Learning to Classify

11

A **classifier** is a procedure that accepts a set of features and produces a class label for them. Classifiers are immensely useful, and find wide application, because many problems are naturally classification 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, credit card companies must decide whether a transaction is good or fraudulent.

All these examples are two class classifiers, but in many cases it is natural to have more classes. You can think of sorting laundry as applying a multi-class classifier. You can think of doctors as complex multi-class classifiers: a doctor accepts a set of features (your complaints, answers to questions, and so on) and then produces a response which we can describe as a class. The grading procedure for any class is a multi-class classifier: it accepts a set of features—performance in tests, homeworks, and so on—and produces a class label (the letter grade).

A classifier is usually trained by obtaining a set of labelled training examples and then searching for a classifier that optimizes some cost function which is evaluated on the training data. What makes training classifiers interesting is that performance on training data doesn't really matter. What matters is performance on run-time data, which may be extremely hard to evaluate because one often does not know the correct answer for that data. For example, we wish to classify credit-card transactions as safe or fraudulent. We could obtain a set of transactions with true labels, and train with those. But what we care about is new transactions, where it would be very difficult to know whether the classifier's answers are right. To be able to do anything at all, the set of labelled examples must be representative of future examples in some strong way. We will always assume that the labelled examples are an IID sample from the set of all possible examples, though we never use the assumption explicitly.

Definition 11.1 (Classifier) A classifier is a procedure that accepts a set of features and produces a label. Classifiers are trained on labelled examples, but the goal is to get a classifier that performs well on data which is not seen at the time of training. Training a classifier requires labelled data that is representative of future data.

11.1 Classification: The Big Ideas

We will write the training dataset $(\mathbf{x}_i, y_i)$. For the i'th example, $\mathbf{x}_i$ represents the values taken by a collection of features. In the simplest case, $\mathbf{x}_i$ would be a vector of real numbers. In some cases, $\mathbf{x}_i$ could contain categorical data or even unknown values. Although $\mathbf{x}_i$ isn't guaranteed to be a vector, it's usually referred to as a **feature vector**. The y_i are labels giving the type of the object that generated the example. We must use these labelled examples to come up with a classifier.

D. Forsyth, *Probability and Statistics for Computer Science*,
https://doi.org/10.1007/978-3-319-64410-3_11

11.1.1 The Error Rate, and Other Summaries of Performance

We can summarize the performance of any particular classifier using the **error** or **total error rate** (the percentage of classification attempts that gave the wrong answer) and the **accuracy** (the percentage of classification attempts that give the right answer). For most practical cases, even the best choice of classifier will make mistakes. For example, an alien tries to classify humans into male and female, using only height as a feature. Whatever the alien's classifier does with that feature, it will make mistakes. This is because the classifier must choose, for each value of height, whether to label the humans with that height male or female. But for the vast majority of heights, there are some males and some females with that height, and so the alien's classifier must make some mistakes.

As the example suggests, a particular feature vector $\mathbf{x}$ may appear with different labels (so the alien will see six foot males and six foot females, quite possibly in the training dataset and certainly in future data). Labels appear with some probability conditioned on the observations, $P(y|\mathbf{x})$. If there are parts of the feature space where $P(\mathbf{x})$ is relatively large (so we expect to see observations of that form) *and* where $P(y|\mathbf{x})$ has relatively large values for more than one label, even the best possible classifier will have a high error rate. If we knew $P(y|\mathbf{x})$ (which is seldom the case), we could identify the classifier with the smallest error rate and compute its error rate. The minimum expected error rate obtained with the best possible classifier applied to a particular problem is known as the **Bayes risk** for that problem. In most cases, it is hard to know what the Bayes risk is, because to compute it requires knowing $P(y|\mathbf{x})$, which isn't usually known.

The error rate of a classifier is not that meaningful on its own, because we don't usually know the Bayes risk for a problem. It is more helpful to compare a particular classifier with some natural alternatives, sometimes called **baselines**. The choice of baseline for a particular problem is almost always a matter of application logic. The simplest general baseline is a know-nothing strategy. Imagine classifying the data without using the feature vector at all—how well does this strategy do? If each of the C classes occurs with the same frequency, then it's enough to label the data by choosing a label uniformly and at random, and the error rate for this strategy is $1 - 1/C$. If one class is more common than the others, the lowest error rate is obtained by labelling everything with that class. This comparison is often known as **comparing to chance**.

It is very common to deal with data where there are only two labels. You should keep in mind this means the highest possible error rate is 50%—if you have a classifier with a higher error rate, you can improve it by switching the outputs. If one class is much more common than the other, training becomes more complicated because the best strategy—labelling everything with the common class—becomes hard to beat.

11.1.2 More Detailed Evaluation

The error rate is a fairly crude summary of the classifier's behavior. For a two-class classifier and a 0–1 loss function, one can report the **false positive rate** (the percentage of negative test data that was classified positive) and the **false negative rate** (the percentage of positive test data that was classified negative). Note that it is important to provide both, because a classifier with a low false positive rate tends to have a high false negative rate, and vice versa. As a result, you should be suspicious of reports that give one number but not the other. Alternative numbers that are reported sometimes include the **sensitivity** (the percentage of true positives that are classified positive) and the **specificity** (the percentage of true negatives that are classified negative).

The false positive and false negative rates of a two-class classifier can be generalized to evaluate a multi-class classifier, yielding the **class confusion matrix**. This is a table of cells, where the i, j'th cell contains the count of cases where the true label was i and the predicted label was j (some people show the fraction of cases rather than the count). Table 11.1 gives an example. This is a class confusion matrix from a classifier built on a dataset where one tries to predict the degree of heart disease from a collection of physiological and physical measurements. There are five classes ($0 \ldots 4$). The i, j'th cell of the table shows the number of data points of true class i that were classified to have class j. As I find it hard to recall whether rows or columns represent true or predicted classes, I have marked this on the table. For each row, there is a **class error rate**, which is the percentage of data points of that class that were misclassified. The first thing to look at in a table like this is the diagonal; if the largest values appear there, then the classifier is working well. This clearly isn't what is happening for Table 11.1. Instead, you can see that the method is very good at telling whether a data point is in class 0 or not (the class error rate is rather small), but cannot distinguish between the other classes. This is a strong hint that the data can't be used to draw the distinctions that we want. It might be a lot better to work with a different set of classes.

Table 11.1 The class confusion matrix for a multiclass classifier

True	Predict					
	0	1	2	3	4	Class error
0	151	7	2	3	1	7.9%
1	32	5	9	9	0	91%
2	10	9	7	9	1	81%
3	6	13	9	5	2	86%
4	2	3	2	6	0	100%

This is a table of cells, where the i, j'th cell contains the count of cases where the true label was i and the predicted label was j (some people show the fraction of cases rather than the count). Further details about the dataset and this example appear in Worked example 11.20

11.1.3 Overfitting and Cross-Validation

Choosing and evaluating a classifier takes some care. The goal is to get a classifier that works well on future data *for which we might never know the true label*, using a training set of labelled examples. This isn't necessarily easy. For example, think about 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; otherwise, it chooses randomly between the classes.

