Contents

CHAPTER 1

Learning to Classify

A **classifier** is a procedure that accepts a set of features and produces a class label for them. There could be two, or many, classes, though it is usual to produce multi-class classifiers out of two-class classifiers. Classifiers are immensely useful, and find wide application, because many problems are naturally decision problems. For example, if you wish to determine whether to place an advert on a web-page or not, you would use a classifier (i.e. look at the page, and say yes or no according to some rule). As another example, if you have a program that you found for free on the web, you would use a classifier to decide whether it was safe to run it (i.e. look at the program, and say yes or no according to some rule). As yet another example, you can think of doctors as extremely complex multi-class classifiers.

Classifiers are built by taking a set of labeled examples and using them to come up with a rule that assigns a label to any new example. In the general problem, we have a training dataset $(\mathbf{x}_i, y_i)$; each of the **feature vectors** $\mathbf{x}_i$ 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.

TODO: clean this up

Classifiers are a crucial tool in high-level vision, because many problems can be abstracted in a form that looks like classification. In this chapter, we describe the basic ideas and methods of classification, abstracted away from any vision problem (Chapter **??** applies classifiers to vision problems). Section 1.1 describes basic notions. In Section 1.2, we describe different ways to build classifiers. Finally, Section 1.3 gives some important practical tricks.

1.1 CLASSIFICATION, ERROR, AND LOSS

You should think of a classifier as a rule, though it might not be implemented that way. We pass in a feature vector, and the rule returns a class label. We know the relative costs of mislabeling each class and must come up with a rule that can take any plausible $\mathbf{x}$ and assign a class to it, in such a way that the expected mislabeling cost is as small as possible, or at least tolerable. For most of this chapter, we will assume that there are two classes, labeled 1 and -1. Section 1.3.2 shows methods for building multi-class classifiers from two-class classifiers.

1.1.1 Using Loss to Determine Decisions

The choice of classification rule must depend on the cost of making a mistake. A two-class classifier can make two kinds of mistake. A **false positive** occurs when a negative example is classified positive; a **false negative** occurs when a positive example is classified negative. For example, pretend there is only one disease; then doctors would be classifiers, deciding whether a patient had it or not. If this disease is dangerous, but is safely and easily treated, then false negatives are expensive errors, but false positives are cheap. Similarly, if it is not dangerous, but the treatment is difficult and unpleasant, then false positives are expensive errors and false negatives are cheap.

Generally, we write outcomes as $(i \to j)$, meaning that an item of type i is classified as an item of type j. There are four outcomes for the two-class case. Each outcome has its own cost, which is known as a **loss**. Hence, we have a loss function that we 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 that strategy, as a function of the kind of item. The **total risk** is the total expected loss when using the classifier. The total risk depends on the strategy, but not on the examples. Write $p(-1 \to 1|\text{using } s)$ for the probability that class -1 is labeled class 1 (and so on). Then, if there were two classes, the total risk of using strategy s would be

$$R(s) = p(-1 \to 1|\text{using } s)L(-1 \to 1) + p(1 \to -1|\text{using } s)L(-1 \to 1).$$

The desirable strategy is one that minimizes this total risk.

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. If the decision boundary is optimal, then *for points on the decision boundary*, either choice of class has the same expected loss; if this weren'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 label -1 yields the same expected loss as choosing label 1.

Now write $p(-1|\mathbf{x})$ for the posterior probability of label -1 given feature vector $\mathbf{x}$ (and so on). Although this might be very hard to know in practice, we can manipulate the abstraction and gain some insight. A choice of label $y = 1$ for a point $\mathbf{x}$ at the decision boundary yields an expected loss

$$p(-1|\mathbf{x})L(-1 \to 1) + p(1|\mathbf{x})L(1 \to 1) = p(-1|\mathbf{x})L(-1 \to 1),$$

and if we choose the other label, the expected loss is

$$p(1|\mathbf{x})L(1 \to -1),$$

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

$$p(-1|\mathbf{x})L(-1 \to 1) = p(1|\mathbf{x})L(1 \to -1).$$

At points off the boundary, we must choose the class with the *lowest* expected loss. Recall that if we choose label 1 for a point $\mathbf{x}$, the expected loss is

$$p(-1|\mathbf{x})L(-1 \to 1),$$

and so on. This means that we should choose label -1 if

$$p(-1|\mathbf{x})L(-1 \to 1) > p(1|\mathbf{x})L(1 \to -1)$$

and label 1 if the inequality is reversed. Notice it does not matter which label we choose at the decision boundary.

A Multi-class Classifier that Minimizes Total Risk

Analyzing expected loss gives a strategy for choosing from any number of classes. We allow the option of refusing to decide which class an object belongs to, which is useful in some problems. Refusing to decide costs d. Conveniently, if d is larger than any misclassification loss, we will never refuse to decide. This means our analysis covers the case when we are forced to decide. The same reasoning applies as above, but there are more boundaries to consider. The simplest case, which is widely dominant in vision, is when loss is **0-1 loss**; here the correct answer has zero loss, and any error costs one.

In this case, the best strategy, known as the **Bayes classifier**, is given in Algorithm 1.1. The total risk associated with this rule is known as the **Bayes risk**; this is the smallest possible risk that we can have using a classifier for this problem. 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 behavior of the risk as the number of examples increases (e.g., one might want the risk to converge to the Bayes risk in probability if the number of examples is large). The Bayes risk is seldom zero, as Figure 1.1 illustrates.

For a loss function

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

the best strategy is

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

Algorithm 1.1: *T*he Bayes Classifier.

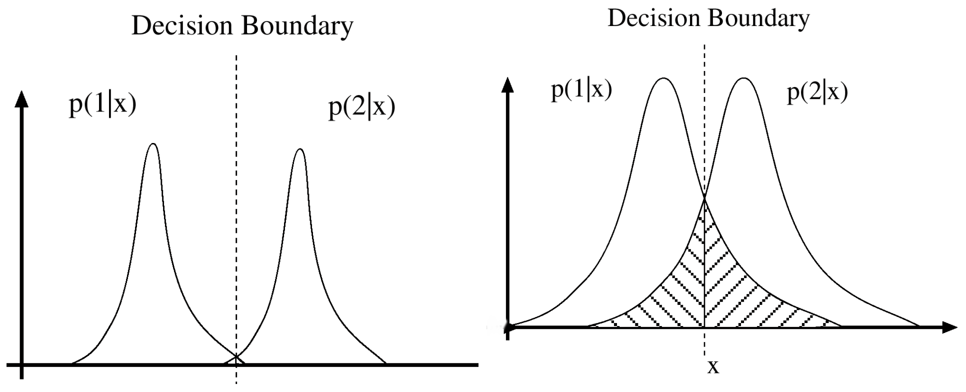

FIGURE 1.1: *T*his figure shows typical elements of a two-class classification problem. We have plotted $p(\text{class}|x)$ as a function of the feature x. Assuming that $L(-1 \rightarrow 1) = L(1 \rightarrow -1)$, we have marked the classifier boundaries. In this case, the Bayes risk is the sum of the 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 is small; for the case on the right, the Bayes risk is rather large.

