Chapter 21

FINDING TEMPLATES USING CLASSIFIERS

There are a number of important object recognition problems that involve looking for image windows which have a simple shape and stylised content. For example, frontal faces appear as oval windows, and (at a coarse scale) all faces look pretty much the same — a dark horizontal bar at the eyes and mouth, a light vertical bar along the nose, and not much texture on the cheeks and forehead. As another example, a camera mounted on the front of a car will always see relevant stop signs as having about the same shape and appearance.

This suggests a view of object recognition where we take all image windows of a particular shape and test them to tell if the relevant object is present. If we don't know how big the object will be, we can search over scale, too; if we don't know its orientation, we might search over orientation as well, etc. Generally, this approach is referred to as **template matching**. There are some objects that can be found very effectively with a template matcher. Faces and roadsigns are important examples. Secondly, while many objects appear to be hard to find with simple template matchers (it would be hard to find a person this way, because the collection of possible image windows that represent a person is immense), there is some evidence that reasoning about *relations* between many different kinds of templates can be an effective way to find objects. In chapter **??**, we explore this line of reasoning further.

The main issue to study in template matching is how one builds a test that can tell whether an oval represents a face or not. Ideally, this test will be obtained using a large set of examples. The test is known as a **classifier** — a classifier is anything that takes a feature set as an input and produces a class label. In this chapter, we describe a variety of techniques for building classifiers, with examples of their use in vision applications. We first present the key ideas and terminology used (section 21.1); we then show two successful classifiers built using histograms (section 21.2); for more complex classifiers, we need to choose the features a classifier should use, and we discuss two methods in (section 21.3). Finally, we describe two different methods for building classifiers with current applications in vision. Sec-

tion **??** is an introduction to the use of neural nets in classification; and section 21.5 describes a very useful classifier known as a support vector machine.

21.1 Classifiers

Classifiers are built by taking a set of labelled examples and using them to come up with a rule that will assign a label to any new example. In the general problem, we have a training data set $(\mathbf{x}_i, y_i)$; each of the $\boldsymbol{x}_i$ consists of measurements of the properties of different types of object, and the y_i are labels giving the type of the object that generated the example. We know the relative costs of mislabelling each class, and must come up with a rule that can take any plausible $\boldsymbol{x}$ and assign a class to it.

The cost of an error significantly affects the decision that is made. In section 21.1.1, we study this question. It will emerge that the probability of a class label given a measurement is the key matter. In section 21.1.2, we discuss methods for building appropriate models in a general way. Finally, we discuss how to estimate the performance of a given classifier (section 21.1.5).

21.1.1 Using Loss to Determine Decisions

The choice of classification rule must depend on the cost of making a mistake. For example, doctors engage in classification all the time — given a patient, they produce the name of a disease. A doctor who decided that a patient suffering from a dangerous and easily treated disease is well, is going to have problems. It would be better to err on the side of misclassifying healthy patients as sick even if doing so involves treating some healthy patients unnecessarily.

The cost depends on what is misclassified to what. Generally, we write outcomes as $(i \to j)$, meaning that an item of type i is classified as an item of type j. Each outcome has its own cost, which is known as a **loss**. Hence, we have a loss function which we shall write as $L(i \to j)$, meaning the loss incurred when an object of type i is classified as having type j. Since losses associated with correct classification should not affect the design of the classifier, $L(i \to i)$ must be zero; but the other losses could be any positive numbers.

The **risk function** of a particular classification strategy is the expected loss when using it, as a function of the kind of item. The **total risk** is the total expected loss when using the classifier. Thus if there were two classes, the total risk of using strategy s would be:

$$R(s) = Pr\{1 \to 2|\text{using } s\} L(1 \to 2) + Pr\{2 \to 1|\text{using } s\} L(2 \to 1)$$

The desirable strategy is one that minimizes this total risk.

Building a Two Class Classifier that Minimizes Total Risk

Assume that the classifier can choose between two classes, and we have a known loss function. There is some boundary in the feature space — which we call the

decision boundary — such that points on one side belong to class one and points on the other side to class two.

We can resort to a trick to determine where the decision boundary is. It must be the case that, *for points on the decision boundary of the optimal classifier*, either choice of class has the same expected loss — if this wasn't so, we could obtain a better classifier by always choosing one class (and so moving the boundary). This means that, for measurements on the decision boundary, choosing class one yields the same expected loss as choosing class two.

A choice of class one for a point $\boldsymbol{x}$ at the decision boundary yields an expected loss

$$\begin{aligned} P\left\{\text{class is } 2|\boldsymbol{x}\right\} L(2 \to 1) + P\left\{\text{class is } 1|\boldsymbol{x}\right\} L(1 \to 1) &= P\left\{\text{class is } 2|\boldsymbol{x}\right\} L(2 \to 1) + 0 \\ &= p(2|\boldsymbol{x})L(2 \to 1) \end{aligned}$$

(you should watch the one's and two's closely here). Similarly, a choice of class two for this point yields an expected loss

$$P\left\{\text{class is } 1|\boldsymbol{x}\right\} L(1 \to 2) = p(1|\boldsymbol{x})L(1 \to 2)$$

and these two terms must be equal. This means our decision boundary consists of the points $\boldsymbol{x}$ where

$$p(1|\boldsymbol{x})L(1 \to 2) = p(2|\boldsymbol{x})L(2 \to 1)$$

We can come up with an expression that is often slightly more practical, by using Bayes' rule. Rewrite our expression as

$$\frac{p(\boldsymbol{x}|1)p(1)}{p(\boldsymbol{x})}L(1 \to 2) = \frac{p(\boldsymbol{x}|2)p(2)}{p(\boldsymbol{x})}L(2 \to 1)$$

and clear denominators to get

$$p(\boldsymbol{x}|1)p(1)L(1 \to 2) = p(\boldsymbol{x}|2)p(2)L(2 \to 1)$$

This expression identifies points $\boldsymbol{x}$ on a class boundary; we now need to know how to classify points off a boundary.

At points off the boundary, we must choose the class with the *lowest* expected loss. Recall that, if we choose class two for a point $\boldsymbol{x}$, the expected loss is

$$p(1|\boldsymbol{x})L(1 \to 2)$$

etc. This means that we should choose class one if

$$p(1|\boldsymbol{x})L(1 \to 2) > p(2|\boldsymbol{x})L(2 \to 1)$$

and class two if

$$p(1|\boldsymbol{x})L(1 \to 2) < p(2|\boldsymbol{x})L(2 \to 1)$$

A Classifier for Multiple Classes

From now on, we shall assume that $L(i \to j)$ is zero for $i = j$ and one otherwise – that is, that each outcome has the same loss. In some problems, there is another option, which is to refuse to decide which class an object belongs to. This option involves some loss, too, which we shall assume to be $d < 1$ (if the loss involved in refusing to decide is greater than the loss involved in any decision, then we'd never refuse to decide).

For our loss function, the best strategy — which is known as the **Bayes classifier** — is given in algorithm 1. The total risk associated with this rule is known as the **Bayes risk** — this is the smallest possible risk that we can have in using a classifier. It is usually rather difficult to know what the Bayes classifier — and hence the Bayes risk — is, because the probabilities involved are not known exactly. In a few cases it is possible to write the rule out explicitly. One way to tell the effectiveness of a technique for building classifiers is to study the behaviour of the risk as the number of examples increases — for example, one might want the risk to converge to the Bayes risk in probability if the number of examples is very large. The Bayes risk is seldom zero, as figure 21.1 illustrates.

For a loss function

$$L(i \to j) = \begin{cases} 1 & i \neq j \\ 0 & i = j \\ d < 1 & \text{no decision} \end{cases}$$

the best strategy is

- *if* $Pr\{k|\mathbf{x}\} > Pr\{i|\mathbf{x}\}$ *for all* i *not equal to* k*, and if this probability is greater than* $1 - d$*, choose type* k
- *if there are several classes* $k_1 ... k_j$ *for which* $Pr\{k_1|\mathbf{x}\} = Pr\{k_2|\mathbf{x}\} = ... = Pr\{k_j|\mathbf{x}\} > Pr\{i|\mathbf{x}\}$ *for all* i *not in* $k_1, ..k_j$*, choose uniformly and at random between* $k_1, ..k_j$
- *if for all* k *we have* $Pr\{k|\mathbf{x}\} > Pr\{i|\mathbf{x}\} \leq 1 - d$*, refuse to decide.*

Algorithm 21.1: *The Bayes classifier classifies points using the posterior probability that an object belongs to a class, the loss function, and the prospect of refusing to decide.*

21.1.2 Overview: Methods for Building Classifiers

Usually, we do not know $Pr\{\mathbf{x}|k\}$ exactly — which are often called **class-conditional densities** — or $Pr\{k\}$, and must determine a classifier from an example data set. There are two rather general strategies:

- **Explicit probability models:** we can use the example data set to build a probability model (of either the likelihood or the posterior, depending on

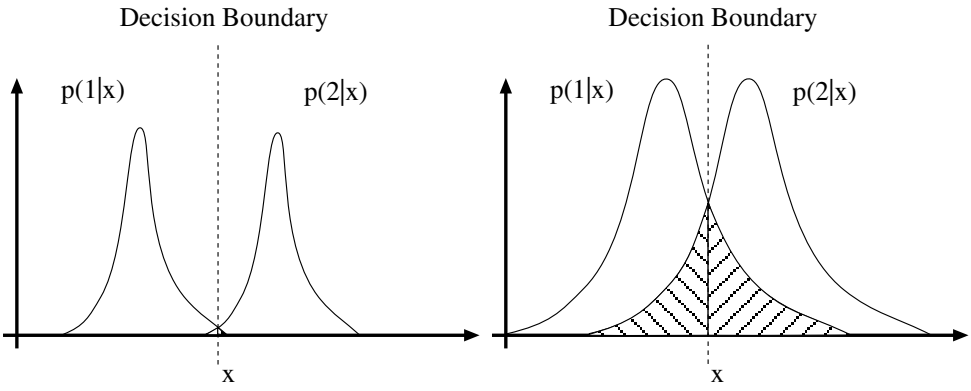


Figure 21.1. This figure shows typical elements of a two class classification problem. We have plotted $p(\text{class}|x)$ as a function of the feature x. Assuming that $L(1 \to 2) = L(2 \to 1)$, we have marked the classifier boundaries. In this case, the Bayes risk is the sum of the amount of the posterior for class one in the class two region and the amount of the posterior for class two in the class one region (the hatched area in the figures). For the case on the left, the classes are well separated, which means that the Bayes risk will be small; for the case on the right, the Bayes risk is rather large.

taste). There are a very wide variety of ways of doing this, some of which we shall see in the following sections. In the very simplest case, we know that the class-conditional densities come from some known parametric form of distribution. In this case, we can compute estimates of the parameters from the data set, and plug these estimates into the Bayes rule. This strategy is often known as a **"plug-in" classifier** (section **??**). This approach covers other parametric density models and other methods of estimating parameters. One subtlety is that the "best" estimate of a parameter may not give the best classifier, because the parametric model may not be correct. Another subtlety is that a good classifier may be obtained using a parametric density model that is not a very accurate description of the data (see figure 21.2). In many cases, it is hard to obtain a satisfactory model with a small number of parameters. More sophisticated modelling tools (such as neural nets, which we deal with in some detail in section 21.4) provide very flexible density models that can be fitted using data.

- **Determining decision boundaries directly:** Quite bad probability models can produce good classifiers, as figure 21.2 indicates. This is because the decision boundaries are what determine the performance of a classifier, not the details of the probability model (the main role of the probability model in the Bayes classifier is to identify the decision boundaries). This suggests that we might ignore the probability model, and attempt to construct good decision boundaries directly. This approach is often extremely successful; it is

particularly attractive when there is no reasonable prospect of modelling the data source. One strategy assumes that the decision boundary comes from one or another class, and constructs an extremisation problem to choose the best element of that class. A particularly important case comes when the data is **linearly separable** — which means that there exists a hyperplane with all the positive points on one side and all the negative points on the other — and thus that a hyperplane is all that is needed to separate the data (section **??**).

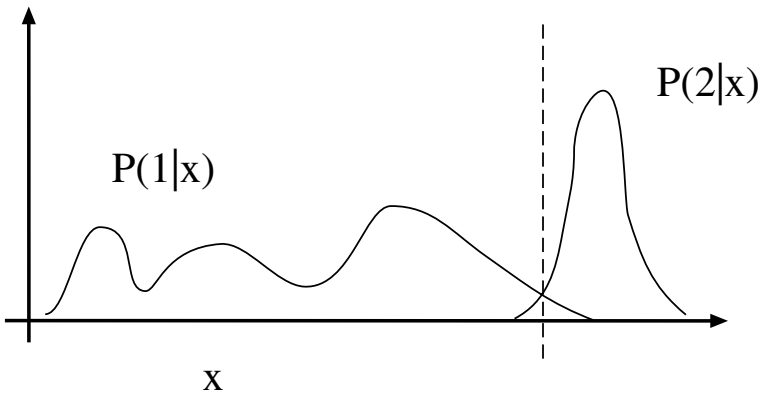


Figure 21.2. The figure shows posterior densities for two classes. The optimal decision boundary is shown as a dashed line. Notice that, while a normal density may provide rather a poor fit *to the posteriors*, the quality of the classifier it provides depends only on *how well it predicts the position of the boundaries*. In this case, assuming that the posteriors are normal may provide a fairly good classifier, because $P(2|x)$ looks normal, and the mean and covariance of $P(1|x)$ look as though they would predict the boundary in the right place.

21.1.3 Example: A Plug-in Classifier for Normal Class-conditional Densities

An important plug-in classifier occurs when the class-conditional densities are known to be normal. We can either assume that the priors are known, or estimate the priors by counting the number of data items from each class. Now we need to provide the parameters for the class-conditional densities. We do this as an estimation problem, using the data items to estimate the mean $\boldsymbol{\mu}_k$ and covariance Σ_k for each class. Now, since $\log a > \log b$ implies $a > b$, we can work with the logarithm of the posterior. This yields a classifier of the form in algorithm 2.

