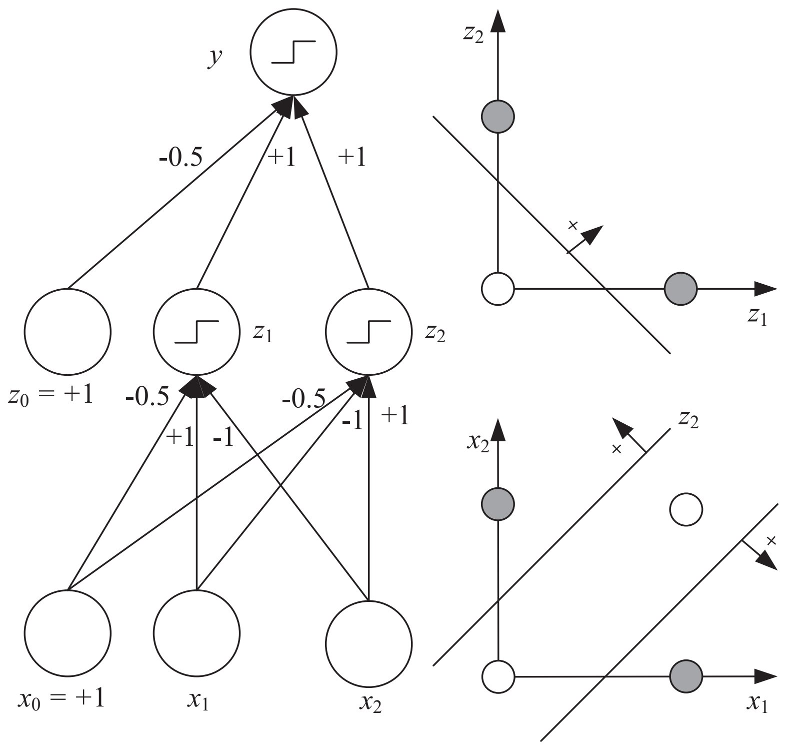


Figure 11.7 The multilayer perceptron that solves the XOR problem. The hidden units and the output have the threshold activation function with threshold at 0.

hidden unit and the disjunction by the output unit. For example,

x_1 XOR x_2 = (x_1 AND $\sim x_2$) OR ($\sim x_1$ AND x_2)

We have seen previously how to implement AND and OR using perceptrons. So two perceptrons can in parallel implement the two AND, and another perceptron on top can OR them together (see figure 11.7). We see that the first layer maps inputs from the (x_1, x_2) to the (z_1, z_2) space defined by the first-layer perceptrons. Note that both inputs, (0,0) and (1,1), are mapped to (0,0) in the (z_1, z_2) space, allowing linear separability in this second space.

Thus in the binary case, for every input combination where the output is 1, we define a hidden unit that checks for that particular conjunction of the input. The output layer then implements the disjunction. Note that this is just an existence proof, and such networks may not be practical as up to 2^d hidden units may be necessary when there are d inputs. Such an architecture implements table lookup and does not generalize.

UNIVERSAL APPROXIMATION

We can extend this to the case where inputs are continuous to show that similarly, any arbitrary function with continuous input and outputs can be approximated with a multilayer perceptron. The proof of *universal approximation* is easy with two hidden layers. For every input case or region, that region can be delimited by hyperplanes on all sides using hidden units on the first hidden layer. A hidden unit in the second layer then ANDs them together to bound the region. We then set the weight of the connection from that hidden unit to the output unit equal to the desired function value. This gives a *piecewise constant approximation* of the function; it corresponds to ignoring all the terms in the Taylor expansion except the constant term. Its accuracy may be increased to the desired value by increasing the number of hidden units and placing a finer grid on the input. Note that no formal bounds are given on the number of hidden units required. This property just reassures us that there is a solution; it does not help us in any other way. It has been proven that an MLP with *one* hidden layer (with an arbitrary number of hidden units) can learn any nonlinear function of the input (Hornik, Stinchcombe, and White 1989).

PIECEWISE CONSTANT APPROXIMATION

11.7 Backpropagation Algorithm

Training a multilayer perceptron is the same as training a perceptron; the only difference is that now the output is a nonlinear function of the input thanks to the nonlinear basis function in the hidden units. Considering the hidden units as inputs, the second layer is a perceptron and we already know how to update the parameters, v_{ij}, in this case, given the inputs z_h. For the first-layer weights, w_{hj}, we use the chain rule to calculate the gradient:

$$\frac{\partial E}{\partial w_{hj}} = \frac{\partial E}{\partial y_i}\frac{\partial y_i}{\partial z_h}\frac{\partial z_h}{\partial w_{hj}}$$

BACKPROPAGATION

It is as if the error propagates from the output y back to the inputs and hence the name *backpropagation* was coined (Rumelhart, Hinton, and Williams 1986a).

11.7.1 Nonlinear Regression

Let us first take the case of nonlinear regression (with a single output) calculated as

$$y^t = \sum_{h=1}^{H} v_h z_h^t + v_0 \tag{11.13}$$

with z_h computed by equation 11.11. The error function over the whole sample in regression is

$$E(\mathbf{W}, \boldsymbol{v}|\mathcal{X}) = \frac{1}{2}\sum_t (r^t - y^t)^2 \tag{11.14}$$

The second layer is a perceptron with hidden units as the inputs, and we use the least-squares rule to update the second-layer weights:

$$\Delta v_h = \eta \sum_t (r^t - y^t) z_h^t \tag{11.15}$$

The first layer also consists of perceptrons with the hidden units as the output units, but in updating the first-layer weights, we cannot use the least-squares rule directly because we do not have a desired output specified for the hidden units. This is where the chain rule comes into play. We write

$$\begin{aligned}
\Delta w_{hj} &= -\eta \frac{\partial E}{\partial w_{hj}} \\
&= -\eta \sum_t \frac{\partial E^t}{\partial y^t}\frac{\partial y^t}{\partial z_h^t}\frac{\partial z_h^t}{\partial w_{hj}} \\
&= -\eta \sum_t \underbrace{-(r^t - y^t)}_{\partial E^t/\partial y^t}\ \underbrace{v_h}_{\partial y^t/\partial z_h^t}\ \underbrace{z_h^t(1-z_h^t)x_j^t}_{\partial z_h^t/\partial w_{hj}} \\
&= \eta \sum_t (r^t - y^t) v_h z_h^t (1 - z_h^t) x_j^t
\end{aligned} \tag{11.16}$$

The product of the first two terms $(r^t - y^t)v_h$ acts like the error term for hidden unit h. This error is *backpropagated* from the error to the hidden unit. $(r^t - y^t)$ is the error in the output, weighted by the "responsibility" of the hidden unit as given by its weight v_h. In the third term, $z_h(1 - z_h)$ is the derivative of the sigmoid and x_j^t is the derivative of the weighted sum with respect to the weight w_{hj}. Note that the change in the first-layer weight, Δw_{hj}, makes use of the second-layer weight, v_h. Therefore, we should calculate the changes in both layers and update the first-layer weights, making use of the *old* value of the second-layer weights, then update the second-layer weights.

Weights, w_{hj}, v_h are started from small random values initially, for example, in the range $[-0.01, 0.01]$, so as not to saturate the sigmoids. It is also a good idea to normalize the inputs so that they all have 0 mean and unit variance and have the same scale, since we use a single η parameter.

BATCH LEARNING

With the learning equations given here, for each pattern, we compute the direction in which each parameter needs be changed and the magnitude of this change. In *batch learning*, we accumulate these changes over all patterns and make the change once after a complete pass over the whole training set is made, as shown in the previous update equations.

EPOCH

It is also possible to have online learning, by updating the weights after each pattern, thereby implementing stochastic gradient descent. A complete pass over all the patterns in the training set is called an *epoch*. The learning factor, η, should be chosen smaller in this case and patterns should be scanned in a random order. Online learning converges faster because there may be similar patterns in the dataset, and the stochasticity has an effect like adding noise and may help escape local minima.

An example of training a multilayer perceptron for regression is shown in figure 11.8. As training continues, the MLP fit gets closer to the underlying function and error decreases (see figure 11.9). Figure 11.10 shows how the MLP fit is formed as a sum of the outputs of the hidden units.

It is also possible to have multiple output units, in which case a number of regression problems are learned at the same time. We have

$$y_i^t = \sum_{h=1}^{H} v_{ih} z_h^t + v_{i0} \tag{11.17}$$

and the error is

$$E(\mathbf{W}, \mathbf{V}|\mathcal{X}) = \frac{1}{2} \sum_t \sum_i (r_i^t - y_i^t)^2 \tag{11.18}$$

The batch update rules are then

$$\Delta v_{ih} = \eta \sum_t (r_i^t - y_i^t) z_h^t \tag{11.19}$$

$$\Delta w_{hj} = \eta \sum_t \left[\sum_i (r_i^t - y_i^t) v_{ih} \right] z_h^t (1 - z_h^t) x_j^t \tag{11.20}$$

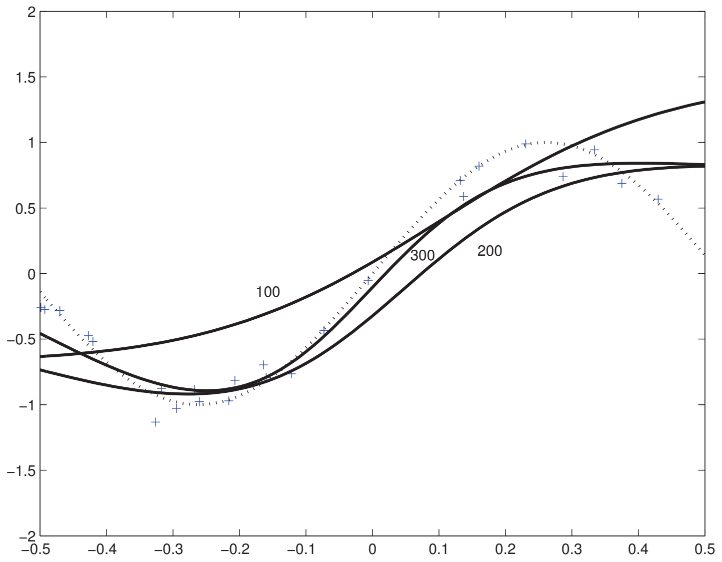


Figure 11.8 Sample training data shown as '+', where $x^t \sim U(-0.5, 0.5)$, and $y^t = f(x^t) + \mathcal{N}(0, 0.1)$. $f(x) = \sin(6x)$ is shown by a dashed line. The evolution of the fit of an MLP with two hidden units after 100, 200, and 300 epochs is drawn.

$\sum_i (r_i^t - y_i^t)v_{ih}$ is the accumulated backpropagated error of hidden unit h from all output units. Pseudocode is given in figure 11.11. Note that in this case, all output units share the same hidden units and thus use the same hidden representation, hence, we are assuming that corresponding to these different outputs, we have related prediction problems. An alternative is to train separate multilayer perceptrons for the separate regression problems, each with its own separate hidden units.

11.7.2 Two-Class Discrimination

When there are two classes, one output unit suffices:

$$y^t = \text{sigmoid}\left(\sum_{h=1}^{H} v_h z_h^t + v_0\right) \tag{11.21}$$

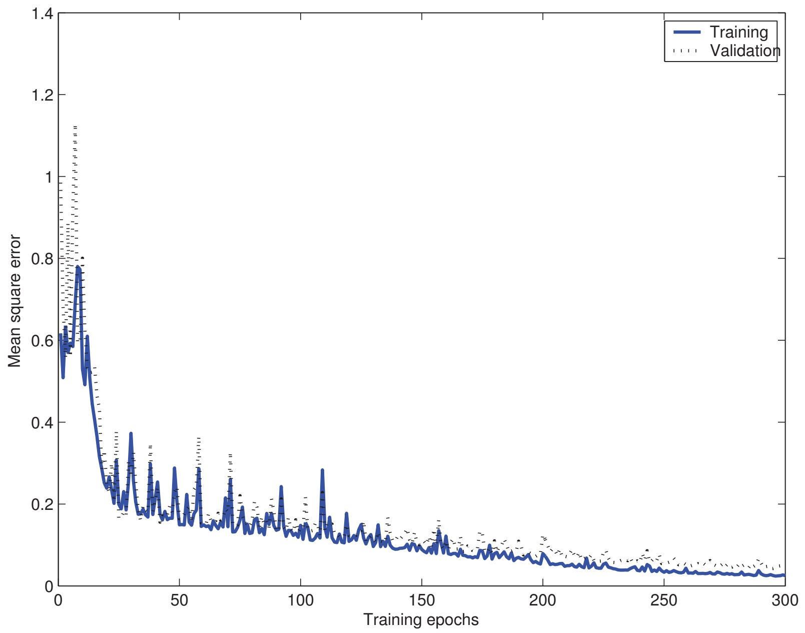


Figure 11.9 The mean square error on training and validation sets as a function of training epochs.

which approximates $P(C_1|\boldsymbol{x}^t)$ and $\hat{P}(C_2|\boldsymbol{x}^t) \equiv 1 - y^t$. We remember from section 10.7 that the error function in this case is

$$E(\mathbf{W}, \boldsymbol{v}|\mathcal{X}) = -\sum_t r^t \log y^t + (1 - r^t)\log(1 - y^t) \tag{11.22}$$

The update equations implementing gradient descent are

$$\Delta v_h = \eta \sum_t (r^t - y^t) z_h^t \tag{11.23}$$

$$\Delta w_{hj} = \eta \sum_t (r^t - y^t) v_h z_h^t (1 - z_h^t) x_j^t \tag{11.24}$$

As in the simple perceptron, the update equations for regression and classification are identical (which does not mean that the values are).

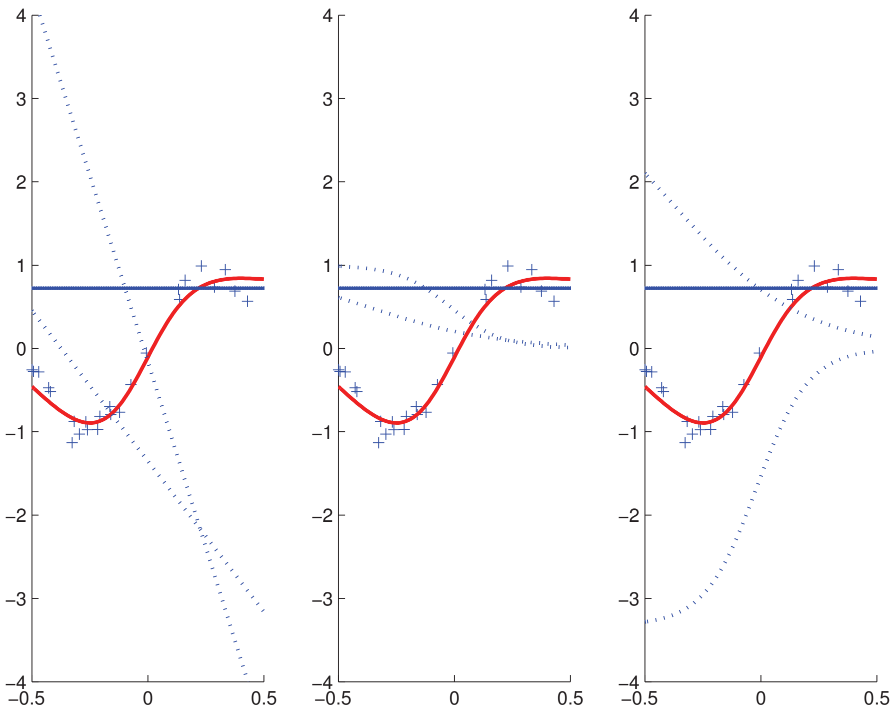


Figure 11.10 (a) The hyperplanes of the hidden unit weights on the first layer, (b) hidden unit outputs, and (c) hidden unit outputs multiplied by the weights on the second layer. Two sigmoid hidden units slightly displaced, one multiplied by a negative weight, when added, implement a bump. With more hidden units, a better approximation is attained (see figure 11.12).

11.7.3 Multiclass Discrimination

In a $(K > 2)$-class classification problem, there are K outputs

$$o_i^t = \sum_{h=1}^{H} v_{ih} z_h^t + v_{i0} \tag{11.25}$$

and we use softmax to indicate the dependency between classes; namely, they are mutually exclusive and exhaustive:

$$y_i^t = \frac{\exp o_i^t}{\sum_k \exp o_k^t} \tag{11.26}$$

```
Initialize all $v_{ih}$ and $w_{hj}$ to rand$(-0.01, 0.01)$
Repeat
    For all $(\boldsymbol{x}^t, r^t) \in \mathcal{X}$ in random order
        For $h = 1, \ldots, H$
            $z_h \leftarrow \text{sigmoid}(\boldsymbol{w}_h^T \boldsymbol{x}^t)$
        For $i = 1, \ldots, K$
            $y_i = \boldsymbol{v}_i^T \boldsymbol{z}$
        For $i = 1, \ldots, K$
            $\Delta \boldsymbol{v}_i = \eta(r_i^t - y_i^t)\boldsymbol{z}$
        For $h = 1, \ldots, H$
            $\Delta \boldsymbol{w}_h = \eta(\sum_i (r_i^t - y_i^t) v_{ih}) z_h (1 - z_h) \boldsymbol{x}^t$
        For $i = 1, \ldots, K$
            $\boldsymbol{v}_i \leftarrow \boldsymbol{v}_i + \Delta \boldsymbol{v}_i$
        For $h = 1, \ldots, H$
            $\boldsymbol{w}_h \leftarrow \boldsymbol{w}_h + \Delta \boldsymbol{w}_h$
Until convergence
```

Figure 11.11 Backpropagation algorithm for training a multilayer perceptron for regression with K outputs. This code can easily be adapted for two-class classification (by setting a single sigmoid output) and to $K > 2$ classification (by using softmax outputs).

where y_i approximates $P(C_i|\boldsymbol{x}^t)$. The error function is

$$E(\mathbf{W}, \mathbf{V}|\mathcal{X}) = -\sum_t \sum_i r_i^t \log y_i^t \tag{11.27}$$

and we get the update equations using gradient descent:

$$\Delta v_{ih} = \eta \sum_t (r_i^t - y_i^t) z_h^t \tag{11.28}$$

$$\Delta w_{hj} = \eta \sum_t \left[\sum_i (r_i^t - y_i^t) v_{ih} \right] z_h^t (1 - z_h^t) x_j^t \tag{11.29}$$

Richard and Lippmann (1991) have shown that given a network of enough complexity and sufficient training data, a suitably trained multilayer perceptron estimates posterior probabilities.

11.7.4 Multiple Hidden Layers

As we saw before, it is possible to have multiple hidden layers each with its own weights and applying the sigmoid function to its weighted sum. For regression, let us say, if we have a multilayer perceptron with two hidden layers, we write

$$\begin{aligned}
z_{1h} &= \text{sigmoid}(\boldsymbol{w}_{1h}^T \boldsymbol{x}) = \text{sigmoid}\left(\sum_{j=1}^{d} w_{1hj} x_j + w_{1h0}\right), \ h = 1, \dots, H_1 \\
z_{2l} &= \text{sigmoid}(\boldsymbol{w}_{2l}^T \boldsymbol{z}_1) = \text{sigmoid}\left(\sum_{h=0}^{H_1} w_{2lh} z_{1h} + w_{2l0}\right), \ l = 1, \dots, H_2 \\
y &= \boldsymbol{v}^T \boldsymbol{z}_2 = \sum_{l=1}^{H_2} v_l z_{2l} + v_0
\end{aligned}$$

where $\boldsymbol{w}_{1h}$ and $\boldsymbol{w}_{2l}$ are the first- and second-layer weights, z_{1h} and z_{2h} are the units on the first and second hidden layers, and $\boldsymbol{v}$ are the third-layer weights. Training such a network is similar except that to train the first-layer weights, we need to backpropagate one more layer (exercise 5).

11.8 Training Procedures

11.8.1 Improving Convergence

Gradient descent has various advantages. It is simple. It is local; namely, the change in a weight uses only the values of the presynaptic and postsynaptic units and the error (suitably backpropagated). When online training is used, it does not need to store the training set and can adapt as the task to be learned changes. Because of these reasons, it can be (and is) implemented in hardware. But by itself, gradient descent converges slowly. When learning time is important, one can use more sophisticated optimization methods (Battiti 1992). Bishop (1995) discusses in detail the application of conjugate gradient and second-order methods to the training of multilayer perceptrons. However, there are two frequently used simple techniques that improve the performance of the gradient descent considerably, making gradient-based methods feasible in real applications.

Momentum

Let us say w_i is any weight in a multilayer perceptron in any layer, including the biases. At each parameter update, successive Δw_i^t values may be so different that large oscillations may occur and slow convergence. t is the time index that is the epoch number in batch learning and the iteration number in online learning. The idea is to take a running average by incorporating the previous update in the current change as if there is a MOMENTUM *momentum* due to previous updates:

$$\Delta w_i^t = -\eta \frac{\partial E^t}{\partial w_i} + \alpha \Delta w_i^{t-1} \tag{11.30}$$

α is generally taken between 0.5 and 1.0. This approach is especially useful when online learning is used, where as a result we get an effect of averaging and smooth the trajectory during convergence. The disadvantage is that the past Δw_i^{t-1} values should be stored in extra memory.

Adaptive Learning Rate

In gradient descent, the learning factor η determines the magnitude of change to be made in the parameter. It is generally taken between 0.0 and 1.0, mostly less than or equal to 0.2. It can be made adaptive for faster convergence, where it is kept large when learning takes place and is decreased when learning slows down:

$$\Delta \eta = \begin{cases} +a & \text{if } E^{t+\tau} < E^t \\ -b\eta & \text{otherwise} \end{cases} \tag{11.31}$$

Thus we increase η by a constant amount if the error on the training set decreases and decrease it geometrically if it increases. Because E may oscillate from one epoch to another, it is a better idea to take the average of the past few epochs as E^t.

11.8.2 Overtraining

A multilayer perceptron with d inputs, H hidden units, and K outputs has $H(d+1)$ weights in the first layer and $K(H+1)$ weights in the second layer. Both the space and time complexity of an MLP is $\mathcal{O}(H \cdot (K+d))$. When e denotes the number of training epochs, training time complexity is $\mathcal{O}(e \cdot H \cdot (K+d))$.

In an application, d and K are predefined and H is the parameter that we play with to tune the complexity of the model. We know from previous chapters that an overcomplex model memorizes the noise in the training set and does not generalize to the validation set. For example, we have previously seen this phenomenon in the case of polynomial regression where we noticed that in the presence of noise or small samples, increasing the polynomial order leads to worse generalization. Similarly in an MLP, when the number of hidden units is large, the generalization accuracy deteriorates (see figure 11.12), and the bias/variance dilemma also holds for the MLP, as it does for any statistical estimator (Geman, Bienenstock, and Doursat 1992).

A similar behavior happens when training is continued too long: As more training epochs are made, the error on the training set decreases, but the error on the validation set starts to increase beyond a certain point (see figure 11.13). Remember that initially all the weights are close to 0 and thus have little effect. As training continues, the most important weights start moving away from 0 and are utilized. But if training is continued further on to get less and less error on the training set, almost all weights are updated away from 0 and effectively become parameters. Thus as training continues, it is as if new parameters are added to the system, increasing the complexity and leading to poor generalization. Learning should be *stopped early* to alleviate this problem of *overtraining*. The optimal point to stop training, and the optimal number of hidden units, is determined through cross-validation, which involves testing the network's performance on validation data unseen during training.

EARLY STOPPING
OVERTRAINING

Because of the nonlinearity, the error function has many minima and gradient descent converges to the nearest minimum. To be able to assess expected error, the same network is trained a number of times starting from different initial weight values, and the average of the validation error is computed.

11.8.3 Structuring the Network

In some applications, we may believe that the input has a local structure. For example, in vision we know that nearby pixels are correlated and there are local features like edges and corners; any object, for example, a handwritten digit, may be defined as a combination of such primitives. Similarly, in speech, locality is in time and inputs close in time can be grouped as speech primitives. By combining these primitives, longer ut-

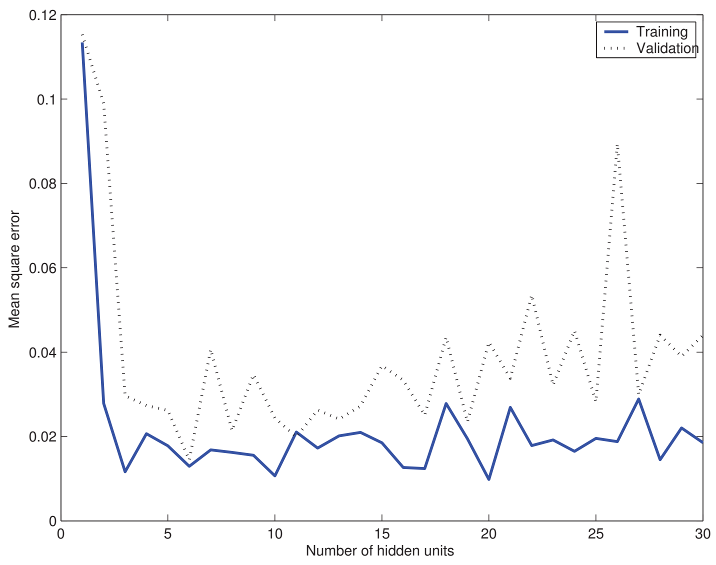


Figure 11.12 As complexity increases, training error is fixed but the validation error starts to increase and the network starts to overfit.

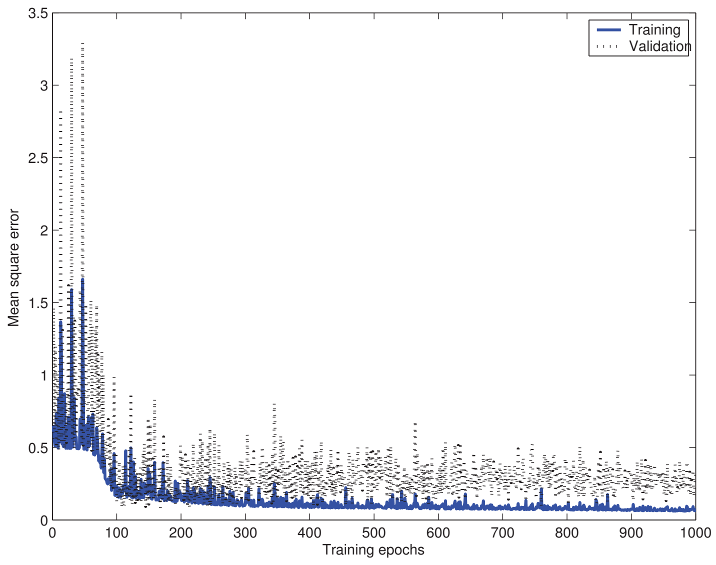


Figure 11.13 As training continues, the validation error starts to increase and the network starts to overfit.

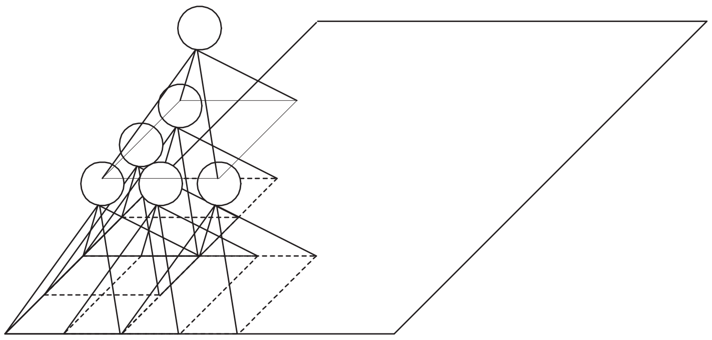


Figure 11.14 A structured MLP. Each unit is connected to a local group of units below it and checks for a particular feature—for example, edge, corner, and so forth—in vision. Only one hidden unit is shown for each region; typically there are many to check for different local features.

terances, for example, speech phonemes, may be defined. In such a case when designing the MLP, hidden units are not connected to all input units because not all inputs are correlated. Instead, we define hidden units that define a window over the input space and are connected to only a small local subset of the inputs. This decreases the number of connections and therefore the number of free parameters (Le Cun et al. 1989).

We can repeat this in successive layers where each layer is connected to a small number of local units below and checks for a more complicated feature by combining the features below in a larger part of the input space until we get to the output layer (see figure 11.14). For example, the input may be pixels. By looking at pixels, the first hidden layer units may learn to check for edges of various orientations. Then by combining edges, the second hidden layer units can learn to check for combinations of edges—for example, arcs, corners, line ends—and then combining them in upper layers, the units can look for semi-circles, rectangles, or in the case of a face recognition application, eyes, mouth, and so forth. This is the example of a *hierarchical cone* where features get more complex, abstract, and fewer in number as we go up the network until we get to classes. Such an architecture is called a *convolutional neural network* where the work of each hidden unit is considered to be a convolution of its input with its weight vector; an earlier similar architecture is the *neocognitron* (Fukushima 1980).

HIERARCHICAL CONE

CONVOLUTIONAL NEURAL NETWORK

NEOCOGNITRON

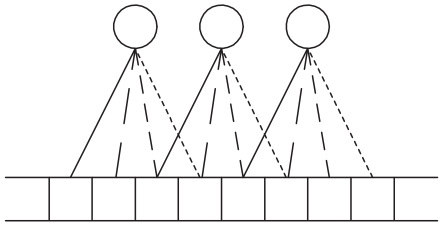


Figure 11.15 In weight sharing, different units have connections to different inputs but share the same weight value (denoted by line type). Only one set of units is shown; there should be multiple sets of units, each checking for different features.

WEIGHT SHARING

In such a case, we can further reduce the number of parameters by *weight sharing*. Taking the example of visual recognition again, we can see that when we look for features like oriented edges, they may be present in different parts of the input space. So instead of defining independent hidden units learning different features in different parts of the input space, we can have copies of the same hidden units looking at different parts of the input space (see figure 11.15). During learning, we calculate the gradients by taking different inputs, then we average these up and make a single update. This implies a single parameter that defines the weight on multiple connections. Also, because the update on a weight is based on gradients for several inputs, it is as if the training set is effectively multiplied.

11.8.4 Hints

The knowledge of local structure allows us to prestructure the multilayer network, and with weight sharing it has fewer parameters. The alternative of an MLP with completely connected layers has no such structure and is more difficult to train. Knowledge of any sort related to the application should be built into the network structure whenever possible.

HINTS

These are called *hints* (Abu-Mostafa 1995) and are the properties of the target function that are known to us independent of the training examples.

In image recognition, there are invariance hints: The identity of an object does not change when it is rotated, translated, or scaled (see figure 11.16). Hints are auxiliary information that can be used to guide the

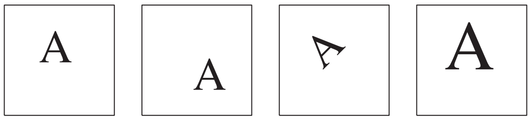


Figure 11.16 The identity of the object does not change when it is translated, rotated, or scaled. Note that this may not always be true, or may be true up to a point: 'b' and 'q' are rotated versions of each other. These are hints that can be incorporated into the learning process to make learning easier.

learning process and are especially useful when the training set is limited. There are different ways in which hints can be used:

VIRTUAL EXAMPLES

1. Hints can be used to create *virtual examples*. For example, knowing that the object is invariant to scale, from a given training example, we can generate multiple copies at different scales and add them to the training set with the same label. This has the advantage that we increase the training set and do not need to modify the learner in any way. The problem may be that too many examples may be needed for the learner to learn the invariance.

2. The invariance may be implemented as a preprocessing stage. For example, optical character readers have a preprocessing stage where the input character image is centered and normalized for size and slant. This is the easiest solution, when it is possible.

3. The hint may be incorporated into the network structure. Local structure and weight sharing, which we saw in section 11.8.3, is one example where we get invariance to small translations and rotations.

4. The hint may also be incorporated by modifying the error function. Let us say we know that $\boldsymbol{x}$ and $\boldsymbol{x}'$ are the same from the application's point of view, where $\boldsymbol{x}'$ may be a "virtual example" of $\boldsymbol{x}$. That is, $f(\boldsymbol{x}) = f(\boldsymbol{x}')$, when $f(\boldsymbol{x})$ is the function we would like to approximate. Let us denote by $g(\boldsymbol{x}|\theta)$, our approximation function, for example, an MLP where θ are its weights. Then, for all such pairs $(\boldsymbol{x}, \boldsymbol{x}')$, we define the penalty function

$$E_h = \left[g(\boldsymbol{x}|\theta) - g(\boldsymbol{x}'|\theta)\right]^2$$

and add it as an extra term to the usual error function:

$$E' = E + \lambda_h \cdot E_h$$

This is a penalty term penalizing the cases where our predictions do not obey the hint, and λ_h is the weight of such a penalty (Abu-Mostafa 1995).

Another example is the approximation hint: Let us say that for x, we do not know the exact value, $f(x)$, but we know that it is in the interval, $[a_x, b_x]$. Then our added penalty term is

$$E_h = \begin{cases} 0 & \text{if } g(x|\theta) \in [a_x, b_x] \\ (g(x) - a_x)^2 & \text{if } g(x|\theta) < a_x \\ (g(x) - b_x)^2 & \text{if } g(x|\theta) > b_x \end{cases}$$

This is similar to the error function used in support vector regression (section 13.10), which tolerates small approximation errors.

TANGENT PROP

Still another example is the *tangent prop* (Simard et al. 1992) where the transformation against which we are defining the hint—for example, rotation by an angle—is modeled by a function. The usual error function is modified (by adding another term) so as to allow parameters to move along this line of transformation without changing the error.