1.1.2 Example: Building a Classifier out of Histograms

TODO: write this; pizza and skin

As we saw in chapter **??**, histograms reveal a great deal about data. A histogram is a representation of a probability distribution, and so we can use histograms to build classifiers. Recall that in the pizza data of chapter **??** (you can find a version of this dataset, along with a neat backstory, at `http://www.amstat.org/publications/jse/jse_data_archive.htm`), there were pizzas from two manufacturers — EagleBoys and Dominos. For these pizza's, we had the diameter, and some information about the topping. We could then try to predict the manufacturer from the diameter of the pizza.

We have two classes (EagleBoys and Dominos). Write x for the diameter of a pizza. As we have seen, to classify we need a model of $p(E|x)$. Since there are two classes, $p(D|x) = 1 - p(E|x)$, so the model of $p(E|x)$ is enough. A natural way to get this model is to use Bayes rule:

$$p(E|x) = \frac{p(x|E)p(E)}{p(x)}$$

We can model $p(x|E)$, $p(E)$ and $p(x)$ with histograms. We construct a set of boxes of appropriate sizes, then count data items into the boxes. The conditional probability that the diameter of a pizza lies in a box, conditioned on its coming from EagleBoys is $p(x|E)$; this can be estimated by taking the number of EagleBoys pizzas in that box, and dividing by the total number of EagleBoys pizzas. Similarly, we can estimate $p(E)$ by the fraction of all pizzas that come from EagleBoys. Finally, we could get $p(x)$ by taking the fraction of all pizzas in the relevant histogram box.

However, we don't really need $p(x)$. Write $L(E \to D)$, etc. for the loss of misclassifying pizzas. Recall that we want to test

$$p(E|x)L(E \to D) > p(D|x)L(D \to E)$$

but this is

$$\frac{p(x|E)p(E)}{p(x)}L(E \to D) > \frac{p(x|D)p(D)}{p(x)}L(D \to E).$$

The first thing to notice about this expression is that only the ratio of the losses matters; we could write an equivalent test as

$$\frac{p(x|E)}{p(x|D)} > \frac{p(D)}{p(E)}\frac{L(D \to E)}{L(E \to D)}.$$

If we do not know the losses, we can simply test the ratio $\frac{p(x|E)}{p(x|D)}$ against a variety of different thresholds, and then choose the one that works best (for example, in a test on a different dataset).

Listing **??** shows a simple histogram-based classifier for the pizza data. To train this classifier, I split the pizza data into two pools — training data, and test data (more on this in the next section). I built the histograms on the training data, then evaluated the classifier on test data for different values of the threshold in listing **??**

Different values of the threshold give different false-positive and false-negative rates. This information can be summarized with a **receiver operating characteristic curve**, or **ROC**. This curve is a plot of the **detection rate** or **true positive rate** as a function of the **false positive rate** for a particular model as the threshold changes (Figure **??**). An ideal model would detect all positive cases and produce no false positives, for any threshold value; in this case, the curve would be a single point. A model that has no information about whether an example is a positive or a negative will produce the line from $(0, 0)$ to $(1, 1)$. If the ROC lies below this line, then we can produce a better classifier by inverting the decision of the original classifier, so this line is the worst possible classifier. The detection rate never goes down as the false positive rate goes up, so the ROC is the graph of a non-decreasing function. Figure 1.2 shows the ROC for the pizza classifier.

This general strategy can work for data of moderate dimension. A classic example in computer vision is a skin detector, built by Jones and Rehg (). There are two main sources of skin color: melanin, which causes darker or lighter skin; and blood, which tends to tint the skin red. Skin with strong hues looks strange or unhealthy (even a mild blue or purple tint makes skin look cyanotic or very cold; mild green tints suggest decomposition). This means that digital images tend to be adjusted so that skin has a relatively narrow range of hues and of intensities, even though there is a very wide variation in melanin content of skin. Jones and Rehg built quite a good skin pixel classifier by constructing a histogram of the r, g, and b values of all skin (resp. non-skin) pixels, and following the recipe for pizza classification above. Figure 1.3 shows the ROC for this classifier.

Models of a classification problem can be compared by comparing their ROC's. Alternatively, we can build a summary of the ROC. Most commonly used in computer vision is the area under the ROC (the **AUC**), which is 1 for a perfect classifier,

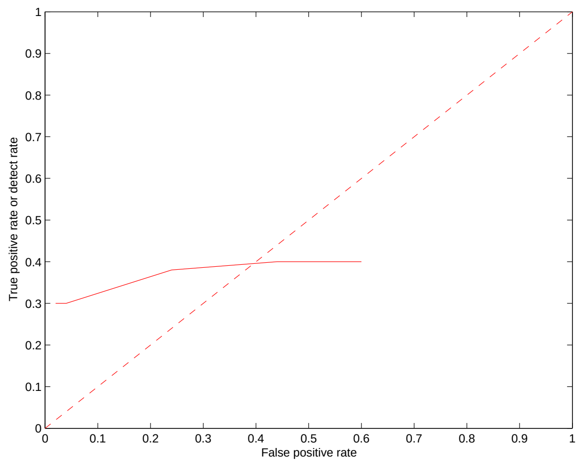

FIGURE 1.2: *T*he receiver operating curve for a histogram-based classifier to tell EagleBoys pizza from Domino's pizza using the diameter. This curve plots the detection rate against the false-negative rate for a variety of values of the parameter θ. A perfect classifier has an ROC that, on these axes, is a horizontal line at 100% detection. This classifier isn't very good, as you would expect if you look back at the histograms for the pizza data.

and 0.5 for a classifier that has no information about the problem. The area under the ROC has the following interpretation: assume we select one positive example and one negative example uniformly at random, and display them to the classifier; the AUC is the probability that the classifier tells correctly which of these two is positive.

1.1.3 Training Error, Test Error, and Overfitting

It can be quite difficult to know a good loss function, but one can usually come up with a plausible model. If we knew the posterior probabilities, building a classifier would be straightforward. Usually we don't, and must build a model from data. This model could be a model of the posterior probabilities, or an estimate of the decision boundaries. In either case, we have only the training data to build it with. **Training error** is the error a model makes on the training data set.

Generally, we will try to make this training error small. However, what we really want to minimize is the **test error**, the error the classifier makes on test data. We cannot minimize this error directly, because we don't know the test set (if we did, special procedures in training apply **?**). However, classifiers that have small training error might not have small test error. 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 learned from data, and has a zero error rate on the training dataset; it is likely to be unhelpful on any other dataset, however.