The **training error** of a classifier is the error rate on examples used to train the classifier. In contrast, the **test error** is error on examples not used to train the classifier. Classifiers that have small training error might not have small test error, because the classification procedure is chosen to do well on the training data. This effect is 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 must generalize from the training data to the test data. The effect occurs because the classifier has been chosen to perform well *on the training dataset*. An efficient training procedure is quite likely to find special properties of the training dataset that aren't representative of the test dataset, because the training dataset is not the same as the test dataset. The training dataset is typically a sample of all the data one might like to have classified, and so is quite likely a lot smaller than the test dataset. Because it is a sample, it may have quirks that don't appear in the test dataset. One consequence of overfitting is that classifiers should always be evaluated on data that was not used in training.

Now assume that we want to estimate the error rate of the classifier on test data. We cannot estimate the error rate of the classifier using data that was used to train the classifier, because the classifier has been trained to do well on that data, which will mean our error rate estimate will be too low. An alternative is to separate out some training data to form a **validation set** (confusingly, this is sometimes called a test set), then train the classifier on the rest of the data, and evaluate on the validation set. The error estimate on the validation set is the value of a random variable, because the validation set is a sample of all possible data you might classify. But this error estimate is **unbiased**, meaning that the expected value of the error estimate is the true value of the error. You can see this by thinking about the error estimate as a sample mean and applying the ideas of Chap. 6.

However, separating out some training data presents the difficulty that the classifier will not be the best possible, because we 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. Each different split is usually called a **fold**. This procedure yields an estimate of the likely future performance of a classifier, at the expense of substantial computation. A common form of this algorithm uses a single data item to form a validation set. This is known as **leave-one-out cross-validation**.

Remember this: *Classifiers usually perform better on training data than on test data, because the classifier was chosen to do well on the training data. This effect is known as overfitting. To get an accurate estimate of future performance, classifiers should always be evaluated on data that was not used in training.*

11.2 Classifying with Nearest Neighbors

Assume we have a labelled dataset consisting of N pairs $(\mathbf{x}_i, y_i)$. Here $\mathbf{x}_i$ is the i'th feature vector, and y_i is the i'th class label. We wish to predict the label y for any new example $\mathbf{x}$; this is often known as a query example or query. Here is a really effective strategy: Find the labelled example $\mathbf{x}_c$ that is closest to $\mathbf{x}$, and report the class of that example.

How well can we expect this strategy to work? A precise analysis would take us way out of our way, but simple reasoning is informative. Assume there are two classes, 1 and -1 (the reasoning will work for more, but the description is slightly more involved). We expect that, if $\mathbf{u}$ and $\mathbf{v}$ are sufficiently close, then $p(y|\mathbf{u})$ is similar to $p(y|\mathbf{v})$. This means that if a labelled example $\mathbf{x}_i$ is close to $\mathbf{x}$, then $p(y|\mathbf{x})$ is similar to $p(y|\mathbf{x}_i)$. Furthermore, we expect that queries are "like" the labelled dataset, in the sense that points that are common (resp. rare) in the labelled data will appear often (resp. seldom) in the queries.

Now imagine the query comes from a location where $p(y = 1|\mathbf{x})$ is large. The closest labelled example $\mathbf{x}_c$ should be nearby (because queries are "like" the labelled data) and should be labelled with 1 (because nearby examples have similar label probabilities). So the method should produce the right answer with high probability.

Alternatively, imagine the query comes from a location where $p(y = 1|\mathbf{x})$ is about the same as $p(y = -1|\mathbf{x})$. The closest labelled example $\mathbf{x}_c$ should be nearby (because queries are "like" the labelled data). But think about a set of examples that are about as close. The labels in this set should vary significantly (because $p(y = 1|\mathbf{x})$ is about the same as $p(y = -1|\mathbf{x})$. This means that, if the query is labelled 1 (resp. -1), a small change in the query will cause it to be labelled -1 (resp. 1). In these regions the classifier will tend to make mistakes more often, as it should. Using a great deal more of this kind of reasoning, nearest neighbors can be shown to produce an error that is no worse than twice the best error rate, if the method has enough examples. There is no prospect of seeing enough examples in practice for this result to apply.

One important generalization is to find the k nearest neighbors, then choose a label from those. 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). In practice, one seldom uses more than three nearest neighbors.

11.2.1 Practical Considerations for Nearest Neighbors

One practical difficulty in using nearest neighbor classifiers is you need a lot of labelled examples for the method to work. For some problems, this means you can't use the method. A second practical difficulty is you need to use a sensible 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? 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. Another possibility is to transform the features so that the covariance matrix is the identity (this is sometimes known as **whitening**; the method follows from the ideas of Chap. 10). This can be hard to do if the dimension is so large that the covariance matrix is hard to estimate.

A third practical difficulty is you need to be able to find the nearest neighbors for your query point. This is surprisingly difficult to do faster than simply checking the distance to each training example separately. If your intuition tells you to use a tree and the difficulty will go away, your intuition isn't right. It turns out that nearest neighbors in high dimensions is one of those problems that is a lot harder than it seems, because high dimensional spaces are quite hard to reason about informally. There's a long history of methods that appear to be efficient but, once carefully investigated, turn out to be bad.

Fortunately, it is usually enough to use an **approximate nearest neighbor**. This is an example that is, with high probability, almost as close to the query point as the nearest neighbor is. Obtaining an approximate nearest neighbor is very much easier than obtaining a nearest neighbor. We can't go into the details here, but there are several distinct methods for finding approximate nearest neighbors. Each involves a series of tuning constants and so on, and, on different datasets, different methods and different choices of tuning constant produce the best results. If you want to use a nearest neighbor classifier on a lot of run-time data, it is usually worth a careful search over methods and tuning constants to find an algorithm that yields a very fast response to a query. It is known how to do this search, and there is excellent software available (FLANN, http://www.cs.ubc.ca/~mariusm/index.php/FLANN/FLANN, by Marius Muja and David G. Lowe).

It is straightforward to use cross-validation to estimate the error rate of a nearest neighbor classifier. Split the labelled training data into two pieces, a (typically large) training set and a (typically small) validation set. Now take each element of the validation set and label it with the label of the closest element of the training set. Compute the fraction of these attempts that produce an error (the true label and predicted labels differ). Now repeat this for a different split, and average the errors over splits. With care, the code you'll write is shorter than this description.

Worked example 11.1 (Classifying Usi- ng Nearest Neighbors) Build a nearest neighbor classifier to classify the MNIST digit data. This dataset is very widely used to check simple methods. It was originally constructed by Yann Lecun, Corinna Cortes, and Christopher J.C. Burges. It has been extensively studied. You can find this dataset in several places. The original dataset is at http://yann.lecun.com/exdb/mnist/. The version I used was used for a Kaggle competition (so I didn't have to decompress Lecun's original format). I found it at http://www.kaggle.com/c/digit-recognizer.