The term $\delta(\mathbf{x}; \boldsymbol{\mu}_k, \Sigma_k)$ in this algorithm is known as the **Mahalanobis distance** [**?**]. The algorithm can be interpreted geometrically as saying that the correct class is the one whose mean is closest to the data item, *taking into account the variance*. In particular, distance from a mean along a direction where there

Assume we have N classes, and the k'th class contains N_k examples, of which the i'th is written as $\boldsymbol{x}_{k,i}$.
For each class k, estimate the mean and standard deviation for that class-conditional density.

$$\boldsymbol{\mu}_k = \frac{1}{N_k}\sum_{i=1}^{N_k}\boldsymbol{x}_{k,i}$$

$$\Sigma_k = \frac{1}{N_k - 1}\sum_{i=1}^{N_k}(\boldsymbol{x}_{k,i} - \boldsymbol{\mu}_k)(\boldsymbol{x}_{k,i} - \boldsymbol{\mu}_k)^T$$

To classify an example $\boldsymbol{x}$

Choose the class k with the smallest value of $\delta(\mathbf{x}; \boldsymbol{\mu}_k, \Sigma_k)^2 - Pr\{k\}$

where

$$\delta(\mathbf{x}; \boldsymbol{\mu}_k, \Sigma_k) = \frac{1}{2}\left((\mathbf{x} - \boldsymbol{\mu}_k)^T\Sigma_k^{-1}(\mathbf{x} - \boldsymbol{\mu}_k)\right)^{(1/2)}$$

Algorithm 21.2: *A plug-in classifier can be used to classify objects into classes if the class-conditional densities are known to be normal*

is little variance has a large weight and distance from the mean along a direction where there is a large variance has little weight. This classifier can be simplified by assuming that each class has the same covariance (with the advantage that we have fewer parameters to estimate). In this case, because the term $\boldsymbol{x}^T\Sigma^{-1}\boldsymbol{x}$ is common to all expressions, the classifier actually involves comparing expressions that are *linear in* $\boldsymbol{x}$ (exercise **??**). If there are only two classes the process boils down to determining whether a linear expression in $\mathbf{x}$ is greater than or less than zero (exercise **??**).

21.1.4 Example: A Non-Parametric Classifier using Nearest Neighbours

It is reasonable to assume that example points "near" an unclassified point should indicate the class of that point. **Nearest neighbours** methods build classifiers using this heuristic. We could classify a point by using the class of the nearest example whose class is known, or use several example points, and make them vote. It is reasonable to require that some minimum number of points vote for the class we choose.

A (k, l) nearest neighbour classifier finds the k example points closest to the point being considered, and classifies this point with the class that has the highest number of votes, as long as this class has more than l votes (otherwise the point is classified as unknown). A $(k, 0)$-nearest neighbour classifier is usually known as a

k-nearest neighbour classifier, and a $(1, 0)$-nearest neighbour classifier is usually known as a **nearest neighbour classifier**.

Nearest neighbour classifiers are known to be good, in the sense that the risk of using a nearest neighbour classifier with a sufficiently large number of examples lies within quite good bounds of the Bayes risk. As k grows, the difference between the Bayes risk and the risk of using a k-nearest neighbour classifier goes down as $1/\sqrt{k}$; in practice, one seldom uses more than three nearest neighbours. Furthermore, if the Bayes risk is zero, the expected risk of using a k-nearest neighbour classifier is also zero (see [Devroye *et al.*, 1996] for more detail on all these points).

Nearest neighbour classifiers come with some computational subtleties, however. The first is the question of finding the k nearest points, which is no mean task in a high-dimensional space. This task can be simplified by noticing that some of the example points may be superfluous. If, when we remove a point from the example set, the set still classifies every point in the space in the same way (the decision boundaries have not moved), then that point is redundant and can be removed. The decision regions for (k, l)-nearest neighbour classifiers are convex polytopes; this makes familiar algorithms available in 2D — where Voronoi diagrams implement the nearest neighbour classifier — but leads to complications in high dimensions, where optimal algorithms are not known as of writing.

Given an feature vector $\boldsymbol{x}$

1. determine the k training examples that are nearest, $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k$;
2. determine the class c that has the largest number of representatives n in this set;
3. if $n > l$, classify $\boldsymbol{x}$ as c, otherwise refuse to classify it.

Algorithm 21.3: *A (k, l) nearest neighbour classifier uses the type of the nearest training examples to classify a feature vector*

A second difficulty in building such classifiers is the choice of distance. For features that are obviously of the same type, such as lengths, the usual metric may be good enough. But what if one feature is a length, one is a colour, and one is an angle? One possibility is to use a covariance estimate to compute a Mahalanobis-like distance.

21.1.5 Estimating and Improving Performance

Typically, classifiers are chosen to work well on the training set, and this can mean that the performance of the classifier on the training set is a poor guide to its overall performance. One example of this problem is the (silly) classifier that takes any data point and, if it is the same as a point in the training set, emits the class of that point and otherwise chooses randomly between the classes. This classifier has

been learnt from data, and has a zero error rate on the training data set; it is likely to be unhelpful on any other data set, though.

The difficulty occurs because classifiers are subject to **overfitting** effects. The phenomenon, which is known by a variety of names (**selection bias** is quite widely used), has to do with the fact that the classifier is chosen to perform well *on the training data set*. The training data is a (possibly representative) subset of the available possibilities. The term overfitting is descriptive of the source of the problem, which is that the classifier's performance on the training data set may have to do with quirks of that data set that don't occur in other sets of examples. If the classifier does this, it is quite possible that it will perform very well on the training data and very badly on any other data set (this phenomenon is often referred to as **generalising badly**).

Generally, we expect classifiers to perform somewhat better on the training set than on the test set (for example, see figure 21.16, which shows training set and test set errors for a classifier that is known to work very well). Overfitting can result in a substantial difference between performance on the training set and performance on the test set. This leaves us with the problem of predicting performance. There are two possible approaches: we can hold back some training data to check the performance of the classifier (an approach we describe below), or we can use theoretical methods to bound the future error rate of the classifier (see, for example, []).

Estimating Total Risk with Cross-Validation

We can make direct estimates of the expected risk of using a classifier, if we split the data set into two subsets, train the classifier on one subset and test it on the other. This is a waste of data, particularly if we have very few data items for a particular class, and may lead to an inferior classifier. However, if the size of the test subset is small, the difficulty may not be significant. In particular, we could then estimate total risk by averaging over all possible splits. This technique, known as **cross-validation**, allows an estimate of the likely future performance of a classifier, at the expense of substantial computation.

The most usual form of this algorithm involves omitting single items from the data set, and is known as **leave-one-out cross-validation**. Errors are usually estimated by simply averaging over the class, but more sophisticated estimates are available [**?**]. We will not justify this tool mathematically; however, it is worth noticing that leave-one-out cross-validation in some sense looks at the sensitivity of the classifier to a small change in the training set. If a classifier performs well under this test, then large subsets of the data set look similar to one another, which suggests that a representation of the relevant probabilities derived from the data set might be quite good.

Using Bootstrapping to Improve Performance

Generally, more training data leads to a better classifier. However, training classifiers with very large data sets can be difficult, and there are diminishing returns.

```
Choose some class of subsets of the training set,
for example, singletons.

For each element of that class, construct a classifier by
omitting that element in training, and compute the
classification errors (or risk) on the omitted subset.

Average these errors over the class of subsets to estimate
the risk of using the classifier trained on the entire training
data set.
```

Algorithm 21.4: *Cross-Validation*

Typically, only a relatively small number of example items are really important in determining the behaviour of a classifier (we see this phenomenon in greater detail in section **??**). The really important examples tend to be rare cases that are quite hard to discriminate — this is because these cases affect the position of the decision boundary most significantly. We need a large data set to ensure that these cases are present, but it appears inefficient to go to great effort to train on a large data set, most of whose elements aren't particularly important.

There is a useful trick that avoids much redundant work. We train on a subset of the examples, run the resulting classifier on the rest of the examples, and then insert the false positives and false negatives into the training set to retrain the classifier. This is because the false positives and false negatives are the cases that give the most information about errors in the configuration of the decision boundaries. This strategy is known as **bootstrapping** (the name is potentially confusing, because there is an unrelated statistical procedure known as bootstrapping; nonetheless, we're stuck with it at this point).

21.2 Building Classifiers from Class Histograms

One simple way to build a probability model for a classifier is to use a histogram. If a histogram is divided by the total number of pixels, we get a representation of the class-conditional probability density function. It is a fact that, as the data set gets larger and the histogram bins get smaller, the histogram divided by the total number of data items will almost certainly converge to the probability density function []. In low dimensional problems, this approach can work quite well (section 21.2.1). It isn't practical for high dimensional data because the number of histogram bins required quickly becomes intractable, unless we use strong independence assumptions to control the complexity (section 21.5).

21.2.1 Finding Skin Pixels using a Classifier

As we indicated in section **??**, skin-finding is useful for activities like building gesture based interfaces. Skin has a quite characteristic range of colours, suggesting that we can build a skin finder by classifying pixels on their colour. Jones and Rehg construct a histogram of RGB values due to skin pixels, and a second histogram of RGB values due to non-skin pixels. These histograms serve as models of the class-conditional densities.

We write $\boldsymbol{x}$ for a vector containing the colour values at a pixel. We subdivide this colour space into boxes, and count the percentage of skin pixels that fall into each box — this histogram supplies $p(\boldsymbol{x}|\text{skin pixel})$, which we can evaluate by determining the box corresponding to $\boldsymbol{x}$ and then reporting the percentage of skin pixels in this box. Similarly, a count of the percentage of non-skin pixels that fall into each box supplies $p(\boldsymbol{x}|\text{not skin pixel})$. We need $p(\text{skin pixel})$ and $p(\text{not skin pixel})$ — or rather, we need only one of the two, as they sum to one. Assume for the moment that the prior is known. We can now build a classifier, using Bayes' rule to obtain the posterior (keep in mind that $p(\boldsymbol{x})$ is easily computed as $p(\boldsymbol{x}|\text{skin pixel}) + p(\boldsymbol{x}|\text{not skin pixel})$).

One way to estimate the prior is to model $p(\text{skin pixel})$ as the fraction of skin pixels in some (ideally large) training set. Notice that our classifier compares

$$\frac{p(\boldsymbol{x}|\text{skin})p(\text{skin})}{p(\boldsymbol{x})}L(\text{skin} \rightarrow \text{not skin})$$

with

$$\frac{p(\boldsymbol{x}|\text{not skin})p(\text{not skin})}{p(\boldsymbol{x})}L(\text{not skin} \rightarrow \text{skin})$$

Now by rearranging terms and noticing that $p(\text{skin}|\boldsymbol{x}) = 1 - p(\text{not skin}|\boldsymbol{x})$, our classifier becomes

- if $p(\text{skin}|\boldsymbol{x}) > \theta$, classify as skin
- if $p(\text{skin}|\boldsymbol{x}) < \theta$, classify as not skin
- if $p(\text{skin}|\boldsymbol{x}) = \theta$, choose classes uniformly and at random

where θ is an expression that doesn't depend on $\boldsymbol{x}$, and encapsulates the relative loss, etc. This yields a family of classifiers, one for each choice of θ.

Each classifier in this family has a different false-positive and false-negative rate. These rates are functions of θ, so we can plot a parametric curve that captures the performance of the family of classifiers. This curve is known as a **receiver operating curve** (or **ROC** for short). Figure 21.4 shows the ROC for a skin finder built using this approach. The ROC is invariant to choice of prior (exercises)— this means that if we change the value of $p(\text{skin})$, we can choose some new value of θ to get a classifier with the same performance. This yields another approach to estimating a prior. We choose some value rather arbitrarily, plot the loss on the training set as a function of θ, and then select the value of θ that minimizes this loss.

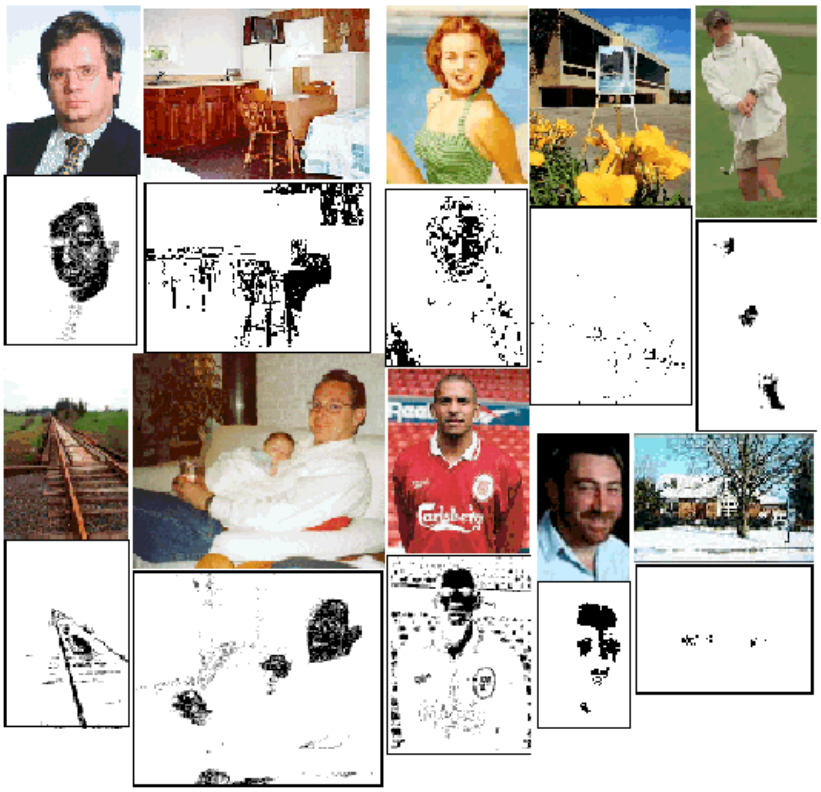


Figure 21.3. The figure shows a variety of images together with the output of the skin detector of Jones and Rehg applied to the image. Pixels marked black are skin pixels, and white are background. Notice that this process is relatively effective, and could certainly be used to focus attention on, say, faces and hands. *figure from Jones and Rehg, Statistical color models with application to skin detection, p.10, in the fervent hope of receiving permission*

21.2.2 Face Finding Assuming Independent Template Responses

Histogram models become impractical in high dimensions, because the number of boxes required goes up as a power of the dimension. We can dodge this phenomenon. Recall from chapter **??** that independence assumptions reduce the number of parameters that must be learned in a probabilistic model; by assuming that terms are independent, we can reduce the dimension sufficiently to use histograms. While this appears to be an aggressive oversimplification, it can result in useful systems. In one such system, due to Schneiderman and Kanade, this model is used to find faces. Assume that the face occurs at a fixed, known scale (we could search smoothed and resampled versions of the image to find larger faces) and will occupy a region of

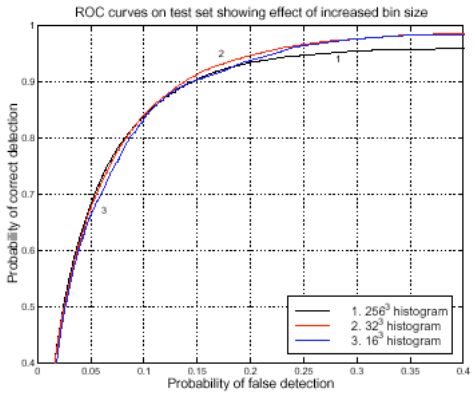


Figure 21.4. The receiver operating curve for the skin detector of Jones and Rehg. This plots the detection rate against the false negative rate for a variety of values of the parameter θ. A perfect classifier has an ROC that, on these axes, is a horizontal line at 100% detection. Notice that the ROC varies slightly with the number of boxes in the histogram. *figure from Jones and Rehg, Statistical color models with application to skin detection, p.11, in the fervent hope of receiving permission*

known shape. In the case of frontal faces, this might be an oval or a square; for a lateral face, this might be some more complicated polygon.