The phenomenon that causes test error to be worse than training error is

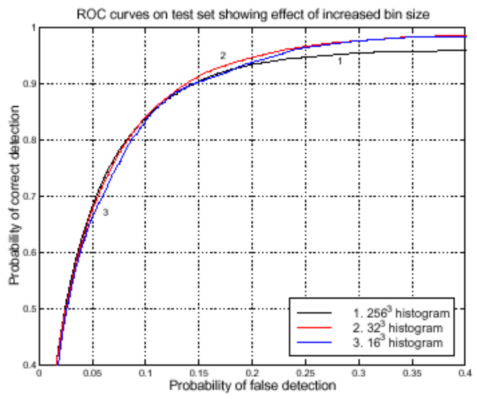

FIGURE 1.3: *T*he receiver operating curve for the skin detector of Jones and Rehg. This plots the detection rate against the false-negative rate for a variety of values of the parameter θ. A perfect classifier has an ROC that, on these axes, is a horizontal line at 100% detection. Notice that the ROC varies slightly with the number of boxes in the histogram. *This figure was originally published as Figure 7 of "Statistical color models with application to skin detection," by M.J. Jones and J. Rehg, Proc. IEEE CVPR, 1999 © IEEE, 1999.*

sometimes called **overfitting** (other names include **selection bias**, because the training data has been selected and so isn't exactly like the test data, and **generalizing badly**, because the classifier fails to generalize). It occurs because the classifier has been trained to perform well *on the training dataset*. The training dataset is not the same as the test dataset. First, it is quite likely smaller. Second, it might be biased through a variety of accidents. This means that small training error may have to do with quirks of the training dataset that don't occur in other sets of examples. It is quite possible that, in this case, the test error will be larger than the training error. Generally, we expect classifiers to perform somewhat better on the training set than on the test set. Overfitting can result in a substantial difference between performance on the training set and performance on the test set. One consequence of overfitting is that classifiers should always be evaluated on test data. Doing this creates other problems, which we discuss in Section 1.1.4.

A procedure called **regularization** attaches a penalty term to the training error to get a better estimate of the test error. This penalty term could take a variety of different forms, depending on the requirements of the application. Section **??** describes regularization in further detail.

1.1.4 Error Rate and Cross-Validation

There are a variety of methods to describe the performance of a classifier. Natural, straightforward choices are to report the **error rate**, the percentage of classification attempts on a test set that result in the wrong answer. This presents an important difficulty. We cannot estimate the error rate of the classifier using training data, because the classifier has been trained to do well on that data, which will mean our error rate estimate will be an underestimate. An alternative is to split some training data to form a validation set, then train the classifier on the rest of the data, and evaluate on the validation set. This has the difficulty that the classifier will not be the best estimate possible, because we have left out some training data when we trained it. This issue can become a significant nuisance when we are trying to tell which of a set of classifiers to use—did the classifier perform poorly on validation data because it is not suited to the problem representation or because it was trained on too little data?

We can resolve this problem with **cross-validation**, which involves repeatedly: splitting data into training and validation sets uniformly and at random, training a classifier on the training set, evaluating it on the validation set, and then averaging the error over all splits. This allows an estimate of the likely future performance of a classifier, at the expense of substantial computation.

```
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
dataset.
```

Algorithm 1.2: *C*ross-Validation

The most usual form of this algorithm involves omitting single items from the dataset 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 (see, e.g., **?**). We do 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 dataset look similar to one another, which suggests that a representation of the relevant probabilities derived from the dataset might be quite good.

For a multi-class classifier, it is often helpful to know which classes were misclassified. We can compute a **class-confusion matrix**, a table whose i, jth entry is the number of times an item of true class i was labeled j by the classifier (notice that this definition is not symmetric). If there are many classes, this matrix can be rendered as an image (Figure 1.4), where the intensity values correspond to counts; typically, larger values are lighter. Such images are quite easy to assess at a glance. One looks for a light diagonal (because the diagonal elements are the counts of correct classifications), for any row that seems dark (which means that there were few elements in that class), and for bright off-diagonal elements (which

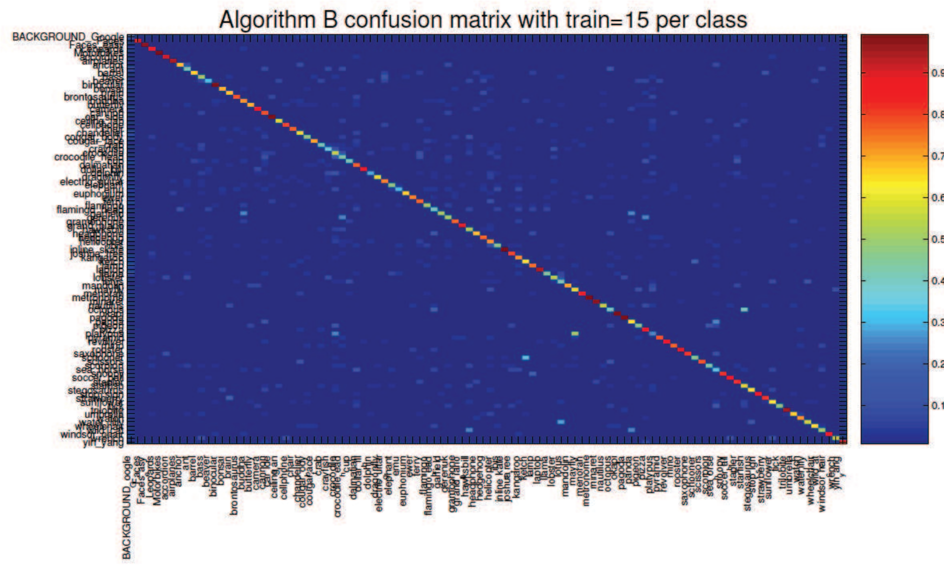

FIGURE 1.4: *An* example of a class confusion matrix from a recent image classification system, due to **?**. The vertical bar shows the mapping of color to number (warmer colors are larger numbers). Note the redness of the diagonal; this is good, because it means the diagonal values are large. There are spots of large off-diagonal values, and these are informative, too. For example, this system confuses: schooners and ketches (understandable); waterlily and lotus (again, understandable); and platypus and mayfly (which might suggest some feature engineering would be a good idea). *This figure was originally published as Figure 5 of "SVM-KNN: Discriminative Nearest Neighbor Classification for Visual Category Recognition," by H. Zhang, A. Berg, M. Maire, and J. Malik, Proc. IEEE CVPR, 2006, © IEEE, 2006.*

are high-frequency misclassifications).

1.2 MAJOR CLASSIFICATION STRATEGIES

Usually, we do not know $p(1|\mathbf{x})$, or $p(1)$, or $p(\mathbf{x}|1)$ exactly, and we must determine a classifier from an example dataset. 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 taste and circumstance). There is a wide variety of ways of doing this, some of which we see in the following sections.

- **Determining decision boundaries directly:** Quite bad probability models can produce good classifiers, as Figure 1.5 indicates. This is because the decision boundaries, rather than the details of the probability model, are what determine the performance of a classifier (the main role of the probability model in the Bayes classifier is to identify the decision boundaries).

This suggests that we could 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 modeling the data source.