11.9 Tuning the Network Size

Previously we saw that when the network is too large and has too many free parameters, generalization may not be well. To find the optimal network size, the most common approach is to try many different architectures, train them all on the training set, and choose the one that generalizes best to the validation set. Another approach is to incorporate this *structural adaptation* into the learning algorithm. There are two ways this can be done:

STRUCTURAL ADAPTATION

1. In the *destructive* approach, we start with a large network and gradually remove units and/or connections that are not necessary.

2. In the *constructive* approach, we start with a small network and gradually add units and/or connections to improve performance.

WEIGHT DECAY

One destructive method is *weight decay* where the idea is to remove unnecessary connections. Ideally to be able to determine whether a unit or connection is necessary, we need to train once with and once without and check the difference in error on a separate validation set. This is costly since it should be done for all combinations of such units/connections.

Given that a connection is not used if its weight is 0, we give each connection a tendency to decay to 0 so that it disappears unless it is reinforced explicitly to decrease error. For any weight w_i in the network, we use the update rule:

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i} - \lambda w_i \tag{11.32}$$

This is equivalent to doing gradient descent on the error function with an added penalty term, penalizing networks with many nonzero weights:

$$E' = E + \frac{\lambda}{2} \sum_i w_i^2 \tag{11.33}$$

Simpler networks are better generalizers is a hint that we implement by adding a penalty term. Note that we are not saying that simple networks are always better than large networks; we are saying that if we have two networks that have the same training error, the simpler one—namely, the one with fewer weights—has a higher probability of better generalizing to the validation set.

The effect of the second term in equation 11.32 is like that of a spring that pulls each weight to 0. Starting from a value close to 0, unless the actual error gradient is large and causes an update, due to the second term, the weight will gradually decay to 0. λ is the parameter that determines the relative importances of the error on the training set and the complexity due to nonzero parameters and thus determines the speed of decay: With large λ, weights will be pulled to 0 no matter what the training error is; with small λ, there is not much penalty for nonzero weights. λ is fine-tuned using cross-validation.

DYNAMIC NODE CREATION

Instead of starting from a large network and *pruning* unnecessary connections or units, one can start from a small network and add units and associated connections should the need arise (figure 11.17). In *dynamic node creation* (Ash 1989), an MLP with one hidden layer with one hidden unit is trained and after convergence, if the error is still high, another hidden unit is added. The incoming weights of the newly added unit and its outgoing weight are initialized randomly and trained with the previ-

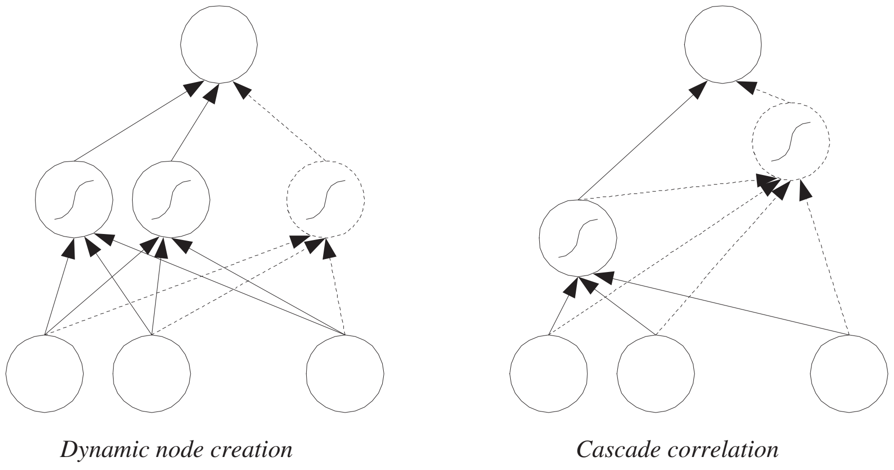


Figure 11.17 Two examples of constructive algorithms. Dynamic node creation adds a unit to an existing layer. Cascade correlation adds each unit as a new hidden layer connected to all the previous layers. Dashed lines denote the newly added unit/connections. Bias units/weights are omitted for clarity.

ously existing weights that are not reinitialized and continue from their previous values.

CASCADE CORRELATION

In *cascade correlation* (Fahlman and Lebiere 1990), each added unit is a new hidden unit in another hidden layer. Every hidden layer has only one unit that is connected to all of the hidden units preceding it and the inputs. The previously existing weights are frozen and are not trained; only the incoming and outgoing weights of the newly added unit are trained.

Dynamic node creation adds a new hidden unit to an existing hidden layer and never adds another hidden layer. Cascade correlation always adds a new hidden layer with a single unit. The ideal constructive method should be able to decide when to introduce a new hidden layer and when to add a unit to an existing layer. This is an open research problem.

Incremental algorithms are interesting because they correspond to modifying not only the parameters but also the model structure during learning. We can think of a space defined by the structure of the multilayer perceptron and operators corresponding to adding/removing unit(s) or layer(s) to move in this space (Aran et al. 2009). Incremental algorithms then do a search in this state space where operators are tried (according to some order) and accepted or rejected depending on some goodness measure, for example, some combination of complexity and validation error. Another example would be a setting in polynomial regression where high-order terms are added/removed during training automatically, fitting model complexity to data complexity. As the cost of computation gets lower, such automatic model selection should be a part of the learning process done automatically without any user interference.

11.10 Bayesian View of Learning

The Bayesian approach in training neural networks considers the parameters, namely, connection weights, w_i, as random variables drawn from a prior distribution $p(w_i)$ and computes the posterior probability given the data

$$p(\boldsymbol{w}|\mathcal{X}) = \frac{p(\mathcal{X}|\boldsymbol{w})p(\boldsymbol{w})}{p(\mathcal{X})} \tag{11.34}$$

where $\boldsymbol{w}$ is the vector of all weights of the network. The MAP estimate $\hat{\boldsymbol{w}}$ is the mode of the posterior

$$\hat{\boldsymbol{w}}_{MAP} = \arg\max_{\boldsymbol{w}} \log p(\boldsymbol{w}|\mathcal{X}) \tag{11.35}$$

Taking the log of equation 11.34, we get

$$\log p(\boldsymbol{w}|\mathcal{X}) = \log p(\mathcal{X}|\boldsymbol{w}) + \log p(\boldsymbol{w}) + C$$

The first term on the right is the log likelihood, and the second is the log of the prior. If the weights are independent and the prior is taken as Gaussian, $\mathcal{N}(0, 1/2\lambda)$

$$p(\boldsymbol{w}) = \prod_i p(w_i) \text{ where } p(w_i) = c \cdot \exp\left[-\frac{w_i^2}{2(1/2\lambda)}\right] \tag{11.36}$$

the MAP estimate minimizes the augmented error function

$$E' = E + \lambda\|\boldsymbol{w}\|^2 \tag{11.37}$$

where E is the usual classification or regression error (negative log likelihood). This augmented error is exactly the error function we used in weight decay (equation 11.33). Using a large λ assumes small variability in parameters, puts a larger force on them to be close to 0, and takes into account the prior more than the data; if λ is small, then the allowed variability of the parameters is larger. This approach of removing unnecessary parameters is known as *ridge regression* in statistics.

RIDGE REGRESSION

REGULARIZATION

This is another example of *regularization* with a cost function, combining the fit to data and model complexity

$$\text{cost} = \text{data-misfit} + \lambda \cdot \text{complexity} \tag{11.38}$$

The use of Bayesian estimation in training multilayer perceptrons is treated in MacKay 1992a, b. We are going to talk about Bayesian estimation in more detail in chapter 16.

Empirically, it has been seen that after training, most of the weights of a multilayer perceptron are distributed normally around 0, justifying the use of weight decay. But this may not always be the case. Nowlan and Hinton (1992) proposed *soft weight sharing* where weights are drawn from a mixture of Gaussians, allowing them to form multiple clusters, not one. Also, these clusters may be centered anywhere and not necessarily at 0, and have variances that are modifiable. This changes the prior of equation 11.36 to a mixture of $M \geq 2$ Gaussians

SOFT WEIGHT SHARING

$$p(w_i) = \sum_{j=1}^{M} \alpha_j p_j(w_i) \tag{11.39}$$

where α_j are the priors and $p_j(w_i) \sim \mathcal{N}(m_j, s_j^2)$ are the component Gaussians. M is set by the user and α_j, m_j, s_j are learned from the data. Using such a prior and augmenting the error function with its log during training, the weights converge to decrease error and also are grouped automatically to increase the log prior.

11.11 Dimensionality Reduction

In a multilayer perceptron, if the number of hidden units is less than the number of inputs, the first layer performs a dimensionality reduction. The form of this reduction and the new space spanned by the hidden units depend on what the MLP is trained for. If the MLP is for classification with output units following the hidden layer, then the new space is defined and the mapping is learned to minimize classification error (see figure 11.18).

We can get an idea of what the MLP is doing by analyzing the weights. We know that the dot product is maximum when the two vectors are

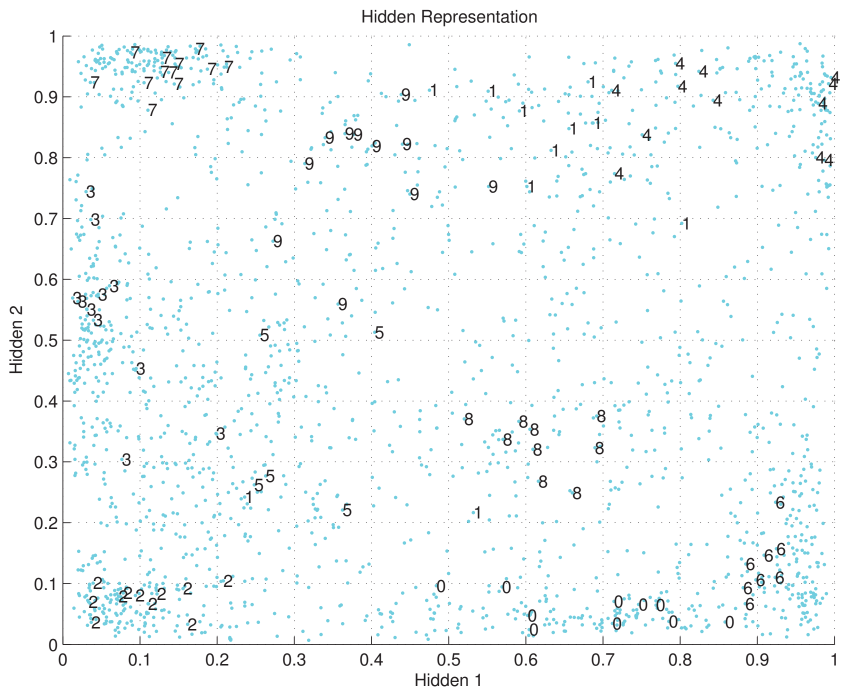


Figure 11.18 Optdigits data plotted in the space of the two hidden units of an MLP trained for classification. Only the labels of one hundred data points are shown. This MLP with sixty-four inputs, two hidden units, and ten outputs has 80 percent accuracy. Because of the sigmoid, hidden unit values are between 0 and 1 and classes are clustered around the corners. This plot can be compared with the plots in chapter 6, which are drawn using other dimensionality reduction methods on the same dataset.

identical. So we can think of each hidden unit as defining a template in its incoming weights, and by analyzing these templates, we can extract knowledge from a trained MLP. If the inputs are normalized, weights tell us of their relative importance. Such analysis is not easy but gives us some insight as to what the MLP is doing and allows us to peek into the black box.

AUTOENCODER

An interesting architecture is the *autoencoder* (Cottrell, Munro, and Zipser 1987), which is an MLP architecture where there are as many outputs as there are inputs, and the required outputs are defined to be equal

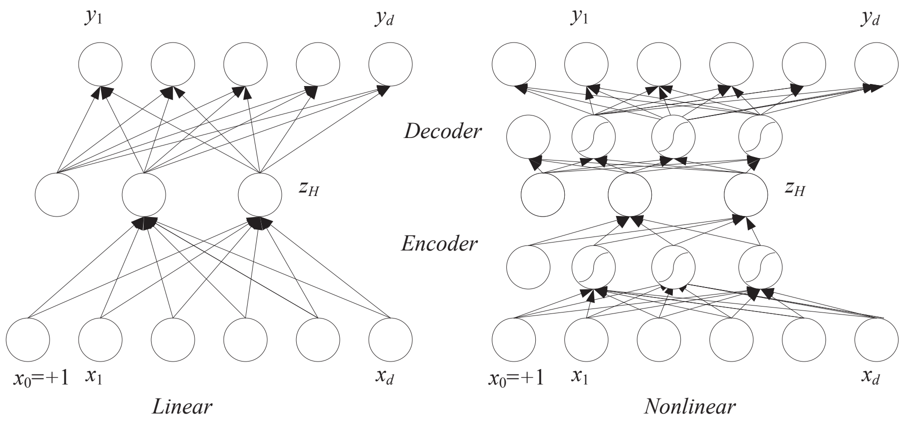


Figure 11.19 In the autoencoder, there are as many outputs as there are inputs and the desired outputs are the inputs. When the number of hidden units is less than the number of inputs, the MLP is trained to find the best coding of the inputs on the hidden units, performing dimensionality reduction. On the left, the first layer acts as an encoder and the second layer acts as the decoder. On the right, if the encoder and decoder are multilayer perceptrons with sigmoid hidden units, the network performs nonlinear dimensionality reduction.

to the inputs (see figure 11.19). To be able to reproduce the inputs again at the output layer, the MLP is forced to find the best representation of the inputs in the hidden layer. When the number of hidden units is less than the number of inputs, this implies dimensionality reduction. Once the training is done, the first layer from the input to the hidden layer acts as an encoder, and the values of the hidden units make up the encoded representation. The second layer from the hidden units to the output units acts as a decoder, reconstructing the original signal from its encoded representation.

It has been shown (Bourlard and Kamp 1988) that an autoencoder MLP with one hidden layer of units implements principal components analysis (section 6.3), except that the hidden unit weights are not the eigenvectors sorted in importance using the eigenvalues but span the same space as the H principal eigenvectors. If the encoder and decoder are not one layer but multilayer perceptrons with sigmoid nonlinearity in the hidden units, the encoder implements nonlinear dimensionality reduction. In section 11.13, we discuss “deep” networks composed of multiple nonlinear hidden layers.

Another way to use an MLP for dimensionality reduction is through multidimensional scaling (section 6.7). Mao and Jain (1995) show how an MLP can be used to learn the *Sammon mapping*. Recalling equation 6.37, Sammon stress is defined as

SAMMON MAPPING

$$E(\theta|\mathcal{X}) = \sum_{r,s} \left[\frac{\|\boldsymbol{g}(\boldsymbol{x}^r|\theta) - \boldsymbol{g}(\boldsymbol{x}^s|\theta)\| - \|\boldsymbol{x}^r - \boldsymbol{x}^s\|}{\|\boldsymbol{x}^r - \boldsymbol{x}^s\|} \right]^2 \tag{11.40}$$

An MLP with d inputs, H hidden units, and $k < d$ output units is used to implement $g(\boldsymbol{x}|\theta)$, mapping the d-dimensional input to a k-dimensional vector, where θ corresponds to the weights of the MLP. Given a dataset of $\mathcal{X} = \{\boldsymbol{x}^t\}_t$, we can use gradient descent to minimize the Sammon stress directly to learn the MLP, namely, $g(\boldsymbol{x}|\theta)$, such that the distances between the k-dimensional representations are as close as possible to the distances in the original space.

11.12 Learning Time

Until now, we have been concerned with cases where the input is fed once, all together. In some applications, the input is temporal where we need to learn a temporal sequence. In others, the output may also change in time. Examples are as follows:

- *Sequence recognition.* This is the assignment of a given sequence to one of several classes. Speech recognition is one example where the input signal sequence is the spoken speech and the output is the code of the word spoken. That is, the input changes in time but the output does not.
- *Sequence reproduction.* Here, after seeing part of a given sequence, the system should predict the rest. Time-series prediction is one example where the input is given but the output changes.
- *Temporal association.* This is the most general case where a particular output sequence is given as output after a specific input sequence. The input and output sequences may be different. Here both the input and the output change in time.

11.12.1 Time Delay Neural Networks

The easiest way to recognize a temporal sequence is by converting it to a spatial sequence. Then any method discussed up to this point can be uti-

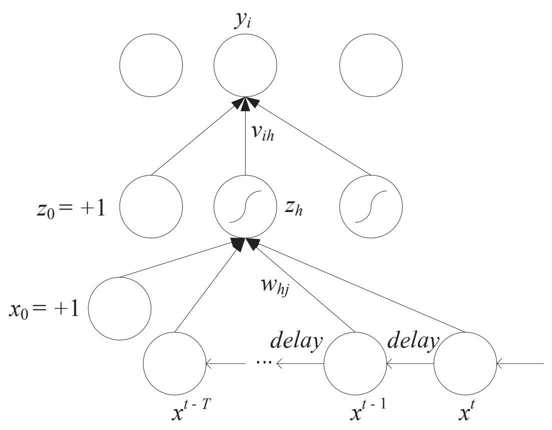


Figure 11.20 A time delay neural network. Inputs in a time window of length T are delayed in time until we can feed all T inputs as the input vector to the MLP.

TIME DELAY NEURAL NETWORK

lized for classification. In a *time delay neural network* (Waibel et al. 1989), previous inputs are delayed in time so as to synchronize with the final input, and all are fed together as input to the system (see figure 11.20). Backpropagation can then be used to train the weights. To extract features local in time, one can have layers of structured connections and weight sharing to get translation invariance in time. The main restriction of this architecture is that the size of the time window we slide over the sequence should be fixed a priori.

11.12.2 Recurrent Networks

RECURRENT NETWORK

In a *recurrent network*, additional to the feedforward connections, units have self-connections or connections to units in the previous layers. This recurrency acts as a short-term memory and lets the network remember what happened in the past.

Most frequently, one uses a partially recurrent network where a limited number of recurrent connections are added to a multilayer perceptron (see figure 11.21). This combines the advantage of the nonlinear approximation ability of a multilayer perceptron with the temporal representation ability of the recurrency, and such a network can be used to implement any of the three temporal association tasks. It is also possible to have hidden units in the recurrent backward connections, these being

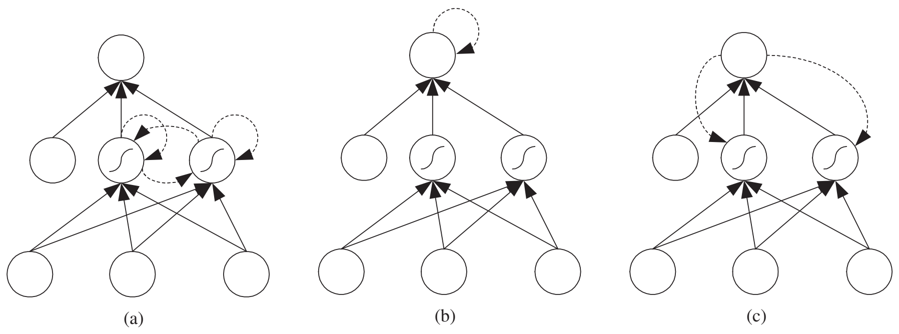


Figure 11.21 Examples of MLP with partial recurrency. Recurrent connections are shown with dashed lines: (a) self-connections in the hidden layer, (b) self-connections in the output layer, and (c) connections from the output to the hidden layer. Combinations of these are also possible.

known as *context units*. No formal results are known to determine how to choose the best architecture given a particular application.

UNFOLDING IN TIME

If the sequences have a small maximum length, then *unfolding in time* can be used to convert an arbitrary recurrent network to an equivalent feedforward network (see figure 11.22). A separate unit and connection is created for copies at different times. The resulting network can be trained with backpropagation with the additional requirement that all copies of each connection should remain identical. The solution, as in weight sharing, is to sum up the different weight changes in time and change the weight by the average. This is called *backpropagation through time* (Rumelhart, Hinton, and Willams 1986b). The problem with this approach is the memory requirement if the length of the sequence is large. *Real time recurrent learning* (Williams and Zipser 1989) is an algorithm for training recurrent networks without unfolding and has the advantage that it can use sequences of arbitrary length.

BACKPROPAGATION THROUGH TIME

REAL TIME RECURRENT LEARNING

11.13 Deep Learning

When a linear model is not sufficient, one possibility is to define new features that are nonlinear functions of the input, for example, higher-order terms, and then build a linear model in the space of those features; we discussed this in section 10.2. This requires us to know what such

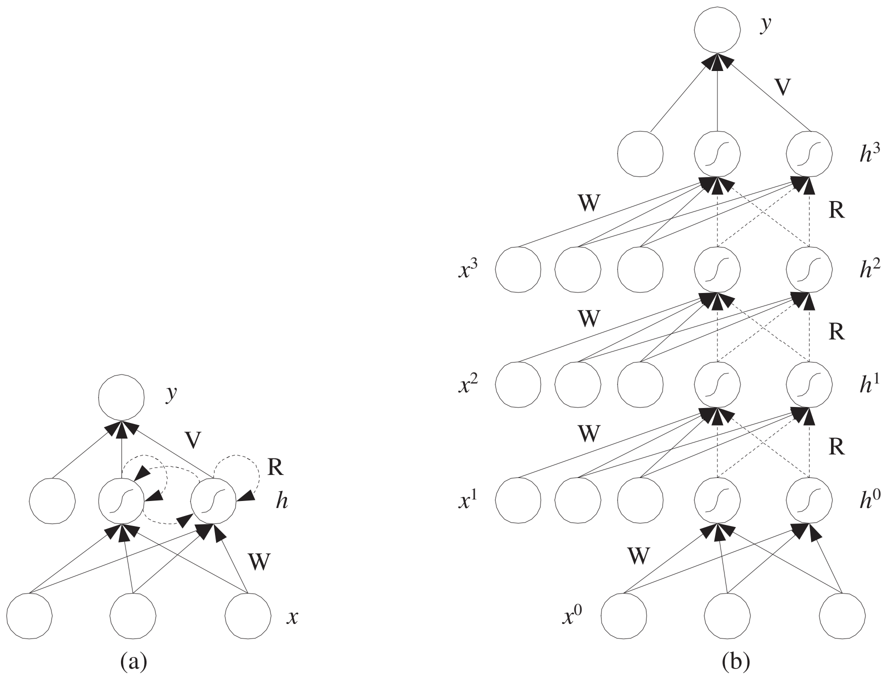


Figure 11.22 Backpropagation through time: (a) recurrent network, and (b) its equivalent unfolded network that behaves identically in four steps.

good basis functions are. Another possibility is to use one of the feature extraction methods we discussed in chapter 6 to learn the new space, for example, PCA or Isomap; such methods have the advantage that they are trained on data. Still, the best approach seems to be to use an MLP that extracts such features in its hidden layer; the MLP has the advantage that the first layer (feature extraction) and the second layer (how those features are combined to predict the output) are learned together in a coupled and supervised manner.

An MLP with one hidden layer has limited capacity, and using an MLP with multiple hidden layers can learn more complicated functions of the input. That is the idea behind *deep neural networks* where, starting from raw input, each hidden layer combines the values in its preceding layer and learns more complicated functions of the input.

DEEP NEURAL NETWORKS

Another aspect of deep networks is that successive hidden layers correspond to more abstract representations until we get to the output layer where the outputs are learned in terms of these most abstract concepts.

We saw an example of this in the convolutional neural networks (section 11.8.3) where starting from pixels, we get to edges, and then to corners, and so on, until we get to a digit. But user knowledge is necessary to define the connectivity and the overall architecture. Consider a face recognizer MLP where inputs are the image pixels and each hidden unit is connected to all the inputs; in such a case, the network has no knowledge that the inputs are face images, or even that the input is two-dimensional; the input is just a vector of values. Using a convolutional network where hidden units are fed with localized two-dimensional patches is a way to feed this information such that correct abstractions can be learned.

DEEP LEARNING

In *deep learning*, the idea is to learn feature levels of increasing abstraction with minimum human contribution (Bengio 2009). This is because in most applications, we do not know what structure there is in the input, and any sort of dependencies that are, for example, locality, should be automatically discovered during training. It is this extraction of dependencies, or patterns, or regularities, that allows abstraction and learning general descriptions.

One major problem with training an MLP with multiple hidden layers is that in backpropagating the error to an early layer, we need to multiply the derivatives in all the layers afterward and the gradient vanishes. This is also why unfolded recurrent neural networks (section 11.12.2) learn very slowly. This does not happen in convolutional neural networks because the fan-in and fan-out of hidden units are typically small.

A deep neural network is typically trained one layer at a time (Hinton and Salakhutdinov 2006). The aim of each layer is to extract the salient features in the data that is fed to it, and a method such as the autoencoder that we discussed in section 11.11 can be used for this purpose; there is the extra advantage that we can use unlabeled data for this purpose. So starting from the raw input, we train an autoencoder, and the encoded representation learned in its hidden layer is then used as input to train the next autoencoder and so on, until we get to the final layer that is trained in a supervised manner with the labeled data. Once all the layers are trained in this way one by one, they are all assembled and the whole network is fine-tuned with the labeled data.

If a lot of labeled data and a lot of computational power are available, the whole deep network can be trained in a supervised manner, but the consensus is that using an unsupervised method to initialize the weights works much better than random initialization—learning can be done much faster and with fewer labeled data.

Deep learning methods are attractive mainly because they need less manual interference. We do not need to craft the right features or suitable basis functions (or kernels—chapter 13), or worry about the right network architecture. Once we have data (and nowadays we have “big” data) and sufficient computation available, we just wait and let the learning algorithm discover all that is necessary by itself.

The idea of multiple layers of increasing abstraction that lies underneath deep learning is intuitive. Not only in vision—in handwritten digits or face images—but in many applications, we can think of layers of abstraction, and discovering such abstract representations would be informative; for example, it allows visualization and also a better description of the problem.

Consider machine translation. For example, starting with an English sentence, in multiple levels of processing and abstraction that are learned automatically from a very large corpus of English sentences to code the lexical, syntactic, and semantic rules of the English language, we would get to the most abstract representation. Now consider the same sentence in French. The levels of processing learned this time from a French corpus would be different, but if two sentences mean the same, at the most abstract, language-independent level, they should have very similar representations.

11.14 Notes

Research on artificial neural networks is as old as the digital computer. McCulloch and Pitts (1943) proposed the first mathematical model for the artificial neuron. Rosenblatt (1962) proposed the perceptron model and a learning algorithm in 1962. Minsky and Papert (1969) showed the limitation of single-layer perceptrons, for example, the XOR problem, and since there was no algorithm to train a multilayer perceptron with a hidden layer at that time, the work on artificial neural networks almost stopped except at a few places. The renaissance of neural networks came with the paper by Hopfield (1982). This was followed by the two-volume parallel distributed processing (PDP) book written by the PDP Research Group (Rumelhart, McClelland, and the PDP Research Group 1986). It seems as though backpropagation was invented independently in several places almost at the same time and the limitation of a single-layer perceptron no longer held.

Starting in the mid-1980s, there has been a huge explosion of work on artificial neural network models from various disciplines: physics, statistics, psychology, cognitive science, neuroscience, and lingustics, not to mention computer science, electrical engineering, and adaptive control. Perhaps the most important contribution of research on artificial neural networks is this synergy that bridged various disciplines, especially statistics and engineering. It is thanks to this that the field of machine learning is now well established.

The field is much more mature now; aims are more modest and better defined. One of the criticisms of backpropagation was that it was not biologically plausible! Though the term "neural network" is still widely used, it is generally understood that neural network models, for example, multilayer perceptrons, are nonparametric estimators and that the best way to analyze them is by using statistical methods.

PROJECTION PURSUIT

For example, a statistical method similar to the multilayer perceptron is *projection pursuit* (Friedman and Stuetzle 1981), which is written as

$$y = \sum_{h=1}^{H} \phi_h(\boldsymbol{w}_h^T \boldsymbol{x})$$

the difference being that each "hidden unit" has its own separate function, $\phi_h(\cdot)$, though in an MLP, all are fixed to be sigmoid. In chapter 12, we will see another neural network structure, named radial basis functions, which uses the Gaussian function at the hidden units.

There are various textbooks on artificial neural networks: Hertz, Krogh, and Palmer 1991, the earliest, is still readable. Bishop 1995 has a pattern recognition emphasis and discusses in detail various optimization algorithms that can be used for training, as well as the Bayesian approach, generalizing weight decay. Ripley 1996 analyzes neural networks from a statistical perspective.

Artificial neural networks, for example, multilayer perceptrons, have various successful applications. In addition to their various successful applications in adaptive control, speech recognition, and vision, two are noteworthy: Tesauro's TD-Gammon program (Tesauro 1994) uses reinforcement learning (chapter 18) to train a multilayer perceptron and plays backgammon at a master level. Pomerleau's ALVINN is a neural network that autonomously drives a van up to 20 miles per hour after learning by observing a driver for five minutes (Pomerleau 1991).

Research in neural networks has seen a major boost recently with the advent of deep learning and deep neural networks, and we see them applied in many areas, for example, finance, biology, natural language processing, and so on, with impressive results—see `deeplearning.net` for more information. With bigger data and cheaper processing hardware every year, they promise to be more popular in the near future.

11.15 Exercises

1. Show the perceptron that calculates NOT of its input.

   SOLUTION:

   $y = s(-x + 0.5)$

2. Show the perceptron that calculates NAND of its two inputs.
3. Show the perceptron that calculates the parity of its three inputs.

   SOLUTION:

$$\begin{aligned} h_1 &= s(-x_1 - x_2 + 2x_3 - 1.5) \quad (001) \\ h_2 &= s(-x_1 + 2x_2 - x_3 - 1.5) \quad (010) \\ h_3 &= s(2x_1 - x_2 - x_3 - 1.5) \quad (100) \\ h_4 &= s(x_1 + x_2 + x_3 - 2.5) \quad (111) \\ y &= s(h_1 + h_2 + h_3 + h_4 - 0.5) \end{aligned}$$

   The four hidden units corresponding to the four cases of (x_1, x_2, x_3) values where the parity is 1, namely, 001, 010, 100, and 111. We then OR them to calculate the overall output. Note that another possibility is to calculate the three-bit parity in terms of two-bit parity (XOR) as $(x_1 \text{ XOR } x_2) \text{ XOR } x_3$.

4. Derive the update equations when the hidden units use tanh, instead of the sigmoid. Use the fact that $\tanh' = (1 - \tanh^2)$.
5. Derive the update equations for an MLP with two hidden layers.

   SOLUTION: Let us first define the forward equations:

$$\begin{aligned} z_{1h} &= \text{sigmoid}(\boldsymbol{w}_{1h}^T \boldsymbol{x}) = \text{sigmoid}\left(\sum_{j=1}^{d} w_{1hj} x_j + w_{1h0}\right), \; h = 1, \ldots, H_1 \\ z_{2l} &= \text{sigmoid}(\boldsymbol{w}_{2l}^T \boldsymbol{z}_1) = \text{sigmoid}\left(\sum_{h=0}^{H_1} w_{2lh} z_{1h} + w_{2l0}\right), \; l = 1, \ldots, H_2 \\ y_i &= \boldsymbol{v}_i^T \boldsymbol{z}_2 = \sum_{l=1}^{H_2} v_{il} z_{2l} + v_0 \end{aligned}$$

Let us take the case of regression:

$$E = \frac{1}{2}\sum_t \sum_i (r_i^t - y_i^t)^2$$

We just backpropagate, that is, continue the chain rule, and we can write error in a layer as a function of the error in the layer that follows it, carrying the supervised error in the output layer to the layers before:

$$\begin{aligned}
err_i &\equiv r_i^t - y_i^t \Rightarrow \Delta v_{il} = \eta \sum_t err_i z_{2l} \\
err_{2l} &\equiv \left[\sum_i err_i v_i\right] z_{2l}(1 - z_{2l}) \Rightarrow \Delta w_{2lh} = \eta \sum_t err_{2l} z_{1h} \\
err_{1h} &\equiv \left[\sum_l err_{2l} w_{2lh}\right] z_{1h}(1 - z_{1h}) \Rightarrow \Delta w_{1hj} = \eta \sum_t err_{1h} x_j
\end{aligned}$$

6. Consider an MLP architecture with one hidden layer where there are also direct weights from the inputs directly to the output units. Explain when such a structure would be helpful and how it can be trained.
7. Parity is cyclic shift invariant; for example, "0101" and "1010" have the same parity. Propose a multilayer perceptron to learn the parity function using this hint.
8. In cascade correlation, what are the advantages of freezing the previously existing weights?
9. Derive the update equations for an MLP implementing Sammon mapping that minimizes Sammon stress (equation 11.40).
10. In section 11.6, we discuss how an MLP with two hidden layers can implement piecewise constant approximation. Show that if the weight in the last layer is not a constant but a linear function of the input, we can implement piecewise linear approximation.
11. Derive the update equations for soft weight sharing.

    SOLUTION: Assume a single-layer network for two-class classification for simplicity:

$$y^t = \text{sigmoid}\left(\sum_i w_i x_i^t\right)$$

the augmented error is

$$E' = \log \sum_t r^t \log y^t + \lambda \sum_i \log \sum_{j=1}^{M} \alpha_j p_j(w_i)$$

where $p_j(w_i) \sim \mathcal{N}(m_j, s_j^2)$. Note that $\{w_i\}_i$ includes all the weights including the bias. When we use gradient descent, we get

$$\Delta w_i^t = \eta(r^t - y^t)x_i^t - \eta\lambda \sum_j \pi_j(w_i)\frac{(w_i - m_j)}{s_j^2}$$

where

$$\pi_j(w_i) = \frac{\alpha_j p_j(w_i)}{\sum_l \alpha_l p_l(w_i)}$$

is the posterior probability that w_i belongs to component j. The weight is updated to both decrease the cross-entropy and move it closer to the mean of the nearest Gaussian. Using such a scheme, we can also update the mixture parameters, for example:

$$\Delta m_j = \eta\lambda \sum_i \pi_j(w_i)\frac{(w_i - m_j)}{s_j^2}$$

$\pi_j(w_i)$ is close to 1 if it is highly probable that w_i comes from component j; in such a case, m_j is updated to be closer to the weight w_i it represents. This is an iterative clustering procedure, and we will discuss such methods in more detail in chapter 12; see for example, equation 12.5.