We now need to model the image pattern generated by the face. This is a likelihood model — we want a model giving $P(\text{image pattern}|\text{face})$. As usual, it is helpful to think in terms of generative models — the process by which a face gives rise to an image patch. The set of possible image patches is somewhat difficult to deal with, because it is big, but we can avoid this by dividing the image patch into a set of subregions, and then labelling the subregions, using a small set of labels.

An appropriate labelling can be obtained using a clustering algorithm and a large number of example images. For example, we might cluster the subregions in a large number of example images using k-means; now each cluster center represents a "typical" form of subregion. The subregions in our image patch can then be labelled with the cluster center to which they are closest. This approach has the advantage that minor variations in the image pattern — caused perhaps by noise, or by skin irregularities, etc. — are suppressed.

At this point, a number of models are available. The simplest practical model is to assume that the probability of encountering each pattern is independent of the configuration of the other patterns (but not of position) given that a face is present. This means that our model is:

$$\begin{aligned} P(\text{image}|\text{face}) &= P(\text{label 1 at } (x_1, y_1), \ldots, \text{label } k \text{ at } (x_k, y_k)|\text{face}) \\ &= P(\text{label 1 at } (x_1, y_1)|\text{face}) \ldots P(\text{label } k \text{ at } (x_k, y_k)|\text{face}) \end{aligned}$$

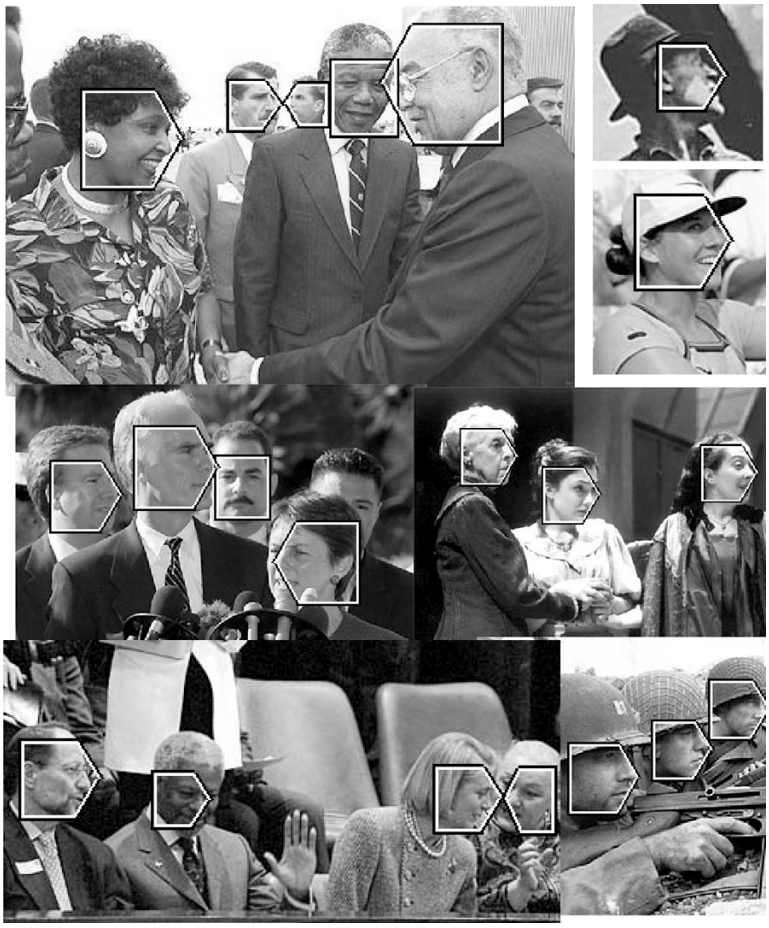


Figure 21.5. Faces found using the method of section 21.5. Image windows at various scales are classified as frontal face, lateral face or non-face, using a likelihood model learned from data. Subregions in the image window are classified into a set of classes learned from data; the face model assumes that labels from these classes are emitted independently of one another, at different positions. This likelihood model yields a posterior value for each class and for each window, and the posterior value is used to identify the window. *figure from Schneiderman and Kanade, A Statistical Method for 3D Object Detection Applied to Faces and Cars, p.6, in the fervent hope of receiving permission*

In this case, each term of the form $P(\text{label } k \text{ at } (x_k, y_k)|\text{face})$ can be learned fairly easily by labelling a large number of example images, and then forming a histogram. Because the histograms are now two dimensional, the number of boxes is no longer problematic. A similar line of reasoning leads to a model of $P(\text{image}|\text{no face})$. A classifier follows from the line of reasoning given above. This approach has been

used successfully by Schneiderman and Kanade to build detectors for faces and cars (figure 21.5).

21.3 Feature Selection

Assume we have a set of pixels that we believe belong together, and that should be classified. What features should we present to a classifier? One approach is to present all the pixel values: this gives the classifier the maximum possible amount of information about the set of pixels, but creates a variety of problems.

Firstly, high dimensional spaces are "big" in the sense that very large numbers of examples can be required to represent the available possibilities fairly. For example, a face at low resolution has a fairly simple structure: it consists (rather roughly) of some dark bars (the eyebrows and eyes) and light bars (the specular reflections from the nose and forehead) on a textureless background. However, if we are working with high resolution faces, it might be very difficult to supply enough examples to make determine that this structure is significant and that minor variations in skin texture, etc. are irrelevant. Instead, we would like to choose a feature space that would make these properties "obvious", typically by imposing some form of structure on the examples.

Secondly, we may know some properties of the patterns in advance; for example, we have models of the behaviour of illumination. Forcing a classifier to use examples to, in essence, come up with a model that we already know is a waste of examples. We would like to use features that are consistent with our knowledge of the patterns. This might involve preprocessing regions (for example, to remove the effects of illumination changes), or choosing features that are invariant to some kinds of transformation (for example, scaling an image region to a standard size).

You should notice a similarity between feature selection and model selection (as described in sections **??** and **??**). In model selection, we were attempting to obtain a model that best explains a data set; here we are attempting to a set of features that best classifies a data set. The two are basically the same activity in slightly distinct forms (you can view a set of features as a model, and classification as explanation); here we will describe methods that are used mainly for feature selection. We concentrate on two standard methods for obtaining **linear features**, features which are a linear function of the initial feature set.

21.3.1 Principal Component Analysis

The core goal in feature selection is to obtain a smaller set of features that "accurately represents" the original set. What this means rather depends on the application; however, one important possibility is that the new set of features should capture as much of the old set's variance as possible. The easiest way to see this is to consider an extreme example; if the value of one feature can be predicted precisely from the value of the others, then it is clearly redundant and can be dropped. By this argument, if we are going to drop a feature, the best one to drop is the one

whose value is most accurately predicted by the others. We can do more than drop features: we can make new features as functions of the old features.

In **principal component analysis**, the new features are linear functions of the old features. In principal component analysis, we take a set of data points and construct a lower dimensional linear subspace that "best explains" the variation of these data points from their mean. This method (also known as the Karhunen-Loéve transform) is a classical technique from statistical pattern recognition [Duda and Hart, 1973; Oja, 1983; Fukunaga, 1990].

Assume we have a set of n feature vectors $\boldsymbol{x}_i$ $(i = 1, \ldots, n)$ in $\mathbb{R}^d$. The mean of this set of feature vectors is $\boldsymbol{\mu}$ (you should think of the mean as the center of gravity in this case), and their covariance is Σ (you can think of the variance as a matrix of second moments). We use the mean as an origin, and study the offsets from the mean, $(\boldsymbol{x}_i - \boldsymbol{\mu})$.

Our features will be linear combinations of the original features; this means it is natural to consider the projection of these offsets onto various different directions. A unit vector $\boldsymbol{v}$ represents a direction in the original feature space; we can interpret this direction as a new feature $v(\boldsymbol{x})$. The value of u on the i'th data point is given by $v(\boldsymbol{x}_i) = \boldsymbol{v}^T(\boldsymbol{x}_i - \boldsymbol{\mu})$. A good feature will capture as much of the variance of the original data set as possible. Notice that v has zero mean; then the variance of v is

$$\begin{aligned}
\text{var}(v) &= \frac{1}{n-1}\sum_{i=1}^{n} v(\boldsymbol{x}_i)v(\boldsymbol{x}_i)^T \\
&= \frac{1}{n}\sum_{i=1}^{n-1} \boldsymbol{v}^T(\boldsymbol{x}_i - \boldsymbol{\mu})(\boldsymbol{v}^T(\boldsymbol{x}_i - \boldsymbol{\mu}))^T \\
&= \boldsymbol{v}^T \left\{ \sum_{i=1}^{n-1} (\boldsymbol{x}_i - \boldsymbol{\mu})(\boldsymbol{x}_i - \boldsymbol{\mu})^T \right\} \boldsymbol{v} \\
&= \boldsymbol{v}^T \Sigma \boldsymbol{v}
\end{aligned}$$

Now we should like to maximise $\boldsymbol{v}^T\Sigma\boldsymbol{v}$ subject to the constraint that $\boldsymbol{v}^T\boldsymbol{v} = 1$. This is an eigenvalue problem; the eigenvector of Σ corresponding to the largest eigenvalue is the solution. Now if we were to project the data onto a space *perpendicular* to this eigenvector, we would obtain a collection of $d-1$ dimensional vectors. The highest variance feature for this collection would be the eigenvector of Σ with second largest eigenvalue; and so on.

This means that the eigenvectors of Σ — which we write as $\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_d$, where the order is given by the size of the eigenvalue and $\boldsymbol{v}_1$ has the largest eigenvalue — give a set of features with the following properties:

- They are independent (because the eigenvectors are orthogonal).
- Projection onto the basis $\{\boldsymbol{v}_1, \ldots, \boldsymbol{v}_k\}$ gives the k-dimensional set of linear features that preserves the most variance.

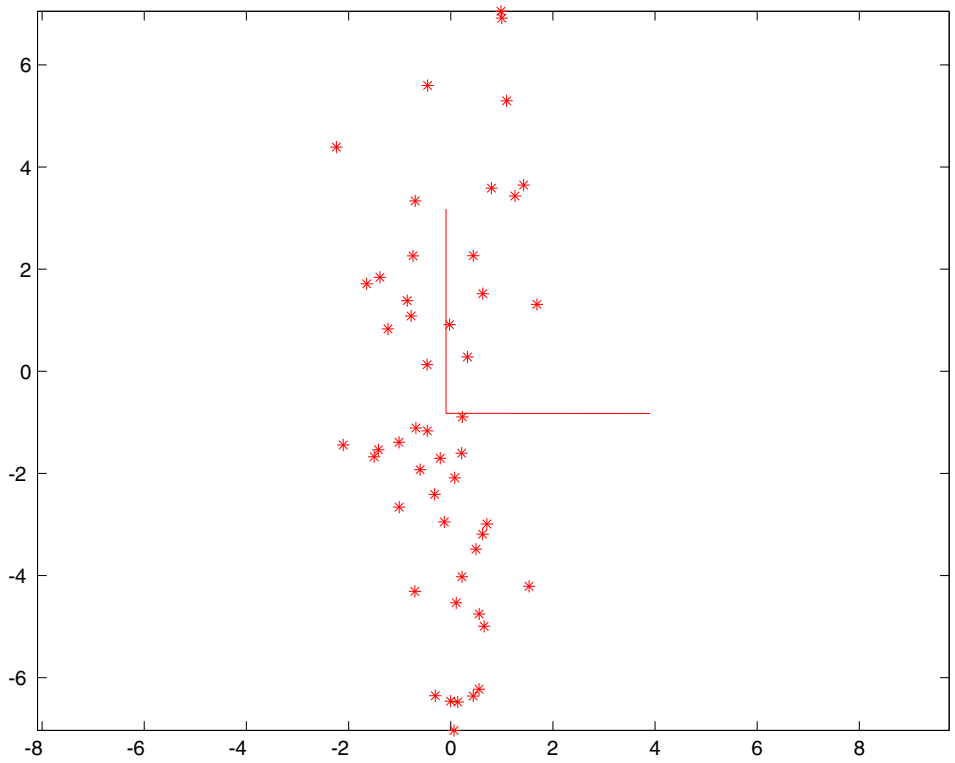


Figure 21.6. A data set which is well represented by a principal component analysis. The axes represent the directions obtained using PCA; the vertical axis is the first principal component, and is the direction in which the variance is highest.

You should notice that, depending on the data source, principal components can give a very good or a very bad representation of a data set (see figures 21.6 and 21.7, and figure 21.8).

21.3.2 Canonical Variates

Principal component analysis yields a set of linear features of a particular dimension that best represents the variance in a high-dimensional dataset. There is no guarantee that this set of features is good for *classification*. For example, figure 21.8 shows a dataset where the first principal component would yield a very bad classifier and the second principal component would yield quite a good one, despite not capturing the variance of the data set.

Linear features that emphasize the distinction between classes are known as **canonical variates**. To construct canonical variates, assume that we have a set of

Assume we have a set of n feature vectors $\boldsymbol{x}_i$ $(i = 1, \ldots, n)$ in $\mathbb{R}^d$. Write

$$\boldsymbol{\mu} = \frac{1}{n}\sum_i \boldsymbol{x}_i$$

$$\Sigma = \frac{1}{n-1}\sum_i (\boldsymbol{x}_i - \boldsymbol{\mu})(\boldsymbol{x}_i - \boldsymbol{\mu})^T$$

The unit eigenvectors of Σ — which we write as $\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_d$, where the order is given by the size of the eigenvalue and $\boldsymbol{v}_1$ has the largest eigenvalue — give a set of features with the following properties:

- They are independent.
- Projection onto the basis $\{\boldsymbol{v}_1, \ldots, \boldsymbol{v}_k\}$ gives the k-dimensional set of linear features that preserves the most variance.