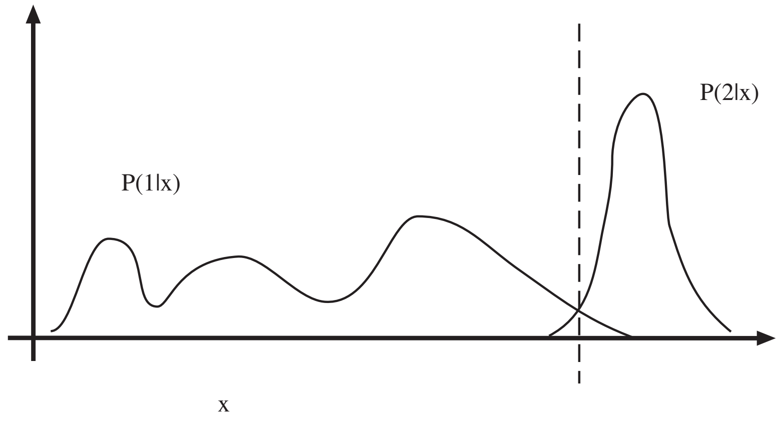

FIGURE 1.5: *T*he figure shows posterior densities for two classes. The optimal decision boundary is shown as a dashed line. Notice that although a normal density may provide rather a poor fit *to the posteriors*, the quality of the classifier it provides depends only on *how well it predicts the position 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 if they would predict the boundary in the right place.

1.2.1 Example: A Nonparametric Classifier Using Nearest Neighbors

It is reasonable to assume that example points near an unclassified point should indicate the class of that point. **Nearest neighbors** 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 neighbor 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 neighbor classifier is usually known as a **k-nearest neighbor classifier**, and a $(1, 0)$-nearest neighbor classifier is usually known as a **nearest neighbor classifier**.

Nearest neighbor classifiers are known to be good, in the sense that the risk of using a nearest neighbor 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 neighbor classifier goes down as $1/\sqrt{k}$. In practice, one seldom uses more than three nearest neighbors. Furthermore, if the Bayes risk is zero, the expected risk of using a k-nearest neighbor classifier is also zero (see **?** for more detail on all these points). Finding the k nearest points for a particular query can be difficult, and Section **??** reviews this point.

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 color, and one is an angle? One possibility is to use a covariance estimate to compute a Mahalanobis-like distance. It is almost always a good idea to scale each feature independently so that the variance of each feature is the same, or at least consistent; this prevents features with very large scales dominating those with very small scales.

1.2.2 Example: Logistic Regression

Logistic regression is a classifier that models the class-conditional densities by requiring that

$$\log \frac{p(1|\mathbf{x})}{p(-1|\mathbf{x})} = \mathbf{a}^T\mathbf{x}$$

where $\mathbf{a}$ is a vector of parameters. The decision boundary here will be a hyperplane passing through the origin of the feature space. Notice that we can turn this into a general hyperplane in the original feature space by extending each example's feature vector by attaching a 1 as the last component. This trick simplifies notation, which is why we adopt it here. It is straightforward to estimate $\mathbf{a}$ using maximum likelihood. Note that

$$p(1|\mathbf{x}) = \frac{\exp \mathbf{a}^T\mathbf{x}}{1 + \exp \mathbf{a}^T\mathbf{x}}$$

and

$$p(-1|\mathbf{x}) = \frac{1}{1 + \exp \mathbf{a}^T\mathbf{x}},$$

so that we can estimate the correct set of parameters $\hat{\mathbf{a}}$ by solving for the minimum of the negative log-likelihood, i.e.,

$$\hat{\mathbf{a}} = \underset{\mathbf{a}}{\text{argmin}} \left[- \sum_{i \in \text{examples}} (\frac{1+y_i}{2})\mathbf{a}^T\mathbf{x} - \log\left(1 + \exp \mathbf{a}^T\mathbf{x}\right) \right].$$

It turns out that this problem is convex, and is easily solved by Newton's method (e.g., **?**).

In fact, when we use maximum likelihood, we are choosing a classifier boundary that minimizes a loss function, and this is a better way to think about the problem. For example i, we write $\gamma_i = \mathbf{a}^T\mathbf{x}_i$. Our classifier will be:

$$\text{choose} \begin{cases} 1 \text{ if } \gamma_i > 0 \\ -1 \text{ if } \gamma_i < 0 \\ \text{randomly if } \gamma_i = 0. \end{cases}$$

Now write the loss for the ith example

$$\begin{aligned} L(y_i, \gamma_i) &= -\left[\frac{1}{2}(1 + y_i)\gamma_i - \log(1 + \exp \gamma_i)\right] \\ &= \log(1 + \exp(-y_i\gamma_i)) \end{aligned}$$

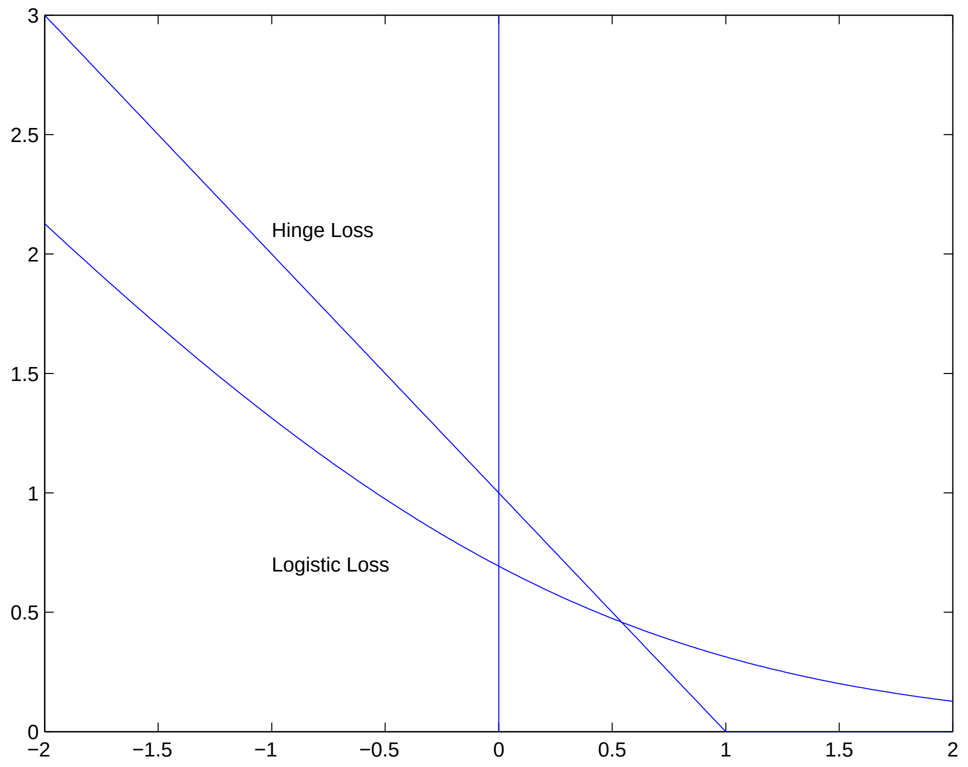

FIGURE 1.6: *The* logistic loss and the hinge loss, plotted for the case $y_i = 1$. In the case of the logistic loss, the horizontal variable is the $\gamma_i = \mathbf{a} \cdot \mathbf{x}_i$ of the text. In the case of the hinge loss, the horizontal variable is the $\mathbf{w} \cdot \mathbf{x}_i + b$ of the text. Notice that in each case, giving a strong negative response to this positive example causes a loss, that grows linearly as the magnitude of the response grows (if it grew faster, we might fear robustness problems). Notice also that giving an insufficiently positive response also causes a loss. The hinge loss isn't differentiable, and the logistic loss is.