Solution I used R for this problem. As you'd expect, R has nearest neighbor code that seems quite good (I haven't had any real problems with it, at least). There isn't really all that much to say about the code. I used the R FNN package. I trained on 1000 of the 42,000 examples in the Kaggle version, and I tested on the next 200 examples. For this (rather small) case, I found the following class confusion matrix:

True	Predict									
	0	1	2	3	4	5	6	7	8	9
0	12	0	0	0	0	0	0	0	0	0
1	0	20	4	1	0	1	0	2	2	1
2	0	0	20	1	0	0	0	0	0	0
3	0	0	0	12	0	0	0	0	4	0
4	0	0	0	0	18	0	0	0	1	1
5	0	0	0	0	0	19	0	0	1	0
6	1	0	0	0	0	0	18	0	0	0
7	0	0	1	0	0	0	0	19	0	2
8	0	0	1	0	0	0	0	0	16	0
9	0	0	0	2	3	1	0	1	1	14

There are no class error rates here, because I couldn't recall the magic line of R to get them. However, you can see the classifier works rather well for this case. MNIST is comprehensively explored in the exercises.

Remember this: *Nearest neighbors has good properties. With enough training data and a low enough dimension, the error rate is guaranteed to be no more than twice the best error rate. The method is wonderfully flexible about the labels the classifier predicts. Nothing changes when you go from a two-class classifier to a multi-class classifier.*

There are important difficulties. You need a large training dataset. If you don't have a reliable measure of how far apart two things are, you shouldn't be doing nearest neighbors. And you need to be able to query a large dataset of examples to find the nearest neighbor of a point.

11.3 Classifying with Naive Bayes

One straightforward source of a classifier is a probability model. For the moment, assume we know $p(y|\mathbf{x})$ for our data. Assume also that all errors in classification are equally important. Then the following rule produces smallest possible expected classification error rate:

For a test example $\mathbf{x}$, report the class y that has the highest value of $(p(y|\mathbf{x}))$. If the largest value is achieved by more than one class, choose randomly from that set of classes.

Usually, we do not have $p(y|\mathbf{x})$. If we have $p(\mathbf{x}|y)$ (often called either a **likelihood** or **class conditional probability**, compare Sect. 9.1), and $p(y)$ (often called a **prior**, compare Sect. 9.2) then we can use Bayes' rule to form

$$p(y|\mathbf{x}) = \frac{p(\mathbf{x}|y)p(y)}{p(\mathbf{x})}$$

(the **posterior**, compare Sect. 9.2). This isn't much help in this form, but write $x^{(j)}$ for the j'th component of $\mathbf{x}$. Now *assume* that features are conditionally independent conditioned on the class of the data item. Our assumption is

$$p(\mathbf{x}|y) = \prod_j p(x^{(j)}|y).$$

It is very seldom the case that this assumption is true, but it turns out to be fruitful to pretend that it is. This assumption means that

$$\begin{aligned} p(y|\mathbf{x}) &= \frac{p(\mathbf{x}|y)p(y)}{p(\mathbf{x})} \\ &= \frac{\left(\prod_j p(x^{(j)}|y)\right) p(y)}{p(\mathbf{x})} \\ &\propto \left(\prod_j p(x^{(j)}|y)\right) p(y). \end{aligned}$$

Now to make a decision, we need to choose the class that has the largest value of $p(y|\mathbf{x})$. In turn, this means we need only know the posterior values up to scale at $\mathbf{x}$, so we don't need to estimate $p(\mathbf{x})$. In the case of where all errors have the same cost, this yields the rule

$$\text{choose } y \text{ such that } \left[\left(\prod_j p(x^{(j)}|y)\right) p(y)\right] \text{ is largest.}$$

This rule suffers from a practical problem. You can't actually multiply a large number of probabilities and expect to get an answer that a floating point system thinks is different from zero. Instead, you should add the log probabilities. Notice that the logarithm function has one nice property: it is monotonic, meaning that $a > b$ is equivalent to $\log a > \log b$. This means the following, more practical, rule is equivalent:

$$\text{choose } y \text{ such that } \left[\left(\sum_j \log p(x^{(j)}|y)\right) + \log p(y)\right] \text{ is largest.}$$

To use this rule, we need models for $p(y)$ and for $p(x^{(j)}|y)$ for each j. The usual way to find a model of $p(y)$ is to count the number of training examples in each class, then divide by the number of classes.

It turns out that simple parametric models work really well for $p(x^{(j)}|y)$. For example, one could use a normal distribution for each $x^{(j)}$ in turn, for each possible value of y, using the training data. The parameters of this normal distribution are chosen using maximum likelihood. The logic of the measurements might suggest other distributions, too. If one of the $x^{(j)}$'s was a count, we might fit a Poisson distribution (again, using maximum likelihood). If it was a 0–1 variable, we might fit a Bernoulli distribution. If it was a discrete variable, then we might use a multinomial model. Even if the $x^{(j)}$ is continuous, we can use a multinomial model by quantizing to some fixed set of values; this can be quite effective.

A naive bayes classifier that has poorly fitting models for each feature could classify data very well. This (reliably confusing property) occurs because classification doesn't require a good model of $p(\mathbf{x}|y)$, or even of $p(y|\mathbf{x})$. All that needs to happen is that, at any $\mathbf{x}$, the score for the right class is higher than the score for all other classes. Figure 11.1 shows an example where a normal model of the class-conditional histograms is poor, but the normal model will result in a good naive bayes classifier. This works because a data item from (say) class one will reliably have a larger probability under the normal model for class one than it will for class two.

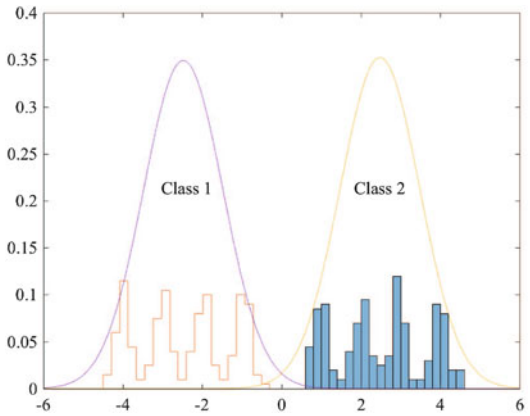

Fig. 11.1 The figure shows class conditional histograms of a feature x for two different classes. The histograms have been normalized so that the counts sum to one, so you can think of them as probability distributions. It should be fairly obvious that a normal model (superimposed) doesn't describe these histograms well. However, the normal model will result in a good naive bayes classifier

Worked example 11.2 (Classifying Breast Tissue Samples). The "breast tissue" dataset at https://archive.ics.uci.edu/ml/datasets/Breast+Tissue contains measurements of a variety of properties of six different classes of breast tissue. Build and evaluate a naive bayes classifier to distinguish between the classes automatically from the measurements.