12. In the autoencoder network, how can we decide on the number of hidden units?
13. Incremental learning of the structure of a MLP can be viewed as a state space search. What are the operators? What is the goodness function? What type of search strategies are appropriate? Define these in such a way that dynamic node creation and cascade-correlation are special instantiations.
14. For the MLP given in figure 11.22, derive the update equations for the unfolded network.

11.16 References

Abu-Mostafa, Y. 1995. “Hints.” *Neural Computation* 7:639–671.

Aran, O., O. T. Yıldız, and E. Alpaydın. 2009. “An Incremental Framework Based on Cross-Validation for Estimating the Architecture of a Multilayer Perceptron.” *International Journal of Pattern Recognition and Artificial Intelligence* 23:159–190.

Ash, T. 1989. “Dynamic Node Creation in Backpropagation Networks.” *Connection Science* 1:365–375.

Battiti, R. 1992. “First- and Second-Order Methods for Learning: Between Steepest Descent and Newton’s Method.” *Neural Computation* 4:141–166.

Bengio, Y. 2009. “Learning Deep Architectures for AI.” *Foundations and Trends in Machine Learning* 2 (1): 1–127.

Bishop, C. M. 1995. *Neural Networks for Pattern Recognition.* Oxford: Oxford University Press.

Bourlard, H., and Y. Kamp. 1988. “Auto-Association by Multilayer Perceptrons and Singular Value Decomposition.” *Biological Cybernetics* 59:291–294.

Cottrell, G. W., P. Munro, and D. Zipser. 1987. “Learning Internal Representations from Gray-Scale Images: An Example of Extensional Programming.” In *Ninth Annual Conference of the Cognitive Science Society*, 462–473. Hillsdale, NJ: Erlbaum.

Durbin, R., and D. E. Rumelhart. 1989. “Product Units: A Computationally Powerful and Biologically Plausible Extension to Backpropagation Networks.” *Neural Computation* 1:133–142.

Fahlman, S. E., and C. Lebiere. 1990. “The Cascade Correlation Architecture.” In *Advances in Neural Information Processing Systems 2*, ed. D. S. Touretzky, 524–532. San Francisco: Morgan Kaufmann.

Friedman, J. H., and W. Stuetzle. 1981. “Projection Pursuit Regression.” *Journal of the American Statistical Association* 76:817–823.

Fukushima, K. 1980. “Neocognitron: A Self-Organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position.” *Biological Cybernetics* 36:193–202.

Geman, S., E. Bienenstock, and R. Doursat. 1992. “Neural Networks and the Bias/Variance Dilemma.” *Neural Computation* 4:1–58.

Hertz, J., A. Krogh, and R. G. Palmer. 1991. *Introduction to the Theory of Neural Computation.* Reading, MA: Addison-Wesley.

Hinton, G. E., and R. R. Salakhutdinov. 2006. “Reducing the dimensionality of data with neural networks.” *Science* 313:504–507.

Hopfield, J. J. 1982. “Neural Networks and Physical Systems with Emergent Collective Computational Abilities.” *Proceedings of the National Academy of Sciences USA* 79:2554–2558.

Hornik, K., M. Stinchcombe, and H. White. 1989. “Multilayer Feedforward Networks Are Universal Approximators.” *Neural Networks* 2:359–366.

Le Cun, Y., B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. 1989. “Backpropagation Applied to Handwritten Zipcode Recognition.” *Neural Computation* 1:541–551.

MacKay, D. J. C. 1992a. “Bayesian Interpolation.” *Neural Computation* 4:415–447.

MacKay, D. J. C. 1992b. “A Practical Bayesian Framework for Backpropagation Networks” *Neural Computation* 4:448–472.

Mao, J., and A. K. Jain. 1995. “Artificial Neural Networks for Feature Extraction and Multivariate Data Projection.” *IEEE Transactions on Neural Networks* 6:296–317.

Marr, D. 1982. *Vision.* New York: Freeman.

McCulloch, W. S., and W. Pitts. 1943. “A Logical Calculus of the Ideas Immenent in Nervous Activity.” *Bulletin of Mathematical Biophysics* 5:115–133.

Minsky, M. L., and S. A. Papert. 1969. *Perceptrons.* Cambridge, MA: MIT Press. (Expanded ed. 1990.)

Nowlan, S. J., and G. E. Hinton. 1992. “Simplifying Neural Networks by Soft Weight Sharing.” *Neural Computation* 4:473–493.

Pomerleau, D. A. 1991. “Efficient Training of Artificial Neural Networks for Autonomous Navigation.” *Neural Computation* 3:88–97.

Posner, M. I., ed. 1989. *Foundations of Cognitive Science.* Cambridge, MA: MIT Press.

Richard, M. D., and R. P. Lippmann. 1991. “Neural Network Classifiers Estimate Bayesian *a Posteriori* Probabilities.” *Neural Computation* 3:461–483.

Ripley, B. D. 1996. *Pattern Recognition and Neural Networks.* Cambridge, UK: Cambridge University Press.

Rosenblatt, F. 1962. *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms.* New York: Spartan.

Rumelhart, D. E., G. E. Hinton, and R. J. Williams. 1986a. “Learning Representations by Backpropagating Errors.” *Nature* 323:533–536.

Rumelhart, D. E., G. E. Hinton, and R. J. Williams. 1986b. “Learning Internal Representations by Error Propagation.” In *Parallel Distributed Processing*, ed. D. E. Rumelhart, J. L. McClelland, and the PDP Research Group, 318–362. Cambridge, MA: MIT Press.

Rumelhart, D. E., J. L. McClelland, and the PDP Research Group, eds. 1986. *Parallel Distributed Processing.* Cambridge, MA: MIT Press.

Simard, P., B. Victorri, Y, Le Cun, and J. Denker. 1992. “Tangent Prop: A Formalism for Specifying Selected Invariances in an Adaptive Network.” In *Advances in Neural Information Processing Systems 4*, ed. J. E. Moody, S. J. Hanson, and R. P. Lippman, 895–903. San Francisco: Morgan Kaufmann.

Tesauro, G. 1994. “TD-Gammon, a Self-Teaching Backgammon Program, Achieves Master-Level Play.” *Neural Computation* 6:215–219.

Thagard, P. 2005. *Mind: Introduction to Cognitive Science.* 2nd ed. Cambridge, MA: MIT Press.

Waibel, A., T. Hanazawa, G. Hinton, K. Shikano, and K. Lang. 1989. “Phoneme Recognition Using Time-Delay Neural Networks.” *IEEE Transactions on Acoustics, Speech, and Signal Processing* 37:328–339.

Williams, R. J., and D. Zipser. 1989. “A Learning Algorithm for Continually Running Fully Recurrent Neural Networks.” *Neural Computation* 1:270–280.

12 *Local Models*

We continue our discussion of multilayer neural networks with models where the first layer contains locally receptive units that respond to instances in a localized region of the input space. The second layer on top learns the regression or classification function for these local regions. We discuss learning methods for finding the local regions of importance as well as the models responsible in there.

12.1 Introduction

One way to do function approximation is to divide the input space into local patches and learn a separate fit in each local patch. In chapter 7, we discussed statistical methods for clustering that allowed us to group input instances and model the input distribution. Competitive methods are neural network methods for online clustering. In this chapter, we discuss the online version of *k*-means, as well as two neural network extensions, adaptive resonance theory (ART), and the self-organizing map (SOM).

We then discuss how supervised learning is implemented once the inputs are localized. If the fit in a local patch is constant, then the technique is named the radial basis function (RBF) network; if it is a linear function of the input, it is called the mixture of experts (MoE). We discuss both regression and classification, and also compare this approach with MLP, which we discussed in chapter 11.

12.2 Competitive Learning

In chapter 7, we used the semiparametric Gaussian mixture density, which assumes that the input comes from one of k Gaussian sources. In this section, we make the same assumption that there are k groups (or clusters) in the data, but our approach is not probabilistic in that we do not enforce a parametric model for the sources. Another difference is that the learning methods we propose are online. We do not have the whole sample at hand during training; we receive instances one by one and update model parameters as we get them. The term *competitive learning* is used because it is as if these groups, or rather the units representing these groups, compete among themselves to be the one responsible for representing an instance. The model is also called *winner-take-all*; it is as if one group wins and gets updated, and the others are not updated at all.

COMPETITIVE LEARNING

WINNER-TAKE-ALL

These methods can be used by themselves for online clustering, as opposed to the batch methods discussed in chapter 7. An online method has the usual advantages that (1) we do not need extra memory to store the whole training set; (2) updates at each step are simple to implement, for example, in hardware; and (3) the input distribution may change in time and the model adapts itself to these changes automatically. If we were to use a batch algorithm, we would need to collect a new sample and run the batch method from scratch over the whole sample.

Starting in section 12.3, we will also discuss how such an approach can be followed by a supervised method to learn regression or classification problems. This will be a two-stage system that can be implemented by a two-layer network, where the first stage (-layer) models the input density and finds the responsible local model, and the second stage is that of the local model generating the final output.

12.2.1 Online k-Means

In equation 7.3, we defined the reconstruction error as

$$E(\{\boldsymbol{m}_i\}_{i=1}^k|\mathcal{X}) = \frac{1}{2}\sum_t \sum_i b_i^t \|\boldsymbol{x}^t - \boldsymbol{m}_i\|^2 \tag{12.1}$$

where

$$b_i^t = \begin{cases} 1 & \text{if } \|\boldsymbol{x}^t - \boldsymbol{m}_i\| = \min_l \|\boldsymbol{x}^t - \boldsymbol{m}_l\| \\ 0 & \text{otherwise} \end{cases} \tag{12.2}$$

$\mathcal{X} = \{\boldsymbol{x}^t\}_t$ is the sample and $\boldsymbol{m}_i, i = 1, \ldots, k$ are the cluster centers. b_i^t is 1 if $\boldsymbol{m}_i$ is the closest center to $\boldsymbol{x}^t$ in Euclidean distance. It is as if all $\boldsymbol{m}_l, l = 1, \ldots, k$ compete and $\boldsymbol{m}_i$ wins the competition because it is the closest.

The batch algorithm, k-means, updates the centers as

$$\boldsymbol{m}_i = \frac{\sum_t b_i^t \boldsymbol{x}^t}{\sum_t b_i^t} \tag{12.3}$$

which minimizes equation 12.1, once the winners are chosen using equation 12.2. As we saw before, these two steps of calculating b_i^t and updating $\boldsymbol{m}_i$ are iterated until convergence.

ONLINE k-MEANS

We can obtain *online k-means* by doing stochastic gradient descent, considering the instances one by one, and doing a small update at each step, not forgetting the effect of the previous updates. The reconstruction error for a single instance is

$$E^t(\{\boldsymbol{m}_i\}_{i=1}^k | \boldsymbol{x}^t) = \frac{1}{2}\sum_i b_i^t \|\boldsymbol{x}^t - \boldsymbol{m}_i\|^2 = \frac{1}{2}\sum_i \sum_{j=1}^d b_i^t (x_j^t - m_{ij})^2 \tag{12.4}$$

where b_i^t is defined as in equation 12.2. Using gradient descent on this, we get the following update rule for each instance $\boldsymbol{x}^t$:

$$\Delta m_{ij} = -\eta \frac{\partial E^t}{\partial m_{ij}} = \eta b_i^t (x_j^t - m_{ij}) \tag{12.5}$$

This moves the closest center (for which $b_i^t = 1$) toward the input by a factor given by η. The other centers have their $b_l^t, l \neq i$ equal to 0 and are not updated (see figure 12.1). A batch procedure can also be defined by summing up equation 12.5 over all t. Like in any gradient descent procedure, a momentum term can also be added. For convergence, η is gradually decreased to 0. But this implies the *stability-plasticity dilemma*: If η is decreased toward 0, the network becomes stable but we lose adaptivity to novel patterns that may occur in time because updates become too small. If we keep η large, $\boldsymbol{m}_i$ may oscillate.

STABILITY-PLASTICITY DILEMMA

The pseudocode of online k-means is given in figure 12.2. This is the online version of the batch algorithm given in figure 7.3.

The competitive network can be implemented as a one-layer recurrent network as shown in figure 12.3. The input layer contains the input vector $\boldsymbol{x}$; note that there is no bias unit. The values of the output units are the b_i and they are perceptrons:

$$b_i = \boldsymbol{m}_i^T \boldsymbol{x} \tag{12.6}$$

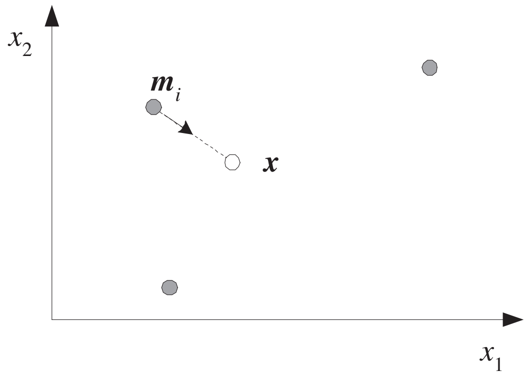


Figure 12.1 Shaded circles are the centers and the empty circle is the input instance. The online version of k-means moves the closest center along the direction of $(\boldsymbol{x} - \boldsymbol{m}_i)$ by a factor specified by η.

Then we need to choose the maximum of the b_i and set it equal to 1, and set the others, $b_l, l \neq i$ to 0. If we would like to do everything purely neural, that is, using a network of concurrently operating processing units, the choosing of the maximum can be implemented through *lateral inhibition*. As shown in figure 12.3, each unit has an excitatory recurrent connection (i.e., with a positive weight) to itself, and inhibitory recurrent connections (i.e., with negative weights) to the other output units. With an appropriate nonlinear activation function and positive and negative recurrent weight values, such a network, after some iterations, converges to a state where the maximum becomes 1 and all others become 0 (Grossberg 1980; Feldman and Ballard 1982).

LATERAL INHIBITION

The dot product used in equation 12.6 is a similarity measure, and we saw in section 5.5 (equation 5.26) that if $\boldsymbol{m}_i$ have the same norm, then the unit with the minimum Euclidean distance, $\|\boldsymbol{m}_i - \boldsymbol{x}\|$, is the same as the one with the maximum dot product, $\boldsymbol{m}_i^T \boldsymbol{x}$.

Here, and later when we discuss other competitive methods, we use the Euclidean distance, but we should keep in mind that using the Euclidean distance implies that all input attributes have the same variance and that they are not correlated. If this is not the case, this should be reflected in the distance measure, that is, by using the Mahalanobis distance, or suitable normalization should be done, for example, by PCA, at

```
Initialize $\boldsymbol{m}_i, i = 1, \ldots, k$, for example, to $k$ random $\boldsymbol{x}^t$
Repeat
    For all $\boldsymbol{x}^t \in \mathcal{X}$ in random order
        $i \leftarrow \arg\min_j \|\boldsymbol{x}^t - \boldsymbol{m}_j\|$
        $\boldsymbol{m}_i \leftarrow \boldsymbol{m}_i + \eta(\boldsymbol{x}^t - \boldsymbol{m}_i)$
Until $\boldsymbol{m}_i$ converge
```

Figure 12.2 Online k-means algorithm. The batch version is given in figure 7.3.

a preprocessing stage before the Euclidean distance is used.

We can rewrite equation 12.5 as

(12.7) $$\Delta m_{ij}^t = \eta b_i^t x_j^t - \eta b_i^t m_{ij}$$

Let us remember that m_{ij} is the weight of the connection from x_j to b_i. An update of the form, as we see in the first term

(12.8) $$\Delta m_{ij}^t = \eta b_i^t x_j^t$$

HEBBIAN LEARNING

is *Hebbian learning*, which defines the update as the product of the values of the presynaptic and postsynaptic units. It was proposed as a model for neural plasticity: A synapse becomes more important if the units before and after the connection fire simultaneously, indicating that they are correlated. However, with only Hebbian learning, the weights grow without bound ($x_j^t \geq 0$), and we need a second force to decrease the weights that are not updated. One possibility is to explicitly normalize the weights to have $\|\boldsymbol{m}_i\| = 1$; if $\Delta m_{ij} > 0$ and $\Delta m_{il} = 0, l \neq j$, once we normalize $\boldsymbol{m}_i$ to unit length, m_{il} decrease. Another possibility is to introduce a weight decay term (Oja 1982), and the second term of equation 12.7 can be seen as such. Hertz, Krogh, and Palmer (1991) discuss competitive networks and Hebbian learning in more detail and show, for example, how such networks can learn to do PCA. Mao and Jain (1995) discuss online algorithms for PCA and LDA.

As we saw in chapter 7, one problem is to avoid dead centers, namely, the ones that are there but are not effectively utilized. In the case of competitive networks, this corresponds to centers that never win the competition because they are initialized far away from any input. There are various ways we can avoid this:

1. We can initialize $\boldsymbol{m}_i$ by randomly chosen input instances, and make sure that they start from where there is data.

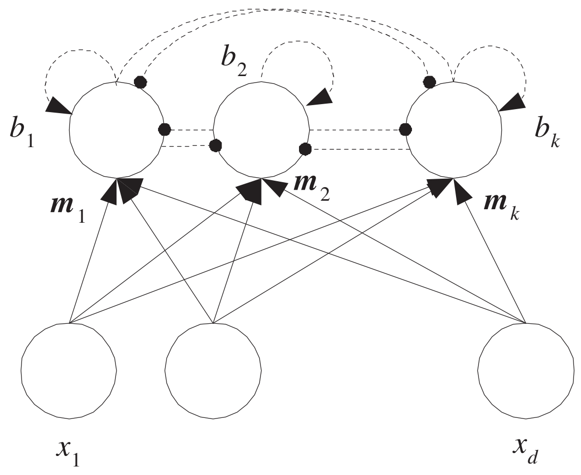


Figure 12.3 The winner-take-all competitive neural network, which is a network of k perceptrons with recurrent connections at the output. Dashed lines are recurrent connections, of which the ones that end with an arrow are excitatory and the ones that end with a circle are inhibitory. Each unit at the output reinforces its value and tries to suppress the other outputs. Under a suitable assignment of these recurrrent weights, the maximum suppresses all the others. This has the net effect that the one unit whose $\boldsymbol{m}_i$ is closest to $\boldsymbol{x}$ ends up with its b_i equal to 1 and all others, namely, $b_l, l \neq i$ are 0.

2. We can use a leader-cluster algorithm and add units one by one, always adding them at a place where they are needed. One example is the ART model, which we discuss in section 12.2.2.

3. When we update, we do not update only the center of the closest unit but some others as well. As they are updated, they also move toward the input, move gradually toward parts of the input space where there are inputs, and eventually win the competition. One example that we discuss in section 12.2.3 is SOM.

4. Another possibility is to introduce a *conscience* mechanism (DeSieno 1988): A unit that has won the competition recently feels guilty and allows others to win.

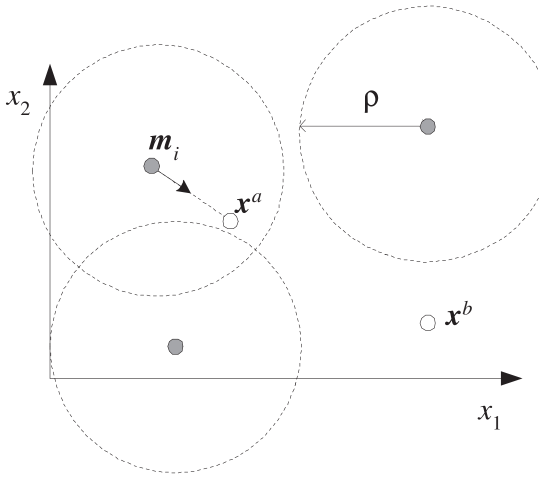


Figure 12.4 The distance from $\boldsymbol{x}^a$ to the closest center is less than the vigilance value ρ and the center is updated as in online k-means. However, $\boldsymbol{x}^b$ is not close enough to any of the centers and a new group should be created at that position.

12.2.2 Adaptive Resonance Theory

The number of groups, k, should be known and specified before the parameters can be calculated. Another approach is *incremental*, where one starts with a single group and adds new groups as they are needed. We discuss the *adaptive resonance theory* (ART) algorithm (Carpenter and Grossberg 1988) as an example of an incremental algorithm. In ART, given an input, all of the output units calculate their values and the one most similar to the input is chosen. This is the unit with the maximum value if the unit uses the dot product as in equation 12.6, or it is the unit with the minimum value if the unit uses the Euclidean distance.

ADAPTIVE RESONANCE THEORY

Let us assume that we use the Euclidean distance. If the minimum value is smaller than a certain threshold value, named the *vigilance*, the update is done as in online k-means. If this distance is larger than vigilance, a new output unit is added and its center is initialized with the instance. This defines a hypersphere whose radius is given by the vigilance defining the volume of scope of each unit; we add a new unit whenever we have an input that is not covered by any unit (see figure 12.4).

VIGILANCE

Denoting vigilance by ρ, we use the following equations at each update:

$$b_i = \|\boldsymbol{m}_i - \boldsymbol{x}^t\| = \min_{l=1}^{k} \|\boldsymbol{m}_l - \boldsymbol{x}^t\| \tag{12.9}$$

$$\begin{cases} \boldsymbol{m}_{k+1} \leftarrow \boldsymbol{x}^t & \text{if } b_i > \rho \\ \Delta\boldsymbol{m}_i = \eta(\boldsymbol{x}^t - \boldsymbol{m}_i) & \text{otherwise} \end{cases}$$

Putting a threshold on distance is equivalent to putting a threshold on the reconstruction error per instance, and if the distance is Euclidean and the error is defined as in equation 12.4, this indicates that the maximum reconstruction error allowed per instance is the square of vigilance.

12.2.3 Self-Organizing Maps

SELF-ORGANIZING MAP

One way to avoid having dead units is by updating not only the winner but also some of the other units as well. In the *self-organizing map* (SOM) proposed by Kohonen (1990, 1995), unit indices, namely, i as in $\boldsymbol{m}_i$, define a *neighborhood* for the units. When $\boldsymbol{m}_i$ is the closest center, in addition to $\boldsymbol{m}_i$, its neighbors are also updated. For example, if the neighborhood is of size 2, then $\boldsymbol{m}_{i-2}, \boldsymbol{m}_{i-1}, \boldsymbol{m}_{i+1}, \boldsymbol{m}_{i+2}$ are also updated but with less weight as the neighborhood increases. If i is the index of the closest center, the centers are updated as

$$\Delta\boldsymbol{m}_l = \eta\, e(l,i)(\boldsymbol{x}^t - \boldsymbol{m}_l) \tag{12.10}$$

where $e(l,i)$ is the *neighborhood function*. $e(l,i) = 1$ when $l = i$ and decreases as $|l-i|$ increases, for example, as a Gaussian, $\mathcal{N}(i,\sigma)$:

$$e(l,i) = \frac{1}{\sqrt{2\pi}\sigma}\exp\left[-\frac{(l-i)^2}{2\sigma^2}\right] \tag{12.11}$$

For convergence, the support of the neighborhood function decreases in time, for example, σ decreases, and at the end, only the winner is updated.

Because neighboring units are also moved toward the input, we avoid dead units since they get to win competition sometime later, after a little bit of initial help from their neighboring friends (see figure 12.5).

Updating the neighbors has the effect that, even if the centers are randomly initialized, because they are moved toward the same input together, once the system converges, units with neighboring indices will also be neighbors in the input space.

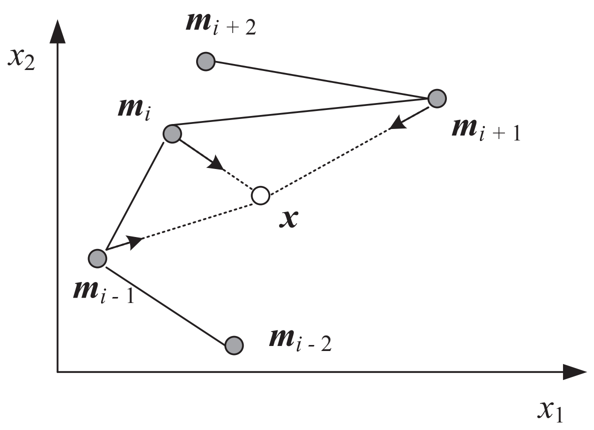


Figure 12.5 In the SOM, not only the closest unit but also its neighbors, in terms of indices, are moved toward the input. Here, neighborhood is 1; $\boldsymbol{m}_i$ and its 1-nearest neighbors are updated. Note here that $\boldsymbol{m}_{i+1}$ is far from $\boldsymbol{m}_i$, but as it is updated with $\boldsymbol{m}_i$, and as $\boldsymbol{m}_i$ will be updated when $\boldsymbol{m}_{i+1}$ is the winner, they will become neighbors in the input space as well.

In most applications, the units are organized as a two-dimensional *map*. That is, each unit will have two indices, $\boldsymbol{m}_{i,j}$, and the neighborhood will be defined in two dimensions. If $\boldsymbol{m}_{i,j}$ is the closest center, the centers are updated as

$$\Delta \boldsymbol{m}_{k,l} = \eta e(k,l,i,j)(\boldsymbol{x}^t - \boldsymbol{m}_{k,l}) \tag{12.12}$$

TOPOGRAPHICAL MAP

where the neighborhood function is now in two dimensions. After convergence, this forms a two-dimensional *topographical map* of the original d-dimensional input space. The map contains many units in parts of the space where density is high, and no unit will be dedicated to parts where there is no input. Once the map converges, inputs that are close in the original space are mapped to units that are close in the map. In this regard, the map can be interpreted as doing a nonlinear form of multidimensional scaling, mapping from the original $\boldsymbol{x}$ space to the two dimensions, (i, j). Similarly, if the map is one-dimensional, the units are placed on the curve of maximum density in the input space, as a *principal curve*.

12.3 Radial Basis Functions

In a multilayer perceptron (chapter 11) where hidden units use the dot product, each hidden unit defines a hyperplane and with the sigmoid nonlinearity, a hidden unit has a value between 0 and 1, coding the position of the instance with respect to the hyperplane. Each hyperplane divides the input space in two, and typically for a given input, many of the hidden units have nonzero output. This is called a *distributed representation* because the input is encoded by the simultaneous activation of many hidden units.

DISTRIBUTED REPRESENTATION

LOCAL REPRESENTATION

Another possibility is to have a *local representation* where for a given input, only one or a few units are active. It is as if these *locally tuned units* partition the input space among themselves and are selective to only certain inputs. The part of the input space where a unit has nonzero response is called its *receptive field*. The input space is then paved with such units.

RECEPTIVE FIELD

Neurons with such response characteristics are found in many parts of the cortex. For example, cells in the visual cortex respond selectively to stimulation that is both local in retinal position and local in angle of visual orientation. Such locally tuned cells are typically arranged in topogrophical cortical maps in which the values of the variables to which the cells respond vary by their position in the map, as in a SOM.

The concept of locality implies a distance function to measure the similarity between the given input $\boldsymbol{x}$ and the position of unit h, $\boldsymbol{m}_h$. Frequently this measure is taken as the Euclidean distance, $\|\boldsymbol{x} - \boldsymbol{m}_h\|$. The response function is chosen to have a maximum where $\boldsymbol{x} = \boldsymbol{m}_h$ and decreasing as they get less similar. Commonly we use the Gaussian function (see figure 12.6):

$$p_h^t = \exp\left[-\frac{\|\boldsymbol{x}^t - \boldsymbol{m}_h\|^2}{2s_h^2}\right] \tag{12.13}$$

Strictly speaking, this is not Gaussian density, but we use the same name anyway. $\boldsymbol{m}_j$ and s_j respectively denote the center and the spread of the local unit j, and as such define a radially symmetric basis function. One can use an elliptic one with different spreads on different dimensions, or even use the full Mahalanobis distance to allow correlated inputs, at the expense of using a more complicated model (exercise 2).

The idea in using such local basis functions is that in the input data, there are groups or clusters of instances and for each such cluster, we

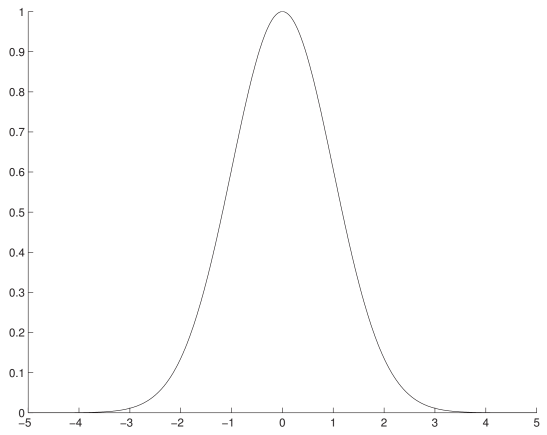


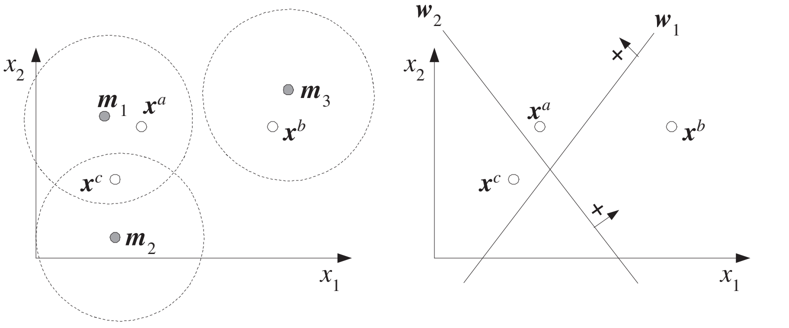
Figure 12.6 The one-dimensional form of the bell-shaped function used in the radial basis function network. This one has $m = 0$ and $s = 1$. It is like a Gaussian but it is not a density; it does not integrate to 1. It is nonzero between $(m - 3s, m + 3s)$, but a more conservative interval is $(m - 2s, m + 2s)$.

define a basis function, p_h^t, which becomes nonzero if instance $\boldsymbol{x}^t$ belongs to cluster h. One can use any of the online competitive methods discussed in section 12.2 to find the centers, $\boldsymbol{m}_h$. There is a simple and effective heuristic to find the spreads: Once we have the centers, for each cluster, we find the most distant instance covered by that cluster and set s_h to half its distance from the center. We could have used one-third, but we prefer to be conservative. We can also use the statistical clustering method, for example, EM on Gaussian mixtures, that we discussed in chapter 7 to find the cluster parameters, namely, means, variances (and covariances).

$p_h^t, h = 1, \ldots, H$ define a new H-dimensional space and form a new representation of $\boldsymbol{x}^t$. We can also use b_h^t (equation 12.2) to code the input but b_h^t are 0/1; p_h^t have the additional advantage that they code the distance to their center by a value in $(0, 1)$. How fast the value decays to 0 depends on s_h. Figure 12.7 gives an example and compares such a *local representation* with a *distributed* representation as used by the multilayer perceptron. Because Gaussians are local, typically we need many more local units than what we would need if we were to use a distributed representation, especially if the input is high-dimensional.

DISTRIBUTED VS LOCAL REPRESENTATION

In the case of supervised learning, we can then use this new local rep-



Local representation in the space of (p_1, p_2, p_3)		Distributed representation in the space of (h_1, h_2)	
$\boldsymbol{x}^a$:	(1.0, 0.0, 0.0)	$\boldsymbol{x}^a$:	(1.0, 1.0)
$\boldsymbol{x}^b$:	(0.0, 0.0, 1.0)	$\boldsymbol{x}^b$:	(0.0, 1.0)
$\boldsymbol{x}^c$:	(1.0, 1.0, 0.0)	$\boldsymbol{x}^c$:	(1.0, 0.0)

Figure 12.7 The difference between local and distributed representations. The values are hard, 0/1, values. One can use soft values in $(0, 1)$ and get a more informative encoding. In the local representation, this is done by the Gaussian RBF that uses the distance to the center, $\boldsymbol{m}_i$, and in the distributed representation, this is done by the sigmoid that uses the distance to the hyperplane, $\boldsymbol{w}_i$.

resentation as the input. If we use a perceptron, we have

$$y^t = \sum_{h=1}^{H} w_h p_h^t + w_0 \tag{12.14}$$

where H is the number of basis functions. This structure is called a *radial basis function* (RBF) network (Broomhead and Lowe 1988; Moody and Darken 1989). Normally, people do not use RBF networks with more than one layer of Gaussian units. H is the complexity parameter, like the number of hidden units in a multilayer perceptron. Previously we denoted it by k, when it corresponded to the number of centers in the case of unsupervised learning.

RADIAL BASIS FUNCTION

Here, we see the advantage of using p_h instead of b_h. Because b_h are 0/1, if equation 12.14 contained b_h instead of the p_h, it would give a piecewise constant approximation with discontuinities at the unit region boundaries. p_h values are soft and lead to a smooth approximation, taking a weighted average while passing from one region to another. We can easily see that such a network is a universal approximator in that it can approximate any function with desired accuracy, given enough units. We can form a grid in the input space to our desired accuracy, define a unit that will be active for each cell, and set its outgoing weight, w_h, to the desired output value.