(where the step follows from simple manipulations; see the exercises). This is plotted in Figure 1.6. This loss is sometimes known as the **logistic loss**. Notice that this loss very strongly penalizes a large positive γ_i if y_i is negative (and vice versa). However, there is no significant advantage to having a large positive γ_i if y_i is positive. This means that the significant components of the loss function will be due to examples that the classifier gets wrong, but also due to examples that have γ_i near zero (i.e., the example is close to the decision boundary). Now the total risk of applying this classifier to our set of examples is

$$\sum_{i \in \text{examples}} -\left[\frac{1}{2}(1+y_i)\gamma_i - \log\left(1+\exp\gamma_i\right)\right],$$

and it is natural to minimize this risk as a function of $\mathbf{a}$ using Newton's method (see **?**). The Hessian will be

$$\mathcal{H} = \sum_{i \in \text{examples}} \frac{\exp \gamma_i}{(1 + \exp \gamma_i)^2} \mathbf{x}_i \mathbf{x}_i^T.$$

Notice that data points where γ_i has a large absolute value make little contribution to the Hessian—it is affected mainly by points where γ_i is small, that is, points near the boundary. For these points, the Hessian looks like a weighted covariance matrix. Now if we have features that are strongly correlated, we can expect that the Hessian is poorly conditioned, because the covariance matrix will have some small eigenvalues. These will be caused by the high covariance of the features. We would typically maximize using Newton's method, which involves updating an estimate $\mathbf{a}^{(n)}$ by computing $\mathbf{a}^{(n+1)} = \mathbf{a}^{(n)} + \delta\mathbf{a}$, where we get the step $\delta\mathbf{a}$ by solving $\mathcal{H}(\delta\mathbf{a}) = -\nabla f$. When this linear system is very poorly conditioned, it means that a wide range of different $\mathbf{a}^{(n+1)}$ have essentially the same value of loss. In turn, many choices of $\mathbf{a}$ will give about the same loss *on the training data*. The training data offers no reason to choose between these $\mathbf{a}$.

However, $\mathbf{a}$ with very large norm may behave badly on future test data, because they will tend to produce large values of $\mathbf{a}^T\mathbf{x}$ for test data items $\mathbf{x}$. In turn, these can produce large losses, particularly if the sign is wrong. This suggests that we should use a value of $\mathbf{a}$ that gives small training loss, and also has a small norm. In turn, this suggests we change the objective function by adding a term that discourages $\mathbf{a}$ with large norm. This term is referred to as a **regularizer**, because it tends to discourage solutions that are large (and so have possible high loss on future test data) but are not strongly supported by the training data. The objective function becomes

$$\text{Training Loss} + \text{Regularizer}$$

which is

$$\text{Training Loss} + \lambda\,(\text{Norm of } \mathbf{a})$$

which is

$$\sum_{i \in \text{examples}} \left(\frac{1}{2}(1 + y_i)\gamma_i - \log\left(1 + \exp \gamma_i\right) \right) + \lambda \mathbf{a}^T\mathbf{a}$$

where $\lambda > 0$ is a constant chosen for good performance. Too large a value of λ, and the classifier will behave poorly on training and test data; too small a value, and the classifier will behave poorly on test data.

Usually, the value of λ is set with a validation dataset. We train classifiers with different values of λ on a test dataset, then evaluate them on a validation set—data whose labels are known, but which is not used for training—and finally choose the λ that gets the best validation error.

Regularizing training loss using the norm is a general recipe, and can be applied to most of the classifiers we describe. For some classifiers, the reasons this approach works are more recondite than those sketched here, but the model here is informative. Norms other than L_2—that is, $\| x \|_2^2 = \mathbf{x}^T\mathbf{x}$—can be used successfully. The most commonly used alternative is L_1—that is, $\| x \|_1 = \sum_i \mathsf{abs}\,(x_i)$—which leads to much more intricate minimization problems but strongly encourages zeros in the coefficients of the classifier, which is sometimes desirable.

1.2.3 Example: Class-Conditional Histograms and Naive Bayes

If we have enough labeled data, we could model the class-conditional densities with histograms. This really is practical only in low dimensions, but is sometimes useful. We obtain $p(\mathbf{x}|y = 1)$ by producing a histogram of the features of the positive examples, $p(\mathbf{x}|y = -1)$ from a histogram of the features of the negative examples, and $p(y = 1)$ by counting positive versus negative examples. Then,

$$p(y = 1|\mathbf{x}) = \frac{p(\mathbf{x}|y = 1)p(y = 1)}{p(\mathbf{x}|y = 1)p(y = 1) + p(\mathbf{x}|y = -1)(1 - p(y = 1))},$$

and we can plot an ROC.

Models like this become impractical in high dimensions because the number of boxes required goes up as a power of the dimension. We can dodge this phenomenon by assuming that features are independent conditioned on the class. Although this appears to be an aggressive oversimplification—it is known by the pejorative name **naive Bayes**—it is often very well-behaved, and is competitive for many problems. In particular, we assume that

$$p(\mathbf{x}|y = 1) = p([x_0, x_1, \ldots, x_n]|y = 1) = p(x_0|y = 1)p(x_1|y = 1)\ldots p(x_n|y = 1).$$

Now each of these conditional distributions is low-dimensional, and so easy to model (either a normal distribution or a histogram are good candidates).

1.2.4 Example: The Linear Support Vector Machine

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

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

We seek a rule that predicts the sign of y for any point $\mathbf{x}$; 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.

Support Vector Machines for Linearly Separable Datasets In a linearly separable dataset, there is some choice of $\mathbf{w}$ and b (which represent a hyperplane) such that

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

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 $\mathbf{w}$ and b. These constraints express the constraint 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 farthest 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 maximizes the minimum distance from example points to the hyperplane (Figure 1.7).