Solution I used R for this example, because I could then use packages easily. The main difficulty here is finding appropriate packages, understanding their documentation, and checking they're right (unless you want to write the source yourself, which really isn't all that hard). I used the R package `caret` to do train-test splits, cross-validation, etc. on the naive bayes classifier in the R package `klaR`. I separated out a test set randomly (approx 20% of the cases for each class, chosen at random), then trained with cross-validation on the remainder. I used a normal model for each feature. The class-confusion matrix on the test set was:

True	Predict					
	adi	car	con	fad	gla	mas
adi	2	0	0	0	0	0
car	0	3	0	0	0	1
con	2	0	2	0	0	0
fad	0	0	0	0	1	0
gla	0	0	0	0	2	1
mas	0	1	0	3	0	1

which is fairly good. The accuracy is 52%. In the training data, the classes are nearly balanced and there are six classes, meaning that chance is about 17%. These numbers, and the class-confusion matrix, will vary with test-train split. I have not averaged over splits, which would give a somewhat more accurate estimate of accuracy.

11.3.1 Cross-Validation to Choose a Model

Naive bayes presents us with a new problem. We can choose from several different types of model for $p(x^{(j)}|y)$ (eg normal models vs. Poisson models), and we need to know which one produces the best classifier. We also need to know how well that classifier will work. It is natural to use cross-validation to estimate how well each type of model works. You can't just look at every type of model for every variable, because that would yield too many models. Instead, choose M types of model that seem plausible (for example, by looking at histograms of feature components conditioned on class and using your judgement). Now compute a cross-validated error for each of M types of model, and choose the type of model with lowest cross-validated error. Computing the cross-validated error involves repeatedly splitting the training set into two pieces, fitting the model on one and computing the error on the other, then averaging the errors. Notice this means the model you fit to each fold will have slightly different parameter values, because each fold has slightly different training data.

However, once we have chosen the type of model, we have two problems. First, we do not know the correct values for the parameters of the best type of model. For each fold in the cross-validation, we estimated slightly different parameters because we trained on slightly different data, and we don't know which estimate is right. Second, we do not have a good estimate of how well the best model works. This is because we chose the type of model with the smallest error estimate, which is likely smaller than the true error estimate for that type of model.

This problem is easily dealt with if you have a reasonably sized dataset. Split the labelled dataset into two pieces. One (call it the training set) is used for training and for choosing a model type, the other (call it the test set) is used only for evaluating the final model. Now for each type of model, compute the cross-validated error on the training set.

Now use the cross-validated error to choose the type of model. Very often this just means you choose the type that produces the lowest cross-validated error, but there might be cases where two types produce about the same error and one is a lot faster to evaluate, etc. Take the entire training set, and use this to estimate the parameters for that type of model. This estimate should be (a little) better than any of the estimates produced in the cross-validation, because it uses (slightly) more data. Finally, evaluate the resulting model on the test set.

This procedure is rather harder to describe than to do (there's a pretty natural set of nested loops here). There are some strong advantages. First, the estimate of how well a particular model type works is unbiased, because we evaluated on data not used on training. Second, once you have chosen a type of model, the parameter estimate you make is the best you can because you used all the training set to obtain it. Finally, your estimate of how well that particular model works is unbiased, too, because you obtained it using data that wasn't used to train or to select a model.

Remember this: *Naive bayes classifiers are straightforward to build, and very effective. Experience has shown they are particularly effective at high dimensional data. A straightforward variant of cross-validation helps select which particular model to use.*

11.4 The Support Vector Machine

Assume we have a labelled dataset consisting of N pairs $(\mathbf{x}_i, y_i)$. Here $\mathbf{x}_i$ is the i'th feature vector, and y_i is the i'th class label. We will assume that there are two classes, and that y_i is either 1 or -1. We wish to predict the sign of y for any point $\mathbf{x}$. We will use a linear classifier, so that for a new data item $\mathbf{x}$, we will predict

$$\text{sign}\left(\mathbf{a}^T\mathbf{x} + b\right)$$

and the particular classifier we use is given by our choice of $\mathbf{a}$ and b.

You should think of $\mathbf{a}$ and b as representing a hyperplane, given by the points where $\mathbf{a}^T\mathbf{x} + b = 0$. Notice that the magnitude of $\mathbf{a}^T\mathbf{x} + b$ grows as the point $\mathbf{x}$ moves further away from the hyperplane. This hyperplane separates the positive data from the negative data, and is an example of a **decision boundary**. When a point crosses the decision boundary, the label predicted for that point changes. All classifiers have decision boundaries. Searching for the decision boundary that yields the best behavior is a fruitful strategy for building classifiers.

Example 11.1 (A Linear Model with a Single Feature) Assume we use a linear model with one feature. For an example with feature value x, predicts $\text{sign}\,(ax + b)$. Equivalently, the model tests x against the threshold $-b/a$.

Example 11.2 (A Linear Model with Two Features) Assume we use a linear model with two features. For an example with feature vector $\mathbf{x}$, the model predicts $\text{sign}\left(\mathbf{a}^T\mathbf{x} + b\right)$. The sign changes along the line $\mathbf{a}^T\mathbf{x} + b = 0$. You should check that this is, indeed, a line. On one side of this line, the model makes positive predictions; on the other, negative. Which side is which can be swapped by multiplying $\mathbf{a}$ and b by -1.

This family of classifiers may look bad to you, and it is easy to come up with examples that it misclassifies badly. In fact, the family is extremely strong. First, it is easy to estimate the best choice of rule for very large datasets. Second, linear classifiers have a long history of working very well in practice on real data. Third, linear classifiers are fast to evaluate.

In practice, examples that are classified badly by the linear rule usually are classified badly because there are too few features. Remember the case of the alien who classified humans into male and female by looking at their heights; if that alien had looked at their chromosomes as well as height, the error rate would have been smaller. In practical examples, experience shows that the error rate of a poorly performing linear classifier can usually be improved by adding features to the vector $\mathbf{x}$.

We will choose $\mathbf{a}$ and b by choosing values that minimize a cost function. The cost function must achieve two goals. First, the cost function needs a term that ensures each training example should be on the right side of the decision boundary (or, at least, not be too far on the wrong side). Second, the cost function needs a term that should penalize errors on query examples. The appropriate cost function has the form:

$$\text{Training error cost} + \lambda \text{ penalty term}$$

where λ is an unknown weight that balances these two goals. We will eventually set the value of λ by a search process.

11.4.1 The Hinge Loss

Write

$$\gamma_i = \mathbf{a}^T\mathbf{x}_i + b$$

for the value that the linear function takes on example i. Write $C(\gamma_i, y_i)$ for a function that compares γ_i with y_i. The training error cost will be of the form

$$(1/N)\sum_{i=1}^{N} C(\gamma_i, y_i).$$

A good choice of C should have some important properties.

- If γ_i and y_i have different signs, then C should be large, because the classifier will make the wrong prediction for this training example. Furthermore, if γ_i and y_i have different signs *and* γ_i has large magnitude, then the classifier will very likely make the wrong prediction for test examples that are close to $\mathbf{x}_i$. This is because the magnitude of $(\mathbf{a}^T\mathbf{x} + b)$ grows as $\mathbf{x}$ gets further from the decision boundary. So C should get larger as the magnitude of γ_i gets larger in this case.
- If γ_i and y_i have the same signs, but γ_i has small magnitude, then the classifier will classify $\mathbf{x}_i$ correctly, but might not classify points that are nearby correctly. This is because a small magnitude of γ_i means that $\mathbf{x}_i$ is close to the decision boundary, so there will be points nearby that are on the other side of the decision boundary. We want to discourage this, so C should not be zero in this case.
- Finally, if γ_i and y_i have the same signs and γ_i has large magnitude, then C can be zero because $\mathbf{x}_i$ is on the right side of the decision boundary and so are all the points near to $\mathbf{x}_i$.