This architecture bears much similarity to the nonparametric estimators, for example, Parzen windows, we saw in chapter 8, and p_h may be seen as kernel functions. The difference is that now we do not have a kernel function over all training instances but group them using a clustering method to make do with fewer kernels. H, the number of units, is the complexity parameter, trading off simplicity and accuracy. With more units, we approximate the training data better, but we get a complex model and risk overfitting; too few may underfit. Again, the optimal value is determined by cross-validation.

Once $\boldsymbol{m}_h$ and s_h are given and fixed, p_h are also fixed. Then w_h can be trained easily batch or online. In the case of regression, this is a linear regression model (with p_h as the inputs) and the w_h can be solved analytically without any iteration (section 4.6). In the case of classification, we need to resort to an iterative procedure. We discussed learning methods for this in chapter 10 and do not repeat them here.

What we do here is a two-stage process: We use an unsupervised method for determining the centers, then build a supervised layer on top of that. This is called *hybrid learning*. We can also learn all parameters, including $\boldsymbol{m}_h$ and s_h, in a supervised manner. The radial basis function of equation 12.13 is differentiable and we can backpropagate, just as we backpropagated in a multilayer perceptron to update the first-layer weights. The structure is similar to a multilayer perceptron with p_h as the hidden units, $\boldsymbol{m}_h$ and s_h as the first-layer parameters, the Gaussian as the activation function in the hidden layer, and w_h as the second-layer weights (see figure 12.8).

HYBRID LEARNING

But before we discuss this, we should remember that training a two-layer network is slow. Hybrid learning trains one layer at a time and is faster. Another technique, called the *anchor* method, sets the centers to the randomly chosen patterns from the training set without any further update. It is adequate if there are many units.

ANCHOR

On the other hand, the accuracy normally is not as high as when a completely supervised method is used. Consider the case when the input is uniformly distributed. Then k-means clustering places the units uniformly. If the function is changing significantly in a small part of the

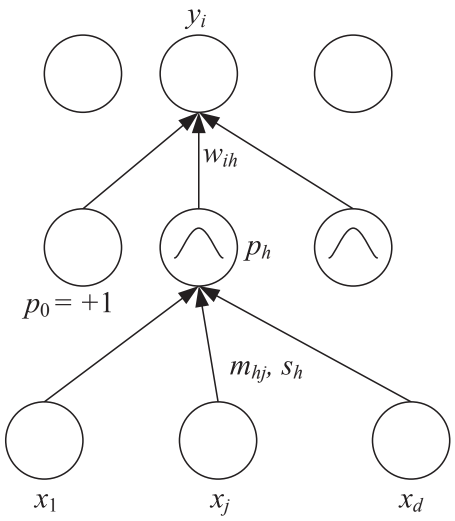


Figure 12.8 The RBF network where p_h are the hidden units using the bell-shaped activation function. $\boldsymbol{m}_h, s_h$ are the first-layer parameters, and $\boldsymbol{w}_i$ are the second-layer weights.

space, it is a better idea to have as many centers in places where the function changes fast, to make the error as small as possible; this is what the completely supervised method would do.

Let us discuss how all of the parameters can be trained in a fully supervised manner. The approach is the same as backpropagation applied to multilayer perceptrons. Let us see the case of regression with multiple outputs. The batch error is

$$E(\{\boldsymbol{m}_h, s_h, w_{ih}\}_{i,h}|\mathcal{X}) = \frac{1}{2}\sum_t \sum_i (r_i^t - y_i^t)^2 \quad (12.15)$$

where

$$y_i^t = \sum_{h=1}^{H} w_{ih} p_h^t + w_{i0} \quad (12.16)$$

Using gradient descent, we get the following update rule for the second-layer weights:

$$\Delta w_{ih} = \eta \sum_t (r_i^t - y_i^t) p_h^t \quad (12.17)$$

This is the usual perceptron update rule, with p_h as the inputs. Typically, p_h do not overlap much and at each iteration, only a few p_h are nonzero and only their w_h are updated. That is why RBF networks learn very fast, and faster than multilayer perceptrons that use a distributed representation.

Similarly, we can get the update equations for the centers and spreads by backpropagation (chain rule):

$$\Delta m_{hj} = \eta \sum_t \left[\sum_i (r_i^t - y_i^t) w_{ih} \right] p_h^t \frac{(x_j^t - m_{hj})}{s_h^2} \tag{12.18}$$

$$\Delta s_h = \eta \sum_t \left[\sum_i (r_i^t - y_i^t) w_{ih} \right] p_h^t \frac{\|\boldsymbol{x}^t - \boldsymbol{m}_h\|^2}{s_h^3} \tag{12.19}$$

Let us compare equation 12.18 with equation 12.5: First, here we use p_h instead of b_h, which means that not only the closest one but all units are updated, depending on their centers and spreads. Second, here the update is supervised and contains the backpropagated error term. The update depends not only on the input but also on the final error $(r_i^t - y_i^t)$, the effect of the unit on the output, w_{ih}, the activation of the unit, p_h, and the input, $(\boldsymbol{x} - \boldsymbol{m}_h)$.

In practice, equations 12.18 and 12.19 need some extra control. We need to explicitly check that s_h do not become very small or very large to be useless; we also need to check that $\boldsymbol{m}_h$ stay in the valid input range.

In the case of classification, we have

$$y_i^t = \frac{\exp\left[\sum_h w_{ih} p_h^t + w_{i0}\right]}{\sum_k \exp\left[\sum_h w_{kh} p_h^t + w_{k0}\right]} \tag{12.20}$$

and the cross-entropy error is

$$E(\{\boldsymbol{m}_h, s_h, w_{ih}\}_{i,h} | \mathcal{X}) = -\sum_t \sum_i r_i^t \log y_i^t \tag{12.21}$$

Update rules can similarly be derived using gradient descent (exercise 3).

Let us look again at equation 12.14. For any input, if p_h is nonzero, then it contributes w_h to the output. Its contribution is a constant fit, as given by w_h. Normally Gaussians do not overlap much, and one or two of them have a nonzero p_h value. In any case, only few units contribute to the output. w_0 is the constant offset and is added to the weighted sum of the active (nonzero) units. We also see that $y = w_0$ if all p_h are 0. We can therefore view w_0 as the "default" value of y: If no Gaussian is active, then the output is given by this value. So a possibility is to make this "default model" a more powerful "rule." For example, we can write

$$y^t = \underbrace{\sum_{h=1}^{H} w_h p_h^t}_{exceptions} + \underbrace{\boldsymbol{v}^T \boldsymbol{x}^t + v_0}_{rule} \tag{12.22}$$

In this case, the rule is linear: $\boldsymbol{v}^T \boldsymbol{x}^t + v_0$. When they are nonzero, Gaussians work as localized "exceptions" and modify the output to make up for the difference between the desired output and the rule output. Such a model can be trained in a supervised manner, and the rule can be trained together with the exceptions (exercise 4). We discuss a similar model, cascading, in section 17.11 where we see it as a combination of two learners, one general rule and the other formed by a set of exceptions.

12.4 Incorporating Rule-Based Knowledge

PRIOR KNOWLEDGE

The training of any learning system can be much simpler if we manage to incorporate *prior knowledge* to initialize the system. For example, prior knowledge may be available in the form of a set of rules that specify the input/output mapping that the model, for example, the RBF network, has to learn. This occurs frequently in industrial and medical applications where rules can be given by experts. Similarly, once a network has been trained, rules can be extracted from the solution in such a way as to better understand the solution to the problem.

The inclusion of prior knowledge has the additional advantage that if the network is required to extrapolate into regions of the input space where it has not seen any training data, it can rely on this prior knowledge. Furthermore, in many control applications, the network is required to make reasonable predictions right from the beginning. Before it has seen sufficient training data, it has to rely primarily on this prior knowledge.

In many applications we are typically told some basic rules that we try to follow in the beginning but that are then refined and altered through experience. The better our initial knowledge of a problem, the faster we can achieve good performance and the less training that is required.

Such inclusion of prior knowledge or extraction of learned knowledge is easy to do with RBF networks because the units are local. This makes RULE EXTRACTION *rule extraction* easier (Tresp, Hollatz, and Ahmad 1997). An example is

$$\text{IF } ((x_1 \approx a) \text{ AND } (x_2 \approx b)) \text{ OR } (x_3 \approx c) \text{ THEN } y = 0.1 \tag{12.23}$$

where $x_1 \approx a$ means "x_1 is approximately a." In the RBF framework, this rule is encoded by two Gaussian units as

$$\begin{aligned} p_1 &= \exp\left[-\frac{(x_1-a)^2}{2s_1^2}\right] \cdot \exp\left[-\frac{(x_2-b)^2}{2s_2^2}\right] \text{ with } w_1 = 0.1 \\ p_2 &= \exp\left[-\frac{(x_3-c)^2}{2s_3^2}\right] \text{ with } w_2 = 0.1 \end{aligned}$$

"Approximately equal to" is modeled by a Gaussian where the center is the ideal value and the spread denotes the allowed difference around this ideal value. Conjunction is the product of two univariate Gaussians that is a bivariate Gaussian. Then, the first product term can be handled by a two-dimensional, namely, $\boldsymbol{x} = [x_1, x_2]$, Gaussian centered at (a, b), and the spreads on the two dimensions are given by s_1 and s_2. Disjunction is modeled by two separate Gaussians, each one handling one of the disjuncts.

Given labeled training data, the parameters of the RBF network so constructed can be fine-tuned after the initial construction, using a small value of η.

This formulation is related to the fuzzy logic approach where equation 12.23 is named a *fuzzy rule*. The Gaussian basis function that checks for approximate equality corresponds to a *fuzzy membership function* (Berthold 1999; Cherkassky and Mulier 1998).

FUZZY RULE

FUZZY MEMBERSHIP FUNCTION

12.5 Normalized Basis Functions

In equation 12.14, for an input, it is possible that all of the p_h are 0. In some applications, we may want to have a *normalization* step to make sure that the values of the local units sum up to 1, thus making sure that for any input there is at least one nonzero unit:

$$g_h^t = \frac{p_h^t}{\sum_{l=1}^H p_l^t} = \frac{\exp[-\|\boldsymbol{x}^t - \boldsymbol{m}_h\|^2/2s_h^2]}{\sum_l \exp[-\|\boldsymbol{x}^t - \boldsymbol{m}_l\|^2/2s_l^2]} \tag{12.24}$$

An example is given in figure 12.9. Taking p_h as $p(\boldsymbol{x}|h)$, g_h correspond to $p(h|\boldsymbol{x})$, the posterior probability that $\boldsymbol{x}$ belongs to unit h. It is as if

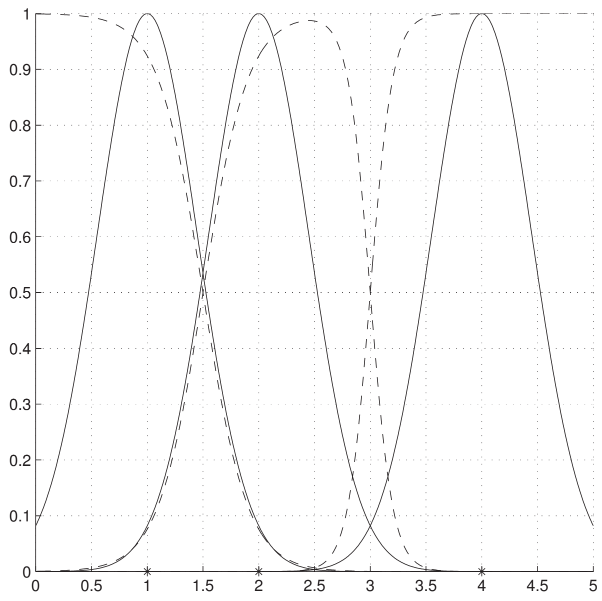


Figure 12.9 (-) Before and (- -) after normalization for three Gaussians whose centers are denoted by '*'. Note how the nonzero region of a unit depends also on the positions of other units. If the spreads are small, normalization implements a harder split; with large spreads, units overlap more.

the units divide the input space among themselves. We can think of g_h as a classifier in itself, choosing the responsible unit for a given input. This classification is done based on distance, as in a parametric Gaussian classifier (chapter 5).

The output is a weighted sum

$$y_i^t = \sum_{h=1}^{H} w_{ih} g_h^t \tag{12.25}$$

where there is no need for a bias term because there is at least one nonzero g_h for each $\boldsymbol{x}$. Using g_h instead of p_h does not introduce any extra parameters; it only couples the units together: p_h depends only on $\boldsymbol{m}_h$ and s_h, but g_h, because of normalization, depends on the centers and spreads of all of the units.

In the case of regression, we have the following update rules using gradient descent:

$$\Delta w_{ih} = \eta \sum_t (r_i^t - y_i^t) g_h^t \tag{12.26}$$

$$\Delta m_{hj} = \eta \sum_t \sum_i (r_i^t - y_i^t)(w_{ih} - y_i^t) g_h^t \frac{(x_j^t - m_{hj})}{s_h^2} \tag{12.27}$$

The update rule for s_h as well as the rules for classification can similarly be derived. Let us compare these with the update rules for the RBF with unnormalized Gaussians (equation 12.17). Here, we use g_h instead of p_h, which makes a unit's update dependent not only on its own parameters, but also on the centers and spreads of other units as well. Comparing equation 12.27 with equation 12.18, we see that instead of w_{ih}, we have $(w_{ih} - y_i^t)$, which shows the role of normalization on the output. The "responsible" unit wants to decrease the difference between its output, w_{ih}, and the final output, y_i^t, proportional to its responsibility, g_h.

12.6 Competitive Basis Functions

As we have seen up until now, in an RBF network the final output is determined as a weighted sum of the contributions of the local units. Though the units are local, it is the final weighted sum that is important and that we want to make as close as possible to the required output. For example, in regression we minimize equation 12.15, which is based on the probabilistic model

$$p(\boldsymbol{r}^t|\boldsymbol{x}^t) = \prod_i \frac{1}{\sqrt{2\pi}\sigma} \exp\left[-\frac{(r_i^t - y_i^t)^2}{2\sigma^2}\right] \tag{12.28}$$

where y_i^t is given by equation 12.16 (unnormalized) or equation 12.25 (normalized). In either case, we can view the model as a *cooperative* one since the units cooperate to generate the final output, y_i^t. We now discuss the approach using *competitive basis functions* where we assume that the output is drawn from a mixture model

COMPETITIVE BASIS FUNCTIONS

$$p(\boldsymbol{r}^t|\boldsymbol{x}^t) = \sum_{h=1}^{H} p(h|\boldsymbol{x}^t) p(\boldsymbol{r}^t|h, \boldsymbol{x}^t) \tag{12.29}$$

$p(h|\boldsymbol{x}^t)$ are the mixture proportions and $p(\boldsymbol{r}^t|h, \boldsymbol{x}^t)$ are the mixture components generating the output if that component is chosen. Note that both of these terms depend on the input $\boldsymbol{x}$.

The mixture proportions are

$$p(h|\boldsymbol{x}) = \frac{p(\boldsymbol{x}|h)p(h)}{\sum_l p(\boldsymbol{x}|l)p(l)} \tag{12.30}$$

$$g_h^t = \frac{a_h \exp[-\|\boldsymbol{x}^t - \boldsymbol{m}_h\|^2/2s_h^2]}{\sum_l a_l \exp[-\|\boldsymbol{x}^t - \boldsymbol{m}_l\|^2/2s_l^2]} \tag{12.31}$$

We generally assume a_h to be equal and ignore them. Let us first take the case of regression where the components are Gaussian. In equation 12.28, noise is added to the weighted sum; here, one component is chosen and noise is added to its output, y_{ih}^t.

Using the mixture model of equation 12.29, the log likelihood is

$$\mathcal{L}(\{\boldsymbol{m}_h, s_h, w_{ih}\}_{i,h}|\mathcal{X}) = \sum_t \log \sum_h g_h^t \exp\left[-\frac{1}{2}\sum_i (r_i^t - y_{ih}^t)^2\right] \tag{12.32}$$

where $y_{ih}^t = w_{ih}$ is the constant fit done by component h for output i, which, strictly speaking, does not depend on $\boldsymbol{x}$. (In section 12.8.2, we discuss the case of competitive mixture of experts where the local fit is a linear function of $\boldsymbol{x}$.) We see that if g_h^t is 1, then it is responsible for generating the right output and needs to minimize the squared error of its prediction, $\sum_i (r_i^t - y_{ih}^t)^2$.

Using gradient ascent to maximize the log likelihood, we get

$$\Delta w_{ih} = \eta \sum_t (r_i^t - y_{ih}^t) f_h^t \tag{12.33}$$

where

$$f_h^t = \frac{g_h^t \exp[-\frac{1}{2}\sum_i (r_i^t - y_{ih}^t)^2]}{\sum_l g_l^t \exp[-\frac{1}{2}\sum_i (r_i^t - y_{il}^t)^2]} \tag{12.34}$$

$$p(h|\boldsymbol{r},\boldsymbol{x}) = \frac{p(h|\boldsymbol{x})p(\boldsymbol{r}|h,\boldsymbol{x})}{\sum_l p(l|\boldsymbol{x})p(\boldsymbol{r}|l,\boldsymbol{x})} \tag{12.35}$$

$g_h^t \equiv p(h|\boldsymbol{x}^t)$ is the posterior probability of unit h given the input, and it depends on the centers and spreads of all of the units. $f_h^t \equiv p(h|\boldsymbol{r},\boldsymbol{x}^t)$ is the posterior probability of unit h given the input and the desired output, also taking the error into account in choosing the responsible unit.

Similarly, we can derive a rule to update the centers:

$$\Delta m_{hj} = \eta \sum_t (f_h^t - g_h^t)\frac{(x_j^t - m_{hj})}{s_h^2} \tag{12.36}$$

f_h is the posterior probability of unit h also taking the required output into account, whereas g_h is the posterior probability using only the input space information. Their difference is the error term for the centers. Δs_h can be similarly derived. In the cooperative case, there is no force on the units to be localized. To decrease the error, means and spreads can take any value; it is even possible sometimes for the spreads to increase and flatten out. In the competitive case, however, to increase the likelihood, units are forced to be localized with more separation between them and smaller spreads.

In classification, each component by itself is a multinomial. Then the log likelihood is

$$\mathcal{L}(\{\boldsymbol{m}_h, s_h, w_{ih}\}_{i,h}|\mathcal{X}) = \sum_t \log \sum_h g_h^t \prod_i (y_{ih}^t)^{r_i^t} \tag{12.37}$$

$$= \sum_t \log \sum_h g_h^t \exp\left[\sum_i r_i^t \log y_{ih}^t\right] \tag{12.38}$$

where

$$y_{ih}^t = \frac{\exp w_{ih}}{\sum_k \exp w_{kh}} \tag{12.39}$$

Update rules for $w_{ih}, \boldsymbol{m}_h$, and s_h can be derived using gradient ascent, which will include

$$f_h^t = \frac{g_h^t \exp[\sum_i r_i^t \log y_{ih}^t]}{\sum_l g_l^t \exp[\sum_i r_i^t \log y_{il}^t]} \tag{12.40}$$

In chapter 7, we discussed the EM algorithm for fitting Gaussian mixtures to data. It is possible to generalize EM for supervised learning as well. Actually, calculating f_h^t corresponds to the E-step. $f_h^t \equiv p(\boldsymbol{r}|h, \boldsymbol{x}^t)$ replaces $p(h|\boldsymbol{x}^t)$, which we used in the E-step in chapter 7 when the application was unsupervised. In the M-step for regression, we update the parameters as

$$\boldsymbol{m}_h = \frac{\sum_t f_h^t \boldsymbol{x}^t}{\sum_t f_h^t} \tag{12.41}$$

$$\mathbf{S}_h = \frac{\sum_t f_h^t (\boldsymbol{x}^t - \boldsymbol{m}_h)(\boldsymbol{x}^t - \boldsymbol{m}_h)^T}{\sum_t f_h^t} \tag{12.42}$$

$$w_{ih} = \frac{\sum_t f_h^t r_i^t}{\sum_t f_h^t} \tag{12.43}$$

We see that w_{ih} is a weighted average where weights are the posterior probabilities of units, given the input and the desired output. In the case of classification, the M-step has no analytical solution and we need to resort to an iterative procedure, for example, gradient ascent (Jordan and Jacobs 1994).

12.7 Learning Vector Quantization

Let us say we have H units for each class, already labeled by those classes. These units are initialized with random instances from their classes. At each iteration, we find the unit, $\boldsymbol{m}_i$, that is closest to the input instance in Euclidean distance and use the following update rule:

$$\begin{cases} \Delta \boldsymbol{m}_i = \eta(\boldsymbol{x}^t - \boldsymbol{m}_i) & \text{if } \boldsymbol{x}^t \text{ and } \boldsymbol{m}_i \text{ have the same class label} \\ \Delta \boldsymbol{m}_i = -\eta(\boldsymbol{x}^t - \boldsymbol{m}_i) & \text{otherwise} \end{cases} \tag{12.44}$$

If the closest center has the correct label, it is moved toward the input to better represent it. If it belongs to the wrong class, it is moved away from the input in the expectation that if it is moved sufficiently away, a center of the correct class will be the closest in a future iteration. This is the *learning vector quantization* (LVQ) model proposed by Kohonen (1990, 1995).

LEARNING VECTOR QUANTIZATION

The LVQ update equation is analogous to equation 12.36 where the direction in which the center is moved depends on the difference between two values: Our prediction of the winner unit based on the input distances and what the winner should be based on the required output.

12.8 The Mixture of Experts

In RBFs, corresponding to each local patch we give a constant fit. In the case where for any input, we have one g_h 1 and all others 0, we get a piecewise constant approximation where for output i, the local fit by patch h is given by w_{ih}. From the Taylor expansion, we know that at each point, the function can be written as

$$f(x) = f(a) + (x - a)f'(a) + \cdots \tag{12.45}$$

Thus a constant approximation is good if x is close enough to a and $f'(a)$ is close to 0—that is, if $f(x)$ is flat around a. If this is not the case, we need to divide the space into a large number of patches, which is particularly serious when the input dimensionality is high, due to the curse of dimensionality.

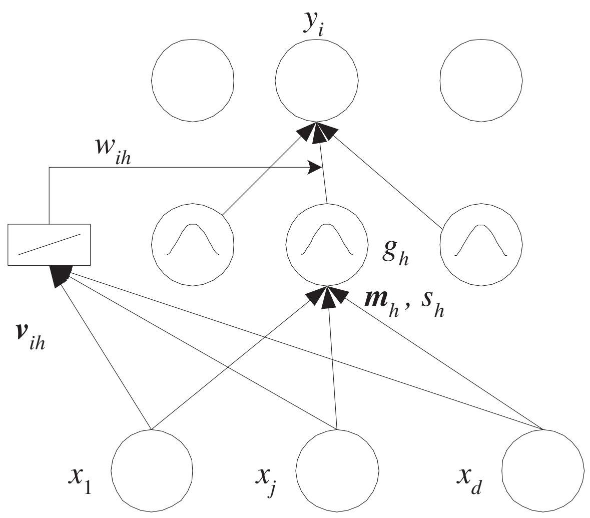


Figure 12.10 The mixture of experts can be seen as an RBF network where the second-layer weights are outputs of linear models. Only one linear model is shown for clarity.

PIECEWISE LINEAR APPROXIMATION

MIXTURE OF EXPERTS

An alternative is to have a *piecewise linear approximation* by taking into account the next term in the Taylor expansion, namely, the linear term. This is what is done by the *mixture of experts* (Jacobs et al. 1991). We write

$$y_i^t = \sum_{h=1}^{H} w_{ih} g_h^t \tag{12.46}$$

which is the same as equation 12.25 but here, w_{ih}, the contribution of patch h to output i is not a constant but a linear function of the input:

$$w_{ih}^t = \boldsymbol{v}_{ih}^T \boldsymbol{x}^t \tag{12.47}$$

$\boldsymbol{v}_{ih}$ is the parameter vector that defines the linear function and includes a bias term, making the mixture of experts a generalization of the RBF network. The unit activations can be taken as normalized RBFs:

$$g_h^t = \frac{\exp[-\|\boldsymbol{x}^t - \boldsymbol{m}_h\|^2/2s_h^2]}{\sum_l \exp[-\|\boldsymbol{x}^t - \boldsymbol{m}_l\|^2/2s_l^2]} \tag{12.48}$$

This can be seen as an RBF network except that the second-layer weights are not constants but are outputs of linear models (see figure 12.10). Ja-

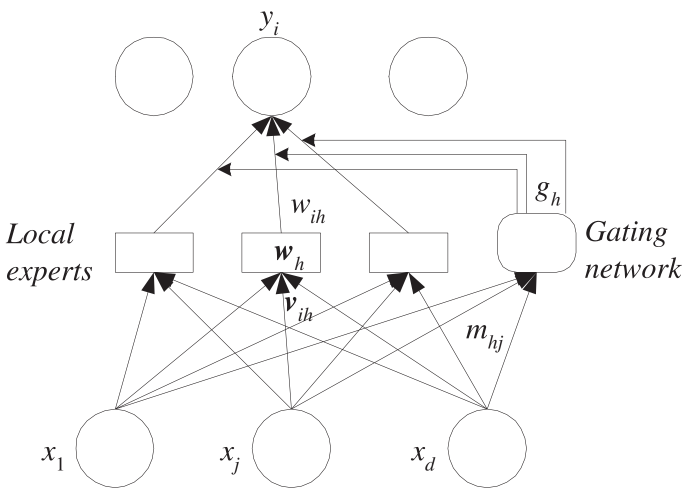


Figure 12.11 The mixture of experts can be seen as a model for combining multiple models. $\boldsymbol{w}_h$ are the models and the gating network is another model determining the weight of each model, as given by g_h. Viewed in this way, neither the experts nor the gating are restricted to be linear.

cobs et al. (1991) view this in another way: They consider $\boldsymbol{w}_h$ as linear models, each taking the input, and call them *experts*. g_h are considered to be the outputs of a *gating network*. The gating network works as a classifier does with its outputs summing to 1, assigning the input to one of the experts (see figure 12.11).

Considering the gating network in this manner, any classifier can be used in gating. When $\boldsymbol{x}$ is high-dimensional, using local Gaussian units may require a large number of experts and Jacobs et al. (1991) propose to take

$$g_h^t = \frac{\exp[\boldsymbol{m}_h^T \boldsymbol{x}^t]}{\sum_l \exp[\boldsymbol{m}_l^T \boldsymbol{x}^t]} \tag{12.49}$$

which is a linear classifier. Note that $\boldsymbol{m}_h$ are no longer centers but hyperplanes, and as such include bias values. This gating network is implementing a classification where it is dividing linearly the input region for which expert h is responsible from the expertise regions of other experts. As we will see again in chapter 17, the mixture of experts is a general architecture for combining multiple models; the experts and the gating may be nonlinear, for example, contain multilayer perceptrons, instead of linear perceptrons (exercise 6).

An architecture similar to the mixture of experts and running line smoother (section 8.8.3) has been proposed by Bottou and Vapnik (1992). In their approach, no training is done initially. When a test instance is given, a subset of the data close to the test instance is chosen from the training set (as in the k-nearest neighbor, but with a large k), a simple model, for example, a linear classifier, is trained with this local data, the prediction is made for the instance, and then the model is discarded. For the next instance, a new model is created, and so on. On a handwritten digit recognition application, this model has less error than the multilayer perceptron, k-nearest neighbor, and Parzen windows; the disadvantage is the need to train a new model on the fly for each test instance.

12.8.1 Cooperative Experts

In the cooperative case, y_i^t is given by equation 12.46, and we would like to make it as close as possible to the required output, r_i^t. In regression, the error function is

$$E(\{\boldsymbol{m}_h, s_h, w_{ih}\}_{i,h}|\mathcal{X}) = \frac{1}{2}\sum_t \sum_i (r_i^t - y_i^t)^2 \tag{12.50}$$

Using gradient descent, second-layer (expert) weight parameters are updated as

$$\Delta \boldsymbol{v}_{ih} = \eta \sum_t (r_i^t - y_i^t) g_h^t \boldsymbol{x}^t \tag{12.51}$$

Compared with equation 12.26, we see that the only difference is that this new update is a function of the input.

If we use softmax gating (equation 12.49), using gradient descent we have the following update rule for the hyperplanes:

$$\Delta m_{hj} = \eta \sum_t \sum_i (r_i^t - y_i^t)(w_{ih}^t - y_i^t) g_h^t x_j^t \tag{12.52}$$

If we use radial gating (equation 12.48), only the last term, $\partial p_h/\partial m_{hj}$, differs.

In classification, we have

$$y_i = \frac{\exp\left[\sum_h w_{ih} g_h^t\right]}{\sum_k \exp\left[\sum_h w_{kh} g_h^t\right]} \tag{12.53}$$

with $w_{ih} = \boldsymbol{v}_{ih}^T \boldsymbol{x}$, and update rules can be derived to minimize the cross-entropy using gradient descent (exercise 7).

12.8.2 Competitive Experts

Just like the competitive RBFs, we have

$$\mathcal{L}(\{\boldsymbol{m}_h, s_h, w_{ih}\}_{i,h}|\mathcal{X}) = \sum_t \log \sum_h g_h^t \exp\left[-\frac{1}{2}\sum_i (r_i^t - y_{ih}^t)^2\right] \tag{12.54}$$

where $y_{ih}^t = w_{ih}^t = \boldsymbol{v}_{ih}\boldsymbol{x}^t$. Using gradient ascent, we get

$$\Delta \boldsymbol{v}_{ih} = \eta \sum_t (r_i^t - y_{ih}^t) f_h^t \boldsymbol{x}^t \tag{12.55}$$

$$\Delta \boldsymbol{m}_h = \eta \sum_t (f_h^t - g_h^t) \boldsymbol{x}^t \tag{12.56}$$

assuming softmax gating as given in equation 12.49.

In classification, we have

$$\mathcal{L}(\{\boldsymbol{m}_h, s_h, w_{ih}\}_{i,h}|\mathcal{X}) = \sum_t \log \sum_h g_h^t \prod_i (y_{ih}^t)^{r_i^t} \tag{12.57}$$

$$= \sum_t \log \sum_h g_h^t \exp\left[\sum_i r_i^t \log y_{ih}^t\right] \tag{12.58}$$

where

$$y_{ih}^t = \frac{\exp w_{ih}^t}{\sum_k \exp w_{kh}^t} = \frac{\exp[\boldsymbol{v}_{ih}\boldsymbol{x}^t]}{\sum_k \exp[\boldsymbol{v}_{kh}\boldsymbol{x}^t]} \tag{12.59}$$

Jordan and Jacobs (1994) generalize EM for the competitive case with local linear models. Alpaydın and Jordan (1996) compare cooperative and competitive models for classification tasks and see that the cooperative model is generally more accurate but the competitive version learns faster. This is because in the cooperative case, models overlap more and implement a smoother approximation, and thus it is preferable in regression problems. The competitive model makes a harder split; generally only one expert is active for an input and therefore learning is faster.

12.9 Hierarchical Mixture of Experts

HIERARCHICAL MIXTURE OF EXPERTS

In figure 12.11, we see a set of experts and a gating network that chooses one of the experts as a function of the input. In a *hierarchical mixture* *of experts*, we replace each expert with a complete system of mixture of experts in a recursive manner (Jordan and Jacobs 1994). Once an architecture is chosen—namely, the depth, the experts, and the gating models—the whole tree can be learned from a labeled sample. Jordan and Jacobs (1994) derive both gradient descent and EM learning rules for such an architecture (see exercise 9).

We may also interpret this architecture as a decision tree (chapter 9) and its gating networks as decision nodes. In the decision trees we discussed earlier, a decision node makes a hard decision and takes one of the branches, so we take only one path from the root to one of the leaves. What we have here is a *soft decision tree* where, because the gating model returns us a probability, we take all the branches but with different probabilities; so we traverse all the paths to all the leaves and we take a weighted sum over all the leaf values where weights are equal to the product of the gating values on the path to each leaf. The advantage of this averaging is that the boundaries between leaf regions are no longer hard but there is a transition from one to the other and this smooths the response (İrsoy, Yıldız, and Alpaydın 2012).

12.10 Notes

An RBF network can be seen as a neural network, implemented by a network of simple processing units. It differs from a multilayer perceptron in that the first and second layers implement different functions. Omohundro (1987) discusses how local models can be implemented as neural networks and also addresses hierarchical data structures for fast localization of relevant local units. Specht (1991) shows how Parzen windows can be implemented as a neural network.

Platt (1991) proposed an incremental version of RBF where new units are added as necessary. Fritzke (1995) similarly proposed a growing version of SOM.

Lee (1991) compares k-nearest neighbor, multilayer perceptron, and RBF network on a handwritten digit recognition application and concludes that these three methods all have small error rates. RBF networks learn faster than backpropagation on a multilayer perceptron but use more parameters. Both of these methods are superior to the k-NN in terms of classification speed and memory need. Such practical constraints like time, memory, and computational complexity may be more important than small differences in error rate in real-world applications.

Kohonen's SOM (1990, 1995) was one of the most popular neural network methods, having been used in a variety of applications including exploratory data analysis and as a preprocessing stage before a supervised learner. One interesting and successful application is the traveling salesman problem (Angeniol, Vaubois, and Le Texier 1988). Just like the difference between *k*-means clustering and EM on Gaussian mixtures (chapter 7), *generative topographic mapping* (GTM) (Bishop, Svensén, and Williams 1998) is a probabilistic version of SOM that optimizes the log likelihood of the data using a mixture of Gaussians whose means are constrained to lie on a two-dimensional manifold (for topological ordering in low dimensions).