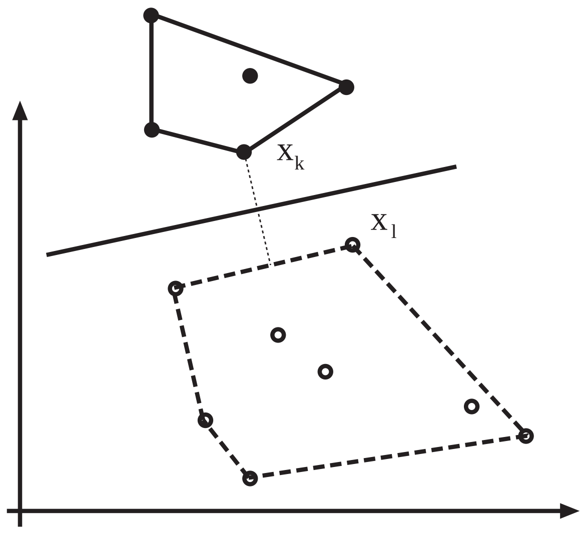

FIGURE 1.7: The hyperplane constructed by a support vector classifier for a plane dataset. The filled circles are data points corresponding to one class, and the empty circles are data points corresponding to the other. We have drawn in the convex hull of each dataset. The most conservative choice of hyperplane is one that maximizes 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 that are associated with a minimum distance between the hulls. We use these points to find the hyperplane in the text.

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

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

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

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

$$dist(\mathbf{x}_k, \text{hyperplane}) + dist(\mathbf{x}_l, \text{hyperplane})$$

which is

$$(\frac{\mathbf{w}}{|\mathbf{w}|} \cdot \mathbf{x}_k + \frac{b}{|\mathbf{w}|}) - (\frac{\mathbf{w}}{|\mathbf{w}|} \cdot \mathbf{x}_1 + \frac{b}{|\mathbf{w}|}),$$

becomes

$$\frac{\mathbf{w}}{|\mathbf{w}|} \cdot (\mathbf{x}_1 - \mathbf{x}_2) = \frac{2}{|\mathbf{w}|}.$$

This means that maximizing the distance is the same as *minimizing* $(1/2)\mathbf{w} \cdot \mathbf{w}$. We now have the constrained minimization problem:

$$\begin{array}{ll} \text{minimize} & (1/2)\mathbf{w} \cdot \mathbf{w} \\ & \\ \text{subject to} & y_i\,(\mathbf{w} \cdot \mathbf{x}_i + b) \geq 1, \end{array}$$

where there is one constraint for each data point.

Support Vector Machines for Non-separable Data

In many cases, a separating hyperplane does 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

$$y_i\,(\mathbf{w} \cdot \mathbf{x}_1 + b) \geq 1 - \xi_i,$$

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

$$\begin{array}{ll} \text{minimize} & \frac{1}{2}\mathbf{w} \cdot \mathbf{w} + C\sum_{i=1}^{N} \xi_i \\ & \\ \text{subject to} & y_i\,(\mathbf{w} \cdot \mathbf{x}_1 + b) \geq 1 - \xi_i \\ \text{and} & \xi_i \geq 0. \end{array}$$

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

The Hinge Loss Support vector machines fit into the recipe, given in Section 1.2.2, of minimizing regularized test loss. The **hinge loss** compares the known value at an example with the response of the SVM at that example. Write $y_i^{(k)}$ for the known value and $y_i^{(p)}$ for the response; then, the hinge loss for that example is

$$L_h(y_i^{(k)}, y_i^{(p)}) = \max(0, 1 - y_i^{(k)} y_i^{(p)}).$$

This loss is always non-negative (Figure 1.6). For the moment, assume $y_i^{(k)} = 1$; then, any prediction by the classifier with value greater than one will incur no loss, and any smaller prediction will incur a cost that is linear in the prediction value (Figure 1.6). This means that minimizing the loss will encourage the classifier to (a) make strong positive (or negative) predictions for positive (or negative) examples and (b) for examples it gets wrong, make the most positive (negative) prediction that it can.

Support vector machines minimize the regularized hinge loss. We can see this by rewriting the constraints, to get $\xi_i \geq 1 - y_i \left(\mathbf{w} \cdot \mathbf{x}_1 + b \right)$. Now ξ_i will take the smallest value that it can, and $\xi_i \geq 0$, so

$$\xi_i = \max \left(0, 1 - y_i \left(\mathbf{w} \cdot \mathbf{x}_1 + b \right) \right) = L_h(y_i, \mathbf{w} \cdot \mathbf{x}_i + b).$$

In turn, solving the SVM above is equivalent to solving the unconstrained problem

$$\text{minimize} \;\; \text{Loss} + \;\; \text{Regularizer} = \sum_{i=1}^{N} L_h(y_i, \mathbf{w} \cdot \mathbf{x}_i + b) + \frac{1}{2C} \mathbf{w} \cdot \mathbf{w}.$$

Solving this problem requires care because it is not differentiable (the max term in the hinge loss is the problem). However, rewriting an SVM in this way is helpful, because it exposes what the SVM does.

1.3 PRACTICAL METHODS FOR BUILDING CLASSIFIERS

We have described several apparently very different classifiers here. But which classifier should one use for a particular application? Generally, this should be dealt with as a practical rather than a conceptual question: that is, one tries several, and uses the one that works best. With all that said, experience suggests that the first thing to try for most problems is a linear SVM or logistic regression, which tends to be much the same thing. Nearest neighbor strategies are always useful, and are consistently competitive with other approaches when there is lots of training data and one has some idea of appropriate relative scaling of the features. The main difficulty with nearest neighbors is actually finding the nearest neighbors of a query. Approximate methods are now very good, and are reviewed in Section **??**. The attraction of these methods is that it is relatively easy to build multi-class classifiers, and to add new classes to a system of classifiers.

The loss function one uses is supposed to be dictated by the natural logic of the underlying problem. This is all very well, but in practice we often do not know what a good loss function is, particularly in multi-class cases. The 0-1 loss is almost universally used, but this loss can impose severe (and, worse, uninformative) penalties in multi-class cases. For example, is labeling a cat with the label "dog" really as bad as labeling it with the label "motorcycle"? The difficulty here is we do not have a good, ready-made loss function that encodes what we really want to do for some classification problems. We explore this point further in Section **??**.

1.3.1 Manipulating Training Data to Improve Performance

Generally, more training data leads to a better classifier. However, training classifiers with large datasets can be difficult, and it can be hard to get enough training data. Typically, only a relatively small number of example items are really important in determining the behavior of a classifier (we see this phenomenon in greater detail in Section 1.2.4). 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

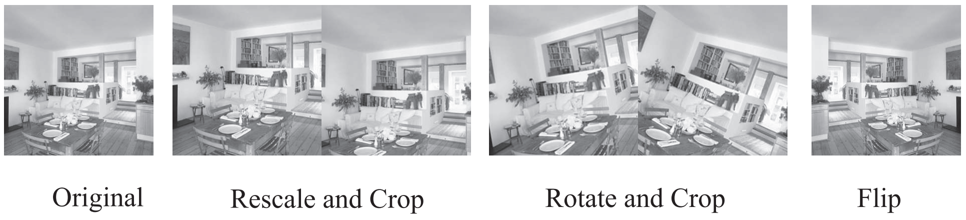

FIGURE 1.8: *A* single positive example can be used to generate numerous positive examples by slight rescaling and cropping, small rotations and crops, or flipping. These transformations can be combined, too. For most applications, these positive examples are informative, because objects usually are not framed and scaled precisely in images. In effect, these examples inform the classifier that, for example, the stove could be slightly more or slightly less to the right of the image or even to the left. *Jake Fitzjones © Dorling Kindersley, used with permission.*

decision boundary most significantly. We need a large dataset to ensure that these cases are present.