The **hinge loss**, which takes the form

$$C(y_i, \gamma_i) = \max(0, 1 - y_i\gamma_i),$$

has these properties (Fig. 11.2).

- If γ_i and y_i have different signs, then C will be large. Furthermore, the cost grows linearly as $\mathbf{x}_i$ moves further away from the boundary on the wrong side.
- If γ_i and y_i have the same sign, but $y_i\gamma_i < 1$ (which means that $\mathbf{x}_i$ is close to the decision boundary), there is some cost, which gets larger as $\mathbf{x}_i$ gets closer to the boundary.
- If $y_i\gamma_i > 1$ (so the classifier predicts the sign correctly *and* $\mathbf{x}_i$ is far from the boundary) there is no cost.

A classifier trained to minimize this loss is encouraged to (a) make strong positive (or negative) predictions for positive (or negative) examples and (b) for examples it gets wrong, make predictions with the smallest magnitude that it can. A linear classifier trained with the hinge loss is known as a **support vector machine** or **SVM**.

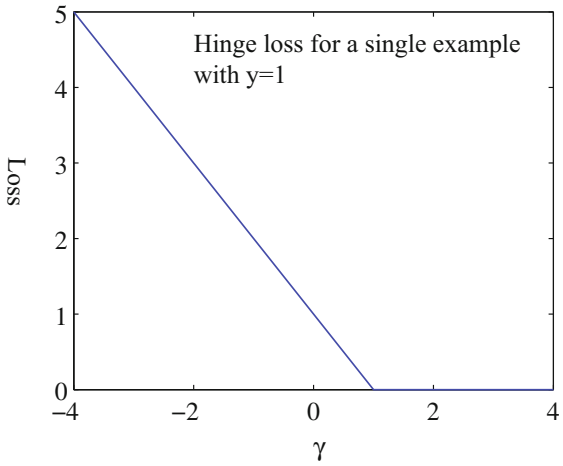

Fig. 11.2 The hinge loss, plotted for the case $y_i = 1$. The horizontal variable is the $\gamma_i = \mathbf{a}^T\mathbf{x}_i + b$ of the text. Notice that giving a strong negative response to this positive example causes a loss that grows linearly as the magnitude of the response grows. Notice also that giving an insufficiently positive response also causes a loss. Giving a strongly positive response is free

11.4.2 Regularization

The penalty term is needed, because the hinge loss has one odd property. Assume that the pair $\mathbf{a}, b$ correctly classifies all training examples, so that $y_i(\mathbf{a}^T x_i + b) > 0$. Then we can always ensure that the hinge loss for the dataset is zero, by scaling $\mathbf{a}$ and b, because you can choose a scale so that $y_j(\mathbf{a}^T x_j + b) > 1$ for *every* example index j. This scale hasn't changed the result of the classification rule on the training data. Now if $\mathbf{a}$ and b result in a hinge loss of zero, then so do $2\mathbf{a}$ and $2b$. This should worry you, because it means we can't choose the classifier parameters uniquely.

Now think about future examples. We don't know what their feature values will be, and we don't know their labels. But we do know that the hinge loss for an example with feature vector $\mathbf{x}$ and unknown label y will be $\max(0, 1 - y\left[\mathbf{a}^T\mathbf{x} + b\right])$. Now imagine the hinge loss for this example *isn't* zero. If the example is classified correctly, then it is close to the decision boundary. We expect that there are fewer of these examples than examples that are far from the decision boundary and on the wrong side, so we concentrate on examples that are misclassified. For misclassified examples, if $\|\mathbf{a}\|$ is small, then at least the hinge loss will be small. By this argument, we would like to achieve a small value of the hinge loss on the training examples using an $\mathbf{a}$ that has small length.

We can do so by adding a penalty term to the hinge loss to favor solutions where $\|\mathbf{a}\|$ is small. To obtain an $\mathbf{a}$ of small length, it is enough to ensure that $(1/2)\mathbf{a}^T\mathbf{a}$ is small (the factor of $1/2$ makes the gradient cleaner). This penalty term will ensure that there is a unique choice of classifier parameters in the case the hinge loss is zero. Experience (and some theory we can't go into here) shows that having a small $\|\mathbf{a}\|$ helps even if there is no pair that classifies all training examples correctly. Doing so improves the error on future examples. Adding a penalty term to improve the solution of a learning problem is sometimes referred to as **regularization**. The penalty term is often 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 parameter λ is often referred to as the **regularization parameter**.

Using the hinge loss to form the training cost, and regularizing with a penalty term $(1/2)\mathbf{a}^T\mathbf{a}$ means our cost function is:

$$S(\mathbf{a}, b; \lambda) = \left[(1/N) \sum_{i=1}^{N} \max(0, 1 - y_i \left(\mathbf{a}^T\mathbf{x}_i + b\right)) \right] + \lambda \left(\frac{\mathbf{a}^T\mathbf{a}}{2} \right).$$

There are now two problems to solve. First, assume we know λ; we will need to find $\mathbf{a}$ and b that minimize $S(\mathbf{a}, b; \lambda)$. Second, we have no theory that tells us how to choose λ, so we will need to search for a good value.

11.4.3 Finding a Classifier with Stochastic Gradient Descent

The usual recipes for finding a minimum are ineffective for our cost function. First, write $\mathbf{u} = [\mathbf{a}, b]$ for the vector obtained by stacking the vector $\mathbf{a}$ together with b. We have a function $g(\mathbf{u})$, and we wish to obtain a value of $\mathbf{u}$ that achieves the

minimum for that function. Sometimes we can solve a problem like this by constructing the gradient and finding a value of $\mathbf{u}$ the makes the gradient zero, but not this time (try it; the max creates problems). We must use a numerical method.

Typical numerical methods take a point $\mathbf{u}^{(n)}$, update it to $\mathbf{u}^{(n+1)}$, then check to see whether the result is a minimum. This process is started from a start point. The choice of start point may or may not matter for general problems, but for our problem a random start point is fine. The update is usually obtained by computing a direction $\mathbf{p}^{(n)}$ such that for small values of η, $g(\mathbf{u}^{(n)} + \eta\mathbf{p}^{(n)})$ is smaller than $g(\mathbf{u}^{(n)})$. Such a direction is known as a **descent direction**. We must then determine how far to go along the descent direction, a process known as **line search**.