GENERATIVE TOPOGRAPHIC MAPPING

In an RBF network, once the centers and spreads are fixed (e.g., by choosing a random subset of training instances as centers, as in the anchor method), training the second layer is a linear model. This model is equivalent to support vector machines with Gaussian kernels where during learning the best subset of instances, named the *support vectors*, are chosen; we discuss them in chapter 13. Gaussian processes (chapter 16) where we interpolate from stored training instances are also similar.

12.11 Exercises

1. Show an RBF network that implements XOR.

   SOLUTION: There are two possibilities (see figure 12.12): (a) We can have two circular Gaussians centered on the two positive instances and the second layer ORs them, or (b) we can have one elliptic Gaussian centered on (0.5, 0.5) with negative correlation to cover the two positive instances.

2. Write down the RBF network that uses elliptic units instead of radial units as in equation 12.13.

   SOLUTION:

   $$p_h^t = \exp\left[-\frac{1}{2}(\boldsymbol{x}^t - \boldsymbol{m}_h)^T \mathbf{S}_h^{-1}(\boldsymbol{x}^t - \boldsymbol{m}_h)\right]$$

   where $\mathbf{S}_h$ is the local covariance matrix.

3. Derive the update equations for the RBF network for classification (equations 12.20 and 12.21).

4. Show how the system given in equation 12.22 can be trained.

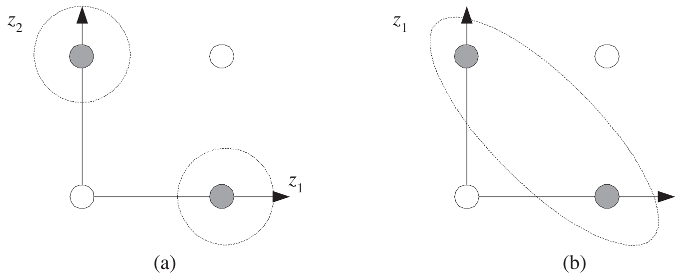


Figure 12.12 Two ways of implementing XOR with RBF.

5. Compare the number of parameters of a mixture of experts architecture with an RBF network.

   SOLUTION: With d inputs, K classes and H Gaussians, an RBF network needs $H \cdot d$ parameters for the centers, H parameters for the spreads and $(H + 1)K$ parameters for the second-layer weights. For the case of the MoE, for each second-layer weight, we need a $d + 1$ dimensional vector of the linear model, but there is no bias; hence we have $HK(d + 1)$ parameters.

   Note that the number of parameters in the first layer is the same with RBF and it is the same whether we have Gaussian or softmax gating: For each hidden unit, in the case of Gaussian gating, we need d parameters for the center and 1 for the spread; in the case of softmax gating, the linear model has $d + 1$ parameters (d inputs and a bias).

6. Formalize a mixture of experts architecture where the experts and the gating network are multilayer perceptrons. Derive the update equations for regression and classification.

7. Derive the update equations for the cooperative mixture of experts for classification.

8. Derive the update equations for the competitive mixture of experts for classification.

9. Formalize the hierarchical mixture of experts architecture with two levels. Derive the update equations using gradient descent for regression and classification.

   SOLUTION: The following is from Jordan and Jacobs 1994; notation is slightly changed to match the notation of the book.

   Let us see the case of regression with a single output: y is the overall output, y_i are the outputs on the first level, and y_{ij} are the outputs on the second level, which are the leaves in a model with two levels. Similarly, g_i are the gating outputs on the first level and $g_{j|i}$ are the outputs on the second level, that is, the gating value of expert j on the second level given that we have chosen the branch i on the first level:

$$\begin{aligned}
y &= \sum_i g_i y_i \\
y_i &= \sum_j g_{j|i} y_{ij} \quad \text{and} \quad g_i = \frac{\exp \boldsymbol{m}_i^T \boldsymbol{x}}{\sum_k \exp \boldsymbol{m}_k^T \boldsymbol{x}} \\
y_{ij} &= \boldsymbol{v}_{ij}^T \boldsymbol{x} \quad \text{and} \quad g_{j|i} = \frac{\exp \boldsymbol{m}_{ij}^T \boldsymbol{x}}{\sum_l \exp \boldsymbol{m}_{il}^T \boldsymbol{x}}
\end{aligned}$$

In regression, the error to be minimized is as follows (note that we are using a competitive version here):

$$E = \sum_t \log \sum_i g_i^t \sum_j g_{j|i}^t \exp\left[-\frac{1}{2}(r^t - y_{ij}^t)^2\right]$$

and using gradient descent, we get the following update equations:

$$\begin{aligned}
\Delta \boldsymbol{v}_{ij} &= \eta \sum_t f_i^t f_{j|i}^t (r^t - y^t) \boldsymbol{x}^t \\
\Delta \boldsymbol{m}_i &= \eta \sum_t (f_i^t - g_i^t) \boldsymbol{x}^t \\
\Delta \boldsymbol{m}_{ij} &= \eta \sum_t f_i^t (f_{j|i}^t - g_{j|i}^t) \boldsymbol{x}^t
\end{aligned}$$

where we make use of the following posteriors:

$$\begin{aligned}
f_i^t &= \frac{g_i^t \sum_j g_{j|i}^t \exp[-(1/2)(r^t - y_{ij}^t)^2]}{\sum_k g_k^t \sum_j g_{j|k}^t \exp[-(1/2)(r^t - y_{kj}^t)^2]} \\
f_{j|i}^t &= \frac{g_{j|i}^t \exp[-(1/2)(r^t - y_{ij}^t)^2]}{\sum_l g_{l|i}^t \exp[-(1/2)(r^t - y_{il}^t)^2]} \\
f_{ij}^t &= \frac{g_i^t g_{j|i}^t \exp[-(1/2)(r^t - y_{ij}^t)^2]}{\sum_k g_k^t \sum_l g_{l|k}^t \exp[-(1/2)(r^t - y_{kl}^t)^2]}
\end{aligned}$$

Note how we multiply the gating values on the path starting from the root to a leaf expert.

For the case of classification with $K > 2$ classes, one possibility is to have K separate HMEs as above (having single output experts), whose outputs we softmax to maximize the log likelihood:

$$\begin{aligned}
\mathcal{L} &= \sum_t \log \sum_i g_i^t \sum_j g_{j|i}^t \exp\left[\sum_c r_c^t \log p_c^t\right] \\
p_c^t &= \frac{\exp y_c^t}{\sum_k \exp y_k^t}
\end{aligned}$$

where each y_c^t denotes the output of one single-output HME. The more interesting case of a single multiclass HME where experts have K softmax outputs is discussed in Waterhouse and Robinson 1994.

10. In the mixture of experts, because different experts specialize in different parts of the input space, they may need to focus on different inputs. Discuss how dimensionality can be locally reduced in the experts.

12.12 References

Alpaydın, E., and M. I. Jordan. 1996. “Local Linear Perceptrons for Classification.” *IEEE Transactions on Neural Networks* 7:788–792.

Angeniol, B., G. Vaubois, and Y. Le Texier. 1988. “Self Organizing Feature Maps and the Travelling Salesman Problem.” *Neural Networks* 1:289–293.

Berthold, M. 1999. “Fuzzy Logic.” In *Intelligent Data Analysis: An Introduction*, ed. M. Berthold and D. J. Hand, 269–298. Berlin: Springer.

Bishop, C. M., M. Svensén, and C. K. I. Williams. 1998. “GTM: The Generative Topographic Mapping.” *Neural Computation* 10:215–234.

Bottou, L., and V. Vapnik. 1992. “Local Learning Algorithms.” *Neural Computation* 4:888–900.

Broomhead, D. S., and D. Lowe. 1988. “Multivariable Functional Interpolation and Adaptive Networks.” *Complex Systems* 2:321–355.

Carpenter, G. A., and S. Grossberg. 1988. “The ART of Adaptive Pattern Recognition by a Self-Organizing Neural Network.” *IEEE Computer* 21 (3): 77–88.

Cherkassky, V., and F. Mulier. 1998. *Learning from Data: Concepts, Theory, and Methods.* New York: Wiley.

DeSieno, D. 1988. “Adding a Conscience Mechanism to Competitive Learning.” In *IEEE International Conference on Neural Networks*, 117–124. Piscataway, NJ: IEEE Press.

Feldman, J. A., and D. H. Ballard. 1982. “Connectionist Models and their Properties.” *Cognitive Science* 6:205–254.

Fritzke, B. 1995. “Growing Cell Structures: A Self Organizing Network for Unsupervised and Supervised Training.” *Neural Networks* 7:1441–1460.

Grossberg, S. 1980. “How Does the Brain Build a Cognitive Code?” *Psychological Review* 87:1–51.

Hertz, J., A. Krogh, and R. G. Palmer. 1991. *Introduction to the Theory of Neural Computation.* Reading, MA: Addison-Wesley.

İrsoy, O., O. T. Yıldız, and E. Alpaydın. 2012. “Soft Decision Trees.” In *International Conference on Pattern Recognition*, 1819–1822. Piscataway, NJ: IEEE Press.

Jacobs, R. A., M. I. Jordan, S. J. Nowlan, and G. E. Hinton. 1991. “Adaptive Mixtures of Local Experts.” *Neural Computation* 3:79–87.

Jordan, M. I., and R. A. Jacobs. 1994. “Hierarchical Mixtures of Experts and the EM Algorithm.” *Neural Computation* 6:181–214.

Kohonen, T. 1990. “The Self-Organizing Map.” *Proceedings of the IEEE* 78:1464–1480.

Kohonen, T. 1995. *Self-Organizing Maps*. Berlin: Springer.

Lee, Y. 1991. “Handwritten Digit Recognition Using *k*-Nearest Neighbor, Radial Basis Function, and Backpropagation Neural Networks.” *Neural Computation* 3:440–449.

Mao, J., and A. K. Jain. 1995. “Artificial Neural Networks for Feature Extraction and Multivariate Data Projection.” *IEEE Transactions on Neural Networks* 6:296–317.

Moody, J., and C. Darken. 1989. “Fast Learning in Networks of Locally-Tuned Processing Units.” *Neural Computation* 1:281–294.

Oja, E. 1982. “A Simplified Neuron Model as a Principal Component Analyzer.” *Journal of Mathematical Biology* 15:267–273.

Omohundro, S. M. 1987. “Efficient Algorithms with Neural Network Behavior.” *Complex Systems* 1:273–347.

Platt, J. 1991. “A Resource Allocating Network for Function Interpolation.” *Neural Computation* 3:213–225.

Specht, D. F. 1991. “A General Regression Neural Network.” *IEEE Transactions on Neural Networks* 2:568–576.

Tresp, V., J. Hollatz, and S. Ahmad. 1997. “Representing Probabilistic Rules with Networks of Gaussian Basis Functions.” *Machine Learning* 27:173–200.

Waterhouse, S. R., and A. J. Robinson. 1994. “Classification Using Hierarchical Mixtures of Experts.” In *IEEE Workshop on Neural Networks for Signal Processing*, 177–186. Piscataway, NJ: IEEE Press.

13 *Kernel Machines*

Kernel machines are maximum margin methods that allow the model to be written as a sum of the influences of a subset of the training instances. These influences are given by application-specific similarity kernels, and we discuss "kernelized" classification, regression, ranking, outlier detection and dimensionality reduction, and how to choose and use kernels.

13.1 Introduction

We now discuss a different approach for linear classification and regression. We should not be surprised to have so many different methods even for the simple case of a linear model. Each learning algorithm has a different inductive bias, makes different assumptions, and defines a different objective function and thus may find a different linear model.

The model that we will discuss in this chapter, called the *support vector machine* (SVM), and later generalized under the name *kernel machine*, has been popular in recent years for a number of reasons:

1. It is a discriminant-based method and uses Vapnik's principle to never solve a more complex problem as a first step before the actual problem (Vapnik 1995). For example, in classification, when the task is to learn the discriminant, it is not necessary to estimate where the class densities $p(\boldsymbol{x}|C_i)$ or the exact posterior probability values $P(C_i|\boldsymbol{x})$; we only need to estimate where the class boundaries lie, that is, $\boldsymbol{x}$ where $P(C_i|\boldsymbol{x}) = P(C_j|\boldsymbol{x})$. Similarly, for outlier detection, we do not need to estimate the full density $p(\boldsymbol{x})$; we only need to find the boundary separating those $\boldsymbol{x}$ that have low $p(\boldsymbol{x})$, that is, $\boldsymbol{x}$ where $p(\boldsymbol{x}) < \theta$, for some threshold $\theta \in (0, 1)$.

2. After training, the parameter of the linear model, the weight vector, can be written down in terms of a subset of the training set, which are the so-called *support vectors*. In classification, these are the cases that are close to the boundary and as such, knowing them allows knowledge extraction: Those are the uncertain or erroneous cases that lie in the vicinity of the boundary between two classes. Their number gives us an estimate of the generalization error, and, as we see below, being able to write the model parameter in terms of a set of instances allows kernelization.

3. As we will see shortly, the output is written as a sum of the influences of support vectors and these are given by *kernel functions* that are application-specific measures of similarity between data instances. Previously, we talked about nonlinear basis functions allowing us to map the input to another space where a linear (smooth) solution is possible; the kernel function uses the same idea.

4. Typically in most learning algorithms, data points are represented as vectors, and either dot product (as in the multilayer perceptrons) or Euclidean distance (as in radial basis function networks) is used. A kernel function allows us to go beyond that. For example, G_1 and G_2 may be two graphs and $K(G_1, G_2)$ may correspond to the number of shared paths, which we can calculate without needing to represent G_1 or G_2 explicitly as vectors.

5. Kernel-based algorithms are formulated as convex optimization problems, and there is a single optimum that we can solve for analytically. Therefore we are no longer bothered with heuristics for learning rates, initializations, checking for convergence, and such. Of course, this does not mean that we do not have any hyperparameters for model selection; we do—any method needs them, to match the algorithm to the data at hand.

We start our discussion with the case of classification, and then generalize to regression, ranking, outlier (novelty) detection, and then dimensionality reduction. We see that in all cases basically we have the similar quadratic program template to maximize the separability, or *margin*, of instances subject to a constraint of the smoothness of solution. Solving for it, we get the support vectors. The kernel function defines the space according to its notion of similarity and a kernel function is good if we have better separation in its corresponding space.

13.2 Optimal Separating Hyperplane

Let us start again with two classes and use labels $-1/+1$ for the two classes. The sample is $\mathcal{X} = \{\boldsymbol{x}^t, r^t\}$ where $r^t = +1$ if $\boldsymbol{x}^t \in C_1$ and $r^t = -1$ if $\boldsymbol{x}^t \in C_2$. We would like to find $\boldsymbol{w}$ and w_0 such that

$$\begin{aligned} \boldsymbol{w}^T\boldsymbol{x}^t + w_0 &\geq +1 \quad \text{for} \quad r^t = +1 \\ \boldsymbol{w}^T\boldsymbol{x}^t + w_0 &\leq -1 \quad \text{for} \quad r^t = -1 \end{aligned}$$

which can be rewritten as

$$r^t(\boldsymbol{w}^T\boldsymbol{x}^t + w_0) \geq +1 \tag{13.1}$$

Note that we do not simply require

$$r^t(\boldsymbol{w}^T\boldsymbol{x}^t + w_0) \geq 0$$

Not only do we want the instances to be on the right side of the hyperplane, but we also want them some distance away, for better generalization. The distance from the hyperplane to the instances closest to it on either side is called the *margin*, which we want to maximize for best generalization. MARGIN

Very early on, in section 2.1, we talked about the concept of the margin when we were talking about fitting a rectangle, and we said that it is better to take a rectangle halfway between S and G, to get a breathing space. This is so that in case noise shifts a test instance slightly, it will still be on the right side of the boundary.

OPTIMAL SEPARATING HYPERPLANE

Similarly, now that we are using the hypothesis class of lines, the *optimal separating hyperplane* is the one that maximizes the margin.

We remember from section 10.3 that the distance of $\boldsymbol{x}^t$ to the discriminant is

$$\frac{|\boldsymbol{w}^T\boldsymbol{x}^t + w_0|}{\|\boldsymbol{w}\|}$$

which, when $r^t \in \{-1, +1\}$, can be written as

$$\frac{r^t(\boldsymbol{w}^T\boldsymbol{x}^t + w_0)}{\|\boldsymbol{w}\|}$$

and we would like this to be at least some value ρ:

$$\frac{r^t(\boldsymbol{w}^T\boldsymbol{x}^t + w_0)}{\|\boldsymbol{w}\|} \geq \rho, \forall t \tag{13.2}$$

We would like to maximize ρ but there are an infinite number of solutions that we can get by scaling $\boldsymbol{w}$ and for a unique solution, we fix $\rho\|\boldsymbol{w}\| = 1$ and thus, to maximize the margin, we minimize $\|\boldsymbol{w}\|$. The task can therefore be defined (see Cortes and Vapnik 1995; Vapnik 1995) as to

$$\min \frac{1}{2}\|\boldsymbol{w}\|^2 \text{ subject to } r^t(\boldsymbol{w}^T\boldsymbol{x}^t + w_0) \geq +1, \forall t \tag{13.3}$$

This is a standard quadratic optimization problem, whose complexity depends on d, and it can be solved directly to find $\boldsymbol{w}$ and w_0. Then, on both sides of the hyperplane, there will be instances that are $1/\|\boldsymbol{w}\|$ away from the hyperplane and the total margin will be $2/\|\boldsymbol{w}\|$.

We saw in section 10.2 that if the problem is not linearly separable, instead of fitting a nonlinear function, one trick is to map the problem to a new space by using nonlinear basis functions. It is generally the case that this new space has many more dimensions than the original space, and, in such a case, we are interested in a method whose complexity does not depend on the input dimensionality.

In finding the optimal hyperplane, we can convert the optimization problem to a form whose complexity depends on N, the number of training instances, and not on d. Another advantage of this new formulation is that it will allow us to rewrite the basis functions in terms of kernel functions, as we will see in section 13.5.

To get the new formulation, we first write equation 13.3 as an unconstrained problem using Lagrange multipliers α^t:

$$\begin{aligned} L_p &= \frac{1}{2}\|\boldsymbol{w}\|^2 - \sum_{t=1}^{N} \alpha^t[r^t(\boldsymbol{w}^T\boldsymbol{x}^t + w_0) - 1] \\ &= \frac{1}{2}\|\boldsymbol{w}\|^2 - \sum_t \alpha^t r^t(\boldsymbol{w}^T\boldsymbol{x}^t + w_0) + \sum_t \alpha^t \end{aligned} \tag{13.4}$$

This should be minimized with respect to $\boldsymbol{w}, w_0$ and maximized with respect to $\alpha^t \geq 0$. The saddle point gives the solution.

This is a convex quadratic optimization problem because the main term is convex and the linear constraints are also convex. Therefore, we can equivalently solve the dual problem, making use of the Karush-Kuhn-Tucker conditions. The dual is to *maximize* L_p with respect to α^t, subject to the constraints that the gradient of L_p with respect to $\boldsymbol{w}$ and w_0 are 0 and also that $\alpha^t \geq 0$:

$$\frac{\partial L_p}{\partial \boldsymbol{w}} = 0 \quad \Rightarrow \quad \boldsymbol{w} = \sum_t \alpha^t r^t \boldsymbol{x}^t \tag{13.5}$$

$$\frac{\partial L_p}{\partial w_0} = 0 \quad \Rightarrow \quad \sum_t \alpha^t r^t = 0 \tag{13.6}$$

Plugging these into equation 13.4, we get the dual

$$\begin{aligned} L_d &= \frac{1}{2}(\boldsymbol{w}^T\boldsymbol{w}) - \boldsymbol{w}^T \sum_t \alpha^t r^t \boldsymbol{x}^t - w_0 \sum_t \alpha^t r^t + \sum_t \alpha^t \\ &= -\frac{1}{2}(\boldsymbol{w}^T\boldsymbol{w}) + \sum_t \alpha^t \\ &= -\frac{1}{2}\sum_t \sum_s \alpha^t \alpha^s r^t r^s (\boldsymbol{x}^t)^T \boldsymbol{x}^s + \sum_t \alpha^t \qquad (13.7) \end{aligned}$$

which we maximize with respect to α^t only, subject to the constraints

$$\sum_t \alpha^t r^t = 0, \text{ and } \alpha^t \geq 0, \forall t$$

This can be solved using quadratic optimization methods. The size of the dual depends on N, sample size, and not on d, the input dimensionality. The upper bound for time complexity is $\mathcal{O}(N^3)$, and the upper bound for space complexity is $\mathcal{O}(N^2)$.

Once we solve for α^t, we see that though there are N of them, most vanish with $\alpha^t = 0$ and only a small percentage have $\alpha^t > 0$. The set of $\boldsymbol{x}^t$ whose $\alpha^t > 0$ are the *support vectors*, and as we see in equation 13.5, $\boldsymbol{w}$ is written as the weighted sum of these training instances that are selected as the support vectors. These are the $\boldsymbol{x}^t$ that satisfy

$$r^t(\boldsymbol{w}^T\boldsymbol{x}^t + w_0) = 1$$

and lie on the margin. We can use this fact to calculate w_0 from any support vector as

$$w_0 = r^t - \boldsymbol{w}^T\boldsymbol{x}^t \tag{13.8}$$

For numerical stability, it is advised that this be done for all support vectors and an average be taken. The discriminant thus found is called the *support vector machine* (SVM) (see figure 13.1).

SUPPORT VECTOR MACHINE

The majority of the α^t are 0, for which $r^t(\boldsymbol{w}^T\boldsymbol{x}^t + w_0) > 1$. These are the $\boldsymbol{x}^t$ that lie more than sufficiently away from the discriminant,

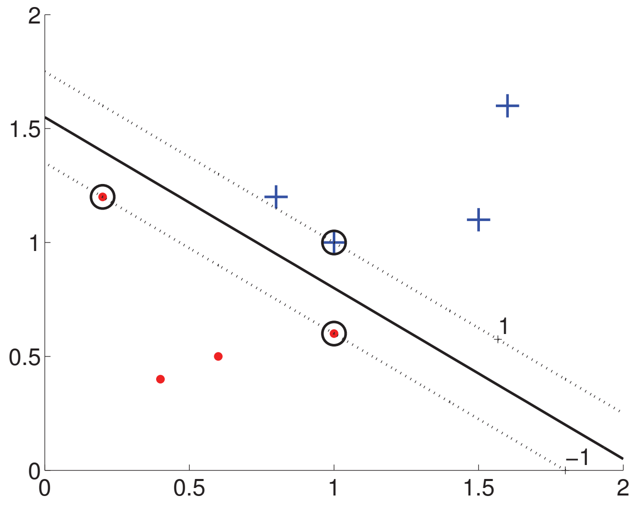


Figure 13.1 For a two-class problem where the instances of the classes are shown by plus signs and dots, the thick line is the boundary and the dashed lines define the margins on either side. Circled instances are the support vectors.

and they have no effect on the hyperplane. The instances that are not support vectors carry no information; even if any subset of them are removed, we would still get the same solution. From this perspective, the SVM algorithm can be likened to the condensed nearest neighbor algorithm (section 8.5), which stores only the instances neighboring (and hence constraining) the class discriminant.

Being a discriminant-based method, the SVM cares only about the instances close to the boundary and discards those that lie in the interior. Using this idea, it is possible to use a simpler classifier before the SVM to filter out a large portion of such instances, thereby decreasing the complexity of the optimization step of the SVM (exercise 1).

During testing, we do not enforce a margin. We calculate $g(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x} + w_0$, and choose according to the sign of $g(\boldsymbol{x})$:

Choose C_1 if $g(\boldsymbol{x}) > 0$ and C_2 otherwise

13.3 The Nonseparable Case: Soft Margin Hyperplane

If the data is not linearly separable, the algorithm we discussed earlier will not work. In such a case, if the two classes are not linearly separable such that there is no hyperplane to separate them, we look for the one that incurs the least error. We define *slack variables*, $\xi^t \geq 0$, which store the deviation from the margin. There are two types of deviation: An instance may lie on the wrong side of the hyperplane and be misclassified. Or, it may be on the right side but may lie in the margin, namely, not sufficiently away from the hyperplane. Relaxing equation 13.1, we require

SLACK VARIABLES

$$r^t(\boldsymbol{w}^T\boldsymbol{x}^t + w_0) \geq 1 - \xi^t \tag{13.9}$$

If $\xi^t = 0$, there is no problem with $\boldsymbol{x}^t$. If $0 < \xi^t < 1$, $\boldsymbol{x}^t$ is correctly classified but in the margin. If $\xi^t \geq 1$, $\boldsymbol{x}^t$ is misclassified (see figure 13.2). The number of misclassifications is $\#\{\xi^t > 1\}$, and the number of non-separable points is $\#\{\xi_t > 0\}$. We define *soft error* as

SOFT ERROR

$$\sum_t \xi^t$$

and add this as a penalty term:

$$L_p = \frac{1}{2}\|\boldsymbol{w}\|^2 + C\sum_t \xi^t \tag{13.10}$$

subject to the constraint of equation 13.9. C is the penalty factor as in any regularization scheme trading off complexity, as measured by the L_2 norm of the weight vector (similar to weight decay in multilayer perceptrons; see sectiona 11.9 and 11.10), and data misfit, as measured by the number of nonseparable points. Note that we are penalizing not only the misclassified points but also the ones in the margin for better generalization, though these latter would be correctly classified during testing.

Adding the constraints, the Lagrangian of equation 13.4 then becomes

$$L_p = \frac{1}{2}\|\boldsymbol{w}\|^2 + C\sum_t \xi^t - \sum_t \alpha^t[r^t(\boldsymbol{w}^T\boldsymbol{x}^t + w_0) - 1 + \xi^t] - \sum_t \mu^t\xi^t \tag{13.11}$$

where μ_t are the new Lagrange parameters to guarantee the positivity of ξ^t. When we take the derivatives with respect to the parameters and set them to 0, we get

$$\frac{\partial L_p}{\partial \boldsymbol{w}} = \boldsymbol{w} - \sum_t \alpha^t r^t \boldsymbol{x}^t = 0 \Rightarrow \boldsymbol{w} = \sum_t \alpha^t r^t \boldsymbol{x}^t \tag{13.12}$$

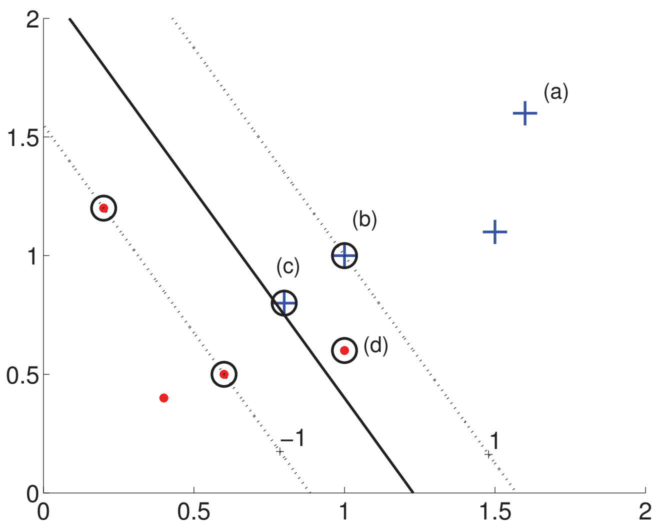


Figure 13.2 In classifying an instance, there are four possible cases: In (a), the instance is on the correct side and far away from the margin; $r^t g(\boldsymbol{x}^t) > 1$, $\xi^t = 0$. In (b), $\xi^t = 0$; it is on the right side and on the margin. In (c), $\xi^t = 1 - g(\boldsymbol{x}^t), 0 < \xi < 1$; it is on the right side but is in the margin and not sufficiently away. In (d), $\xi^t = 1 + g(\boldsymbol{x}^t) > 1$; it is on the wrong side—this is a misclassification. All cases except (a) are support vectors. In terms of the dual variable, in (a), $\alpha^t = 0$; in (b), $\alpha^t < C$; in (c) and (d), $\alpha^t = C$.

$$\frac{\partial L_p}{\partial w_0} = \sum_t \alpha^t r^t = 0 \tag{13.13}$$

$$\frac{\partial L_p}{\partial \xi^t} = C - \alpha^t - \mu^t = 0 \tag{13.14}$$

Since $\mu^t \geq 0$, this last implies that $0 \leq \alpha^t \leq C$. Plugging these into equation 13.11, we get the dual that we maximize with respect to α^t:

$$L_d = \sum_t \alpha^t - \frac{1}{2}\sum_t \sum_s \alpha^t \alpha^s r^t r^s (\boldsymbol{x}^t)^T \boldsymbol{x}^s \tag{13.15}$$

subject to

$$\sum_t \alpha^t r^t = 0 \text{ and } 0 \leq \alpha^t \leq C, \forall t$$

Solving this, we see that as in the separable case, instances that lie on the correct side of the boundary with sufficient margin vanish with their $\alpha^t = 0$ (see figure 13.2). The support vectors have their $\alpha^t > 0$ and they define $\boldsymbol{w}$, as given in equation 13.12. Of these, those whose $\alpha^t < C$ are the ones that are on the margin, and we can use them to calculate w_0; they have $\xi^t = 0$ and satisfy $r^t(\boldsymbol{w}^T\boldsymbol{x}^t + w_0) = 1$. Again, it is better to take an average over these w_0 estimates. Those instances that are in the margin or misclassified have their $\alpha^t = C$.

The nonseparable instances that we store as support vectors are the instances that we would have trouble correctly classifying if they were not in the training set; they would either be misclassified or classified correctly but not with enough confidence. We can say that the number of support vectors is an upper-bound estimate for the expected number of errors. And, actually, Vapnik (1995) has shown that the expected test error rate is

$$E_N[P(error)] \le \frac{E_N[\#\text{ of support vectors}]}{N}$$

where $E_N[\cdot]$ denotes expectation over training sets of size N. The nice implication of this is that it shows that the error rate depends on the number of support vectors and not on the input dimensionality.

HINGE LOSS

Equation 13.9 implies that we define error if the instance is on the wrong side or if the margin is less than 1. This is called the *hinge loss*. If $y^t = \boldsymbol{w}^T\boldsymbol{x}^t + w_0$ is the output and r^t is the desired output, hinge loss is defined as

$$L_{hinge}(y^t, r^t) = \begin{cases} 0 & \text{if } y^t r^t \ge 1 \\ 1 - y^t r^t & \text{otherwise} \end{cases} \tag{13.16}$$

In figure 13.3, we compare hinge loss with 0/1 loss, squared error, and cross-entropy. We see that unlike 0/1 loss, hinge loss also penalizes instances in the margin even though they may be on the correct side, and the loss increases linearly as the instance moves away on the wrong side. This is different from the squared loss that therefore is not as robust as the hinge loss. We see that cross-entropy minimized in logistic discrimination (section 10.7) or by the linear perceptron (section 11.3) is a good continuous approximation to the hinge loss.

C of equation 13.10 is the regularization parameter fine-tuned using cross-validation. It defines the trade-off between margin maximization and error minimization: If it is too large, we have a high penalty for nonseparable points, and we may store many support vectors and overfit.

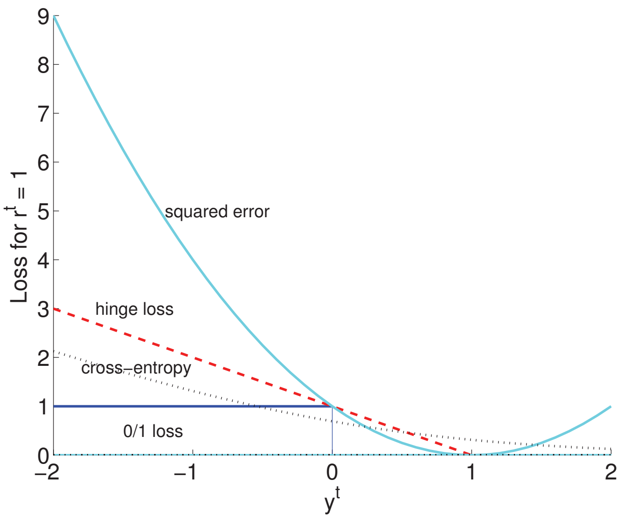


Figure 13.3 Comparison of different loss functions for $r^t = 1$: 0/1 loss is 0 if $y^t = 1$, 1 otherwise. Hinge loss is 0 if $y^t > 1$, $1 - y^t$ otherwise. Squared error is $(1 - y^t)^2$. Cross-entropy is $\log(1/(1 + \exp(-y^t)))$.

If it is too small, we may find too simple solutions that underfit. Typically, one chooses from $[10^{-6}, 10^{-5}, \ldots, 10^{+5}, 10^{+6}]$ in the log scale by looking at the accuracy on a validation set.

13.4 ν-SVM

There is another, equivalent formulation of the soft margin hyperplane that uses a parameter $\nu \in [0, 1]$ instead of C (Schölkopf et al. 2000). The objective function is

$$\min \frac{1}{2}\|\boldsymbol{w}\|^2 - \nu\rho + \frac{1}{N}\sum_t \xi^t \tag{13.17}$$

subject to

$$r^t(\boldsymbol{w}^T\boldsymbol{x}^t + w_0) \geq \rho - \xi^t,\ \xi^t \geq 0,\ \rho \geq 0 \tag{13.18}$$

ρ is a new parameter that is a variable of the optimization problem and scales the margin: The margin is now $2\rho/\|\boldsymbol{w}\|$. ν has been shown to be a lower bound on the fraction of support vectors and an upper bound on the fraction of instances having margin errors ($\sum_t \#\{\xi^t > 0\}$). The dual is

$$L_d = -\frac{1}{2}\sum_t \sum_s \alpha^t \alpha^s r^t r^s (\boldsymbol{x}^t)^T \boldsymbol{x}^s \tag{13.19}$$

subject to

$$\sum_t \alpha^t r^t = 0,\ 0 \le \alpha^t \le \frac{1}{N},\ \sum_t \alpha^t \ge \nu$$

When we compare equation 13.19 with equation 13.15, we see that the term $\sum_t \alpha^t$ no longer appears in the objective function but is now a constraint. By playing with ν, we can control the fraction of support vectors, and this is advocated to be more intuitive than playing with C.