There are two useful tricks that help. First, for many or most cases in computer vision, we can expand the set of training examples with quite simple tricks. For concreteness, imagine we are training a classifier to recognize pictures of kitchens. The first step is to collect many pictures of kitchens. But we aren't guaranteed that an image of a kitchen will appear at a fixed size, or at a fixed rotation, or with a fixed crop. Usually we would resize the images to a fixed size using a uniform scaling, cropping as necessary. However, we could vary the scaling slightly, vary the cropping slightly, or vary the rotation of the image slightly (Figure 1.8). This means that each picture of a kitchen can generate a large number of positive examples. It is usually less helpful to do this with negative examples, because it is usually easy to get a large number of negative examples. A second useful trick can avoid 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. We may repeat this several times, and in the final stages, we may use the classifier to seek false positives. For example, we might collect pictures from the Web, classify them, and then look at the positives for errors. This strategy is sometimes called **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).

There is an extremely important variant of this approach called **hard negative mining**. This applies to situations where we have a moderate supply of positive examples, but an immense number of negative examples. Such situations occur commonly when we use classifiers to detect objects (Section **??**). The general procedure is to test every image window to tell whether it contains, say, a face. There are a lot of image windows, and it is quite easy to obtain a lot of images that are certain not to contain a face. In this case we can't use all the negative examples in training, but we need to search for negative examples that are most likely to improve the classifier's performance. We can do so by selecting a set of negative examples, training with these, and then searching the rest of the negative examples to find ones that generate false positives—these are hard negatives. We can iterate the procedure of training and searching for hard negatives; typically, we expand the pool of negative examples at each iteration.

1.3.2 Building Multi-Class Classifiers Out of Binary Classifiers

There are two standard methods to build multi-class classifiers out of binary classifiers. In the **all-vs-all** approach, we train a binary classifier for each pair of classes. To classify an example, we present it to each of these classifiers. Each classifier decides which of two classes the example belongs to, then records a vote for that class. The example gets the class label with the most votes. This approach is simple, but scales very badly with the number of classes.

In the **one-vs-all** approach, we build a binary classifier for each class. This classifier must distinguish its class from all the other classes. We then take the class with the largest classifier score. One possible concern with this method is that training algorithms usually do not compel classifiers to be good at ranking examples. We train classifiers so that they give positive scores for positive examples, and negative scores for negative examples, but we do nothing explicit to ensure that a more positive score means the example is more like the positive class. Another important concern is that the classifier scores must be calibrated to one another, so that when one classifier gives a larger positive score than another, we can be sure that the first classifier is more certain than the second. Some classifiers, such as logistic regression, report posterior probabilities, which require no calibration. Others, such as the SVM, report numbers with no obvious semantics and need to be calibrated. The usual method to calibrate these numbers is an algorithm due to **?**, which uses logistic regression to fit a simple probability model to SVM outputs. One-vs-all methods tend to be reliable and effective even when applied to uncalibrated classifier outputs, most likely because training algorithms do tend to encourage classifiers to rank examples correctly.

Neither strategy is particularly attractive when the number of classes is large, because the number of classifiers we must train scales poorly (linearly in one case, quadratically in the other) with the number of classes. If we were to allocate each class a distinct binary vector, we would need only $\log N$ bits in the vector for N classes. We could then train one classifier for each bit, and we should be able to classify into N classes with only $\log N$ classifiers. This strategy tends to founder on questions of which class should get which bit string, because this choice has significant effects on the ease of training the classifiers. Nonetheless, it gives an argument that suggests that we should not need as many as N classifiers to tell N classes apart. This question is becoming important because the number of object categories that modern methods can deal with is growing quickly. For example, one now sees methods that do 10,000-class classification for vision objects (**?**). The difference between training 10,000 SVMs and training 14 is very significant, and we can expect considerable research on this matter.

1.3.3 Software for SVM's

We obtain a support vector machine by solving one of the constrained optimization problems given above. These problems have quite special structure, and one would usually use one of the many packages available on the web for SVMs to solve them.

LIBSVM (which can be found using Google, or at `http://www.csie.ntu.edu.tw/~cjlin/libsvm/`) is a dual solver that is now widely used; it searches for nonzero Lagrange multipliers using a clever procedure known as SMO (sequential minimal optimization). A good primal solver is PEGASOS; source code can be found using Google, or at `http://www.cs.huji.ac.il/~shais/code/index.html`.

SVMLight (Google, or `http://svmlight.joachims.org/`) is a comprehensive SVM package with numerous features. It can produce sophisticated estimates of the error rate, learn to rank as well as to classify, and copes with hundreds of thousands of examples. Andrea Vedaldi, Manik Varma, Varun Gulshan, and Andrew Zisserman publish code for a multiple kernel learning-based image classifier at `http://www.robots.ox.ac.uk/~vgg/software/MKL/`. Manik Varma publishes code for general multiple-kernel learning at `http://research.microsoft.com/en-us/um/people/manik/code/GMKL/download.html`, and for multiple-kernel learning using SMO at `http://research.microsoft.com/en-us/um/people/manik/code/SMO-MKL/download.html`. Peter Gehler and Sebastian Nowozin publish code for their recent multiple-kernel learning method at `http://www.vision.ee.ethz.ch/~pgehler/projects/iccv09/index.html`.

1.4 BASIC IDEAS FOR NUMERICAL MINIMIZATION

Assume we have a function $g(\mathbf{x})$, and we wish to obtain a value of $\mathbf{x}$ that achieves the minimum for that function. Sometimes we can solve this problem in closed form, but more usually we need a numerical method. Implementing these numerical methods is a specialized business; it is usual to use general optimization codes. This section is intended to sketch how such codes work, so you can read manual pages, etc. more effectively. Personally, I am a happy user of Matlab's `fminunc`, although the many different settings take some getting used to.

Typical codes take a description of the objective function (typically, the name of a function), a start point for the search, and a collection of parameters. All codes take an estimate $\mathbf{x}^{(i)}$, update it to $\mathbf{x}^{(i+1)}$, then check to see whether the result is a minimum. This process is started from the start point. The update is usually obtained by computing a direction $\mathbf{p}^{(i)}$ such that for small values of h, $g(\mathbf{x}^{(i)}+h\mathbf{p}^{(i)})$ is smaller than $g(\mathbf{x}^{(i)}$. Such a direction is known as a **descent direction**.