Algorithm 21.5: *Principal components analysis identifies a collection of linear features that are independent, and capture as much variance as possible from a dataset.*

data items $\boldsymbol{x}_i$, for $i \in \{1, \ldots, n\}$. We assume that there are p features (i.e. that the $\boldsymbol{x}_i$ are p-dimensional vectors). We have g different classes, and the j'th class has mean $\boldsymbol{\mu}_j$. Write $\overline{\boldsymbol{\mu}}$ for the mean of the class means, i.e.

$$\overline{\boldsymbol{\mu}} = \frac{1}{g}\sum_{j=1}^{g} \boldsymbol{\mu}_j$$

Write

$$\mathcal{B} = \frac{1}{g-1}\sum_{j=1}^{g} (\boldsymbol{\mu}_j - \overline{\boldsymbol{\mu}})(\boldsymbol{\mu}_j - \overline{\boldsymbol{\mu}})^T$$

Note that $\mathcal{B}$ gives the variance of the class means. In the simplest case, we assume that each class has the same covariance Σ, and that this has full rank. We would like to obtain a set of axes where the clusters of data points belonging to a particular class will group together tightly, while the distinct classes will be widely separated. This involves finding a set of features that maximises the ratio of the separation (variance) between the class means to the variance within each class. The separation between the class means is typically referred to as the **between-class variance**, and the variance within a class is typically referred to as the **within-class variance**.

Now we are interested in linear functions of the features, so we concentrate on

$$v(\boldsymbol{x}) = \boldsymbol{v}^T\boldsymbol{x}$$

We should like to maximize the ratio of the between-class variances to the within-class variances for $\boldsymbol{v}_1$.

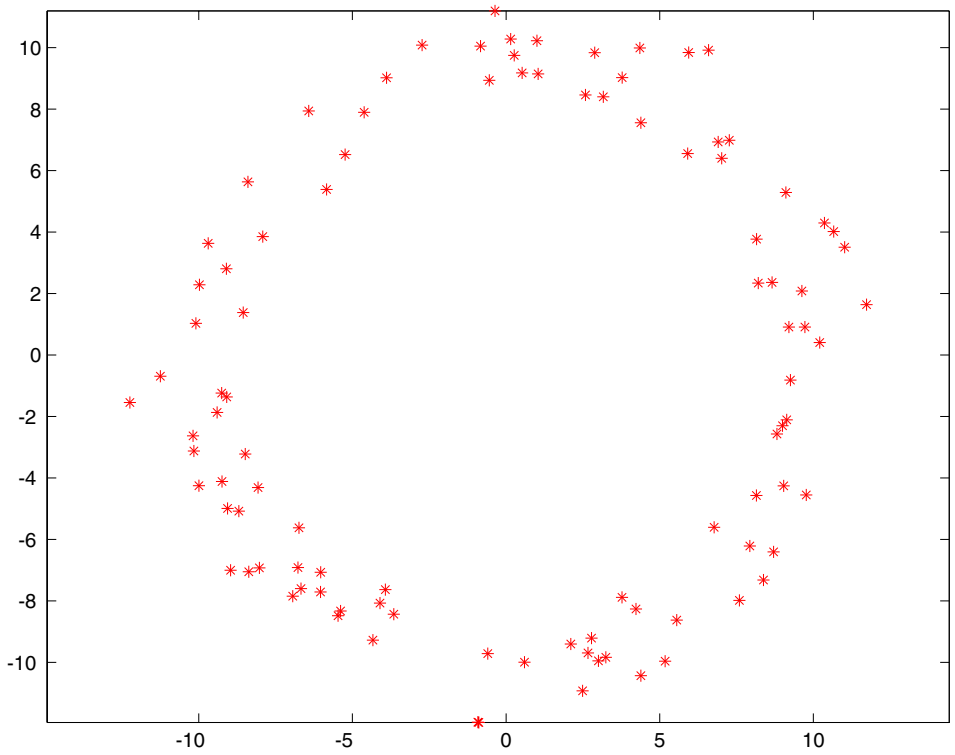


Figure 21.7. Not every data set that is well represented by PCA. The principal components of this data set will be relatively unstable, because the variance in each direction is the same for the source. This means that we may well report significantly different principal components for different datasets from this source. This is a secondary issue — the main difficulty is that projecting the data set onto some axis will suppress the main feature, its circular structure.

Using the same argument as for principal components, we can achieve this by choosing $\boldsymbol{v}$ to maximise

$$\frac{\boldsymbol{v}_1^T \mathcal{B} \boldsymbol{v}_1}{\boldsymbol{v}_1^T \Sigma \boldsymbol{v}_1}$$

This problem is the same as maximising $\boldsymbol{v}_1^T \mathcal{B} \boldsymbol{v}_1$ subject to the constraint that $\boldsymbol{v}_1^T \Sigma \boldsymbol{v}_1 = 1$. In turn, a solution has the property that

$$\mathcal{B} \boldsymbol{v}_1 + \lambda \Sigma \boldsymbol{v}_1 = 0$$

for some constant λ. This is known as a **generalised eigenvalue problem** — if Σ has full rank, we can solve it by finding the eigenvector of $\Sigma^{-1}\mathcal{B}$ with largest

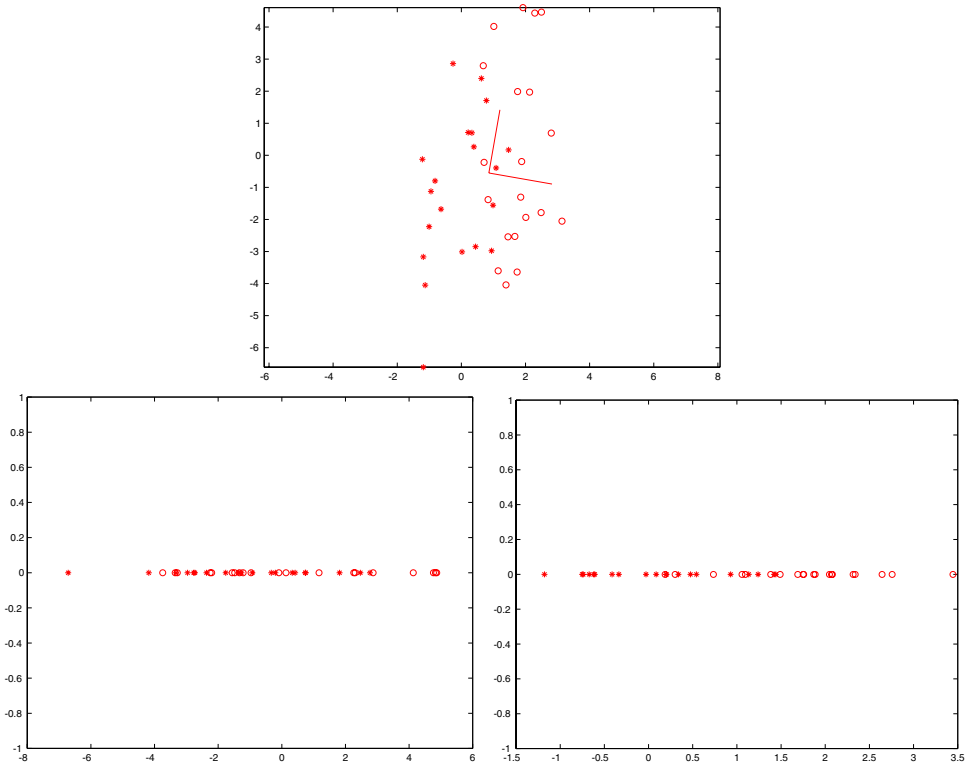


Figure 21.8. Principal component analysis doesn't take into account the fact that there may be more than one class of item in a dataset. This can lead to significant problems. For a classifier, we would like to obtain a set of features that firstly reduces the number of features and secondly makes the difference between classes most obvious. For the data set on the **top**, one class is indicated by circles and the other by stars. PCA would suggest projection onto a vertical axis, which captures the variance in the dataset, but cannot be used to discriminate it, as we can see from the axes obtained by PCA, which are overlaid on the data set. The **bottom row** shows the projections onto those axes. On the **bottom left**, we show the projection onto the first principal component — which has higher variance, but separates the classes poorly — and on the **bottom right**, we show the projection onto the second principal component — which has significantly lower variance (look at the axes) and gives better separation.

eigenvalue (otherwise, we use specialised routines within the relevant numerical software environment).

Now for each $\boldsymbol{v}_l$, for $2 \leq l \leq p$, we should like to find features that extremise the criterion, and are independent of the the previous $\boldsymbol{v}_l$. These are provided by the other eigenvectors of $\Sigma\mathcal{B}$. The eigenvalues give the variance along the features (which are independent). By choosing the $m < p$ eigenvectors with the largest

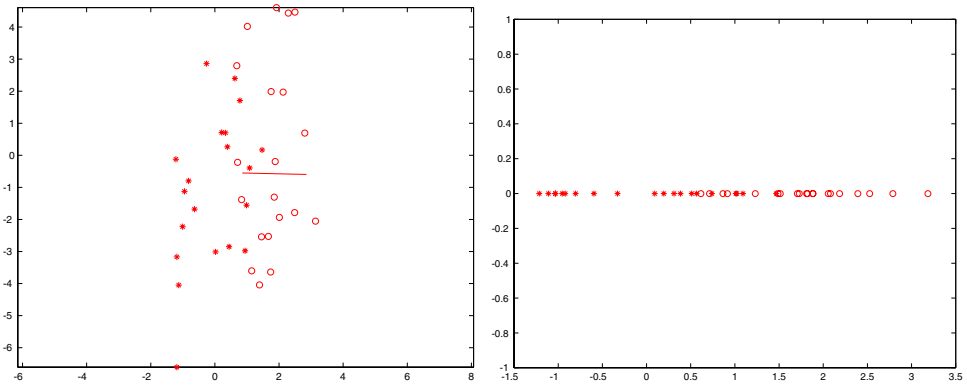


Figure 21.9. Canonical variates use the class of each data item as well as the features in estimating a good set of linear features. In particular, the approach constructs axes that separate different classes as well as possible. The data set used in figure 21.8 is shown on the **left**, with the axis given by the first canonical variate overlaid. On the **bottom right**, we show the projection onto that axis, where the classes are rather well separated.

eigenvalues, we obtain a set of features that reduces the dimension of the feature space will best preserving the separation between classes. This doesn't by any means guarantee the best error rate for a classifier on a reduced number of features, but it offers a good place to start, by reducing the number of features while respecting the category structure (details and examples in [McLachlan and Krishnan, 1996], p— or in [Ripley, 1996], p—).

If the classes don't have the same covariance, it is still possible to construct canonical variates. In this case, we estimate a Σ as the covariance of all of the offsets of each data item *from its own class mean*, and proceed as before. Again, this is an approach without a guarantee of optimality, but one that can work quite well in practice.

21.4 Neural Networks

It is commonly the case that neither simple parametric density models nor histogram models can be used. In this case, we must either use more sophisticated density models (an idea we explore in this section) or look for decision boundaries directly (section **??**).

21.4.1 Key Ideas

A **neural network** is a parametric approximation technique that has proven useful for building density models. Neural networks typically approximate a vector function $\boldsymbol{f}$ of some input $\boldsymbol{x}$ with a series of **layers**. Each layer forms a vector of outputs each of which is obtained by applying the same non-linear function — which we

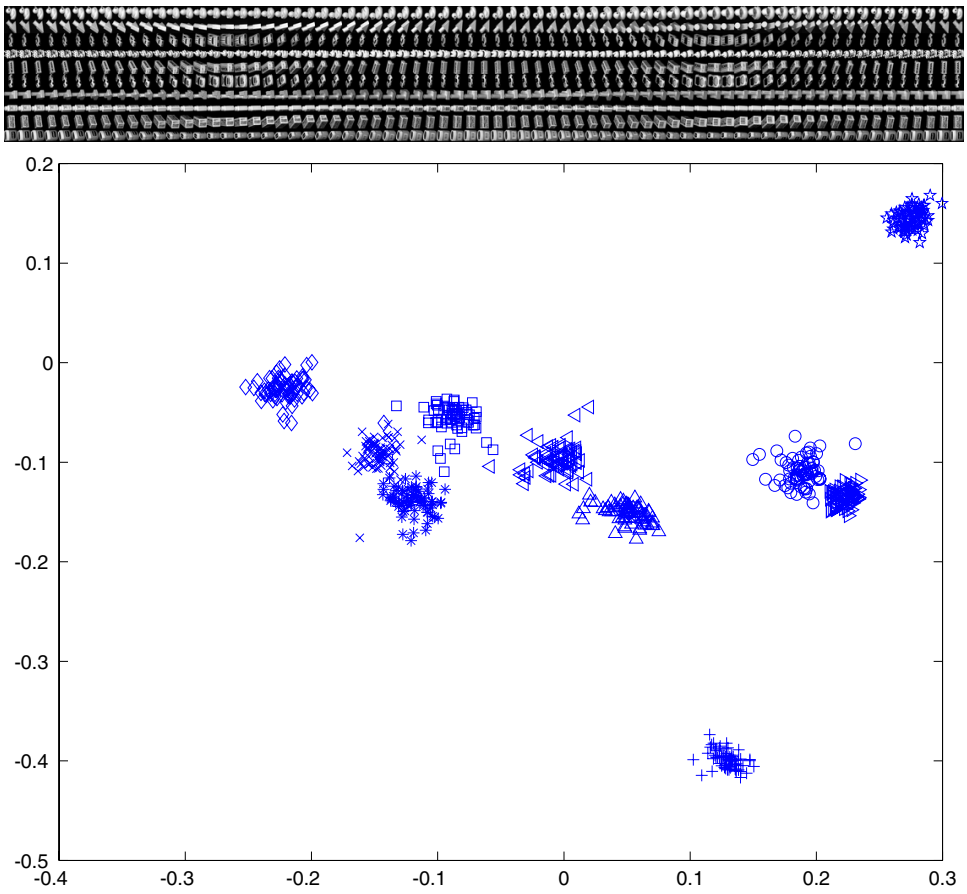


Figure 21.10. Canonical variates are effective for a variety of simple template matching problems. The figure on top shows views of 10 objects at a variety of poses, on a black background (these images are smoothed and resampled versions of images in the well-known COIL database, due to Nene and Nayar and available at `http://www.cs.columbia.edu/*******`). Identifying an object from one of these images is a relatively simple matter, because there is no segmentation. We then used 60 of the images of each object to determine a set of canonical variates. The figure below shows the first two canonical variates for 71 images — the 60 training images and 11 others — of each object (different symbols correspond to different objects). Note that the clusters are tight and well separated; on these two canonical variates alone, we could probably get quite good classification.

shall write as ϕ — to different affine functions of the inputs. We adopt the convenient trick of adding an extra component to the inputs and fixing the value of this component at one, so that we obtain a linear function of this augmented input

Assume that we have a set of data items of g different classes. There are n_k items in each class, and a data item from the k'th class is $\boldsymbol{x}_{k,i}$, for $i \in \{1, \ldots, n_k\}$. The j'th class has mean $\boldsymbol{\mu}_j$. We assume that there are p features (i.e. that the $\boldsymbol{x}_i$ are p-dimensional vectors).
Write $\overline{\boldsymbol{\mu}}$ for the mean of the class means, i.e.

$$\overline{\boldsymbol{\mu}} = \frac{1}{g}\sum_{j=1}^{g} \boldsymbol{\mu}_j$$

Write

$$\mathcal{B} = \frac{1}{g-1}\sum_{j=1}^{g} (\boldsymbol{\mu}_j - \overline{\boldsymbol{\mu}})(\boldsymbol{\mu}_j - \overline{\boldsymbol{\mu}})^T$$

Assume that each class has the same covariance Σ, which is either known or estimated as

$$\Sigma = \frac{1}{N-1}\sum_{c=1}^{g} \left\{ \sum_{i=1}^{n_c} (\boldsymbol{x}_{c,i} - \boldsymbol{\mu}_c)(\boldsymbol{x}_{c,i} - \boldsymbol{\mu}_c)^T \right\}$$

The unit eigenvectors of $\Sigma^{-1}\mathcal{B}$ — which we write as $\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_d$, where the order is given by the size of the eigenvalue and $\boldsymbol{v}_1$ has the largest eigenvalue — give a set of features with the following property:

- Projection onto the basis $\{\boldsymbol{v}_1, \ldots, \boldsymbol{v}_k\}$ gives the k-dimensional set of linear features that best separates the class means.