Obtaining a descent direction: One method to choose a descent direction is **gradient descent**, which uses the negative gradient of the function. Recall our notation that

$$\mathbf{u} = \begin{pmatrix} u_1 \\ u_2 \\ \dots \\ u_d \end{pmatrix}$$

and that

$$\nabla g = \begin{pmatrix} \frac{\partial g}{\partial u_1} \\ \frac{\partial g}{\partial u_2} \\ \dots \\ \frac{\partial g}{\partial u_d} \end{pmatrix}.$$

We can write a Taylor series expansion for the function $g(\mathbf{u}^{(n)} + \eta\mathbf{p}^{(n)})$. We have that

$$g(\mathbf{u}^{(n)} + \eta\mathbf{p}^{(n)}) = g(\mathbf{u}^{(n)}) + \eta\left[(\nabla g)^T\mathbf{p}^{(n)}\right] + O(\eta^2)$$

This means that we can expect that if

$$\mathbf{p}^{(n)} = -\nabla g(\mathbf{u}^{(n)}),$$

we expect that, at least for small values of h, $g(\mathbf{u}^{(n)} + \eta\mathbf{p}^{(n)})$ will be less than $g(\mathbf{u}^{(n)})$. 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.

But recall that our cost function is a sum of a penalty term and one error cost per example. This means the cost function looks like

$$g(\mathbf{u}) = \left[(1/N)\sum_{i=1}^{N} g_i(\mathbf{u})\right] + g_0(\mathbf{u}),$$

as a function of $\mathbf{u}$. Gradient descent would require us to form

$$-\nabla g(\mathbf{u}) = -\left(\left[(1/N)\sum_{i=1}^{N} \nabla g_i(\mathbf{u})\right] + \nabla g_0(\mathbf{u})\right)$$

and then take a small step in this direction. But if N is large, this is unattractive, as we might have to sum a lot of terms. This happens a lot in building classifiers, where you might quite reasonably expect to deal with millions (billions; perhaps trillions) of examples. Touching each example at each step really is impractical.

Stochastic gradient descent is an algorithm that replaces the exact gradient with an approximation that has a random error, but is simple and quick to compute. The term

$$(\frac{1}{N})\sum_{i=1}^{N} \nabla g_i(\mathbf{u}).$$

is a population mean, and we know how to deal with those. We can estimate this term by drawing a random sample (a **batch**) of N_b (the **batch size**) examples, with replacement, from the population of N examples, then computing the mean for that sample. We approximate the population mean by

$$(\frac{1}{N_b}) \sum_{j \in \text{batch}} \nabla g_j(\mathbf{u}).$$

The batch size is usually determined using considerations of computer architecture (how many examples fit neatly into cache?) or of database design (how many examples are recovered in one disk cycle?). One common choice is $N_b = 1$, which is the same as choosing one example uniformly and at random. We form

$$\mathbf{p}_{N_b}^{(n)} = -\left(\left[(1/N_b) \sum_{j \in \text{batch}} \nabla g_i(\mathbf{u}) \right] + \nabla g_0(\mathbf{u}) \right)$$

and then take a small step along $\mathbf{p}_{N_b}^{(n)}$. Our new point becomes

$$\mathbf{u}^{(n+1)} = \mathbf{u}^{(n)} + \eta \mathbf{p}_{N_b}^{(n)},$$

where η is called the **steplength** (or sometimes **step size** or **learning rate**, even though it isn't the size or the length of the step we take, or a rate!).

Because the expected value of the sample mean is the population mean, if we take many small steps along $\mathbf{p}_{N_b}$, they should average out to a step backwards along the gradient. This approach is known as stochastic gradient descent because we're not going along the gradient, but along a random vector which is the gradient only in expectation. It isn't obvious that stochastic gradient descent is a good idea. Although each step is easy to take, we may need to take more steps. The question is then whether we gain in the increased speed of the step what we lose by having to take more steps. Not much is known theoretically, but in practice the approach is hugely successful for training classifiers.

Choosing a steplength: Choosing a steplength η takes some work. We can't search for the step that gives us the best value of g, because we don't want to evaluate the function g (doing so involves looking at each of the g_i terms). Instead, we use an η that is large at the start—so that the method can explore large changes in the values of the classifier parameters—and small steps later—so that it settles down. The choice of how η gets smaller is often known as a **steplength schedule**.

Here are useful examples of steplength schedules. Often, you can tell how many steps are required to have seen the whole dataset; this is called an **epoch**. A common steplength schedule sets the steplength in the e'th epoch to be

$$\eta^{(e)} = \frac{m}{e+n},$$

where m and n are constants chosen by experiment with small subsets of the dataset. When there are a lot of examples, an epoch is a long time to fix the steplength, and this approach can reduce the steplength too slowly. Instead, you can divide training into what I shall call **seasons** (blocks of a fixed number of iterations, smaller than epochs), and make the steplength a function of the season number.

There is no good test for whether stochastic gradient descent has converged to the right answer, because natural tests involve evaluating the gradient and the function, and doing so is expensive. More usual is to plot the error as a function of iteration on the validation set, and interrupt or stop training when the error has reached an acceptable level. The error (resp. accuracy) should vary randomly (because the steps are taken in directions that only approximate the gradient) but should decrease (resp. increase) overall as training proceeds (because the steps do approximate the gradient). Figures 11.3 and 11.4 show examples of these curves, which are sometimes known as **learning curves**.

11.4.4 Searching for λ

We do not know a good value for λ. We will obtain a value by choosing a set of different values, fitting an SVM using each value, and taking the λ value that will yield the best SVM. Experience has shown that the performance of a method is not profoundly sensitive to the value of λ, so that we can look at values spaced quite far apart. It is usual to take some small number (say, $1e-4$), then multiply by powers of 10 (or 3, if you're feeling fussy and have a fast computer). So, for example, we might look at $\lambda \in \{1e-4, 1e-3, 1e-2, 1e-1\}$. We know how to fit an SVM to a particular value of λ (Sect. 11.4.3). The problem is to choose the value that yields the best SVM, and to use that to get the best classifier.

We have seen a version of this problem before (Sect. 11.3.1). There, we chose from several different types of model to obtain the best naive bayes classifier. The recipe from that section is easily adapted to the current problem. We regard each

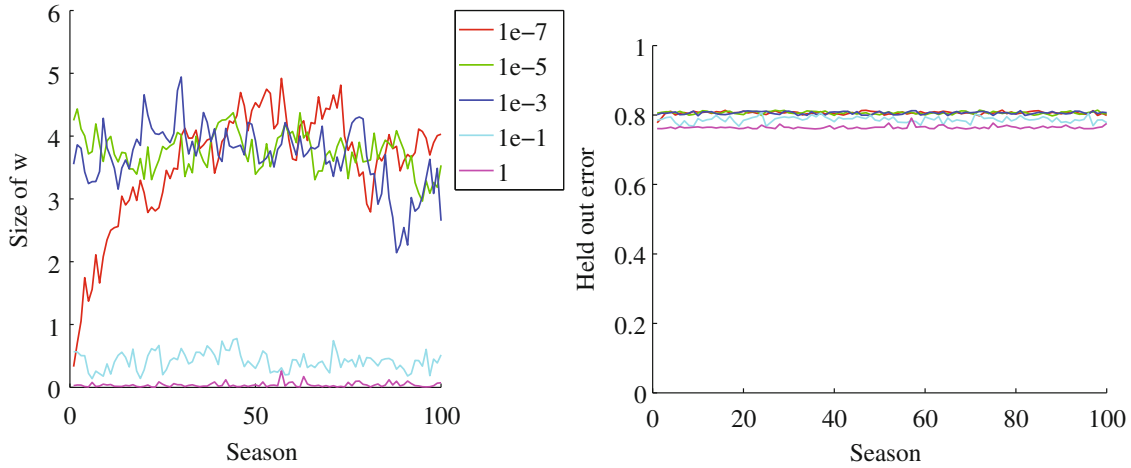

Fig. 11.3 On the *left*, the magnitude of the weight vector **a** at the end of each season for the first training regime described in the text. On the *right*, the accuracy on held out data at the end of each season. Notice how different choices of regularization parameter lead to different magnitudes of **a**; how the method isn't particularly sensitive to choice of regularization parameter (they change by factors of 100); how the accuracy settles down fairly quickly; and how overlarge values of the regularization parameter do lead to a loss of accuracy