Assume we have a descent direction. We must now choose how far to travel along that direction. We can see $g(\mathbf{x}^{(i)} + h\mathbf{p}^{(i)})$ as a function of h. Write this function as $\phi(h)$. We start at $h = 0$ (which is the original value $\mathbf{x}^{(i)}$, so $\phi(0) = g(\mathbf{x}^{(i)})$, and move in the direction of increasing h to find a small value of $\phi(h)$ that is less than $\phi(0)$. The descent direction was chosen so that for small $h > 0$, $\phi(h) < \phi(0)$; one way to tell we are at a minimum is we cannot choose a descent direction. Searching for a good value of h is known as **line search**. Typically, this search involves a sequence of estimated values of h, which we write h_i. One algorithm is to start with (say) $h_0 = 1$; if $\phi(h_i)$ is not small enough (and there are other tests we may need to apply — this is a summary!), we compute $h_{(i+1)} = 1/2h_i$.

This stops when some h_i passes a test, or when it is so small that the step is pointless.

There are two main methods to choose a descent direction. The first, known as **gradient descent**, uses the negative gradient of the function. We write $\mathbf{p}^{(i)} = -\nabla g(\mathbf{x}^{(i)})$. This works (as long as g is differentiable, and quite often when it isn't) because g must go down for at least small steps in this direction. There are two ways to evaluate a gradient. You can require that the software estimate a numerical derivative for you, which usually slows things down somewhat, or you can supply a gradient value. Usually this gradient value must be computed by the same function that computes the objective function value.

One tip: in my experience, about 99% of problems with numerical optimization codes occur because the user didn't check that the gradient their function computed is right. Most codes will compute a numerical gradient for you, then check that against your gradient; if they're sufficiently different, the code will complain. You don't want to do this at runtime, because it slows things up, but it's an excellent idea to check.

The other method to choose a descent direction is **Newton's method**. The point we seek has the property that $\nabla g(\mathbf{x}) = \mathbf{0}$. Now assume that our estimate $\mathbf{x}^{(i)}$ is rather close to the right point. We can then write a Taylor series. Write $\mathcal{H}(\mathbf{x}^{(i)})$ for the matrix of second derivatives of g, evaluated at $\mathbf{x}^{(i)}$. This is usually called the **Hessian**. The Taylor series for $g(\mathbf{x}^{(i)} + \mathbf{p})$ is then

$$g(\mathbf{x}^{(i)} + \mathbf{p}) \approx g(\mathbf{x}^{(i)}) + \nabla g(\mathbf{x}^{(i)})\mathbf{p} + \frac{1}{2}\mathbf{p}^T\mathcal{H}\mathbf{p}$$

and so we have

$$\nabla g(\mathbf{x}^{(i)} + \mathbf{p}) \approx \nabla g(\mathbf{x}^{(i)}) + \mathcal{H}\mathbf{p}.$$

Now we want $\mathbf{x}^{(i+1)} = \mathbf{x}^{(i)} + \mathbf{p}$ to be the right answer. So we want

$$\nabla g(\mathbf{x}^{(i)}) + \mathcal{H}\mathbf{p} = \mathbf{0}$$

or

$$\mathcal{H}\mathbf{p} = -\nabla g(\mathbf{x}^{(i)}).$$

In the ideal case, this update places $\mathbf{x}^{(i+1)}$ on the solution. This doesn't happen all that often (though it can happen in important cases), and we may need to iterate. When we are close to the right point, Newton's method can converge extremely fast. However, it can behave rather badly when we are far away. Typical codes estimate a $\mathbf{p}$ using a modified form of Newton's method, then apply the line search above. To use Newton's method, you must either allow the code to compute a numerical Hessian (which can be rather slow), or provide the Hessian yourself.

Another tip: in my experience, about 99% of problems with codes that use Newton's method occur because the user didn't check that both the gradient *and* the Hessian their function computed is right. Again, most codes will check a numerical estimate against your value if you want. It's an excellent idea to check.

1.5 NOTES

We warn readers that a search over classifiers is not a particularly effective way to solve problems; instead, look to improved feature constructions. However, many application problems have special properties, and so there is an enormous number of different methods to build classifiers. We have described methods that reliably give good results. Classification is now a standard topic, with a variety of important textbooks. Accounts of most mainstream classifiers can be found in major recent texts. We recommend **?**, **?**, **?**, and **?**. An important theoretical account of when classification methods can be expected to work or to fail is in **?**.

Listing 1.1: Matlab code used to train a histogram based classifier for the pizza data

```
[num, txt, raw]=xlsread('~/Current/Courses/Probcourse/SomeData/DataSets/cleanpiz
ndat=size(num, 1);
pdiams=num(:, 5);
labels=zeros(ndat, 1);
for i=1:ndat
    if strcmp(txt(i, 2), 'Dominos')==1
        labels(i)=1;
    else
        labels(i)=-1;
    end
end
% now we do a test-train split
ntrain=floor(0.8*250);
permutev=randperm(ndat);
%% how many boxes in the histogram
nboxes=20;
hn=(1-1e-9)*min(pdiams);
hx=(1+1e-9)*max(pdiams);
hs=(hx-hn)/nboxes;
histd=zeros(nboxes, 1);
histe=zeros(nboxes, 1);
%%
for i=1:ntrain
    pd=pdiams(permutev(i));
    lv=labels(permutev(i));
    hptr=floor((pd-hn)/hs+1);
    if lv>0
        histd(hptr)=histd(hptr)+1;
    else
        histe(hptr)=histe(hptr)+1;
    end
end
histe=histe+1e-9;
histd=histd+1e-9;
pxcd=histd/sum(histd);
pxce=histe/sum(histe);
```

Listing 1.2: Matlab code used to test a histogram based classifier for the pizza data

```
%%
% it's trained.  Now test for various thresholds
thresholds=[0, 0.2, 0.4, 0.6, 0.8, 1, 1.2, 1.4, 1.6, 1.8, 2];
fps=zeros(11, 1);
fns=zeros(11, 1);
tps=zeros(11, 1);
tns=zeros(11, 1);
for th=1:11
    thresh=thresholds(th);
    for i=ntrain+1:ndat
        pd=pdiams(permutev(i));
        lv=labels(permutev(i));
        hptr=floor((pd-hn)/hs+1);
        testv=pxcd(hptr)/pxce(hptr);
        if testv>thresh
            predlabel=1;
        else
            predlabel=-1;
        end
        if predlabel>0&&lv>0
            tps(th)=tps(th)+1;
        elseif predlabel>0&&lv<0
            fps(th)=fps(th)+1;
        elseif predlabel<0&&lv>0
            fns(th)=fns(th)+1;
        else
            tns(th)=tns(th)+1;
        end
    end
end
tpr=tps/(ndat-ntrain);
fpr=fps/(ndat-ntrain);
figure(1); clf;
plot(fpr, tpr, 'r');
axis([0 1 0 1])
ylabel('True positive rate or detect rate');
xlabel('False positive rate');
hold on
plot([0, 1], [0, 1], '--r');
cd('~/Current/Courses/Probcourse/Classification/Figures');
print -depsc2 pizzaroc.eps
```