13.5 Kernel Trick

Section 10.2 demonstrated that if the problem is nonlinear, instead of trying to fit a nonlinear model, we can map the problem to a new space by doing a nonlinear transformation using suitably chosen basis functions and then use a linear model in this new space. The linear model in the new space corresponds to a nonlinear model in the original space. This approach can be used in both classification and regression problems, and in the special case of classification, it can be used with any scheme. In the particular case of support vector machines, it leads to certain simplifications that we now discuss.

Let us say we have the new dimensions calculated through the basis functions

$$\boldsymbol{z} = \boldsymbol{\phi}(\boldsymbol{x}) \text{ where } z_j = \phi_j(\boldsymbol{x}), j = 1, \ldots, k$$

mapping from the d-dimensional $\boldsymbol{x}$ space to the k-dimensional $\boldsymbol{z}$ space where we write the discriminant as

$$\begin{aligned} g(\boldsymbol{z}) &= \boldsymbol{w}^T \boldsymbol{z} \\ g(\boldsymbol{x}) &= \boldsymbol{w}^T \boldsymbol{\phi}(\boldsymbol{x}) \\ &= \sum_{j=1}^{k} w_j \phi_j(\boldsymbol{x}) \end{aligned} \tag{13.20}$$

where we do not use a separate w_0; we assume that $z_1 = \phi_1(\boldsymbol{x}) \equiv 1$. Generally, k is much larger than d and k may also be larger than N, and there lies the advantage of using the dual form whose complexity depends on N, whereas if we used the primal it would depend on k. We also use the more general case of the soft margin hyperplane here because we have no guarantee that the problem is linearly separable in this new space.

The problem is the same

$$L_p = \frac{1}{2}\|\boldsymbol{w}\|^2 + C\sum_t \xi^t \tag{13.21}$$

except that now the constraints are defined in the new space

$$r^t \boldsymbol{w}^T \boldsymbol{\phi}(\boldsymbol{x}^t) \geq 1 - \xi^t \tag{13.22}$$

The Lagrangian is

$$L_p = \frac{1}{2}\|\boldsymbol{w}\|^2 + C\sum_t \xi^t - \sum_t \alpha^t \left[r^t \boldsymbol{w}^T \boldsymbol{\phi}(\boldsymbol{x}^t) - 1 + \xi^t\right] - \sum_t \mu^t \xi^t \tag{13.23}$$

When we take the derivatives with respect to the parameters and set them to 0, we get

$$\frac{\partial L_p}{\partial \boldsymbol{w}} = \boldsymbol{w} = \sum_t \alpha^t r^t \boldsymbol{\phi}(\boldsymbol{x}^t) \tag{13.24}$$

$$\frac{\partial L_p}{\partial \xi^t} = C - \alpha^t - \mu^t = 0 \tag{13.25}$$

The dual is now

$$L_d = \sum_t \alpha^t - \frac{1}{2}\sum_t\sum_s \alpha^t \alpha^s r^t r^s \boldsymbol{\phi}(\boldsymbol{x}^t)^T \boldsymbol{\phi}(\boldsymbol{x}^s) \tag{13.26}$$

subject to

$$\sum_t \alpha^t r^t = 0 \text{ and } 0 \leq \alpha^t \leq C, \forall t$$

KERNEL FUNCTION

The idea in *kernel machines* is to replace the inner product of basis functions, $\boldsymbol{\phi}(\boldsymbol{x}^t)^T\boldsymbol{\phi}(\boldsymbol{x}^s)$, by a *kernel function*, $K(\boldsymbol{x}^t, \boldsymbol{x}^s)$, between instances in the original input space. So instead of mapping two instances $\boldsymbol{x}^t$ and $\boldsymbol{x}^s$ to the $\boldsymbol{z}$-space and doing a dot product there, we directly apply the kernel function in the original space.

$$L_d = \sum_t \alpha^t - \frac{1}{2}\sum_t\sum_s \alpha^t \alpha^s r^t r^s K(\boldsymbol{x}^t, \boldsymbol{x}^s) \tag{13.27}$$

The kernel function also shows up in the discriminant

$$\begin{aligned} g(\boldsymbol{x}) &= \boldsymbol{w}^T\boldsymbol{\phi}(\boldsymbol{x}) = \sum_t \alpha^t r^t \boldsymbol{\phi}(\boldsymbol{x}^t)^T\boldsymbol{\phi}(\boldsymbol{x}) \\ &= \sum_t \alpha^t r^t K(\boldsymbol{x}^t, \boldsymbol{x}) \end{aligned} \tag{13.28}$$

This implies that if we have the kernel function, we do not need to map it to the new space at all. Actually, for any valid kernel, there does exist a corresponding mapping function, but it may be much simpler to use $K(\boldsymbol{x}^t, \boldsymbol{x})$ rather than calculating $\boldsymbol{\phi}(\boldsymbol{x}^t)$, $\boldsymbol{\phi}(\boldsymbol{x})$ and taking the dot product. KERNELIZATION Many algorithms have been *kernelized*, as we will see in later sections, and that is why we have the name "kernel machines."

GRAM MATRIX The matrix of kernel values, $\mathbf{K}$, where $\mathbf{K}_{ts} = K(\boldsymbol{x}^t, \boldsymbol{x}^s)$, is called the *Gram matrix*, which should be symmetric and positive semidefinite. Recently, it has become standard practice in sharing datasets to have available only the $\mathbf{K}$ matrices without providing $\boldsymbol{x}^t$ or $\boldsymbol{\phi}(\boldsymbol{x}^t)$. Especially in bioinformatics or natural language processing applications where $\boldsymbol{x}$ (or $\boldsymbol{\phi}(\boldsymbol{x})$) has hundreds or thousands of dimensions, storing/downloading the $N \times N$ matrix is much cheaper (Vert, Tsuda, and Schölkopf 2004); this, however, implies that we can use only those available for training/testing and cannot use the trained model to make predictions outside this dataset.

13.6 Vectorial Kernels

The most popular, general-purpose kernel functions are

- *polynomials* of degree q:

$$K(\boldsymbol{x}^t, \boldsymbol{x}) = (\boldsymbol{x}^T\boldsymbol{x}^t + 1)^q \tag{13.29}$$

where q is selected by the user. For example, when $q = 2$ and $d = 2$,

$$\begin{aligned} K(\boldsymbol{x}, \boldsymbol{y}) &= (\boldsymbol{x}^T\boldsymbol{y} + 1)^2 \\ &= (x_1y_1 + x_2y_2 + 1)^2 \\ &= 1 + 2x_1y_1 + 2x_2y_2 + 2x_1x_2y_1y_2 + x_1^2y_1^2 + x_2^2y_2^2 \end{aligned}$$

corresponds to the inner product of the basis function (Cherkassky and Mulier 1998):

$$\phi(\boldsymbol{x}) = [1, \sqrt{2}x_1, \sqrt{2}x_2, \sqrt{2}x_1x_2, x_1^2, x_2^2]^T$$

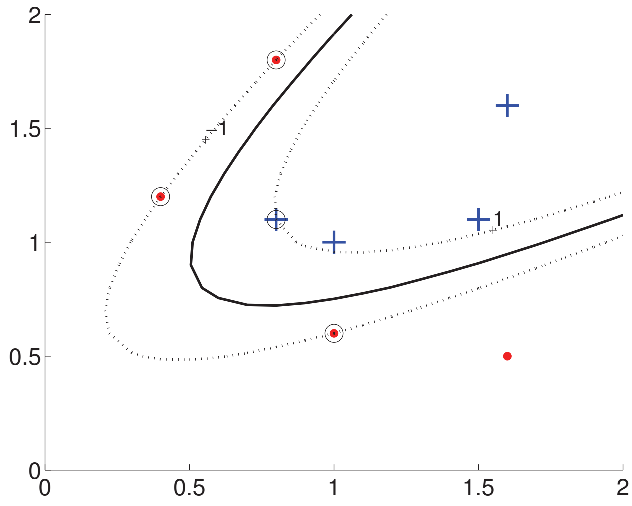


Figure 13.4 The discriminant and margins found by a polynomial kernel of degree 2. Circled instances are the support vectors.

An example is given in figure 13.4. When $q = 1$, we have the *linear kernel* that corresponds to the original formulation.

- *radial-basis functions*:

$$K(\mathbf{x}^t, \mathbf{x}) = \exp\left[-\frac{\|\mathbf{x}^t - \mathbf{x}\|^2}{2s^2}\right] \tag{13.30}$$

defines a spherical kernel as in Parzen windows (chapter 8) where $\mathbf{x}^t$ is the center and s, supplied by the user, defines the radius. This is also similar to radial basis functions that we discuss in chapter 12.

An example is shown in figure 13.5 where we see that larger spreads smooth the boundary; the best value is found by cross-validation. Note that when there are two parameters to be optimized using cross-validation, for example, here C and s^2, one should do a grid (factorial) search in the two dimensions; we will discuss methods for searching the best combination of such factors in section 19.2.

One can have a Mahalanobis kernel, generalizing from the Euclidean distance:

$$K(\mathbf{x}^t, \mathbf{x}) = \exp\left[-\frac{1}{2}(\mathbf{x}^t - \mathbf{x})^T \mathbf{S}^{-1}(\mathbf{x}^t - \mathbf{x})\right] \tag{13.31}$$

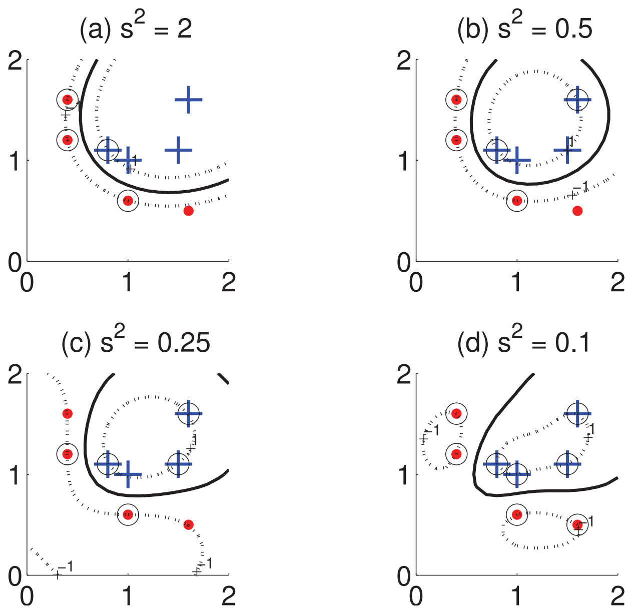


Figure 13.5 The boundary and margins found by the Gaussian kernel with different spread values, s^2. We get smoother boundaries with larger spreads.

where $\mathbf{S}$ is a covariance matrix. Or, in the most general case,

$$K(\boldsymbol{x}^t, \boldsymbol{x}) = \exp\left[-\frac{\mathcal{D}(\boldsymbol{x}^t, \boldsymbol{x})}{2s^2}\right] \tag{13.32}$$

for some distance function $\mathcal{D}(\boldsymbol{x}^t, \boldsymbol{x})$.

- *sigmoidal functions*:

$$K(\boldsymbol{x}^t, \boldsymbol{x}) = \tanh(2\boldsymbol{x}^T\boldsymbol{x}^t + 1) \tag{13.33}$$

where $\tanh(\cdot)$ has the same shape with sigmoid, except that it ranges between -1 and $+1$. This is similar to multilayer perceptrons that we discussed in chapter 11.

13.7 Defining Kernels

It is also possible to define application-specific kernels. Kernels are generally considered to be measures of similarity in the sense that $K(\boldsymbol{x}, \boldsymbol{y})$ takes a larger value as $\boldsymbol{x}$ and $\boldsymbol{y}$ are more "similar," from the point of view of the application. This implies that any prior knowledge we have regarding the application can be provided to the learner through appropriately defined kernels—"kernel engineering"—and such use of kernels can be seen as another example of a "hint" (section 11.8.4).

There are string kernels, tree kernels, graph kernels, and so on (Vert, Tsuda, and Schölkopf 2004), depending on how we represent the data and how we measure similarity in that representation.

For example, given two documents, the number of words appearing in both may be a kernel. Let us say D_1 and D_2 are two documents and one possible representation is called *bag of words* where we predefine M words relevant for the application, and we define $\boldsymbol{\phi}(D_1)$ as the M-dimensional binary vector whose dimension i is 1 if word i appears in D_1 and is 0 otherwise. Then, $\boldsymbol{\phi}(D_1)^T\boldsymbol{\phi}(D_2)$ counts the number of shared words. Here, we see that if we directly define and implement $K(D_1, D_2)$ as the number of shared words, we do not need to preselect M words and can use just any word in the vocabulary (of course, after discarding uninformative words like "of," "and," etc.) and we would not need to generate the bag-of-words representation explicitly and it would be as if we allowed M to be as large as we want.

BAG OF WORDS

Sometimes—for example, in bioinformatics applications—we can calculate a *similarity score* between two objects, which may not necessarily be positive semidefinite. Given two strings (of genes), a kernel measures the *edit distance*, namely, how many operations (insertions, deletions, substitutions) it takes to convert one string into another; this is also called *alignment*. In such a case, a trick is to define a set of M templates and represent an object as the M-dimensional vector of scores to all the templates. That is, if $\boldsymbol{m}_i, i = 1, \dots, M$ are the templates and $s(\boldsymbol{x}^t, \boldsymbol{m}_i)$ is the score between $\boldsymbol{x}^t$ and $\boldsymbol{m}_i$, then we define

EDIT DISTANCE

ALIGNMENT

$$\boldsymbol{\phi}(\boldsymbol{x}^t) = [s(\boldsymbol{x}^t, \boldsymbol{m}_1), s(\boldsymbol{x}^t, \boldsymbol{m}_2), \dots, s(\boldsymbol{x}^t, \boldsymbol{m}_M)]^T$$

and we define the *empirical kernel map* as

EMPIRICAL KERNEL MAP

$$K(\boldsymbol{x}^t, \boldsymbol{x}^s) = \boldsymbol{\phi}(\boldsymbol{x}^t)^T\boldsymbol{\phi}(\boldsymbol{x}^s)$$

which is a valid kernel.

Sometimes, we have a binary score function; for example, two proteins may interact or not, and we want to be able to generalize from this to scores for two arbitrary instances. In such a case, a trick is to define a graph where the nodes are the instances and two nodes are linked if they interact, that is, if the binary score returns 1. Then we say that two nodes that are not immediately linked are "similar" if the path between them is short or if they are connected by many paths. This converts pairwise local interactions to a global similarity measure, rather like defining a geodesic distance used in Isomap (section 6.10), and it is called the *diffusion kernel*. DIFFUSION KERNEL

If $p(\boldsymbol{x})$ is a probability density, then

$$K(\boldsymbol{x}^t, \boldsymbol{x}) = p(\boldsymbol{x}^t)p(\boldsymbol{x})$$

is a valid kernel. This is used when $p(\boldsymbol{x})$ is a generative model for $\boldsymbol{x}$ measuring how likely it is that we see $\boldsymbol{x}$. For example, if $\boldsymbol{x}$ is a sequence, $p(\boldsymbol{x})$ can be a hidden Markov model (chapter 15). With this kernel, $K(\boldsymbol{x}^t, \boldsymbol{x})$ will take a high value if both $\boldsymbol{x}^t$ and $\boldsymbol{x}$ are likely to have been generated by the same model. It is also possible to parametrize the generative model as $p(\boldsymbol{x}|\theta)$ and learn θ from data; this is called the *Fisher kernel* (Jaakkola and Haussler 1998). FISHER KERNEL

13.8 Multiple Kernel Learning

It is possible to construct new kernels by combining simpler kernels. If $K_1(\boldsymbol{x}, \boldsymbol{y})$ and $K_2(\boldsymbol{x}, \boldsymbol{y})$ are valid kernels and c a constant, then

$$K(\boldsymbol{x}, \boldsymbol{y}) = \begin{cases} cK_1(\boldsymbol{x}, \boldsymbol{y}) \\ K_1(\boldsymbol{x}, \boldsymbol{y}) + K_2(\boldsymbol{x}, \boldsymbol{y}) \\ K_1(\boldsymbol{x}, \boldsymbol{y}) \cdot K_2(\boldsymbol{x}, \boldsymbol{y}) \end{cases} \tag{13.34}$$

are also valid.

Different kernels may also be using different subsets of $\boldsymbol{x}$. We can therefore see combining kernels as another way to fuse information from different sources where each kernel measures similarity according to its domain. When we have input from two representations A and B

$$\begin{aligned} K_A(\boldsymbol{x}_A, \boldsymbol{y}_A) + K_B(\boldsymbol{x}_B, \boldsymbol{y}_B) &= \boldsymbol{\phi}_A(\boldsymbol{x}_A)^T \boldsymbol{\phi}_A(\boldsymbol{y}_A) + \boldsymbol{\phi}_B(\boldsymbol{x}_B)^T \boldsymbol{\phi}_B(\boldsymbol{y}_B) \\ &= \boldsymbol{\phi}(\boldsymbol{x})^T \boldsymbol{\phi}(\boldsymbol{y}) \\ &= K(\boldsymbol{x}, \boldsymbol{y}) \end{aligned} \tag{13.35}$$

where $\boldsymbol{x} = [\boldsymbol{x}_A, \boldsymbol{x}_B]$ is the concatenation of the two representations. That is, taking a sum of two kernels corresponds to doing a dot product in the concatenated feature vectors. One can generalize to a number of kernels

$$K(\boldsymbol{x}, \boldsymbol{y}) = \sum_{i=1}^{m} K_i(\boldsymbol{x}, \boldsymbol{y}) \tag{13.36}$$

which, similar to taking an average of classifiers (section 17.4), this time averages over kernels and frees us from the need to choose one particular kernel. It is also possible to take a weighted sum and also learn the weights from data (Lanckriet et al. 2004; Sonnenburg et al. 2006):

$$K(\boldsymbol{x}, \boldsymbol{y}) = \sum_{i=1}^{m} \eta_i K_i(\boldsymbol{x}, \boldsymbol{y}) \tag{13.37}$$

MULTIPLE KERNEL LEARNING

subject to $\eta_i \geq 0$, with or without the constraint of $\sum_i \eta_i = 1$, respectively known as convex or conic combination. This is called *multiple kernel learning* where we replace a single kernel with a weighted sum (Gönen and Alpaydın 2011). The single kernel objective function of equation 13.27 becomes

$$L_d = \sum_t \alpha^t - \frac{1}{2} \sum_t \sum_s \alpha^t \alpha^s r^t r^s \sum_i \eta_i K_i(\boldsymbol{x}^t, \boldsymbol{x}^s) \tag{13.38}$$

which we solve for both the support vector machine parameters α^t and the kernel weights η_i. Then, the combination of multiple kernels also appear in the discriminant

$$g(\boldsymbol{x}) = \sum_t \alpha^t r^t \sum_i \eta_i K_i(\boldsymbol{x}^t, \boldsymbol{x}) \tag{13.39}$$

After training, η_i will take values depending on how the corresponding kernel $K_i(\boldsymbol{x}^t, \boldsymbol{x})$ is useful in discriminating. It is also possible to localize kernels by defining kernel weights as a parameterized function of the input $\boldsymbol{x}$, rather like the gating function in mixture of experts (section 17.8)

$$g(\boldsymbol{x}) = \sum_t \alpha^t r^t \sum_i \eta_i(\boldsymbol{x}|\theta_i) K_i(\boldsymbol{x}^t, \boldsymbol{x}) \tag{13.40}$$

and the gating parameters θ_i are learned together with the support vector machine parameters (Gönen and Alpaydın 2008).

When we have information coming from multiple sources in different representations or modalities—for example, in speech recognition where we may have both acoustic and visual lip image—the usual approach is to feed them separately to different classifiers and then fuse the decisions; we will discuss methods for this in detail in chapter 17. Combining multiple kernels provides us with another way of integrating input from multiple sources, where there is a single classifier that uses different kernels for inputs of different sources, for which there are different notions of similarity (Noble 2004). The localized version can then seen be an extension of this where we can choose between sources, and hence similarity measures, depending on the input.

13.9 Multiclass Kernel Machines

When there are $K > 2$ classes, the straightforward, *one-vs.-all* way is to define K two-class problems, each one separating one class from all other classes combined and learn K support vector machines $g_i(\boldsymbol{x}), i = 1, \ldots, K$. That is, in training $g_i(\boldsymbol{x})$, examples of C_i are labeled $+1$ and examples of $C_k, k \neq i$ are labeled as -1. During testing, we calculate all $g_i(\boldsymbol{x})$ and choose the maximum.

Platt (1999) proposed to fit a sigmoid to the output of a single (2-class) SVM output to convert to a posterior probability. Similarly, one can train one layer of softmax outputs to minimize cross-entropy to generate $K > 2$ posterior probabilities (Mayoraz and Alpaydın 1999):

$$y_i(\boldsymbol{x}) = \sum_{j=1}^{K} v_{ij} f_j(\boldsymbol{x}) + v_{i0} \tag{13.41}$$

where $f_j(\boldsymbol{x})$ are the SVM outputs and y_i are the posterior probability outputs. Weights v_{ij} are trained to minimize cross-entropy. Note, however, that as in stacking (section 17.9), the data on which we train v_{ij} should be different from the data used to train the base SVMs $f_j(\boldsymbol{x})$, to alleviate overfitting.

Instead of the usual approach of building K two-class SVM classifiers to separate one from all the rest, as with any other classifier, one can build $K(K-1)/2$ *pairwise* classifiers (see also section 10.4), each $g_{ij}(\boldsymbol{x})$ taking examples of C_i with the label $+1$, examples of C_j with the label -1, and not using examples of the other classes. Separating classes in pairs is normally expected to be an easier job, with the additional advantage that because we use less data, the optimizations will be faster, noting however that we have $\mathcal{O}(K^2)$ discriminants to train instead of $\mathcal{O}(K)$.

ERROR-CORRECTING OUTPUT CODES

In the general case, both one-vs.-all and pairwise separation are special cases of the *error-correcting output codes* (ECOC) that decompose a multiclass problem to a set of two-class problems (Dietterich and Bakiri 1995) (see also section 17.6). SVMs being two-class classifiers are ideally suited to this (Allwein, Schapire, and Singer 2000), and it is also possible to have an incremental approach where new two-class SVMs are added to better separate pairs of classes that are confused, to ameliorate a poor ECOC matrix (Mayoraz and Alpaydın 1999).

Another possibility is to write a single *multiclass* optimization problem involving all classes (Weston and Watkins 1998):

$$\min \frac{1}{2}\sum_{i=1}^{K}\|\boldsymbol{w}_i\|^2 + C\sum_i\sum_t \xi_i^t \tag{13.42}$$

subject to

$$\boldsymbol{w}_{z^t}\boldsymbol{x}^t + w_{z^t0} \geq \boldsymbol{w}_i\boldsymbol{x}^t + w_{i0} + 2 - \xi_i^t, \forall i \neq z^t \text{ and } \xi_i^t \geq 0$$

where z^t contains the class index of $\boldsymbol{x}^t$. The regularization terms minimizes the norms of all hyperplanes simultaneously, and the constraints are there to make sure that the margin between the actual class and any other class is at least 2. The output for the correct class should be at least $+1$, the output of any other class should be at least -1, and the slack variables are defined to make up any difference.

Though this looks neat, the one-vs.-all approach is generally preferred because it solves K separate N variable problems whereas the multiclass formulation uses $K \cdot N$ variables.

13.10 Kernel Machines for Regression

Now let us see how support vector machines can be generalized for regression. We see that the same approach of defining acceptable margins, slacks, and a regularizing function that combines smoothness and error is also applicable here. We start with a linear model, and later on we see how we can use kernel functions here as well:

$$f(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x} + w_0$$

In regression proper, we use the square of the difference as error:

$$e_2(r^t, f(\boldsymbol{x}^t)) = [r^t - f(\boldsymbol{x}^t)]^2$$

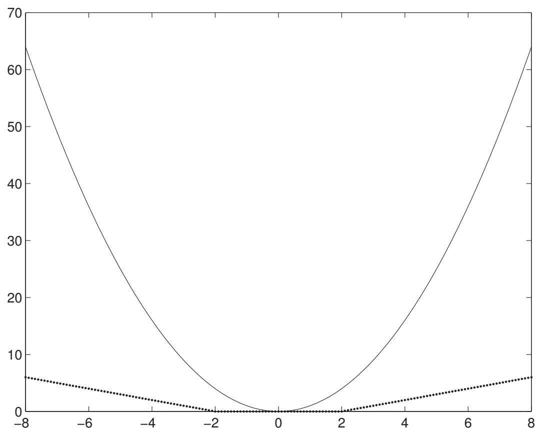


Figure 13.6 Quadratic and ϵ-sensitive error functions. We see that ϵ-sensitive error function is not affected by small errors and also is less affected by large errors and thus is more robust to outliers.

whereas in support vector regression, we use the ϵ-sensitive loss function:

$$e_\epsilon(r^t, f(\boldsymbol{x}^t)) = \begin{cases} 0 & \text{if } |r^t - f(\boldsymbol{x}^t)| < \epsilon \\ |r^t - f(\boldsymbol{x}^t)| - \epsilon & \text{otherwise} \end{cases} \tag{13.43}$$

which means that we tolerate errors up to ϵ and also that errors beyond have a linear effect and not a quadratic one. This error function is therefore more tolerant to noise and is thus more *robust* (see figure 13.6). As in the hinge loss, there is a region of no error, which causes sparseness.

ROBUST REGRESSION

Analogous to the soft margin hyperplane, we introduce slack variables to account for deviations out of the ϵ-zone and we get (Vapnik 1995)

$$\min \frac{1}{2}\|\boldsymbol{w}\|^2 + C\sum_t (\xi_+^t + \xi_-^t) \tag{13.44}$$

subject to

$$\begin{aligned} r^t - (\boldsymbol{w}^T\boldsymbol{x} + w_0) &\le \epsilon + \xi_+^t \\ (\boldsymbol{w}^T\boldsymbol{x} + w_0) - r^t &\le \epsilon + \xi_-^t \\ \xi_+^t, \xi_-^t &\ge 0 \end{aligned}$$

where we use two types of slack variables, for positive and negative deviations, to keep them positive. Actually, we can see this as two hinges added back to back, one for positive and one for negative slacks. This formulation corresponds to the ϵ-sensitive loss function given in equation 13.43. The Lagrangian is

$$\begin{aligned}L_p &= \frac{1}{2}\|\boldsymbol{w}\|^2 + C\sum_t(\xi_+^t + \xi_-^t) \\ &\quad - \sum_t \alpha_+^t\left[\epsilon + \xi_+^t - r^t + (\boldsymbol{w}^T\boldsymbol{x} + w_0)\right] \\ &\quad - \sum_t \alpha_-^t\left[\epsilon + \xi_-^t + r^t - (\boldsymbol{w}^T\boldsymbol{x} + w_0)\right] \\ &\quad - \sum_t(\mu_+^t\xi_+^t + \mu_-^t\xi_-^t) \end{aligned} \tag{13.45}$$

Taking the partial derivatives, we get

$$\frac{\partial L_p}{\partial \boldsymbol{w}} = \boldsymbol{w} - \sum_t(\alpha_+^t - \alpha_-^t)\boldsymbol{x}^t = 0 \Rightarrow \boldsymbol{w} = \sum_t(\alpha_+^t - \alpha_-^t)\boldsymbol{x}^t \tag{13.46}$$

$$\frac{\partial L_p}{\partial w_0} = \sum_t(\alpha_+^t - \alpha_-^t)\boldsymbol{x}^t = 0 \tag{13.47}$$

$$\frac{\partial L_p}{\partial \xi_+^t} = C - \alpha_+^t - \mu_+^t = 0 \tag{13.48}$$

$$\frac{\partial L_p}{\partial \xi_-^t} = C - \alpha_-^t - \mu_-^t = 0 \tag{13.49}$$

The dual is

$$\begin{aligned}L_d &= -\frac{1}{2}\sum_t\sum_s(\alpha_+^t - \alpha_-^t)(\alpha_+^s - \alpha_-^s)(\boldsymbol{x}^t)^T\boldsymbol{x}^s \\ &\quad -\epsilon\sum_t(\alpha_+^t + \alpha_-^t) + \sum_t r^t(\alpha_+^t - \alpha_-^t)\end{aligned} \tag{13.50}$$

subject to

$$0 \le \alpha_+^t \le C\ ,\ 0 \le \alpha_-^t \le C\ ,\ \sum_t(\alpha_+^t - \alpha_-^t) = 0$$

Once we solve this, we see that all instances that fall in the tube have $\alpha_+^t = \alpha_-^t = 0$; these are the instances that are fitted with enough precision (see figure 13.7). The support vectors satisfy either $\alpha_+^t > 0$ or $\alpha_-^t > 0$ and are of two types. They may be instances that are on the boundary of the tube (either α_+^t or α_-^t is between 0 and C), and we use these to calculate w_0. For example, assuming that $\alpha_+^t > 0$, we have $r^t = \boldsymbol{x}^T\boldsymbol{x}^t + w_0 + \epsilon$. Instances that fall outside the ϵ-tube are of the second type; these are

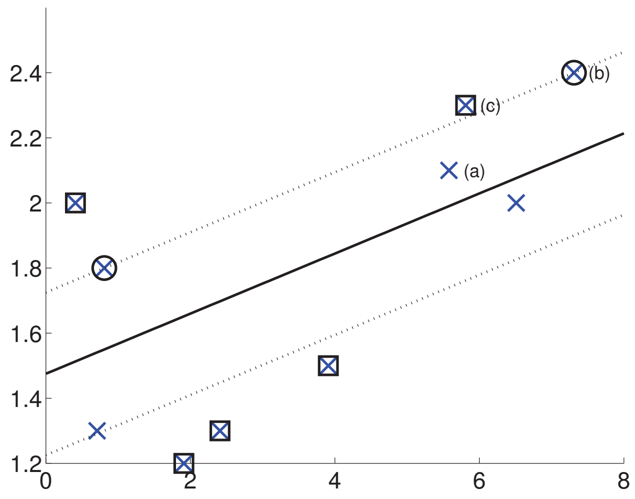


Figure 13.7 The fitted regression line to data points shown as crosses and the ϵ-tube are shown ($C = 10, \epsilon = 0.25$). There are three cases: In (a), the instance is in the tube; in (b), the instance is on the boundary of the tube (circled instances); in (c), it is outside the tube with a positive slack, that is, $\xi_+^t > 0$ (squared instances). (b) and (c) are support vectors. In terms of the dual variable, in (a), $\alpha_+^t = 0, \alpha_-^t = 0$, in (b), $\alpha_+^t < C$, and in (c), $\alpha_+^t = C$.

instances for which we do not have a good fit ($\alpha_+^t = C$), as shown in figure 13.7.

Using equation 13.46, we can write the fitted line as a weighted sum of the support vectors:

$$f(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x} + w_0 = \sum_t (\alpha_+^t - \alpha_-^t)(\boldsymbol{x}^t)^T\boldsymbol{x} + w_0 \tag{13.51}$$

Again, the dot product $(\boldsymbol{x}^t)^T\boldsymbol{x}^s$ in equation 13.50 can be replaced with a kernel $K(\boldsymbol{x}^t, \boldsymbol{x}^s)$, and similarly $(\boldsymbol{x}^t)^T\boldsymbol{x}$ be replaced with $K(\boldsymbol{x}^t, \boldsymbol{x})$ and we can have a nonlinear fit. Using a polynomial kernel would be similar to fitting a polynomial (figure 13.8), and using a Gaussian kernel (figure 13.9) would be similar to nonparametric smoothing models (section 8.8) except that because of the sparsity of solution, we would not need the whole training set but only a subset.

There is also an equivalent ν-SVM formulation for regression (Schölkopf et al. 2000), where instead of fixing ϵ, we fix ν to bound the fraction of

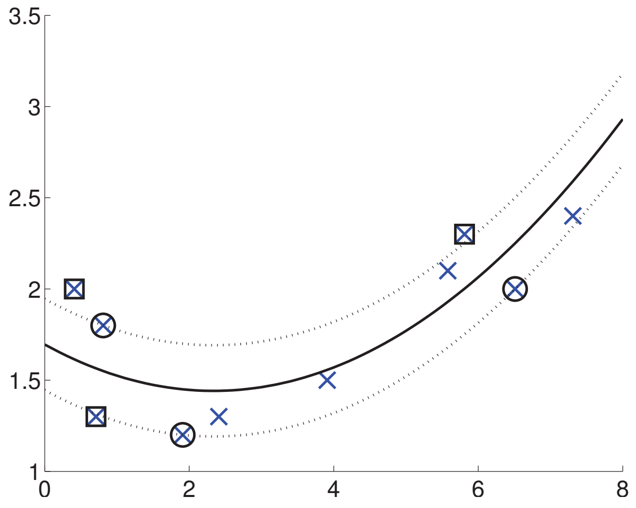


Figure 13.8 The fitted regression line and the ϵ-tube using a quadratic kernel are shown ($C = 10, \epsilon = 0.25$). Circled instances are the support vectors on the margins, squared instances are support vectors which are outliers.