Algorithm 21.6: *Canonical variates identifies a collection of linear features that separating the classes as well as possible.*

vector. This means that a layer with augmented input vector $\boldsymbol{u}$ and output vector $\boldsymbol{v}$ can be written as

$$\boldsymbol{v} = [\phi(\boldsymbol{w}_1 \cdot \boldsymbol{u}), \phi(\boldsymbol{w}_2 \cdot \boldsymbol{u}), \ldots \phi(\boldsymbol{w}_n \cdot \boldsymbol{u})]$$

where the $\boldsymbol{w}_i$ are parameters that can be adjusted to approve the approximation.

Typically, a neural net uses a sequence of layers to approximate a function. Each layer will use augmented input vectors. For example, if we are approximating a vector function $\boldsymbol{g}$ of a vector $\boldsymbol{x}$ with a two layer net, we obtain

$$\boldsymbol{g}(\boldsymbol{x}) \approx \boldsymbol{f}(\boldsymbol{x}) = [\phi(\boldsymbol{w}_{21} \cdot \boldsymbol{y}), \phi(\boldsymbol{w}_{22} \cdot \boldsymbol{y}), \ldots \phi(\boldsymbol{w}_{2n} \cdot \boldsymbol{y})]$$

where

$$\boldsymbol{y}(\boldsymbol{z}) = [\phi(\boldsymbol{w}_{11} \cdot \boldsymbol{z}), \phi(\boldsymbol{w}_{12} \cdot \boldsymbol{z}), \ldots \phi(\boldsymbol{w}_{1m} \cdot \boldsymbol{z}), 1]$$

and

$$\boldsymbol{z}(\boldsymbol{x}) = [x_1, x_2, \ldots, x_p, 1]$$

Some of the elements of $\boldsymbol{w}_{1k}$ or $\boldsymbol{w}_{2k}$ could be clamped at zero; in this case, we are insisting that some elements of $\boldsymbol{y}$ do not affect $\boldsymbol{f}(\boldsymbol{x})$. If this is the case, the layer is referred to as being **partially connected**, otherwise it is known as a **fully connected layer**. Of course, layer two could be either fully or partially connected as well. The parameter n is fixed by the dimension of $\boldsymbol{f}$ and p is fixed by the dimension of $\boldsymbol{x}$, but there is no reason that m should be the same as either n or p. Typically, m is larger than either. A similar construction yields three layer networks (or networks with more layers, which are uncommon). Neural networks are often drawn with circles indicating variables and arrows indicating possibly non-zero connections; this gives a representation that exposes the basic structure of the approximation (figure 21.11).

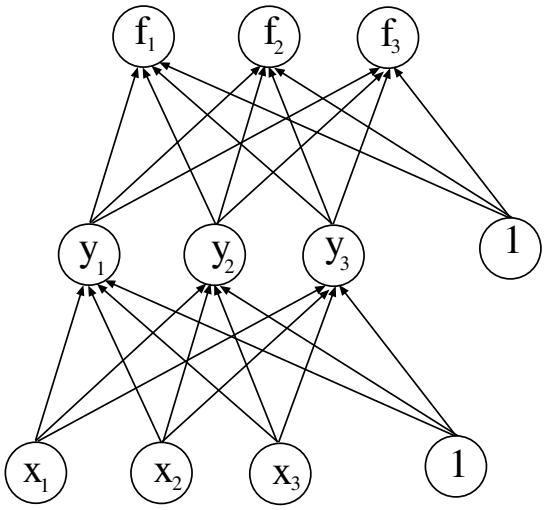


Figure 21.11. Neural networks are often illustrated by diagrams of the form shown above. Each circle represents a variable, and the circles are typically labelled with the variable. The "layers" are obvious in such a drawing. This network is the two layer network given in the text; the arrows indicate that the coefficient coupling the two variables in the affine function could be non-zero. This network is fully connected, because all arrows are present. It is possible to have arrows skip layers, etc.

Choosing a Non-Linearity

There are a variety of possibilities for ϕ. For example, we could use a **threshold function**, which has value one when the argument is positive and zero otherwise. It is quite easy to visualize the response of a layer of that uses a threshold function; each component of the layer changes from zero to one along a hyperplane. This means that the output vector takes different values in each cell of an arrangement of hyperplanes in the input space. Networks that use layers of this form are hard

to train, because the threshold function is not differentiable.

It is more common to use a ϕ that changes smoothly (but rather quickly) from zero to one, often called a **sigmoid function** or **squashing function**. The **logistic function** is one popular example. This is a function of the form

$$\phi(x;\nu) = \frac{e^{x/\nu}}{1+e^{x/\nu}}$$

where ν controls how sharply the function changes at $x = 0$. It isn't crucial that the horizontal assymptotes are zero and one. Another popular squashing function is

$$\phi(x;\nu, A) = A\tanh(\nu x)$$

which has horizontal assymptotes at A and $-A$. Figure 21.12 illustrates these non-linearities.

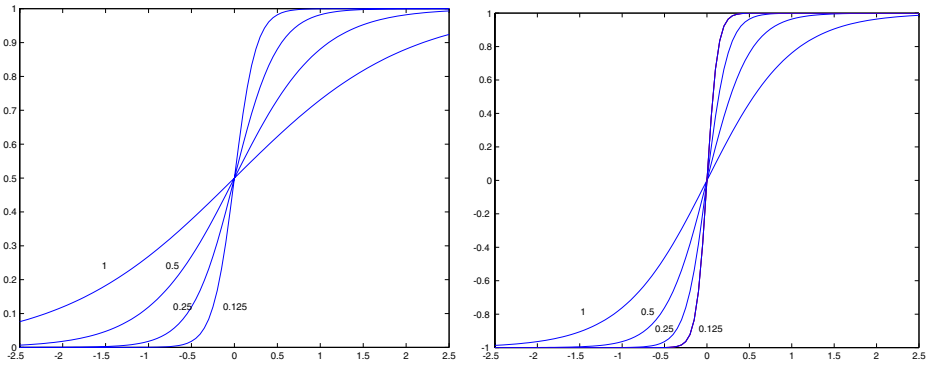


Figure 21.12. On the **left**, a series of squashing functions obtained using $\phi(x;\nu) = \frac{e^{x/\nu}}{1+e^{x/\nu}}$, for different values of ν indicated on the figure. On the **right**, a series of squashing functions obtained using $\phi(x;\nu, A) = A\tanh(x/\nu)$ for different values of ν indicated on the figure. Generally, for x close to the center of the range, the squashing function is linear; for x small or large, it is strongly non-linear.

Producing a Classifier using a Neural Net

To produce a neural net that approximates some function $\boldsymbol{g}(\boldsymbol{x})$, we collect a series of examples $\boldsymbol{x}^e$. We construct a network that has one output for each dimension of $\boldsymbol{g}$. Write this network as $\boldsymbol{n}(\boldsymbol{x};\boldsymbol{p})$, where $\boldsymbol{p}$ is a parameter vector that contains all the $\boldsymbol{w}_{ij}$. We supply a desired output vector $\boldsymbol{o}^e$ for this input; typically, $\boldsymbol{o}^e = \boldsymbol{g}(\boldsymbol{x}^e)$. We now obtain $\hat{\boldsymbol{p}}$ that minimizes

$$Error(\boldsymbol{p}) = \left(\frac{1}{2}\right)\sum_e |\,\boldsymbol{n}(\boldsymbol{x}^e;\boldsymbol{p}) - \boldsymbol{o}^e\,|^2$$

using appropriate optimization software (the half will simplify a little notation later on, but is of no real consequence).

The most significant case occurs when $\boldsymbol{g}(\boldsymbol{x})$ is intended to approximate the posterior on classes, given the data. We do not know this posterior, and so cannot supply its value to the training procedure. Instead, we require that our network has one output for each class. Given an example $\boldsymbol{x}^e$, we construct a desired output $\boldsymbol{o}^e$ as a vector which contains a one in the component corresponding to that example's class and a zero in each other component. We now train the net as above, and regard the output of the neural network as a model of the posterior probability. An input $\boldsymbol{x}$ is then classified by forming $\boldsymbol{n}(\boldsymbol{x}; \hat{\boldsymbol{p}})$, and then choosing the class that corresponds to the largest component of this vector.

21.4.2 Minimizing the Error

Recall that we are training a net by minimizing the sum over the examples of the difference between the desired output and the actual output, i.e., by minimizing

$$Error(\boldsymbol{p}) = \left(\frac{1}{2}\right) \sum_e |\, \boldsymbol{n}(\boldsymbol{x}^e; \boldsymbol{p}) - \boldsymbol{o}^e \,|^2$$

as a function of the parameters $\boldsymbol{p}$. There are a variety of strategies for obtaining $\hat{\boldsymbol{p}}$, the set of parameters that minimize this error. One is gradient descent; from some initial point $\boldsymbol{p}_i$, we compute a new point $\boldsymbol{p}_{i+1}$ by

$$\boldsymbol{p}_{i+1} = \boldsymbol{p}_i - \epsilon(\nabla Error)$$

where ϵ is some small constant.

Stochastic Gradient Descent

Write the error for example e as $Error(\boldsymbol{p}; \boldsymbol{x}^e)$, so the total error is $Error(\boldsymbol{p}) = \sum_e Error(\boldsymbol{p}; \boldsymbol{x}^e)$. Now if we use gradient descent, we are updating parameters using the algorithm

$$\boldsymbol{p}_{i+1} = \boldsymbol{p}_i - \epsilon \nabla Error$$

(where the gradient is with respect to $\boldsymbol{p}$ and is evaluated at $\boldsymbol{p}_i$). This works, because if ϵ is sufficiently small, we have that

$$\begin{aligned} Error(\boldsymbol{p}_{i+1}) &= Error(\boldsymbol{p}_i - \epsilon \nabla Error) \\ &\approx Error(\boldsymbol{p}_i) - \epsilon(\nabla Error \cdot \nabla Error) \\ &\leq Error(\boldsymbol{p}_i) \end{aligned}$$

with equality only at an extremum. This creates a problem: evaluating the error and its gradient is going to involve a sum over all examples, which may be a very large number. We should like to avoid this sum; it turns out that it is possible to do so by selecting an example at random, computing the gradient *for that example*

alone, and updating the parameters using that gradient. In this process, known as **stochastic gradient descent**, we update the parameters using the algorithm

$$\boldsymbol{p}_{i+1} = \boldsymbol{p}_i - \epsilon \nabla Error(\boldsymbol{p}; \boldsymbol{x}^e)$$

(where the gradient is with respect to $\boldsymbol{p}$, is evaluated at $\boldsymbol{p}_i$, and we choose the example uniformly at random, making a different choice at each step). In this case, the error doesn't necessarily go down for each particular choice, but the *expected* value of the error *does* go down, for a sufficiently small value of ϵ. In particular, we have

$$\begin{aligned}
\mathrm{E}(Error(\boldsymbol{p}_{i+1})) &= \mathrm{E}(Error(\boldsymbol{p}_i - \epsilon \nabla Error(\boldsymbol{p}; \boldsymbol{x}^e)) \\
&\approx \mathrm{E}(Error(\boldsymbol{p}_i) - \epsilon(\nabla Error \cdot \nabla Error(\boldsymbol{p}; \boldsymbol{x}^e)) \\
&= Error(\boldsymbol{p}_i) - \epsilon \frac{1}{n} \sum_e (\nabla Error \cdot \nabla Error(\boldsymbol{p}; \boldsymbol{x}^e)) \\
&= Error(\boldsymbol{p}_i) - \epsilon(\nabla Error \cdot (\frac{1}{n} \sum_e \nabla Error(\boldsymbol{p}; \boldsymbol{x}^e))) \\
&= Error(\boldsymbol{p}_i) - \frac{\epsilon}{n}(\nabla Error \cdot \nabla Error) \\
&< Error(\boldsymbol{p}_i) \text{ if } |\nabla E| > 0
\end{aligned}$$

By taking sufficient steps down a gradient computed using only one example (selected uniformly and at random each time we take a step), we can in fact minimize the function. This is because the expected value goes down for each step, unless we're at the minimum.

Backpropagation

The difficulty with this problem is that $\nabla Error$ could be quite hard to compute. There is an effective strategy for computing $\nabla Error$ called **back propagation**. This approach exploits the layered structure of the neural network as a function of a function of a ... to obtain the derivative.