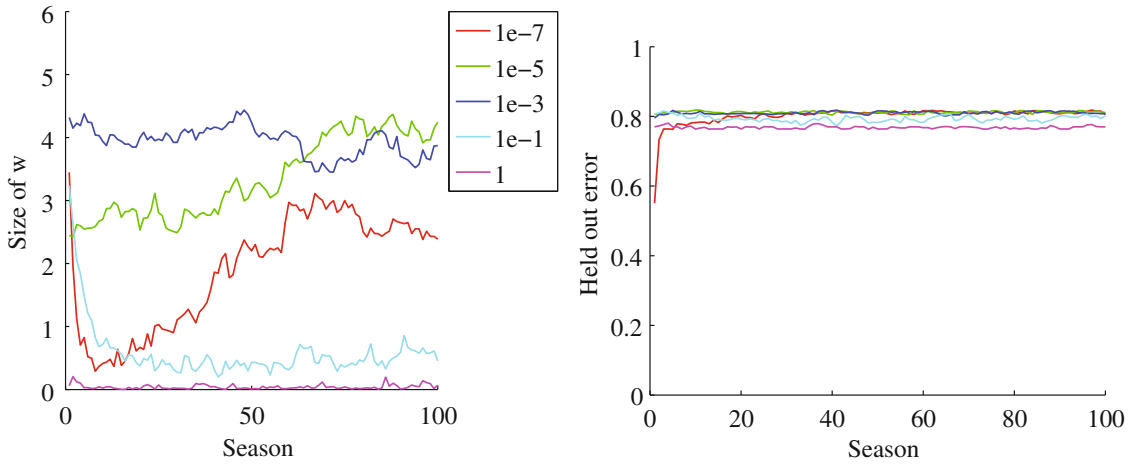

Fig. 11.4 On the *left*, the magnitude of the weight vector **a** at the end of each season for the second training regime described in the text. On the *right*, the accuracy on held out data at the end of each season. Notice how different choices of regularization parameter lead to different magnitudes of **a**; how the method isn't particularly sensitive to choice of regularization parameter (they change by factors of 100); how the accuracy settles down fairly quickly; and how overlarge values of the regularization parameter do lead to a loss of accuracy

different λ value as representing a different model. We split the data into two pieces: one is a training set, used for fitting and choosing models; the other is a test set, used for evaluating the final chosen model.

Now for each value of λ, compute the cross-validated error of an SVM using that λ on the training set. Do this by repeatedly splitting the training set into two pieces (training and validation); fitting the SVM with that λ to the training piece using stochastic gradient descent; evaluating the error on the validation piece; and averaging these errors. Now use the cross-validated error to choose the best λ value. Very often this just means you choose the value that produces the lowest cross-validated error, but there might be cases where two values produce about the same error and one is preferred for some other reason. Notice that you can compute the standard deviation of the cross-validated error as well as the mean, so you can tell whether differences between cross-validated errors are significant.

Now take the entire training set, and use this to fit an SVM for the chosen λ value. This should be (a little) better than any of the SVMs obtained in the cross-validation, because it uses (slightly) more data. Finally, evaluate the resulting SVM on the test set.

This procedure is rather harder to describe than to do (there's a pretty natural set of nested loops here). There are some strong advantages. First, the estimate of how well a particular SVM type works is unbiased, because we evaluated on data

not used on training. Second, once you have chosen the cross-validation parameter, the SVM you fit is the best you can fit because you used all the training set to obtain it. Finally, your estimate of how well that particular SVM works is unbiased, too, because you obtained it using data that wasn't used to train or to select a model.

11.4.5 Example: Training an SVM with Stochastic Gradient Descent

I have summarized the SVM training procedure in a set of boxes, below. You should be aware that the recipe there admits many useful variations, though. One useful practical trick is to rescale the feature vector components so each has unit variance. This doesn't change anything conceptual as the best choice of decision boundary for rescaled data is easily derived from the best choice for unscaled, and vice versa. Rescaling very often makes stochastic gradient descent perform better because the method takes steps that are even in each component.

It is quite usual to use packages to fit SVM's, and good packages may use a variety of tricks which we can't go into to make training more efficient. Nonetheless, you should have a grasp of the overall process, because it follows a pattern that is useful for training other models (among other things, most deep networks are trained using this pattern).

Procedure 11.1 (Training an SVM: Overall) Start with a dataset containing N pairs $(\mathbf{x}_i, y_i)$. Each $\mathbf{x}_i$ is a d-dimensional feature vector, and each y_i is a label, either 1 or -1. Optionally, rescale the $\mathbf{x}_i$ so that each component has unit variance. Choose a set of possible values of the regularization weight λ. Separate the dataset into two sets: test and training. Reserve the test set. For each value of the regularization weight, use the training set to estimate the accuracy of an SVM with that λ value, using cross-validation as in Procedure 11.2 and stochastic gradient descent. Use this information to choose λ_0, the best value of λ (usually, the one that yields the highest accuracy). Now use the training set to fit the best SVM using λ_0 as the regularization constant. Finally, use the test set to compute the accuracy or error rate of that SVM, and report that

Procedure 11.2 (Training an SVM: Estimating the Accuracy) Repeatedly: split the training dataset into two components (training and validation), at random; use the training component to train an SVM; and compute the accuracy on the validation component. Now average the resulting accuracy values.

Procedure 11.3 (Training an SVM: Stochastic Gradient Descent) Obtain $\mathbf{u} = (\mathbf{a}, b)$ by stochastic gradient descent on the cost function

$$g(\mathbf{u}) = \left[(1/N) \sum_{i=1}^{N} g_i(\mathbf{u}) \right] + g_0(\mathbf{u})$$

where $g_0(\mathbf{u}) = \lambda(\mathbf{a}^T\mathbf{a})/2$ and $g_i(\mathbf{u}) = \max(0, 1 - y_i\left(\mathbf{a}^T\mathbf{x}_i + b\right))$.