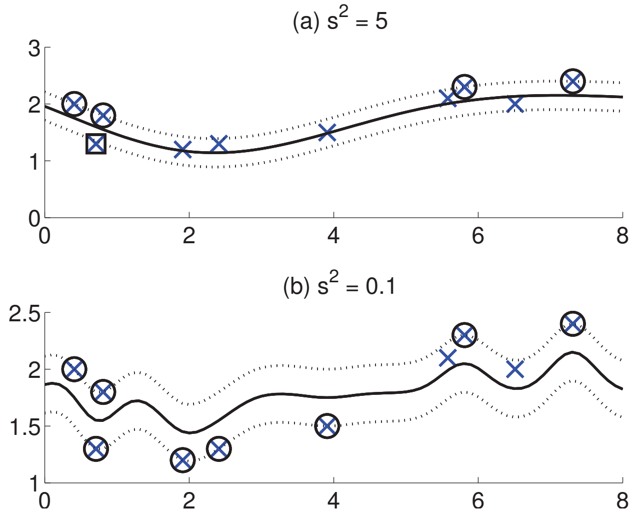


Figure 13.9 The fitted regression line and the ϵ-tube using a Gaussian kernel with two different spreads are shown ($C = 10, \epsilon = 0.25$). Circled instances are the support vectors on the margins, and squared instances are support vectors that are outliers.

support vectors. There is still a need for C though.

13.11 Kernel Machines for Ranking

Remember that in ranking, we have instances that need to be ordered in a certain way (Liu 2011). For example, we may have pairwise constraints such as $r^u \prec r^v$ which means that instance $\boldsymbol{x}^u$ should generate a higher score than $\boldsymbol{x}^v$. In section 10.9, we discuss how we can train a linear model for this purpose using gradient descent. We now discuss how we can do the same using support vector machines.

We consider each pairwise constraint as one data instance $t : r^u \prec r^v$ and minimize

$$L_p = \frac{1}{2}\|\boldsymbol{w}\|^2 + C\sum_t \xi^t \tag{13.52}$$

subject to

$$\begin{aligned} \boldsymbol{w}^T\boldsymbol{x}^u &\geq \boldsymbol{w}^T\boldsymbol{x}^v + 1 - \xi^t, \text{ for each } t : r^u \prec r^v \\ \xi^t &\geq 0 \end{aligned} \tag{13.53}$$

Equation 13.53 requires that the score for $\boldsymbol{x}^u$ be at least 1 unit more than the score for $\boldsymbol{x}^v$ and hence defines a margin. If the constraint is not satisfied, the slack variable is nonzero and equation 13.52 minimizes the sum of such slacks and the complexity term, which again corresponds to making the width of the margin as large as possible (Herbrich, Obermayer, and Graepel 2000; Joachims 2002). Note that the second term of the sum of slacks is the same as the error used in equation 10.46 except for the 1 unit margin, and the complexity term, as we discussed before, can be interpreted as a weight decay term on the linear model (see section 11.10).

Note that there is one constraint for each pair where an ordering is defined, and hence the number of such constraints is $\mathcal{O}(N^2)$. The constraint of equation 13.53 can also be written as

$$\boldsymbol{w}^T(\boldsymbol{x}^u - \boldsymbol{x}^v) \geq 1 - \xi^t$$

That is, we can view this as a two-class classification of pairwise differences, $\boldsymbol{x}^u - \boldsymbol{x}^v$. So by calculating such differences and labeling them as $r^t \in \{-1, +1\}$ depending on whether $r^v \prec r^u$ or $r^u \prec r^v$ respectively, any two-class kernel machine can be used to implement ranking. But this is not the most efficient way to implement, and faster methods have been proposed (Chapelle and Keerthi 2010).

The dual is

$$L_d = \sum_t \alpha^t - \frac{1}{2}\sum_t \sum_s \alpha^t \alpha^s (\boldsymbol{x}^u - \boldsymbol{x}^v)^T (\boldsymbol{x}^k - \boldsymbol{x}^l) \tag{13.54}$$

subject to $0 \le \alpha^t \le C$. Here, t and s are two pairwise constraints, such as $t : r^u \prec r^v$ and $s : r^k \prec r^l$. Solving this, for the constraints that are satisfied, we have $\xi^t = 0$ and $\alpha^t = 0$; for the ones that are satisfied but are in the margin, we have $0 < \xi^t < 1$ and $\alpha^t < C$; and for the ones that are not satisfied (and are misranked), we have $\xi^t > 1$ and $\alpha^t = C$.

For new test instance $\boldsymbol{x}$, the score is calculated as

$$g(\boldsymbol{x}) = \sum_t \alpha^t (\boldsymbol{x}^u - \boldsymbol{x}^v)^T \boldsymbol{x} \tag{13.55}$$

It is straightforward to write the kernelized version of the primal, dual, and score functions, and this is left to the reader (see exercise 7).

13.12 One-Class Kernel Machines

Support vector machines, originally proposed for classification, are extended to regression by defining slack variables for deviations around the regression line, instead of the discriminant. We now see how SVM can be used for a restricted type of unsupervised learning, namely, for estimating regions of high density. We are not doing a full density estimation; rather, we want to find a boundary (so that it reads like a classification problem) that separates volumes of high density from volumes of low density (Tax and Duin 1999). Such a boundary can then be used for *novelty* or *outlier detection*. This is also called *one-class classification*.

OUTLIER DETECTION
ONE-CLASS CLASSIFICATION

We consider a sphere with center $\boldsymbol{a}$ and radius R that we want to enclose as much as possible of the density, measured empirically as the enclosed training set percentage. At the same time, trading off with it, we want to find the smallest radius (see figure 13.10). We define slack variables for instances that lie outside (we only have one type of slack variable because we have examples from one class and we do not have any penalty for those inside), and we have a smoothness measure that is proportional to the radius:

$$\min R^2 + C \sum_t \xi^t \tag{13.56}$$

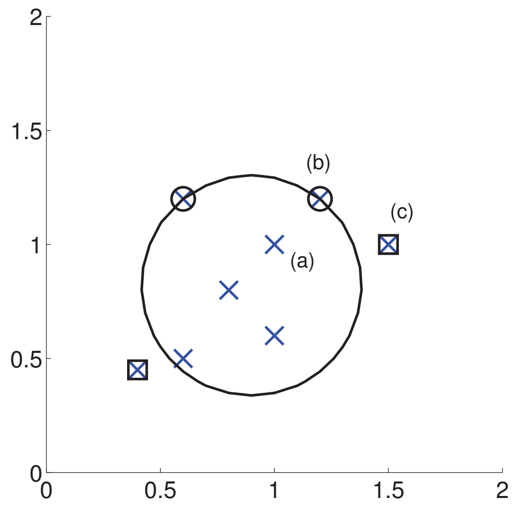


Figure 13.10 One-class support vector machine places the smoothest boundary (here using a linear kernel, the circle with the smallest radius) that encloses as much of the instances as possible. There are three possible cases: In (a), the instance is a typical instance. In (b), the instance falls on the boundary with $\xi^t = 0$; such instances define R. In (c), the instance is an outlier with $\xi^t > 0$. (b) and (c) are support vectors. In terms of the dual variable, we have, in (a), $\alpha^t = 0$; in (b), $0 < \alpha^t < C$; in (c), $\alpha^t = C$.

subject to

$$\|\boldsymbol{x}^t - \boldsymbol{a}\|^2 \le R^2 + \xi^t \text{ and } \xi^t \ge 0, \forall t$$

Adding the constraints, we get the Lagrangian, which we write keeping in mind that $\|\boldsymbol{x}^t - \boldsymbol{a}\|^2 = (\boldsymbol{x}^t - \boldsymbol{a})^T(\boldsymbol{x}^t - \boldsymbol{a})$:

$$L_p = R^2 + C\sum_t \xi^t - \sum_t \alpha^t \left(R^2 + \xi^t - \left[(\boldsymbol{x}^t)^T\boldsymbol{x}^t - 2\boldsymbol{a}^T\boldsymbol{x}^t + \boldsymbol{a}^T\boldsymbol{a}\right]\right) - \sum_t \gamma^t \xi^t \tag{13.57}$$

with $\alpha^t \ge 0$ and $\gamma^t \ge 0$ being the Lagrange multipliers. Taking the derivative with respect to the parameters, we get

$$\frac{\partial L}{\partial R} = 2R - 2R\sum_t \alpha^t = 0 \Rightarrow \sum_t \alpha^t = 1 \tag{13.58}$$

$$\frac{\partial L}{\partial \boldsymbol{a}} = \sum_t \alpha^t(2\boldsymbol{x}^t - 2\boldsymbol{a}) = 0 \Rightarrow \boldsymbol{a} = \sum_t \alpha^t \boldsymbol{x}^t \tag{13.59}$$

$$\frac{\partial L}{\partial \xi^t} = C - \alpha^t - \gamma^t = 0 \tag{13.60}$$

Since $\gamma^t \geq 0$, we can write this last as the constraint: $0 \leq \alpha^t \leq C$. Plugging these into equation 13.57, we get the dual that we maximize with respect to α^t:

$$L_d = \sum_t \alpha^t (\boldsymbol{x}^t)^T \boldsymbol{x}^t - \sum_t \sum_s \alpha^t \alpha^s (\boldsymbol{x}^t)^T \boldsymbol{x}^s \tag{13.61}$$

subject to

$$0 \leq \alpha^t \leq C \text{ and } \sum_t \alpha^t = 1$$

When we solve this, we again see that most of the instances vanish with their $\alpha^t = 0$; these are the typical, highly likely instances that fall inside the sphere (figure 13.10). There are two type of support vectors with $\alpha^t > 0$: There are instances that satisfy $0 < \alpha^t < C$ and lie on the boundary, $\|\boldsymbol{x}^t - \boldsymbol{a}\|^2 = R^2$ ($\xi^t = 0$), which we use to calculate R. Instances that satisfy $\alpha^t = C$ ($\xi^t > 0$) lie outside the boundary and are the outliers. From equation 13.59, we see that the center $\boldsymbol{a}$ is written as a weighted sum of the support vectors.

Then given a test input $\boldsymbol{x}$, we say that it is an outlier if

$$\|\boldsymbol{x} - \boldsymbol{a}\|^2 > R^2$$

or

$$\boldsymbol{x}^t \boldsymbol{x} - 2\boldsymbol{a}^T \boldsymbol{x} + \boldsymbol{a}^T \boldsymbol{a} > R^2$$

Using kernel functions, allow us to go beyond a sphere and define boundaries of arbitrary shapes. Replacing the dot product with a kernel function, we get (subject to the same constraints):

$$L_d = \sum_t \alpha^t K(\boldsymbol{x}^t, \boldsymbol{x}^t) - \sum_t \sum_s \alpha^t \alpha^s K(\boldsymbol{x}^t, \boldsymbol{x}^s) \tag{13.62}$$

For example, using a polynomial kernel of degree 2 allows arbitrary quadratic surfaces to be used. If we use a Gaussian kernel (equation 13.30), we have a union of local spheres. We reject $\boldsymbol{x}$ as an outlier if

$$K(\boldsymbol{x}, \boldsymbol{x}) - 2\sum_t \alpha^t K(\boldsymbol{x}, \boldsymbol{x}^t) + \sum_t \sum_s \alpha^t \alpha^s K(\boldsymbol{x}^t, \boldsymbol{x}^s) > R^2$$

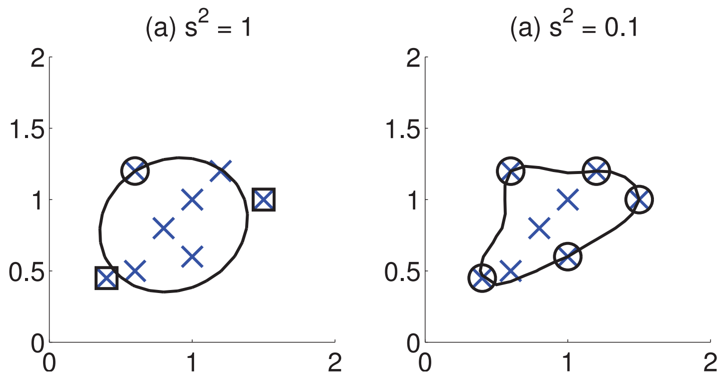


Figure 13.11 One-class support vector machine using a Gaussian kernel with different spreads.

The third term does not depend on $\boldsymbol{x}$ and is therefore a constant (we use this as an equality to solve for R where $\boldsymbol{x}$ is an instance on the margin). In the case of a Gaussian kernel where $K(\boldsymbol{x}, \boldsymbol{x}) = 1$, the condition reduces to

$$\sum_t \alpha^t K_G(\boldsymbol{x}, \boldsymbol{x}^t) < R_c$$

for some constant R_c, which is analogous to the kernel density estimator (section 8.2.2)—except for the sparseness of the solution—with a probability threshold R_c (see figure 13.11).

There is also an alternative, equivalent ν-SVM type of formulation of one-class support vector machines that uses the canonical $(1/2)\|\boldsymbol{w}\|^2$ type of smoothness (Schölkopf et al. 2001).

13.13 Large Margin Nearest Neighbor Classifier

In chapter 8, we discussed nonparametric methods where instead of fitting a global model to the data we interpolate from a subset of neighboring instances, and specifically in section 8.6, we covered the importance of using a good distance measure. We now discuss a method to learn a distance measure from the data. Strictly speaking, this is not a kernel machine, but it uses the idea of keeping a margin in ranking, as we noted in section 13.11.

The basic idea is to view k-nearest neighbor classification (section 8.4) as a ranking problem. Let us say the k-nearest neighbors of $\boldsymbol{x}^i$ contains two instances $\boldsymbol{x}^j$ and $\boldsymbol{x}^l$ such that $\boldsymbol{x}^i$ and $\boldsymbol{x}^j$ are of the same class and $\boldsymbol{x}^l$ belongs to another class. In such a case, we want a distance measure such that the distance between $\boldsymbol{x}^i$ and $\boldsymbol{x}^l$ is more than $\boldsymbol{x}^i$ and $\boldsymbol{x}^j$. Actually, we not only require that it be more but that there be a one-unit margin between them and if this is not satisfied, we have a slack variable for the difference:

$$\mathcal{D}(\boldsymbol{x}^i,\boldsymbol{x}^l) \geq \mathcal{D}(\boldsymbol{x}^i,\boldsymbol{x}^j) + 1 - \xi^{ijl}$$

The distance measure works as a score function in a ranking problem, and each $(\boldsymbol{x}^i,\boldsymbol{x}^j,\boldsymbol{x}^l)$ triple defines one ranking constraint as in equation 13.53.

LARGE MARGIN NEAREST NEIGHBOR

This is the basic idea behind the *large margin nearest neighbor* (LMNN) algorithm (Weinberger and Saul 2009). The error function minimized is

$$(1-\mu)\sum_{i,j}\mathcal{D}(\boldsymbol{x}^i,\boldsymbol{x}^j) + \mu\sum_{i,j,l}(1-y_{il})\xi_{ijl} \tag{13.63}$$

subject to

$$\begin{aligned}\mathcal{D}(\boldsymbol{x}^i,\boldsymbol{x}^l) &\geq \mathcal{D}(\boldsymbol{x}^i,\boldsymbol{x}^j) + 1 - \xi^{ijl},\ \text{if } \boldsymbol{r}^i = \boldsymbol{r}^j \text{ and } \boldsymbol{r}^i \neq \boldsymbol{r}^l \\ \xi^{ijl} &\geq 0\end{aligned} \tag{13.64}$$

Here, $\boldsymbol{x}^j$ is one of the k-nearest neighbors of $\boldsymbol{x}^i$ and they are of the same class: $\boldsymbol{r}^i = \boldsymbol{r}^j$—it is a *target* neighbor. $\boldsymbol{x}^l$ is also one of the k-nearest neighbors of $\boldsymbol{x}^i$; if they are of the same label, then y_{il} is set to 1 and we incur no loss; if they are of different classes, then $\boldsymbol{x}^l$ is an *impostor*, y_{il} is set to 0, and if the condition 13.64 is not satisfied, the slack defines a cost. The second term of equation 13.63 is the sum of such slacks. The first term is the total distance to all target neighbors and minimizing that has an effect of regularization—we want to keep the distances as small as possible.

In LMNN, Mahalanobis distance is used as the distance measure model:

$$\mathcal{D}(\boldsymbol{x}^i,\boldsymbol{x}^j|\mathbf{M}) = (\boldsymbol{x}^i - \boldsymbol{x}^j)^T\mathbf{M}(\boldsymbol{x}^i - \boldsymbol{x}^j) \tag{13.65}$$

and $\mathbf{M}$ matrix is the parameter that is to be optimized. Equation 13.63 defines a convex (more specifically, positive semi-definite) problem and hence has a unique minimum.

When the input dimensionality is high and there are few data, as we discuss in equation 8.21, we can regularize by factoring $\mathbf{M}$ as $\mathbf{L}^T\mathbf{L}$ where $\mathbf{L}$ is $k \times d$ with $k < d$:

$$\mathcal{D}(\boldsymbol{x}^i, \boldsymbol{x}^j|\mathbf{L}) = \|\mathbf{L}\boldsymbol{x}^i - \mathbf{L}\boldsymbol{x}^j\|^2 \tag{13.66}$$

$\mathbf{L}\boldsymbol{x}$ is the k-dimensional projection of $\boldsymbol{x}$, and Mahalanobis distance in the original d-dimensional $\boldsymbol{x}$ space corresponds to the (squared) Euclidean distance in the new k-dimensional space—see figure 8.7 for an example. If we plug equation 13.66 into equation 13.63 as the distance measure, we get the *large margin component analysis* (LMCA) algorithm (Torresani and Lee 2007); unfortunately, this is no longer a convex optimization problem, and if we use gradient descent, we get a locally optimal solution.

LARGE MARGIN COMPONENT ANALYSIS

13.14 Kernel Dimensionality Reduction

We know from section 6.3 that principal components analysis (PCA) reduces dimensionality by projecting on the eigenvectors of the covariance matrix $\boldsymbol{\Sigma}$ with the largest eigenvalues, which, if data instances are centered ($E[\boldsymbol{x}] = 0$), can be written as $\mathbf{X}^T\mathbf{X}$. In the kernelized version, we work in the space of $\boldsymbol{\phi}(\boldsymbol{x})$ instead of the original $\boldsymbol{x}$ and because, as usual, the dimensionality d of this new space may be much larger than the dataset size N, we prefer to work with the $N \times N$ matrix $\mathbf{X}\mathbf{X}^T$ and do feature embedding instead of working with the $d \times d$ matrix $\mathbf{X}^T\mathbf{X}$. The projected data matrix is $\boldsymbol{\Phi} = \boldsymbol{\phi}(\mathbf{X})$, and hence we work with the eigenvectors of $\boldsymbol{\Phi}^T\boldsymbol{\Phi}$ and hence the kernel matrix $\mathbf{K}$.

KERNEL PCA

Kernel PCA uses the eigenvectors and eigenvalues of the kernel matrix and this corresponds to doing a linear dimensionality reduction in the $\phi(\boldsymbol{x})$ space. When $\boldsymbol{c}_i$ and λ_i are the corresponding eigenvectors and eigenvalues, the projected new k-dimensional values can be calculated as

$$z_j^t = \sqrt{\lambda_j}\boldsymbol{c}_j^t, j = 1, \ldots, k, \; t = 1, \ldots, N$$

An example is given in figure 13.12 where we first use a quadratic kernel and then decrease dimensionality to two (out of five) using kernel PCA and implement a linear SVM there. Note that in the general case (e.g., with a Gaussian kernel), the eigenvalues do not necessarily decay and there is no guarantee that we can reduce dimensionality using kernel PCA.

What we are doing here is multidimensional scaling (section 6.7) using kernel values as the similarity values. For example, by taking $k = 2$,

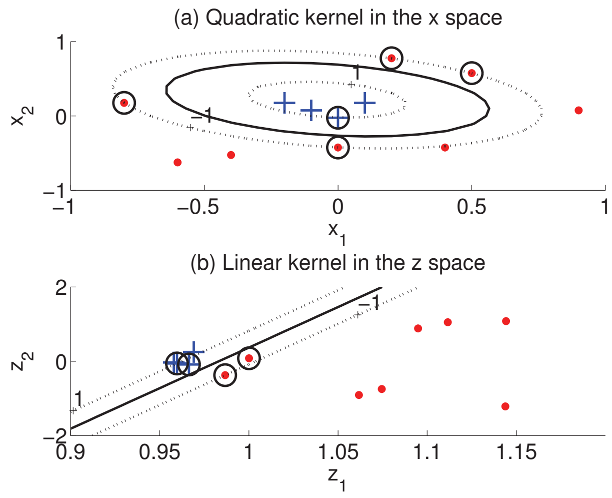


Figure 13.12 Instead of using a quadratic kernel in the original space (a), we can use kernel PCA on the quadratic kernel values to map to a two-dimensional new space where we use a linear discriminant (b); these two dimensions (out of five) explain 80 percent of the variance.

one can visualize the data in the space induced by the kernel matrix, which can give us information as to how similarity is defined by the used kernel. Linear discriminant analysis (LDA) (section 6.8) can similarly be kernelized (Müller et al. 2001). The kernelized version of canonical correlation analysis (CCA) (section 6.9) is discussed in Hardoon, Szedmak, Shawe-Taylor 2004.

In chapter 6, we discussed nonlinear dimensionality reduction methods, Isomap and LLE. In fact, by viewing the elements of the cost matrix in equation 6.58 as kernel evaluations for pairs of inputs, LLE can be seen as kernel PCA for a particular choice of kernel. The same also holds for Isomap when a kernel function is defined as a function of the geodesic distance on the graph.

13.15 Notes

The idea of generalizing linear models by mapping the data to a new space through nonlinear basis functions is old, but the novelty of support vector machines is that of integrating this into a learning algorithm whose parameters are defined in terms of a subset of data instances (the so-called *dual representation*), hence also without needing to explicitly evaluate the basis functions and thereby also limiting complexity by the size of the training set; this is also true for Gaussian processes where the kernel function is called the covariance function (section 16.9).

DUAL REPRESENTATION

The sparsity of the solution shows the advantage over nonparametric estimators, such as *k*-nearest neighbor and Parzen windows, or Gaussian processes, and the flexibility to use kernel functions allows working with nonvectorial data. Because there is a unique solution to the optimization problem, we do not need any iterative optimization procedure as we do in neural networks. Because of all these reasons, support vector machines are now considered to be the best, off-the-shelf learners and are widely used in many domains, especially bioinformatics (Schölkopf, Tsuda, and Vert 2004) and natural language processing applications, where an increasing number of tricks are being developed to derive kernels (Shawe-Taylor and Cristianini 2004).

The use of kernel functions implies a different data representation; we no longer define an instance (object/event) as a vector of attributes by itself, but in terms of how it is similar to or differs from other instances; this is akin to the difference between multidimensional scaling that uses a matrix of distances (without any need to know how they are calculated) and principal components analysis that uses vectors in some space.

The support vector machine is currently considered to be the best off-the-shelf learning algorithm and has been applied successfully in various domains. The fact that we are solving a convex problem and hence optimally and the idea of kernels that allow us to code our prior information has made it quite popular. There is a huge literature on the support vector machine and all types of kernel machines. The classic books are by Vapnik (1995, 1998) and Schölkopf and Smola (2002). Burges 1998 and Smola and Schölkopf 1998 are good tutorials on SVM classification and regression, respectively. Many free software packages are also available, and the ones that are most popular are SVMlight (Joachims 2008) and LIBSVM (Chang and Lin 2011).

13.16 Exercises

1. Propose a filtering algorithm to find training instances that are very unlikely to be support vectors.

   SOLUTION: Support vectors are those instances that are close to the boundaries. So if there is an instance surrounded by a large number of instances all of the same class, it will very probably not be chosen as a support vector. So, for example, we can do an 11-nearest neighbor search for all instances and if all its 11 neighbors are of the same class, we can prune that instance from the training set.

2. In equation 13.31, how can we estimate **S**?

   SOLUTION: We can calculate the covariance matrix of the data and use that as **S**. Another possibility is to have a local $\mathbf{S}^t$ for each support vector, and we can use a number of neighborhood data points to estimate it; we may need to take measures in such a case to make sure that **S** is not singular or decrease dimensionality is some way.

3. In the empirical kernel map, how can we choose the templates?

   SOLUTION: The easiest and most frequently used approach is to use all the training instances, and in such a case $\boldsymbol{\phi}(\cdot)$ is N-dimensional. We can decrease complexity and make the model more efficient by choosing a subset; we can use a randomly chosen subset, do some clustering, and use the cluster centers as templates (as in vector quantization), or use a subset that covers the input space well using as few instances as possible.

4. In the localized multiple kernel of equation 13.40, propose a suitable model for $\eta_i(\boldsymbol{x}|\theta_i)$ and discuss how it can be trained.

5. In kernel regression, what is the relation, if any, between ϵ and noise variance?

6. In kernel regression, what is the effect of using different ϵ on bias and variance?

   SOLUTION: ϵ is a smoothing parameter. When it is too large, we smooth too much, which reduces variance but risks increasing bias. If it is too small, the variance may be large and bias would be small.

7. Derive the kernelized version of the primal, dual, and the score functions for ranking..

   SOLUTION: The primal is

   $$L_p = \frac{1}{2}\|\boldsymbol{w}\|^2 + C\sum_t \xi^t$$

   subject to

   $$\begin{aligned} \boldsymbol{w}^T\boldsymbol{\phi}(\boldsymbol{x}^u - \boldsymbol{x}^v) &\geq 1 - \xi^t \\ \xi^t &\geq 0 \end{aligned}$$

The dual is

$$L_d = \sum_t \alpha^t - \frac{1}{2}\sum_t \sum_s \alpha^t \alpha^s K(\boldsymbol{x}^u - \boldsymbol{x}^v, \boldsymbol{x}^k - \boldsymbol{x}^l)$$

where $K(\boldsymbol{x}^u - \boldsymbol{x}^v, \boldsymbol{x}^k - \boldsymbol{x}^l) = \boldsymbol{\phi}(\boldsymbol{x}^u - \boldsymbol{x}^v)^T \boldsymbol{\phi}(\boldsymbol{x}^k - \boldsymbol{x}^l)$.

For new test instance $\boldsymbol{x}$, the score is calculated as

$$g(\boldsymbol{x}) = \sum_t \alpha^t K(\boldsymbol{x}^u - \boldsymbol{x}^v, \boldsymbol{x})$$

8. How can we use one-class SVM for classification?

   SOLUTION: We can use a separate one-class SVM for each class and then combine them to make a decision. For example, for each class C_i, we fit a one-class SVM to find parameters α_i^t:

   $$\sum_t \alpha_i^t K_G(\boldsymbol{x}, \boldsymbol{x}^t)$$

   and this then can be taken as an estimator for $p(\boldsymbol{x}|C_i)$. If the priors are more or less equal, we can simply choose the class having the largest value; otherwise we can use Bayes' rule for classification.

9. In a setting such as that in figure 13.12, use kernel PCA with a Gaussian kernel.

10. Let us say we have two representations for the same object and associated with each, we have a different kernel. How can we use both to implement a joint dimensionality reduction using kernel PCA?

13.17 References

Allwein, E. L., R. E. Schapire, and Y. Singer. 2000. "Reducing Multiclass to Binary: A Unifying Approach for Margin Classifiers." *Journal of Machine Learning Research* 1:113–141.

Burges, C. J. C. 1998. "A Tutorial on Support Vector Machines for Pattern Recognition." *Data Mining and Knowledge Discovery* 2:121–167.

Chang, C.-C., and C.-J. Lin. 2011. *LIBSVM: A Library for Support Vector Machines. ACM Transactions on Intelligent Systems and Technology* 2: 27:1–27:27.

Chapelle, O., and S. S. Keerthi. 2010. "Efficient Algorithms for Ranking with SVMs." *Information Retrieval* 11:201–215.

Cherkassky, V., and F. Mulier. 1998. *Learning from Data: Concepts, Theory, and Methods.* New York: Wiley.

Cortes, C., and V. Vapnik. 1995. "Support Vector Networks." *Machine Learning* 20:273–297.

Dietterich, T. G., and G. Bakiri. 1995. “Solving Multiclass Learning Problems via Error-Correcting Output Codes.” *Journal of Artificial Intelligence Research* 2: 263–286.

Gönen, M., and E. Alpaydın. 2008. “Localized Multiple Kernel Learning.” In *25th International Conference on Machine Learning*, ed. A. McCallum and S. Roweis, 352–359. Madison, WI: Omnipress.

Gönen, M., and E. Alpaydın. 2011. “Multiple Kernel Learning Algorithms.” *Journal of Machine Learning Research* 12:2211–2268.

Hardoon, D. R., S. Szedmak, J. Shawe-Taylor. 2004. “Canonical Correlation Analysis: An Overview with Application to Learning Methods.” *Neural Computation* 16:2639–2664.

Herbrich, R., K. Obermayer, and T. Graepel. 2000. “Large Margin Rank Boundaries for Ordinal Regression.” In *Advances in Large Margin Classifiers*, ed. A. J. Smola, P. Bartlett, B. Schölkopf and D. Schuurmans, 115–132. Cambridge, MA: MIT Press.

Jaakkola, T., and D. Haussler. 1999. “Exploiting Generative Models in Discriminative Classifiers.” In *Advances in Neural Information Processing Systems 11*, ed. M. J. Kearns, S. A. Solla, and D. A. Cohn, 487–493. Cambridge, MA: MIT Press.

Joachims, T. 2002. “Optimizing Search Engines using Clickthrough Data.” In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 133–142. New York, NY: ACM.

Joachims, T. 2008. *SVMlight*, `http://svmlight.joachims.org`.

Lanckriet, G. R. G, N. Cristianini, P. Bartlett, L. El Ghaoui, and M. I. Jordan. 2004. “Learning the Kernel Matrix with Semidefinite Programming.” *Journal of Machine Learning Research* 5: 27–72.

Liu, T.-Y. 2011. *Learning to Rank for Information Retrieval*. Heidelberg: Springer.

Mayoraz, E., and E. Alpaydın. 1999. “Support Vector Machines for Multiclass Classification.” In *Foundations and Tools for Neural Modeling, Proceedings of IWANN’99, LNCS 1606*, ed. J. Mira and J. V. Sanchez-Andres, 833–842. Berlin: Springer.

Müller, K. R., S. Mika, G. Rätsch, K. Tsuda, and B. Schölkopf. 2001. “An Introduction to Kernel-Based Learning Algorithms.” *IEEE Transactions on Neural Networks* 12:181–201.

Noble, W. S. 2004. “Support Vector Machine Applications in Computational Biology.” In *Kernel Methods in Computational Biology*, ed. B. Schölkopf, K. Tsuda, and J.-P. Vert, 71–92. Cambridge, MA: MIT Press.

Platt, J. 1999. "Probabilities for Support Vector Machines." In *Advances in Large Margin Classifiers*, ed. A. J. Smola, P. Bartlett, B. Schölkopf, and D. Schuurmans, 61–74. Cambridge, MA: MIT Press.

Schölkopf, B., J. Platt, J. Shawe-Taylor, A. J. Smola, and R. C. Williamson. 2001. "Estimating the Support of a High-Dimensional Distribution." *Neural Computation* 13:1443–1471.

Schölkopf, B., and A. J. Smola. 2002. *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond.* Cambridge, MA: MIT Press.

Schölkopf, B., A. J. Smola, R. C. Williamson, and P. L. Bartlett. 2000. "New Support Vector Algorithms." *Neural Computation* 12:1207–1245.

Schölkopf, B., K. Tsuda, and J.-P. Vert, eds. 2004. *Kernel Methods in Computational Biology.* Cambridge, MA: MIT Press.

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

Smola, A., and B. Schölkopf. 1998. *A Tutorial on Support Vector Regression*, NeuroCOLT TR-1998-030, Royal Holloway College, University of London, UK.

Sonnenburg, S., G. Rätsch, C. Schäfer, and B. Schölkopf. 2006. "Large Scale Multiple Kernel Learning." *Journal of Machine Learning Research* 7:1531–1565.

Tax, D. M. J., and R. P. W. Duin. 1999. "Support Vector Domain Description." *Pattern Recognition Letters* 20:1191–1199.

Torresani, L., and K. C. Lee. 2007. "Large Margin Component Analysis." In *Advances in Neural Information Processing Systems 19*, ed. B. Schölkopf, J. Platt, and T. Hoffman, 1385–1392. Cambridge, MA: MIT Press.

Vapnik, V. 1995. *The Nature of Statistical Learning Theory.* New York: Springer.

Vapnik, V. 1998. *Statistical Learning Theory.* New York: Wiley.

Vert, J.-P., K. Tsuda, and B. Schölkopf. 2004. "A Primer on Kernel Methods." In *Kernel Methods in Computational Biology*, ed. B. Schölkopf, K. Tsuda, and J.-P. Vert, 35–70. Cambridge, MA: MIT Press.

Weinberger, K. Q., and L. K. Saul. 2009. "Distance Metric Learning for Large Margin Classification." *Journal of Machine Learning Research* 10:207–244.

Weston, J., and C. Watkins. 1998. "Multiclass Support Vector Machines." *Technical Report CSD-TR-98-04*, Department of Computer Science, Royal Holloway, University of London.

14 *Graphical Models*

Graphical models represent the interaction between variables visually and have the advantage that inference over a large number of variables can be decomposed into a set of local calculations involving a small number of variables making use of conditional independencies. After some examples of inference by hand, we discuss the concept of d-separation and the belief propagation algorithm on a variety of graphs.