Now recall the two layer neural net which we wrote as

$$\boldsymbol{f}(\boldsymbol{x}) = [\phi(\boldsymbol{w}_{21} \cdot \boldsymbol{y}), \phi(\boldsymbol{w}_{22} \cdot \boldsymbol{y}), \ldots \phi(\boldsymbol{w}_{2n} \cdot \boldsymbol{y})]$$

where

$$\boldsymbol{y}(\boldsymbol{z}) = [\phi(\boldsymbol{w}_{11} \cdot \boldsymbol{z}), \phi(\boldsymbol{w}_{12} \cdot \boldsymbol{z}), \ldots \phi(\boldsymbol{w}_{1m} \cdot \boldsymbol{z}), 1]$$

and

$$\boldsymbol{z}(\boldsymbol{x}) = [x_1, x_2, \ldots, x_p, 1]$$

We would like to compute

$$\frac{\partial Error}{\partial w_{kl,m}}$$

Choose $\boldsymbol{p}_o$ (randomly)
Use backpropagation (algorithm **??**) to compute
$\nabla Error(\boldsymbol{x}^e; \boldsymbol{p}_o)$
$\boldsymbol{p}_n = \boldsymbol{p}_o - \epsilon \nabla Error(\boldsymbol{x}^e; \boldsymbol{p}_o)$
Until $|Error(\boldsymbol{p}_n) - Error(\boldsymbol{p}_o)|$ is small
or $|\boldsymbol{p}_o - \boldsymbol{p}_n|$ is small

$\boldsymbol{p}_o = \boldsymbol{p}_n$
Choose an example $(\boldsymbol{x}^e, \boldsymbol{o}^e)$ uniformly and
at random from the training set
Use backpropagation (algorithm **??**) to compute
$\nabla Error(\boldsymbol{x}^e; \boldsymbol{p}_o)$
$\boldsymbol{p}_n = \boldsymbol{p}_o - \epsilon \nabla Error(\boldsymbol{x}^e; \boldsymbol{p}_o)$
end

Algorithm 21.7:

Stochastic gradient descent minimizes the error of a neural net approximation, using backpropagation to compute the derivatives.

where $w_{kl,m}$ is the m'th component of $\boldsymbol{w}_{kl}$. Let us deal with the coefficients of the output layer first, so that we are interested in $w_{2l,m}$, and we get

$$\begin{aligned}
\frac{\partial Error}{\partial w_{2l,m}} &= \sum_k \frac{\partial Error}{\partial f_k} \frac{\partial f_k}{\partial w_{2l,m}} \\
&= \frac{\partial Error}{\partial f_l} \frac{\partial f_l}{\partial w_{2l,m}} \\
&= \sum_e \left\{ (f_l(\boldsymbol{x}^e) - o_l^e) \frac{\partial f_l}{\partial w_{2l,m}} \right\} \\
&= \sum_e \{ (f_l(\boldsymbol{x}^e) - o_l^e) \phi'_{2l}(y_m(\boldsymbol{x}^e)) \} \\
&= \sum_e \{ \delta_{2l}^e (y_m(\boldsymbol{x}^e)) \}
\end{aligned}$$

Here we use the notation

$$\phi'_{2l} = \frac{\partial \phi}{\partial u}$$

where the derivative is evaluated at $u = \boldsymbol{w}_{21} \cdot \boldsymbol{y}$, and we write

$$\delta_{2l}^e = (f_l(\boldsymbol{x}^e) - o_l^e) \phi'_{2l}$$

Notice that evaluating this derivative involves terms in the input of the layer — the terms $y_m(\boldsymbol{x}^e)$ — and in its output — the terms δ_{2l}^e.

Now consider the coefficients of the second layer. We are interested in $w_{1l,m}$, and we get

$$\begin{aligned}
\frac{\partial Error}{\partial w_{1l,m}} &= \sum_k \left\{ \frac{\partial Error}{\partial f_k} \frac{\partial f_k}{\partial w_{1l,m}} \right\} \\
&= \sum_{i,j} \left\{ \frac{\partial Error}{\partial f_i} \frac{\partial f_i}{\partial y_j} \frac{\partial y_j}{\partial w_{1l,m}} \right\} \\
&= \left\{ \sum_k \frac{\partial Error}{\partial f_k} \frac{\partial f_k}{\partial y_l} \right\} \frac{\partial y_l}{\partial w_{1l,m}} \\
&= \sum_e \left\{ \sum_k \left\{ (f_k(\boldsymbol{x}^e) - o_k^e) \frac{\partial f_k}{\partial y_l} \right\} \frac{\partial y_l}{\partial w_{1l,m}} \right\} \\
&= \sum_e \left\{ \sum_k \left\{ (f_k(\boldsymbol{x}^e) - o_k^e) \phi'_{2k} w_{2k,l} \right\} \frac{\partial y_l}{\partial w_{1l,m}} \right\} \\
&= \sum_e \left\{ \sum_k \left\{ (f_k(\boldsymbol{x}^e) - o_k^e) \phi'_{2k} w_{2k,l} \right\} \phi'_{1l} z_m \right\} \\
&= \sum_e \left\{ \sum_k \} \delta_{2k}^e w_{2k,l} \right\} \phi'_{1l} z_m \Bigg\}
\end{aligned}$$

In this expression,

$$\phi'_{2k} = \frac{\partial \phi}{\partial u}$$

evaluated at $u = \boldsymbol{w}_{2k} \cdot \boldsymbol{y}$, and

$$\phi'_{1l} = \frac{\partial \phi}{\partial u}$$

evaluated at $u = \boldsymbol{w}_{1l} \cdot \boldsymbol{z}$. Now if we write

$$\delta_{1l}^e = \sum_k \left\{ \delta_{2k}^e w_{2k,l} \right\} \phi'_{1l}$$

we get

$$\frac{\partial E}{\partial w_{1l,m}} = \sum_e \delta_{1l}^e z_m(\boldsymbol{x}^e)$$

Again, this sum involves a term obtained computing the previous derivative, terms in the derivatives within the layer, and terms in the input. You should convince yourself that, if we had a third layer, the derivative of the error with respect to parameters within this third layer would have a similar form — a function of terms in the derivative of the second layer, terms in the derivatives within the third layer, and terms in the input (all this comes from aggressive application of the chain rule). This suggests a two pass algorithm:

1. Evaluate the net's output on each example. This is usually referred to as a **forward pass**.
2. Evaluate the derivatives using the intermediate terms. This is usually referred to as a **backward pass**.

This process yields the derivatives of the total error with respect to the parameters. We can obtain another simplification: we adopted stochastic gradient descent to avoid having to sum the value of the error and of its gradient over all examples. Because computing a gradient is linear, to compute the gradient of the error on one example alone, we simply drop the sum at the front of our expressions for the gradient. The whole is given in algorithm **??**.

21.4.3 When to Stop Training

Typically, gradient descent is not continued until an exact minimum is found. Surprisingly, this is a source of robustness. The easiest way to understand this is to consider the shape of the error function around the minimum. If the error function changes sharply at the minimum, then the performance of the network is quite sensitive to the choice of parameters. This suggests that the network will generalize badly. You can see this by assuming that the training examples are one half of a larger set; if we had trained the net on the other half, we'd have obtained slightly different set of parameters. This means that the net with our current parameters will perform badly on this other half, because the error changes sharply with a small change in the parameters.

Now if the error function doesn't change sharply at the minimum, there is no particular point in expending effort to be at the minimum value, as long as we are reasonably close — we know that this minimum error value won't be attained on a training set. It is common practice to continue with stochastic gradient descent until (a) each example will have been visited on average rather more than once and (b) the decrease in the value of the function goes below some threshold.

A more difficult question is how many layers to use, and how many units to use in each layer. This question — which is one of model selection — tends to be resolved by experiment. We refer interested readers to [].

21.4.4 Finding Faces using Neural Networks

Face finding is an application that illustrates the usefulness of classifiers. In frontal views at a fairly coarse scale, all faces look basically the same; there are bright regions on the forehead, the cheeks and the nose, and dark regions around the eyes, the eyebrows, the base of the nose and the mouth. This suggests approaching face finding as a search over all image windows of a fixed size for windows that look like a face. Larger or smaller faces can be found by searching coarser or finer scale images.

Because a face illuminated from the left looks very different to a face illuminated from the right, the image windows must be corrected for illumination. Generally,

Notation:
Write the two-layer neural net as

$$\begin{aligned}\boldsymbol{f}(\boldsymbol{x};\boldsymbol{p}) &= [\phi(\boldsymbol{w}_{21}\cdot\boldsymbol{y}),\phi(\boldsymbol{w}_{22}\cdot\boldsymbol{y}),\ldots\phi(\boldsymbol{w}_{2n}\cdot\boldsymbol{y})]\\ \boldsymbol{y}(\boldsymbol{z}) &= [\phi(\boldsymbol{w}_{11}\cdot\boldsymbol{z}),\phi(\boldsymbol{w}_{12}\cdot\boldsymbol{z}),\ldots\phi(\boldsymbol{w}_{1m}\cdot\boldsymbol{z}),1]\\ \boldsymbol{z}(\boldsymbol{x}) &= [x_1,x_2,\ldots,x_p,1]\end{aligned}$$

($\boldsymbol{p}$ is a vector containing all parameters). Write the error on a single example as

$$\begin{aligned}Error^e &= Error(\boldsymbol{p};\boldsymbol{x}^e)\\ &= \left(\frac{1}{2}\right)|\,\boldsymbol{f}(\boldsymbol{x}^e;\boldsymbol{p})-\boldsymbol{o}^e\,|^2\end{aligned}$$

We would like to compute

$$\frac{\partial Error^e}{\partial w_{kl,m}}$$

where $w_{kl,m}$ is the m'th component of $\boldsymbol{w}_{kl}$.

Forward pass: Compute $\boldsymbol{f}(\boldsymbol{x}^e;\boldsymbol{p})$, saving all intermediate variables

Backward pass: Compute

$$\begin{aligned}\delta^e_{2l} &= (f_l(\boldsymbol{x}^e)-o^e_l)\phi'_{2l}\\ \phi'_{2l} &= \frac{\partial\phi}{\partial u}\text{ evaluated at }u=\boldsymbol{w}_{21}\cdot\boldsymbol{y}\\ \frac{\partial Error^e}{\partial w_{2l,m}} &= \sum_e\{\delta^e_{2l}(y_m(\boldsymbol{x}^e))\}\end{aligned}$$

Now compute

$$\begin{aligned}\delta^e_{1l} &= \sum_k\{\delta^e_{2k}w_{2k,l}\}\,\phi'_{1l}\\ \phi'_{1l} &= \frac{\partial\phi}{\partial u}\text{ evaluated at }u=\boldsymbol{w}_{11}\cdot\boldsymbol{z}\\ \frac{\partial E^e}{\partial w_{1l,m}} &= \delta^e_{1l}z_m(\boldsymbol{x}^e)\end{aligned}$$

Algorithm 21.8: *Backpropagation to compute the derivative of the fitting error of a two-layer neural net on a single example with respect to its parameters.*

illumination effects look enough like a linear ramp (one side is bright, the other side is dark, and there is a smooth transition between them) that we can simply fit a linear ramp to the intensity values and subtract that from the image window. Another way to do this would be to log-transform the image, and then subtract a linear ramp fitted to the logs. This has the advantage that (using a rather rough model) illumination effects are additive in the log transform. There doesn't appear to be any evidence in the literature that the log transform makes much difference in practice. Another approach is to histogram equalize the window to ensure that its histogram is the same as that of a set of reference images (histogram equalisation is described in section **??**).

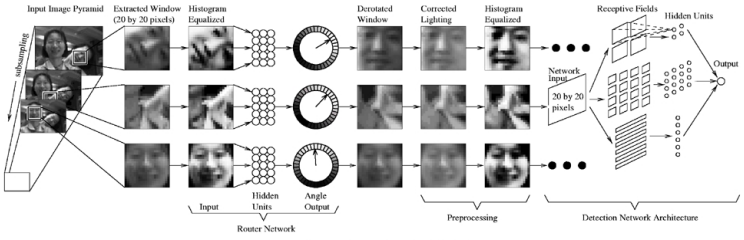


Figure 2. Overview of the algorithm.

Figure 21.13. The architecture of Rowley, Baluja and Kanade's system for finding faces. Image windows of a fixed size are corrected to a standard illumination using histogram equalisation; they are then passed to a neural net that estimates the orientation of the window. The windows are reoriented, and passed to a second net that determines whether a face is present. *figure from Rowley, Baluja and Kanade, Rotation invariant neural-network based face detection p.2, in the fervent hope of receiving permission*

Once the windows have been corrected for illumination, we need to determine whether there is a face present. The orientation isn't known, and so we must either determine it, or produce a classifier that is insensitive to orientation. Rowley, Baluja and Kanade have produced a face finder that finds faces very successfully by firstly estimating the orientation of the window, using one neural net, and then reorienting the window so that it is frontal, and passing the frontal window onto another neural net (see figure 21.13). The orientation finder has 36 output units, each coding for a 10^o range of orientations; the window is reoriented to the orientation given by the largest output. Examples of the output of this system are given in figure 21.14.

21.4.5 Convolutional Neural Nets

Neural networks are not confined to the architecture sketched above; there are a wide variety of alternatives (a good start is to look at []). One architecture that has proven useful in vision applications is the **convolutional neural network**. The motivating idea here is that it appears to be useful to represent image regions with filter outputs. Furthermore, we can obtain a compositional representation we apply

Figure 7. Result of arbitrating between two networks trained with derotated negative examples. The label in the upper left corner of each image (D/T/F) gives the number of faces detected (D), the total number of faces in the image (T), and the number of false detections (F). The label in the lower right corner of each image gives its size in pixels.

Figure 21.14. Typical responses for the Rowley, Baluja and Kanade system for face finding; a mask icon is superimposed on each window that is determined to contain a face. The orientation of the face is indicated by the configuration of the eye-holes in the mask. *figure from Rowley, Baluja and Kanade, Rotation invariant neural-network based face detection p.6, in the fervent hope of receiving permission*

filters to a representation itself obtained using filter outputs. For example, assume that we are looking for handwritten characters; the response of oriented bar filters is likely to be useful here. If we obtain a map of the oriented bars in the image, we

can apply another filter to this map, and the output of this filter indicates spatial relations between the bars.

These observations suggest using a system of filters to build up a set of relations between primitives, and then using a conventional neural network to classify on the resulting representation. There is no particular reason to specify the filters in advance; instead, we could learn them too.

Lecun *et al.* have built a number of classifiers for handwritten digits using a convolutional neural network. The basic architecture is given in figure 21.15. The classifier is applied to a 32x32 image window. The first stage — C1 in the figure — consists of six feature maps. The feature maps are obtained by convolving the image with a 5x5 filter kernel, adding a constant, and applying a sigmoid function. Each map uses a different kernel and constant, and these parameters are learned.

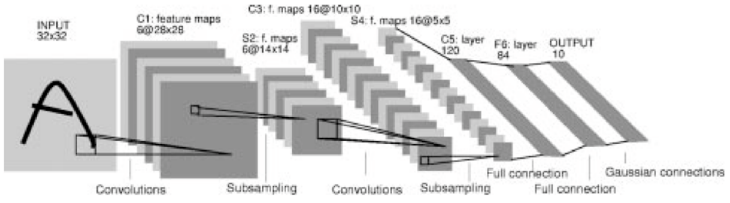


Fig. 2. Architecture of LeNet-5, a convolutional NN, here used for digits recognition. Each plane is a feature map, i.e., a set of units whose weights are constrained to be identical.

Figure 21.15. The architecture of LeNet 5, a convolutional neural net used for recognising handwritten characters. The layers marked C are convolutional layers; those marked S are subsampling layers. The general form of the classifier uses an increasing number of features at increasingly coarse scales to represent the image window; finally, the window is passed to a fully connected neural net, which produces a rectified output that is classified by looking at its distance from a set of canonical templates for characters. *figure from Gradient-Based Learning Applied to Document Recognition Yann Lecun, Léon Bottou, Yoshua Bengio, and Patrick Haffner, p.2283, in the fervent hope of receiving permission*

Because the exact position of a feature should not be important, the resolution of the feature maps is reduced, leading to a new set of six feature maps — S2 in the figure. These maps are, essentially, subsampled versions of the previous layer; this subsampling is achieved by averaging 2x2 neighbourhoods, multiplying by a parameter, adding a parameter, and passing the result through a sigmoid function. The multiplicative and additive parameters are learned. A series of pairs of layers of this form follows, with the number of feature maps increasing as the resolution decreases. Finally, there is a layer with 84 outputs; each of these outputs is supplied by a unit that takes every element of the previous layer as an input.

This network is used to recognise hand printed characters; examples from the training set are given in figure **??**. The outputs are seen as a 7x12 (=84!) image of a character that has been rectified from its hand printed version, and can now be compared with a canonical pattern for that character. The network can rectify

distorted characters very successfully (figure **??** shows some extreme cases that are successfully recognised). The input character is given the class of the character whose canonical pattern is closest to the rectified version. The resulting network has a test error rate of 0.95% (figure 21.16).

Fig. 4. Size-normalized examples from the MNIST database.

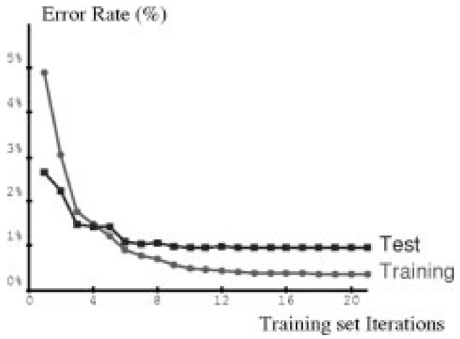


Fig. 5. Training and test error of LeNet-5 as a function of the number of passes through the 60 000 pattern training set (without distortions). The average training error is measured on-the-fly as training proceeds. This explains why the training error appears to be larger than the test error initially. Convergence is attained after 10–12 passes through the training set.

Figure 21.16. On the **left**, a small subset of the MNIST database of handwritten characters, used to train and test LeNet 5. Note the fairly wide variation in the appearance of each character. On the **right**, the error rate of LeNet 5 on a training set and on a test set, plotted as a function of the number of gradient descent passes through the entire training set of 60,000 examples (i.e. if the horizontal axis reads six, the training has taken 360, 000 gradient descent steps). Note that at some point the training error goes down but the test error doesn't; this phenomenon occurs because the system's performance is optimised on the training data. A substantial difference would indicate overfitting. *figure from Gradient-Based Learning Applied to Document Recognition Yann Lecun, Léon Bottou, Yoshua Bengio, and Patrick Haffner, p.2287, in the fervent hope of receiving permission figure from Gradient-Based Learning Applied to Document Recognition Yann Lecun, Léon Bottou, Yoshua Bengio, and Patrick Haffner, p.2288, in the fervent hope of receiving permission*

21.5 The Support Vector Machine

From the perspective of the vision community, classifiers are not an end in themselves, but a means; so when a technique that is simple, reliable and effective becomes available, it tends to be adopted quite widely. The **support vector machine** is such a technique. This should be the first classifier you think of when you wish to build a classifier from examples (unless the examples come from a known distribution, which hardly ever happens). We give a basic introduction to the ideas, and

show some examples where the technique has proven useful.

Assume we have a set of N points $\boldsymbol{x}_i$ that belong to two classes, which we shall indicate by 1 and -1. These points come with their class labels, which we shall write as y_i; thus, our data set can be written as

$$\{(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_N, y_N)\}$$

We should like to determine a rule that predicts the sign of y for any point $\boldsymbol{x}$; this rule is our classifier.

At this point, we distinguish between two cases: either the data is linearly separable, or it isn't. The linearly separable case is much easier, and we deal with it first.

21.5.1 Support Vector Machines for Linearly Separable Datasets

In a linearly separable data set, there is some choice of $\boldsymbol{w}$ and b (which represent a hyperplane) such that

$$y_i \left(\boldsymbol{w} \cdot \boldsymbol{x}_i + b \right) > 0$$

for every example point (notice the devious use of the sign of y_i). There is one of these expressions for each data point, and the set of expressions represents a set of constraints on the choice of $\boldsymbol{w}$ and b. These constraints express the fact that all examples with a negative y_i should be on one side of the hyperplane and all with a positive y_i should be on the other side.

In fact, because the set of examples is finite, there is a family of separating hyperplanes. Each of these hyperplanes must separate the convex hull of one set of examples from the convex hull of the other set of examples. The most conservative choice of hyperplane is the one that is furthest from both hulls. This is obtained by joining the closest points on the two hulls, and constructing a hyperplane perpendicular to this line, and through its midpoint. This hyperplane is as far as possible from each set, in the sense that it maximises the minimum distance from example points to the hyperplane (figure 21.17)

Now we can choose the scale of $\boldsymbol{w}$ and b, because scaling the two together by a positive number doesn't affect the validity of the constraints $y_i(\boldsymbol{w} \cdot \boldsymbol{x}_i + b) > 0$. This means that we can choose $\boldsymbol{w}$ and b such that for every data point we have

$$y_i \left(\boldsymbol{w} \cdot \boldsymbol{x}_i + b \right) \geq 1$$

and such that equality is achieved on at least one point on each side of the hyperplane. Now assume that $\boldsymbol{x}_k$ achieves equality and $y_k = 1$, and $\boldsymbol{x}_l$ achieves equality and $y_l = -1$. This means that $\boldsymbol{x}_k$ is on one side of the hyperplane and $\boldsymbol{x}_l$ is on the other; furthermore, the distance from $\boldsymbol{x}_l$ to the hyperplane is minimal (among the points on the same side as $\boldsymbol{x}_l$), as is the distance from $\boldsymbol{x}_k$ to the hyperplane. Notice that there might be several points with these properties.

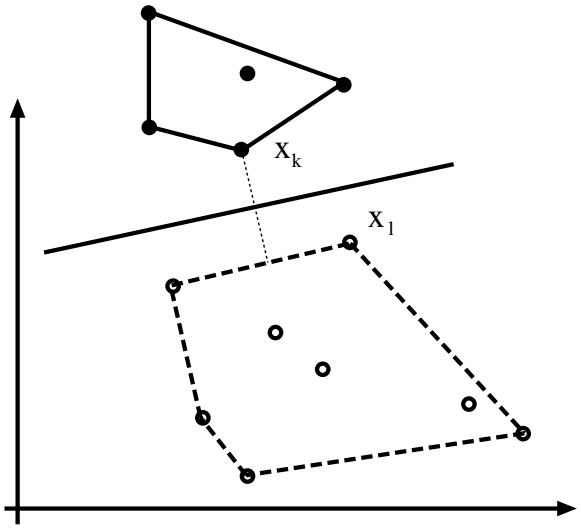


Figure 21.17. The hyperplane constructed by a support vector classifier for a plane data set. The filled circles are data points corresponding to one class, and the empty circles are data points corresponding to the other. We have drawn in the convex hull of each data set. The most conservative choice of hyperplane is one that maximises the minimum distance from each hull to the hyperplane. A hyperplane with this property is obtained by constructing the shortest line segment between the hulls, and then obtaining a hyperplane perpendicular to this line segment and through its midpoint. Only a subset of the data determines the hyperplane. Of particular interest are points on each convex hull which are associated with a minimum distance between the hulls — we will use these points to find the hyperplane in the text.

This means that $\boldsymbol{w} \cdot (\boldsymbol{x}_1 - \boldsymbol{x}_2) = 2$, so that

$$\begin{aligned} dist(\boldsymbol{x}_k, \text{hyperplane}) + dist(\boldsymbol{x}_l, \text{hyperplane}) &= (\frac{\boldsymbol{w}}{|\boldsymbol{w}|} \cdot \boldsymbol{x}_k + \frac{b}{|\boldsymbol{w}|}) - (\frac{\boldsymbol{w}}{|\boldsymbol{w}|} \cdot \boldsymbol{x}_1 + \frac{b}{|\boldsymbol{w}|}) \\ &= \frac{\boldsymbol{w}}{|\boldsymbol{w}|} \cdot (\boldsymbol{x}_1 - \boldsymbol{x}_2) = \frac{2}{|\boldsymbol{w}|} \end{aligned}$$

This means that maximising the distance is the same as *minimising* $(1/2)\boldsymbol{w} \cdot \boldsymbol{w}$. We now have the constrained minimisation problem:

$$\text{minimize} \quad (1/2)\boldsymbol{w} \cdot \boldsymbol{w}$$

$$\text{subject to} \quad y_i \left(\boldsymbol{w} \cdot \boldsymbol{x}_i + b\right) \geq 1$$

where there is one constraint for each data point.

Solving for the Support Vector Machine

We can solve this problem, by introducing Lagrange multipliers α_i to obtain the Lagrangian

$$(1/2)\boldsymbol{w}\cdot\boldsymbol{w} - \sum_1^N \alpha_i \left(y_i \left(\boldsymbol{w}\cdot\boldsymbol{x}_1 + b\right) - 1\right)$$

This Lagrangian needs to be minimised with respect to $\boldsymbol{w}$ and $\boldsymbol{b}$ and maximised with respect to α_i — these are the Karush-Kuhn-Tucker conditions []. A little manipulation leads to the requirements that

$$\sum_1^N \alpha_i y_i = 0$$

and

$$\boldsymbol{w} = \sum_1^N \alpha_i y_i \boldsymbol{x}_i$$

This second expression is why the device is known as a support vector machine. Generally, it will be the case that the hyperplane is determined by a relatively small number of example points, and the position of other examples is irrelevant (see figure 21.17 — everything inside the convex hull of each set of examples is irrelevant to choosing the hyperplane, and most of the hull vertices are, too). This means that we expect that most α_i are zero, and the data points corresponding to non-zero α_i — which are the ones that determine the hyperplane — are known as the **support vectors**.

Now by substituting these expressions into the original problem and manipulating, we obtain the **dual problem** given by

$$\begin{aligned} \text{maximize} \quad & \sum_i^N \alpha_i - \frac{1}{2}\sum_{i,j=1}^N \alpha_i (y_i y_j \boldsymbol{x}_i \cdot \boldsymbol{x}_j)\alpha_j \\ \text{subject to} \quad & \alpha_i \geq 0 \\ \text{and} \quad & \sum_{i=1}^N \alpha_i y_i = 0 \end{aligned}$$

You should notice that the criterion is a quadratic form in the Lagrange multipliers. This problem is a standard numerical problem, known as **quadratic programming**. One can use standard packages quite successfully for this problem, but it does have special features — while there may be a very large number of variables, most will be zero at a solution point — which can be exploited [].

21.5.2 Finding Pedestrians using Support Vector Machines

At a fairly coarse scale, pedestrians have a characteristic, "lollipop-like" appearance — a wide torso on narrower legs. This suggests that they can be found using a

Notation:
We have a training set of N examples

$$\{(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_N, y_N)\}$$

where y_i is either 1 or -1.

Solving for the SVM:
Set up and solve the dual optimization problem:

$$\begin{aligned} \text{maximize} \quad & \sum_i \alpha_i - \frac{1}{2} \sum_{i,j=1}^{N} \alpha_i (y_i y_j \boldsymbol{x}_i \cdot \boldsymbol{x}_j) \alpha_j \\ \text{subject to} \quad & \alpha_i \geq 0 \\ \text{and} \quad & \sum_{i=1}^{N} \alpha_i y_i = 0 \end{aligned}$$

We can then determine $\boldsymbol{w}$ from

$$\boldsymbol{w} = \sum_1^N \alpha_i y_i \boldsymbol{x}_i$$

Now for any example point $\boldsymbol{x}_i$ where α_i is non-zero, we have that

$$y_i(\boldsymbol{w} \cdot \boldsymbol{x}_i + b) = 1$$

which yields the value of b.

Classifying a point:
Any new data point is classified by

$$\begin{aligned} f(\boldsymbol{x}) &= \text{sign}\,(\boldsymbol{w} \cdot \boldsymbol{x} + b) \\ &= \text{sign}\left(\left(\sum_1^N \alpha_i y_i \boldsymbol{x} \cdot \boldsymbol{x}_i\right) + b\right) \\ &= \text{sign}\left(\sum_1^N (\alpha_i y_i \boldsymbol{x} \cdot \boldsymbol{x}_i + b)\right) \end{aligned}$$

Algorithm 21.9: *Finding an SVM for a Linearly Separable Problem*

support vector machine. The general strategy is the same as for the face-finding example in section **??**: each image window of a fixed size is presented to a classifier,

which determines whether the window contains a pedestrian or not. The number of pixels in the window may be large, and we know that many pixels may be irrelevant. In the case of faces, we could deal with this by cropping the image to an oval shape which would contain the face. This is harder to do with pedestrians, because their outline is of a rather variable shape.