14.1 Introduction

GRAPHICAL MODELS
BAYESIAN NETWORKS
BELIEF NETWORKS
PROBABILISTIC NETWORKS

Graphical models, also called *Bayesian networks*, *belief networks*, or *probabilistic networks*, are composed of nodes and arcs between the nodes. Each node corresponds to a random variable, X, and has a value corresponding to the probability of the random variable, $P(X)$. If there is a directed arc from node X to node Y, this indicates that X has a *direct influence* on Y. This influence is specified by the conditional probability $P(Y|X)$. The network is a *directed acyclic graph* (DAG); namely, there are no cycles. The nodes and the arcs between the nodes define the *structure* of the network, and the conditional probabilities are the *parameters* given the structure.

DIRECTED ACYCLIC GRAPH

A simple example is given in figure 14.1, which models that rain causes the grass to get wet. It rains on 40 percent of the days and when it rains, there is a 90 percent chance that the grass gets wet; maybe 10 percent of the time it does not rain long enough for us to really consider the grass wet enough. The random variables in this example are binary; they are either true or false. There is a 20 percent probability that the grass gets wet without its actually raining, for example, when a sprinkler is used.

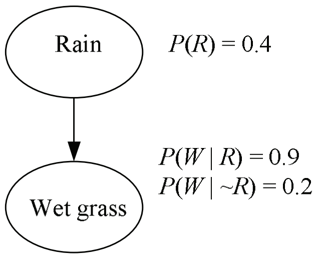


Figure 14.1 Bayesian network modeling that rain is the cause of wet grass.

We see that these three values completely specify the joint distribution of $P(R, W)$. If $P(R) = 0.4$, then $P({\sim}R) = 0.6$, and similarly $P({\sim}W|R) = 0.1$ and $P({\sim}W|{\sim}R) = 0.8$. The joint is written as

$$P(R, W) = P(R)P(W|R)$$

We can calculate the individual (marginal) probability of wet grass by summing up over the possible values that its parent node can take:

$$\begin{aligned} P(W) &= \sum_R P(R, W) = P(W|R)P(R) + P(W|{\sim}R)P({\sim}R) \\ &= 0.9 \cdot 0.4 + 0.2 \cdot 0.6 = 0.48 \end{aligned}$$

If we knew that it rained, the probability of wet grass would be 0.9; if we knew for sure that it did not, it would be as low as 0.2; not knowing whether it rained or not, the probability is 0.48.

CAUSAL GRAPH

Figure 14.1 shows a *causal graph* in that it explains that the cause of wet grass is rain. Bayes' rule allows us to invert the dependencies and have a *diagnosis*. For example, knowing that the grass is wet, the probability that it rained can be calculated as follows:

$$P(R|W) = \frac{P(W|R)P(R)}{P(W)} = 0.75$$

Knowing that the grass is wet increased the probability of rain from 0.4 to 0.75; this is because $P(W|R)$ is high and $P(W|{\sim}R)$ is low.

INDEPENDENCE

We form graphs by adding nodes and arcs and generate dependencies. X and Y are *independent events* if

$$p(X, Y) = P(X)P(Y) \tag{14.1}$$

CONDITIONAL INDEPENDENCE

X and Y are *conditionally independent events* given a third event Z if

(14.2) $$P(X, Y|Z) = P(X|Z)P(Y|Z)$$

which can also be rewritten as

(14.3) $$P(X|Y, Z) = P(X|Z)$$

In a graphical model, not all nodes are connected; actually, in general, a node is connected to only a small number of other nodes. Certain subgraphs imply conditional independence statements, and these allow us to break down a complex graph into smaller subsets in which inferences can be done locally and whose results are later propagated over the graph. There are three canonical cases and larger graphs are constructed using these as subgraphs.

14.2 Canonical Cases for Conditional Independence

Case 1: Head-to-Tail Connection

Three events may be connected serially, as seen in figure 14.2a. We see here that X and Z are independent given Y: Knowing Y tells Z everything; knowing the state of X does not add any extra knowledge for Z; we write $P(Z|Y, X) = P(Z|Y)$. We say that Y *blocks* the path from X to Z, or in other words, it *separates* them in the sense that if Y is removed, there is no path between X to Z. In this case, the joint is written as

(14.4) $$P(X, Y, Z) = P(X)P(Y|X)P(Z|Y)$$

Writing the joint this way implies independence:

(14.5) $$P(Z|X, Y) = \frac{P(X, Y, Z)}{P(X, Y)} = \frac{P(X)P(Y|X)P(Z|Y)}{P(X)P(Y|X)} = P(Z|Y)$$

Typically, X is the cause of Y and Y is the cause of Z. For example, as seen in figure 14.2b, X can be cloudy sky, Y can be rain, and Z can be wet grass. We can propagate information along the chain. If we do not know the state of cloudy, we have

$$\begin{aligned} P(R) &= P(R|C)P(C) + P(R|\sim C)P(\sim C) = 0.38 \\ P(W) &= P(W|R)P(R) + P(W|\sim R)P(\sim R) = 0.48 \end{aligned}$$

Let us say in the morning we see that the weather is cloudy; what can we say about the probability that the grass will be wet? To do this, we

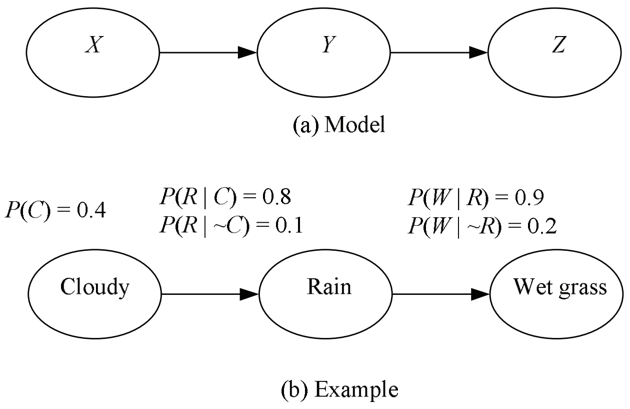


Figure 14.2 Head-to-tail connection. (a) Three nodes are connected serially. X and Z are independent given the intermediate node Y: $P(Z|Y,X) = P(Z|Y)$. (b) Example: Cloudy weather causes rain, which in turn causes wet grass.

need to propagate evidence first to the intermediate node R, and then to the query node W.

$$P(W|C) = P(W|R)P(R|C) + P(W|{\sim}R)P({\sim}R|C) = 0.76$$

Knowing that the weather is cloudy increased the probability of wet grass. We can also propagate evidence back using Bayes' rule. Let us say that we were traveling and on our return, see that our grass is wet; what is the probability that the weather was cloudy that day? We use Bayes' rule to invert the direction:

$$P(C|W) = \frac{P(W|C)P(C)}{P(W)} = 0.65$$

Knowing that the grass is wet increased the probability of cloudy weather from its default (prior) value of 0.4 to 0.65.

Case 2: Tail-to-Tail Connection

X may be the parent of two nodes Y and Z, as shown in figure 14.3a. The joint density is written as

$$P(X,Y,Z) = P(X)P(Y|X)P(Z|X) \tag{14.6}$$

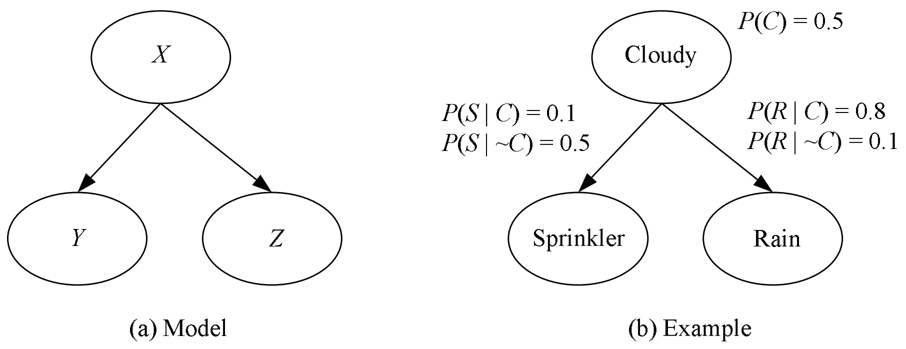


Figure 14.3 Tail-to-tail connection. X is the parent of two nodes Y and Z. The two child nodes are independent given the parent: $P(Y|X,Z) = P(Y|X)$. In the example, cloudy weather causes rain and also makes us less likely to turn the sprinkler on.

Normally Y and Z are dependent through X; given X, they become independent:

$$P(Y,Z|X) = \frac{P(X,Y,Z)}{P(X)} = \frac{P(X)P(Y|X)P(Z|X)}{P(X)} = P(Y|X)P(Z|X) \tag{14.7}$$

When its value is known, X blocks the path between Y and Z or, in other words, separates them.

In figure 14.3b, we see an example where cloudy weather influences both rain and the use of the sprinkler, one positively and the other negatively. Knowing that it rained, for example, we can invert the dependency using Bayes' rule and infer the cause:

$$\begin{aligned} P(C|R) &= \frac{P(R|C)P(C)}{P(R)} = \frac{P(R|C)P(C)}{\sum_C P(R,C)} \\ &= \frac{P(R|C)P(C)}{P(R|C)P(C) + P(R|{\sim}C)P({\sim}C)} = 0.89 \end{aligned} \tag{14.8}$$

Note that this value is larger than $P(C)$; knowing that it rained increased the probability that the weather is cloudy.

In figure 14.3a, if X is not known, knowing Y, for example, we can infer X that we can then use to infer Z. In figure 14.3b, knowing the state of the sprinkler has an effect on the probability that it rained. If we know that the sprinkler is on,

$$P(R|S) = \sum_C P(R,C|S) = P(R|C)P(C|S) + P(R|{\sim}C)P({\sim}C|S) \tag{14.9}$$

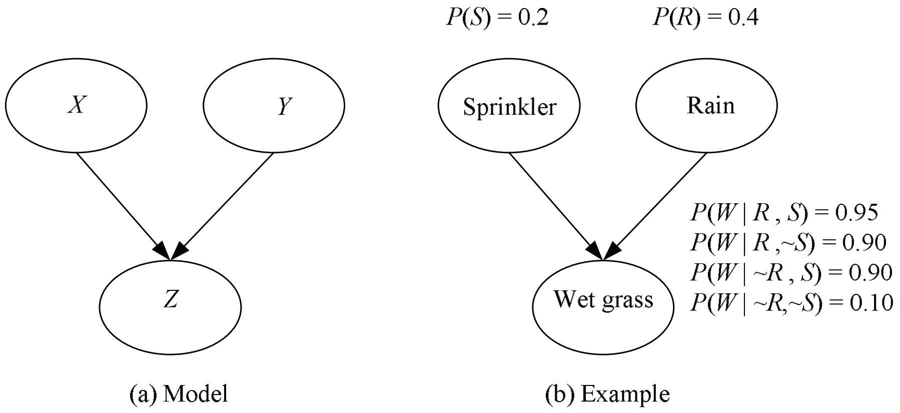


Figure 14.4 Head-to-head connection. A node has two parents that are independent unless the child is given. For example, an event may have two independent causes.

$$\begin{aligned} &= P(R|C)\frac{P(S|C)P(C)}{P(S)} + P(R|{\sim}C)\frac{P(S|{\sim}C)P({\sim}C)}{P(S)} \\ &= 0.22 \end{aligned}$$

This is less than $P(R) = 0.45$; that is, knowing that the sprinkler is on decreases the probability that it rained because sprinkler and rain happens for different states of cloudy weather. If the sprinkler is known to be off, using the same approach, we find that $P(R|{\sim}S) = 0.55$; the probability of rain increases this time.

Case 3: Head-to-Head Connection

In a head-to-head node, there are two parents X and Y to a single node Z, as shown in figure 14.4a. The joint density is written as

$$P(X,Y,Z) = P(X)P(Y)P(Z|X,Y) \tag{14.10}$$

X and Y are independent: $P(X,Y) = P(X) \cdot P(Y)$ (exercise 2); they become dependent when Z is known. The concept of blocking or separation is different for this case: The path between X and Y is blocked, or they are separated, when Z is *not* observed; when Z (or any of its descendants) is observed, they are not blocked, separated, or independent.

We see, for example, in figure 14.4b that node W has two parents, R and S, and thus its probability is conditioned on the values of those two, $P(W|R,S)$.

Not knowing anything else, the probability that grass is wet is calculated by marginalizing over the joint:

$$\begin{aligned}P(W) &= \sum_{R,S} P(W,R,S)\\ &= P(W|R,S)P(R,S) + P(W|{\sim}R,S)P({\sim}R,S)\\ &\quad +P(W|R,{\sim}S)P(R,{\sim}S) + P(W|{\sim}R,{\sim}S)P({\sim}R,{\sim}S)\\ &= P(W|R,S)P(R)P(S) + P(W|{\sim}R,S)P({\sim}R)P(S)\\ &\quad +P(W|R,{\sim}S)P(R)P({\sim}S) + P(W|{\sim}R,{\sim}S)P({\sim}R)P({\sim}S)\\ &= 0.52\end{aligned}$$

Now, let us say that we know that the sprinkler is on, and we check how this affects the probability. This is a causal (predictive) inference:

$$\begin{aligned}P(W|S) &= \sum_{R} P(W,R|S)\\ &= P(W|R,S)P(R|S) + P(W|{\sim}R,S)P({\sim}R|S)\\ &= P(W|R,S)P(R) + P(W|{\sim}R,S)P({\sim}R)\\ &= 0.92\end{aligned}$$

We see that $P(W|S) > P(W)$; knowing that the sprinkler is on, the probability of wet grass increases.

We can also calculate the probability that the sprinkler is on, given that the grass is wet. This is a diagnostic inference.

$$P(S|W) = \frac{P(W|S)P(S)}{P(W)} = 0.35$$

$P(S|W) > P(S)$, that is, knowing that the grass is wet increased the probability of having the sprinkler on. Now let us assume that it rained. Then we have

$$\begin{aligned}P(S|R,W) &= \frac{P(W|R,S)P(S|R)}{P(W|R)} = \frac{P(W|R,S)P(S)}{P(W|R)}\\ &= 0.21\end{aligned}$$

EXPLAINING AWAY

which is less than $P(S|W)$. This is called *explaining away*; given that we know it rained, the probability of the sprinkler causing the wet grass

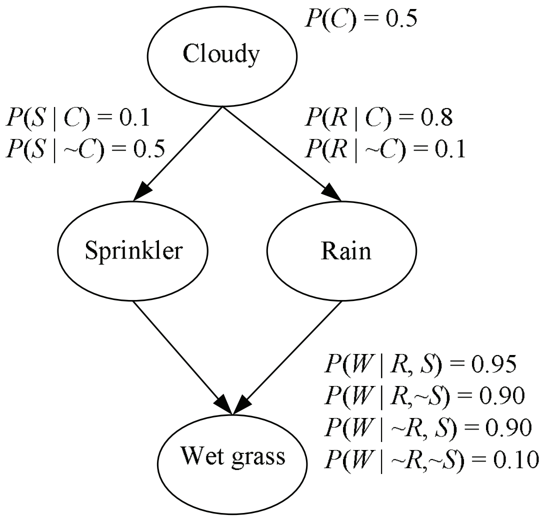


Figure 14.5 Larger graphs are formed by combining simpler subgraphs over which information is propagated using the implied conditional independencies.

decreases. Knowing that the grass is wet, rain and sprinkler become dependent. Similarly, $P(S|{\sim}R, W) > P(S|W)$. We see the same behavior when we compare $P(R|W)$ and $P(R|W, S)$ (exercise 3).

We can construct larger graphs by combining such subgraphs. For example, in figure 14.5 where we combine the two subgraphs, we can, for example, calculate the probability of having wet grass if it is cloudy:

$$\begin{aligned}
P(W|C) &= \sum_{R,S} P(W, R, S|C) \\
&= P(W, R, S|C) + P(W, {\sim}R, S|C) \\
&\quad + P(W, R, {\sim}S|C) + P(W, {\sim}R, {\sim}S|C) \\
&= P(W|R, S, C)P(R, S|C) \\
&\quad + P(W|{\sim}R, S, C)P({\sim}R, S|C) \\
&\quad + P(W|R, {\sim}S, C)P(R, {\sim}S|C) \\
&\quad + P(W|{\sim}R, {\sim}S, C)P({\sim}R, {\sim}S|C) \\
&= P(W|R, S)P(R|C)P(S|C) \\
&\quad + P(W|{\sim}R, S)P({\sim}R|C)P(S|C) \\
&\quad + P(W|R, {\sim}S)P(R|C)P({\sim}S|C)
\end{aligned}$$

$$+P(W|{\sim}R,{\sim}S)P({\sim}R|C)P({\sim}S|C)$$

where we have used that $P(W|R,S,C) = P(W|R,S)$; given R and S, W is independent of C: R and S between them block the path between W and C. Similarly, $P(R,S|C) = P(R|C)P(S|C)$; given C, R and S are independent. We see the advantage of Bayesian networks here, which explicitly encode independencies and allow breaking down inference into calculation over small groups of variables that are propagated from evidence nodes to query nodes.

We can calculate $P(C|W)$ and have a diagnostic inference:

$$P(C|W) = \frac{P(W|C)P(C)}{P(W)}$$

The graphical representation is visual and helps understanding. The network represents conditional independence statements and allows us to break down the problem of representing the joint distribution of many variables into *local* structures; this eases both analysis and computation. Figure 14.5 represents a joint density of four binary variables that would normally require fifteen values ($2^4 - 1$) to be stored, whereas here there are only nine. If each node has a small number of parents, the complexity decreases from exponential to linear (in the number of nodes). As we have seen earlier, inference is also easier as the joint density is broken down into conditional densities of smaller groups of variables:

$$P(C,S,R,W) = P(C)P(S|C)P(R|C)P(W|S,R) \tag{14.11}$$

In the general case, when we have variables $X_1, \ldots, X_d$, we write

$$P(X_1, \ldots, X_d) = \prod_{i=1}^{d} P(X_i|\text{parents}(X_i)) \tag{14.12}$$

Then given any subset of X_i, namely, setting them to certain values due to evidence, we can calculate the probability distribution of some other subset of X_i by marginalizing over the joint. This is costly because it requires calculating an exponential number of joint probability combinations, even though each of them can be simplified as in equation 14.11. Note, however, that given the same evidence, for different X_i, we may be using the same intermediate values (products of conditional probabilities and sums for marginalization), and in section 14.5, we will discuss the belief propagation algorithm to do inference cheaply by doing the local intermediate calculations once which we can use multiple times for different query nodes.

Though in this example we use binary variables, it is straightforward to generalize for cases where the variables are discrete with any number of possible values (with m possible values and k parents, a table of size m^k is needed for the conditional probabilities), or they can be continuous (parameterized, e.g., $p(Y|x) \sim \mathcal{N}(\mu(x|\theta), \sigma^2)$; see figure 14.7).

One major advantage to using a Bayesian network is that we do not need to designate explicitly certain variables as input and certain others as output. The value of any set of variables can be established through evidence and the probabilities of any other set of variables can be inferred, and the difference between unsupervised and supervised learning becomes blurry. From this perspective, a graphical model can be thought of as a "probabilistic database" (Jordan 2004), a machine that can answer queries regarding the values of random variables.

HIDDEN VARIABLES

In a problem, there may also be *hidden variables* whose values are never known through evidence. The advantage of using hidden variables is that the dependency structure can be more easily defined. For example, in basket analysis when we want to find the dependencies among items sold, let us say we know that there is a dependency among "baby food," "diapers," and "milk" in that a customer buying one of these is very much likely to buy the other two. Instead of putting (noncausal) arcs among these three, we may designate a hidden node "baby at home" as the hidden cause of the consumption of these three items. When there are hidden nodes, their values are estimated given the values of observed nodes and filled in.

CAUSALITY

It should be stressed at this point that a link from a node X does not, and need not, always imply a *causality*. It only implies a *direct influence* of X over Y in the sense that the probability of Y is conditioned on the value of X, and two nodes may have a link between them even if there is no direct cause. It is preferable to have the causal relations in constructing the network by providing an explanation of how the data is generated (Pearl 2000) but such causes may not always be accessible.

14.3 Generative Models

GENERATIVE MODEL

Still, graphical models are frequently used to visualize *generative models* for representing the process that we believe has created the data. For example, for the case of classification, the corresponding graphical model is shown in figure 14.6a, with $\boldsymbol{x}$ as the input and C a multinomial variable

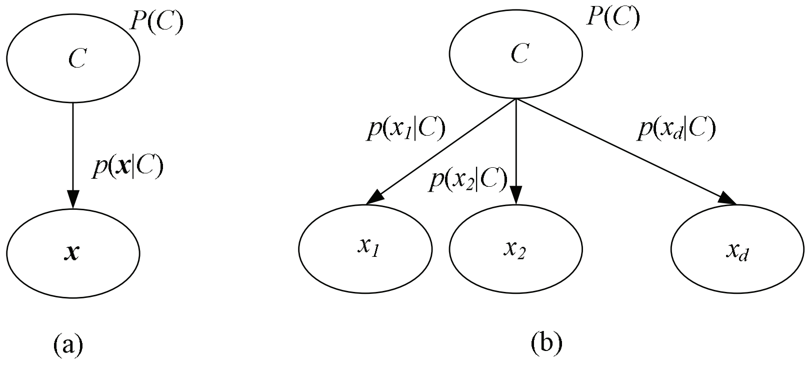


Figure 14.6 (a) Graphical model for classification. (b) Naive Bayes' classifier assumes independent inputs.

taking one of K states for the class code. It is as if we first pick a class C at random by sampling from $P(C)$, and then having fixed C, we pick an $\boldsymbol{x}$ by sampling from $p(\boldsymbol{x}|C)$. Bayes' rule inverts the generative direction and allows a diagnosis, as in the rain and wet grass case we saw in figure 14.1:

$$P(C|\boldsymbol{x}) = \frac{P(C)p(\boldsymbol{x}|C)}{P(\boldsymbol{x})}$$

Note that clustering is similar except that instead of the multinomial class indicator variable C we have the cluster indicator variable Z, and it is not observed during training. The E-step of the expectation-maximization algorithm (section 7.4) uses Bayes' rule to invert the arc and fills in the cluster indicator given the input.

NAIVE BAYES' CLASSIFIER

If the inputs are independent, we have the graph shown in figure 14.6b, which is called the *naive Bayes' classifier*, because it ignores possible dependencies, namely, correlations, among the inputs and reduces a multivariate problem to a group of univariate problems:

$$p(\boldsymbol{x}|C) = \prod_{j=1}^{d} p(x_j|C)$$

We have discussed classification for this case in sections 5.5 and 5.7 for numeric and discrete $\boldsymbol{x}$, respectively.

Linear regression can be visualized as a graphical model, as shown in figure 14.7. Input $\boldsymbol{x}^t$ is drawn from a prior $p(\boldsymbol{x})$, and the dependent variable r^t depends on the input $\boldsymbol{x}$ and the weights $\boldsymbol{w}$. Here, we define a

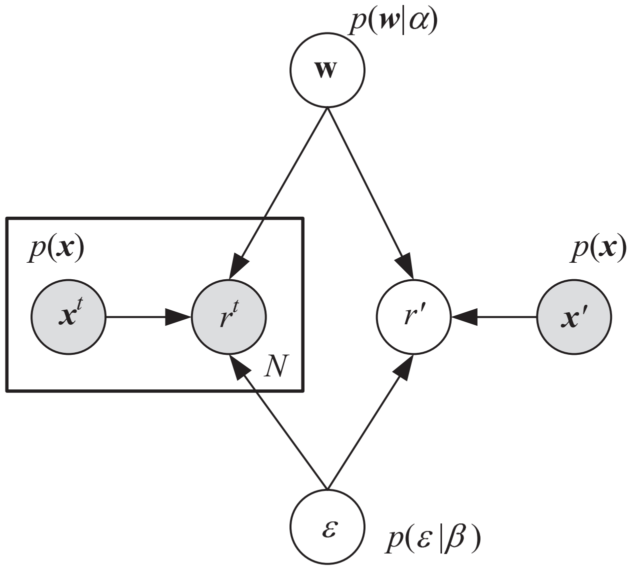


Figure 14.7 Graphical model for linear regression.

node for the weights $\boldsymbol{w}$ with a prior parameterized by α, namely, $p(\boldsymbol{w}) \sim \mathcal{N}(\mathbf{0}, \alpha^{-1}\mathbf{I})$. There is also a node for the noise ϵ variable, parameterized by β, namely, $p(\epsilon) \sim \mathcal{N}(0, \beta^{-1})$:

$$p(r^t|\boldsymbol{x}^t, \boldsymbol{w}) \sim \mathcal{N}(\boldsymbol{w}^T\boldsymbol{x}^t, \beta^{-1}) \tag{14.13}$$

There are N such pairs in the training set, which is shown by the rectangular *plate* in the figure—the plate corresponds to the training set $\mathcal{X}$. Given a new input $\boldsymbol{x}'$, the aim is to estimate r'. The weights $\boldsymbol{w}$ are not given but they can be estimated using the training set of $\mathcal{X}$ which we can divide as $[\mathbf{X}, \boldsymbol{r}]$.

In equation 14.9, where C is the cause of R and S, we write

$$P(R|S) = \sum_C P(R, C|S) = P(R|C)P(C|S) + P(R|{\sim}C)P({\sim}C|S)$$

filling in C using observed S and average over all possible values of C. Similarly here, we write

$$\begin{aligned} p(r'|\boldsymbol{x}', \boldsymbol{r}, \mathbf{X}) &= \int p(r'|\boldsymbol{x}', \boldsymbol{w})p(\boldsymbol{w}|\mathbf{X}, \boldsymbol{r})d\boldsymbol{w} \\ &= \int p(r'|\boldsymbol{x}', \boldsymbol{w})\frac{p(\boldsymbol{r}|\mathbf{X}, \boldsymbol{w})p(\boldsymbol{w})}{p(\boldsymbol{r})}d\boldsymbol{w} \end{aligned}$$

$$\propto \int p(r'|\boldsymbol{x}', \boldsymbol{w}) \prod_t p(r^t|\boldsymbol{x}^t, \boldsymbol{w}) p(\boldsymbol{w}) d\boldsymbol{w} \qquad (14.14)$$

where the second line is due to Bayes' rule and the third line is due to the independence of instances in the training set.

Note that what we have in figure 14.7 is a Bayesian model where we designate parameter $\boldsymbol{w}$ as a random variable with a prior distribution. As we see in equation 14.14, what we are effectively doing is estimating the posterior $p(\boldsymbol{w}|\mathbf{X}, \boldsymbol{r})$ and then integrating over it. We began discussing this in section 4.4, and we discuss it in greater detail in chapter 16, for different generative models and different sets of parameters.

14.4 d-Separation

d-SEPARATION

We now generalize the concept of blocking and separation under the name of *d-separation*, and we define it in a way so that for arbitrary subsets of nodes A, B, and C, we can check if A and B are independent given C. Jordan (2004) visualizes this as a ball bouncing over the graph and calls this the *Bayes' ball*. We set the nodes in C to their values, place a ball at each node in A, let the balls move around according to a set of rules, and check whether a ball reaches any node in B. If this is the case, they are dependent; otherwise, they are independent.

BAYES' BALL

To check whether A and B are d-separated given C, we consider all possible paths between any node in A and any node in B. Any such path is *blocked* if

(a) the directions of the edges on the path either meet head-to-tail (case 1) or tail-to-tail (case 2) and the node is in C, or

(b) the directions of the edges on the path meet head-to-head (case 3) and neither that node nor any of its descendant is in C.

If all paths are blocked, we say that A and B are d-separated, that is, independent, given C; otherwise, they are dependent. Examples are given in figure 14.8.

14.5 Belief Propagation

Having discussed some inference examples by hand, we now are interested in an algorithm that can answer queries such as $P(X|E)$ where X

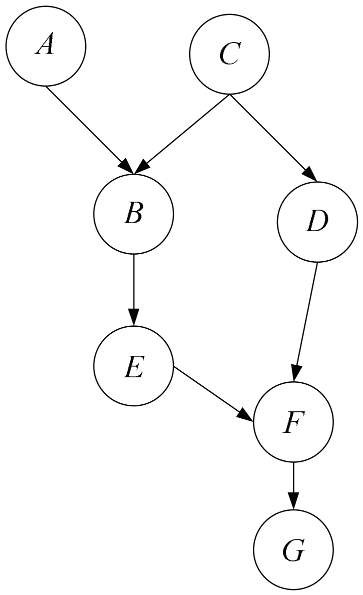


Figure 14.8 Examples of d-separation. The path $BCDF$ is blocked given C because C is a tail-to-tail node. $BEFG$ is blocked by F because F is a head-to-tail node. $BEFD$ is blocked unless F (or G) is given.

is any *query node* in the graph and E is any subset of *evidence nodes* whose values are set to certain value. Following Pearl (1988), we start with the simplest case of chains and gradually move on to more complex graphs. Our aim is to find the graph operation counterparts of probabilistic procedures such as Bayes' rule or marginalization, so that the task of inference can be mapped to general-purpose graph algorithms.

14.5.1 Chains

A *chain* is a sequence of head-to-tail nodes with one *root* node without any parent; all other nodes have exactly one parent node, and all nodes except the very last, *leaf*, have a single child. If evidence is in the ancestors of X, we can just do a diagnostic inference and propagate evidence down the chain; if evidence is in the descendants of X, we can do a causal inference and propagate upward using Bayes' rule. Let us see the general case where we have evidence in both directions, up the chain E^+ and

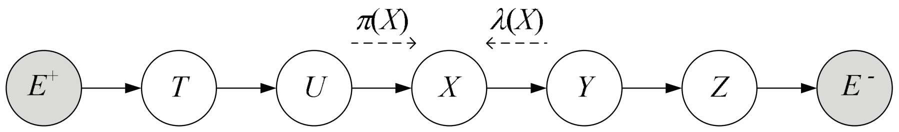


Figure 14.9 Inference along a chain.

down the chain E^- (see figure 14.9). Note that any evidence node separates X from the nodes on the chain on the other side of the evidence and their values do not affect $p(X)$; this is true in both directions.

We consider each node as a processor that receives messages from its neighbors and pass it along after some local calculation. Each node X locally calculates and stores two values: $\lambda(X) \equiv P(E^-|X)$ is the propagated E^- that X receives from its child and forwards to its parent, and $\pi(X) \equiv P(X|E^+)$ is the propagated E^+ that X receives from its parent and passes on to its child.

$$\begin{aligned}
P(X|E) &= \frac{P(E|X)P(X)}{P(E)} = \frac{P(E^+,E^-|X)P(X)}{P(E)} \\
&= \frac{P(E^+|X)P(E^-|X)P(X)}{P(E)} \\
&= \frac{P(X|E^+)P(E^+)P(E^-|X)P(X)}{P(X)P(E)} \\
&= \alpha P(X|E^+)P(E^-|X) = \alpha\pi(X)\lambda(X) \qquad (14.15)
\end{aligned}$$

for some normalizing constant α, not dependent on the value of X. The second line is there because E^+ and E^- are independent given X, and the third line is due to Bayes' rule.

If a node E is instantiated to a certain value $\tilde{e}$, $\lambda(\tilde{e}) \equiv 1$ and $\lambda(e) \equiv 0$, for $e \neq \tilde{e}$. The leaf node X that is not instantiated has its $\lambda(x) \equiv 1$, for all x values. The root node X that is not instantiated takes the prior probabilities as π values: $\pi(x) \equiv P(x), \forall x$.

Given these initial conditions, we can devise recursive formulas to propagate evidence along the chain.

For the π-messages, we have

$$\begin{aligned}
\pi(X) &\equiv P(X|E^+) = \sum_U P(X|U,E^+)P(U|E^+) \\
&= \sum_U P(X|U)P(U|E^+) = \sum_U P(X|U)\pi(U) \qquad (14.16)
\end{aligned}$$

where the second line follows from the fact that U blocks the path between X and E^+.

For the λ-messages, we have

$$\begin{aligned}\lambda(X) &\equiv P(E^-|X) = \sum_Y P(E^-|X,Y)P(Y|X) \\ &= \sum_Y P(E^-|Y)P(Y|X) = \sum_Y P(Y|X)\lambda(Y) \qquad (14.17)\end{aligned}$$

where the second line follows from the fact that Y blocks the path between X and E^-.

When the evidence nodes are set to a value, they initiate traffic and nodes continue updating until there is convergence. Pearl (1988) views this as a parallel machine where each node is implemented by a processor that works in parallel with others and exchanges information through λ- and π-messages with its parent and child.

14.5.2 Trees

Chains are restrictive because each node can have only a single parent and a single child, that is, a single cause and a single symptom. In a *tree*, each node may have several children but all nodes, except the single root, have exactly one parent. The same belief propagation also applies here with the difference from chains being that a node receives different λ-messages from its children, $\lambda_Y(X)$ denoting the message X receives from its child Y, and sends different π-messages to its children, $\pi_Y(X)$ denoting the message X sends to its child Y.

Again, we divide possible evidence to two parts, E^- are nodes that are in the subtree rooted at the query node X, and E^+ are evidence nodes elsewhere (see figure 14.10). Note that this second need not be an ancestor of X but may also be in a subtree rooted at a sibling of X. The important point is that again X separates E^+ and E^- so that we can write $P(E^+,E^-|X) = P(E^+|X)P(E^-|X)$, and hence have

$$P(X|E) = \alpha\pi(X)\lambda(X)$$

where again α is a normalizing constant.

$\lambda(X)$ is the evidence in the subtree rooted at X, and if X has two children Y and Z, as shown in figure 14.10, it can be calculated as

$$\begin{aligned}\lambda(X) &\equiv P(E_X^-|X) = P(E_Y^-, E_Z^-|X) \\ &= P(E_Y^-|X)P(E_Z^-|X) = \lambda_Y(X)\lambda_Z(X) \qquad (14.18)\end{aligned}$$