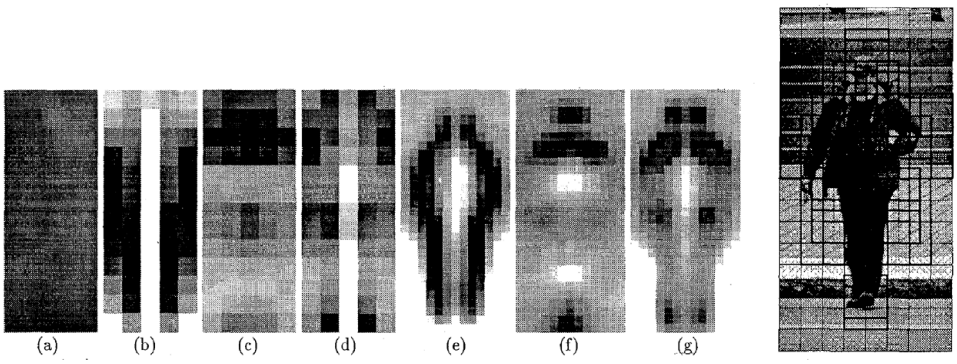


Figure 21.18. On the **left**, averages over the training set of different wavelet coefficients at different positions in the image. Coefficients that are above the (spatial) average value are shown dark, and those that are below are shown light. We expect that noise has the average value, meaning that coefficients that are very light or very dark contain information that could identify pedestrians. On the **right**, a grid showing the support domain for the features computed. Notice that this follows the boundary of the pedestrian fairly closely. *figure from Papageorgiou, Oren and Poggio, A general framework for object detection, p.3, in the fervent hope of receiving permission*

We need to identify features that can help determine whether a window contains a pedestrian or not. It is natural to try to obtain a set of features from a set of examples. A variety of feature selection algorithms might be appropriate here (all of them are variants of search). Papageorgiou, Oren and Poggio chose to look at local features — **wavelet coefficients**, which are the response of specially selected filters with local support — and to use an averaging approach. In particular, they argue that the background in a picture of a pedestrian looks like noise, images that don't contain pedestrians look like noise, and the average noise response of their filters is known. This means that attractive features are ones whose average over many images of pedestrians is different from their noise response. If we average the response of a particular filter in a particular position over a large number of images, and the average is similar to a noise response, then that filter in that position is not particularly informative.

Now that features have been chosen, training follows the lines of section **??**. Papageorgiou, Oren and Poggio use bootstrapping (section **??**), which appears to improve performance significantly.

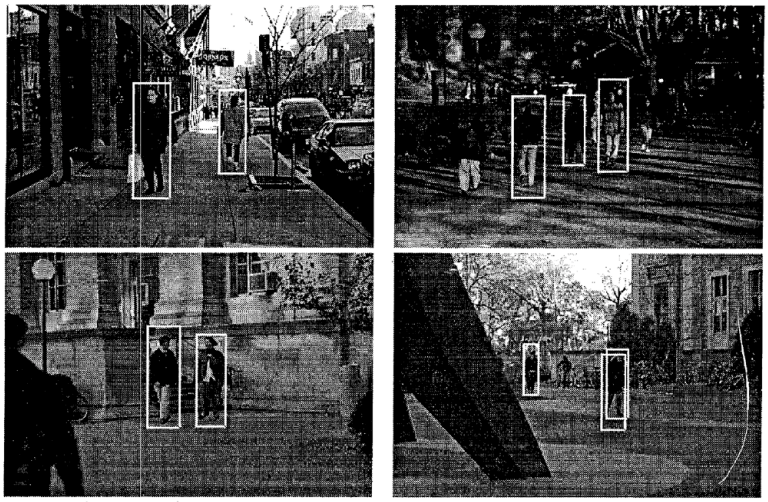
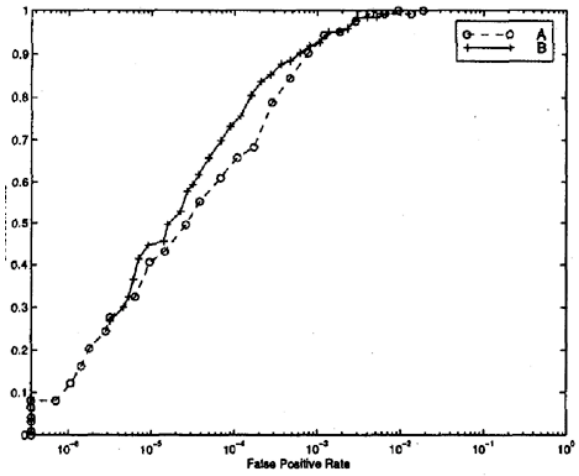


Figure 21.19. Examples of pedestrians detected using the method of Papageorgiou, Oren and Poggio. While not all pedestrians are found, there is a fairly high detection rate. The ROC is in figure 21.20. *figure from Papageorgiou, Oren and Poggio, A general framework for object detection, p.3, in the fervent hope of receiving permission*



(b) People Detection System

Figure 21.20. The receiver operating curve for the pedestrian detection system of Papageorgiou, Oren and Poggio. *figure from Papageorgiou, Oren and Poggio, A general framework for object detection, p.4, in the fervent hope of receiving permission*

21.6 Conclusions

This topic is one on which no orthodoxy is yet established; instead, one tries to use methods that seem likely to work on the problem in hand. For the sake of brevity, we have omitted a vast number of useful techniques; Vapnik's book [**?**], Bishop's book [Bishop, 1995], Ripley's book [Ripley, 1996] and McClachlan's book [**?**] are good places to start.

Choosing a decision boundary is strictly easier than fitting a posterior model. However, with a decision boundary there is no reliable indication of the extent to which an example belongs to one or another class, as there is with a posterior model. Furthermore, fitting a decision boundary requires that we know the classes to which the example objects should be allocated. It is by no means obvious that one can construct an unambiguous class hierarchy for the objects we encounter in recognition problems. Both approaches can require very large numbers of examples to build useful classifiers. Typically, the stronger the model that is applied, the fewer examples required to build a classifier.

It is difficult to build classifiers that are really successful when objects have a large number of degrees of freedom (though see section **??**), and classifiers tend to be difficult to use if the number of features can vary from example to example; in both cases, some form of structural model appears to be necessary. However, estimating, representing and manipulating probability densities in the very high dimensional spaces that occur in vision problems is practically impossible, unless very strong assumptions are applied. Furthermore, it is easy to build probability models for which inference is again practically impossible; it isn't yet known how to build models that are easy to handle of the scale required for vision problems.

This subject is currently at the cutting edge of research in vision and learning. It's hard to know how to choose a method for a given problem, and opportunism seems to be the best approach at present. The examples in this chapter and in the next chapter illustrate a range of approaches that have been taken — some are very successful — but don't yet represent a clear theory.

An alternative approach is to train multiple classifiers and combine their outputs. This strategy is usually known as **boosting**. Boosting is most useful for classifiers with quite simple decision boundaries; these are usually easy to train, but have quite poor performance. Typically, we train a classifier, and then determine the examples in the training set that it gets wrong. These examples are then emphasized — either by weighting errors on them more heavily, or inserting copies into the training set — and a new classifier is trained. We now find out what the new classifier gets wrong, and emphasize these examples and train again; this proces continues through many iterations. Now the outputs of all the classifiers are combined using a set of weights.

Assignments

Exercises

1. Assume that we are dealing with measurements $\boldsymbol{x}$ in some feature space S. There is an open set D where any element is classified as class one, and any element in the interior of $S - D$ is classified as class two.

   - Show that

$$\begin{aligned} R(s) &= Pr\{1 \to 2|\text{using } s\} L(1 \to 2) + Pr\{2 \to 1|\text{using } s\} L(2 \to 1) \\ &= \int_D p(2|\boldsymbol{x}) d\boldsymbol{x} L(1 \to 2) + \int_{S-D} p(1|\boldsymbol{x}) d\boldsymbol{x} L(2 \to 1) \end{aligned}$$

   - Why are we ignoring the boundary of D (which is the same as the boundary of $S - D$) in computing the total risk?

2. In section 2, we said that, if each class-conditional density had the same covariance, the classifier of algorithm 2 boiled down to comparing two expressions that are linear in $\boldsymbol{x}$.

   - Show that this is true.
   - Show that, if there are only two classes, we need only test the sign of a linear expression in $\boldsymbol{x}$.

3. In section 21.3.1, we set up a feature u, where the value of u on the i'th data point is given by $u_i = \boldsymbol{v} \cdot (\boldsymbol{x}_i - \boldsymbol{\mu})$. Show that u has zero mean.

4. In section 21.3.1, we set up a series of features u, where the value of u on the i'th data point is given by $u_i = \boldsymbol{v} \cdot (\boldsymbol{x}_i - \boldsymbol{\mu})$. We then said that the $\boldsymbol{v}$ would be eigenvectors of Σ, the covariance matrix of the data items. Show that the different features are independent, using the fact that the eigenvectors of a symmetric matrix are orthogonal.

5. In section **??**, we said that the ROC was invariant to choice of prior. Prove this.

Programming Assignments

1.

II Appendix: Support Vector Machines for Datasets that are not Linearly Separable

In many cases, a separating hyperplane will not exist. To allow for this case, we introduce a set of **slack variables**, $\xi_i \geq 0$, which represent the amount by which the constraint is violated. We can now write our new constraints as

$$y_i \left(\boldsymbol{w} \cdot \boldsymbol{x}_1 + b \right) \geq 1 - \xi_i$$

and we modify the objective function to take account of the extent of the constraint violations, to get the problem

$$\begin{aligned} \text{minimize} \quad & \frac{1}{2} \boldsymbol{w} \cdot \boldsymbol{w} + C \sum_{i=1}^{N} \xi_i \\ \text{subject to} \quad & y_i \left(\boldsymbol{w} \cdot \boldsymbol{x}_1 + b \right) \geq 1 - \xi_i \\ \text{and} \quad & \xi_i \geq 0 \end{aligned}$$

Here C gives the significance of the constraint violations with respect to the distance between the points and the hyperplane. The dual problem becomes

$$\begin{aligned} \text{maximize} \quad & \sum_{i}^{N} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{N} \alpha_i (y_i y_j \boldsymbol{x}_i \cdot \boldsymbol{x}_j) \alpha_j \\ \text{subject to} \quad & C \geq \alpha_i \geq 0 \\ \text{and} \quad & \sum_{i=1}^{N} \alpha_i y_i = 0 \end{aligned}$$

Again, we have

$$\boldsymbol{w} = \sum_{1}^{N} \alpha_i y_i \boldsymbol{x}_i$$

but recovering b from the solution to the dual problem is slightly more interesting. For each example where $C > \alpha_i > 0$ (note that these are strict inequalities, unlike the constraints) the slack variable ξ_i will be zero. This means that

$$\sum_{j=1}^{N} y_j \alpha_j \boldsymbol{x}_i \cdot \boldsymbol{x}_j + b = y_i$$

for these values of i. This expression yields b. Again, the optimization problem is a quadratic programming problem, though there is no guarantee that many points will have $\alpha_i = 0$.

III Appendix: Using Support Vector Machines with Non-Linear Kernels

For many data sets, it is unlikely that a hyperplane will yield a good classifier. Instead, we want a decision boundary with a more complex geometry. One way to achieve this is to map the feature vector into some new space, and look for a hyperplane in that new space. For example, if we had a plane data set that we were convinced could be separated by plane conics, we might apply the map

$$(x, y) \to (x^2, xy, y^2, x, y)$$

to the dataset. A classifier boundary that is a hyperplane in this new feature space is a conic in the original feature space. In this form, this idea is not particularly useful, because we might need to map the data into a very high dimensional space (for example, assume that we know the classifier boundary has degree two, and the data is 10 dimensional — we would need to map the data into a 65 dimensional space).

Write the map as $\boldsymbol{x}' = \phi(\boldsymbol{x})$. Write out the optimisation problem for the new points $\boldsymbol{x}'_i$; you will notice that the only form in which $\boldsymbol{x}'_i$ appears is in the terms

$$\boldsymbol{x}'_i \cdot \boldsymbol{x}'_j$$

which we could write as $\phi(\boldsymbol{x}_i) \cdot \phi(\boldsymbol{x}_j)$. Apart from always being positive, this term doesn't give us much information about ϕ. In particular, the map doesn't appear explicitly in the optimisation problem. If we did solve the optimisation problem, the final classifier would be

$$\begin{aligned} f(\boldsymbol{x}) &= \text{sign}\left(\sum_1^N (\alpha_i y_i \boldsymbol{x}' \cdot \boldsymbol{x}'_i + b)\right) \\ &= \text{sign}\left(\sum_1^N (\alpha_i y_i \phi(\boldsymbol{x}) \cdot \phi(\boldsymbol{x}_i) + b)\right) \end{aligned}$$

Assume that we have a function $k(\boldsymbol{x}, \boldsymbol{y})$ which is positive for all pairs of $\boldsymbol{x}$, $\boldsymbol{y}$. It can be shown that, under various technical conditions of no interest to us, there is some ϕ such that $k(\boldsymbol{x}, \boldsymbol{y}) = \phi(\boldsymbol{x}) \cdot \phi(\boldsymbol{y})$. All this allows us to adopt a clever trick — instead of constructing ϕ explicitly, we obtain some appropriate $k(\boldsymbol{x}, \boldsymbol{y})$, and use it in place of ϕ. In particular, the dual optimisation problem becomes

$$\begin{aligned} \text{maximize} \quad & \sum_i^N \alpha_i - \frac{1}{2} \sum_{i,j=1}^N \alpha_i (y_i y_j k(\boldsymbol{x}_i, \boldsymbol{x}_j)) \alpha_j \\ \text{subject to} \quad & \alpha_i \geq 0 \\ \text{and} \quad & \sum_{i=1}^N \alpha_i y_i = 0 \end{aligned}$$

and the classifier becomes

$$f(\boldsymbol{x}) = \text{sign}\left(\sum_{1}^{N}(\alpha_i y_i k(\boldsymbol{x}, \boldsymbol{x}_i) + b)\right)$$

Of course, these equations assume that the dataset are separable in the new feature space represented by k. This may not be the case, in which case the problem becomes

$$\begin{aligned} \text{maximize} \quad & \sum_{i}^{N}\alpha_i - \frac{1}{2}\sum_{i,j=1}^{N}\alpha_i(y_i y_j k(\boldsymbol{x}_i, \boldsymbol{x}_j))\alpha_j \\ \text{subject to} \quad & C \geq \alpha_i \geq 0 \\ \text{and} \quad & \sum_{i=1}^{N}\alpha_i y_i = 0 \end{aligned}$$

and the classifier becomes

$$f(\boldsymbol{x}) = \text{sign}\left(\sum_{1}^{N}(\alpha_i y_i k(\boldsymbol{x}, \boldsymbol{x}_i) + b)\right)$$

There are a variety of possible choices for $k(\boldsymbol{x}, \boldsymbol{y})$. The main issue is that it must be positive for all values of $\boldsymbol{x}$ and $\boldsymbol{y}$. Some typical choices are shown in table 21.1. There doesn't appear to be any principled method for choosing between kernels; one tries different forms, and uses the one that gives the best error rate, measured using cross-validation.

Kernel form	Qualitative properties of ϕ represented by this kernel
$(\boldsymbol{x} \cdot \boldsymbol{y})^d$	ϕ is all monomials of degree d
$(\boldsymbol{x} \cdot \boldsymbol{y} + c)^d$	ϕ is all monomials of degree d or below
$\tanh(a\boldsymbol{x} \cdot \boldsymbol{y} + b)$	
$\exp(-\frac{(\boldsymbol{x}-\boldsymbol{y})^T(\boldsymbol{x}-\boldsymbol{y})}{2\sigma^2})$	

Table 21.1. Some support vector kernels