Do so by first choosing a fixed number of items per batch N_b, the number of steps per season N_s, and the number of steps k to take before evaluating the model (this is usually a lot smaller than N_s). Choose a random start point. Now iterate:

- Update the stepsize. In the s'th season, the step size is typically $\eta^{(s)} = \frac{m}{s+n}$ for constants m and n chosen by small-scale experiments.
- Split the training dataset into a training part and a validation part. This split changes each season. Use the validation set to get an unbiased estimate of error during that season's training.
- Now, until the end of the season (i.e. until you have taken N_s steps):

(continued)

– Take k steps. Each step is taken by selecting a batch of N_b data items uniformly and at random from the training part for that season. Write $\mathcal{D}$ for this set. Now compute

$$\mathbf{p}^{(n)} = -\frac{1}{N_b}\left(\sum_{i\in\mathcal{D}} \nabla g_i(\mathbf{u}^{(n)})\right) - \lambda\mathbf{u}^{(n)},$$

and update the model by computing

$$\mathbf{u}^{(n+1)} = \mathbf{u}^{(n)} + \eta\mathbf{p}^{(n)}$$

– Evaluate the current model $\mathbf{u}^{(n)}$ by computing the accuracy on the validation part for that season. Plot the accuracy as a function of step number.

There are two ways to stop. You can choose a fixed number of seasons (or of epochs) and stop when that is done. Alternatively, you can watch the error plot and stop when the error reaches some level or meets some criterion.

Here is an example in some detail. I downloaded the dataset at http://archive.ics.uci.edu/ml/datasets/Adult. This dataset apparently contains 48,842 data items, but I worked with only the first 32,000. Each consists of a set of numeric and categorical features describing a person, together with whether their annual income is larger than or smaller than 50 K$. I ignored the categorical features to prepare these figures. This isn't wise if you want a good classifier, but it's fine for an example. I used these features to predict whether income is over or under 50 K$. I split the data into 5000 test examples, and 27,000 training examples. It's important to do so at random. There are 6 numerical features. I subtracted the mean (which doesn't usually make much difference) and rescaled each so that the variance was 1 (which is often very important).

Setting up stochastic gradient descent: We have estimates $\mathbf{a}^{(n)}$ and $b^{(n)}$ of the classifier parameters, and we want to improve the estimates. I used a batch size of $N_b = 1$. Pick the r'th example at random. The gradient is

$$\nabla\left(\max(0, 1 - y_r\left(\mathbf{a}^T\mathbf{x}_r + b\right)) + \frac{\lambda}{2}\mathbf{a}^T\mathbf{a}\right).$$

Assume that $y_k\left(\mathbf{a}^T\mathbf{x}_r + b\right) > 1$. In this case, the classifier predicts a score with the right sign, and a magnitude that is greater than one. Then the first term is zero, and the gradient of the second term is easy. Now if $y_k\left(\mathbf{a}^T\mathbf{x}_r + b\right) < 1$, we can ignore the max, and the first term is $1 - y_r\left(\mathbf{a}^T\mathbf{x}_r + b\right)$; the gradient is again easy. If $y_r\left(\mathbf{a}^T\mathbf{x}_r + b\right) = 1$, there are two distinct values we could choose for the gradient, because the max term isn't differentiable. It does not matter which value we choose because this situation hardly ever happens. We choose a steplength η, and update our estimates using this gradient. This yields:

$$\mathbf{a}^{(n+1)} = \mathbf{a}^{(n)} - \eta\begin{cases}\lambda\mathbf{a} & \text{if } y_k\left(\mathbf{a}^T\mathbf{x}_k + b\right) \geq 1\\ \lambda\mathbf{a} - y_k\mathbf{x} & \text{otherwise}\end{cases}$$

and

$$b^{(n+1)} = b^{(n)} - \eta\begin{cases}0 & \text{if } y_k\left(\mathbf{a}^T\mathbf{x}_k + b\right) \geq 1\\ -y_k & \text{otherwise}\end{cases}.$$

Training: I used two different training regimes. In the first training regime, there were 100 seasons. In each season, I applied 426 steps. For each step, I selected one data item uniformly at random (sampling with replacement), then stepped down the gradient. This means the method sees a total of 42,600 data items. This means that there is a high probability it has touched each data item once (27,000 isn't enough, because we are sampling with replacement, so some items get seen more than once). I chose five different values for the regularization parameter and trained with a steplength of $1/(0.01 * s + 50)$, where s is the season. At the end of each season, I computed $\mathbf{a}^T\mathbf{a}$ and the accuracy (fraction of examples correctly classified) of the current classifier on the held out test examples. Figure 11.3 shows the results. You should notice that the accuracy changes slightly each season; that for larger regularizer values $\mathbf{a}^T\mathbf{a}$ is smaller; and that the accuracy settles down to about 0.8 very quickly.

In the second training regime, there were 100 seasons. In each season, I applied 50 steps. For each step, I selected one data item uniformly at random (sampling with replacement), then stepped down the gradient. This means the method sees a

total of 5000 data items, and about 3000 unique data items—it hasn't seen the whole training set. I chose five different values for the regularization parameter and trained with a steplength of $1/(0.01 * s + 50)$, where s is the season. At the end of each season, I computed $\mathbf{a}^T\mathbf{a}$ and the accuracy (fraction of examples correctly classified) of the current classifier on the held out test examples. Figure 11.4 shows the results.

This is an easy classification example. Points worth noting are

- the accuracy makes large changes early, then settles down to make slight changes each season;
- quite large changes in regularization constant have small effects on the outcome, but there is a best choice;
- for larger values of the regularization constant, $\mathbf{a}^T\mathbf{a}$ is smaller;
- there isn't much difference between the two training regimes;
- and the method doesn't need to see all the training data to produce a classifier that is about as good as it would be if the method *had* seen all training data.

All of these points are relatively typical of SVM's trained using stochastic gradient descent with very large datasets.

Remember this: *Linear SVM's are a go-to classifier. When you have a binary classification problem, the first step should be to try a linear SVM. Training with stochastic gradient descent is straightforward, and extremely effective. Finding an appropriate value of the regularization constant requires an easy search. There is an immense quantity of good software available.*

11.4.6 Multi-Class Classification with SVMs

I have shown how one trains a linear SVM to make a binary prediction (i.e. predict one of two outcomes). But what if there are three, or more, labels? In principle, you could write a binary code for each label, then use a different SVM to predict each bit of the code. It turns out that this doesn't work terribly well, because an error by one of the SVM's is usually catastrophic.

There are two methods that are widely used. 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 (you have to build $O(N^2)$ different SVM's for N 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 can think up quite good reasons this approach shouldn't work. For one thing, the classifier isn't told that you intend to use the score to tell similarity between classes. In practice, the approach works rather well and is quite widely used. This approach scales a bit better with the number of classes ($O(N)$).

Remember this: *It is straightforward to build a multi-class classifier out of binary classifiers. Any decent SVM package will do this for you.*

11.5 Classifying with Random Forests

One way to build a classifier is to use a sequence of simple tests, where each test is allowed to use the results of all previous tests. This class of rule can be drawn as a tree (Fig. 11.5), where each node represents a test, and the edges represent the possible outcomes of the test. To classify a test item with such a tree, you present it to the first node; the outcome of the test determines which node it goes to next; and so on, until the example arrives at a leaf. When it does arrive at a leaf, we label the test item with the most common label in the leaf. This object is known as a **decision tree**. Notice one attractive feature of this decision tree: it deals with multiple class labels quite easily, because you just label the test item with the most common label in the leaf that it arrives at when you pass it down the tree.

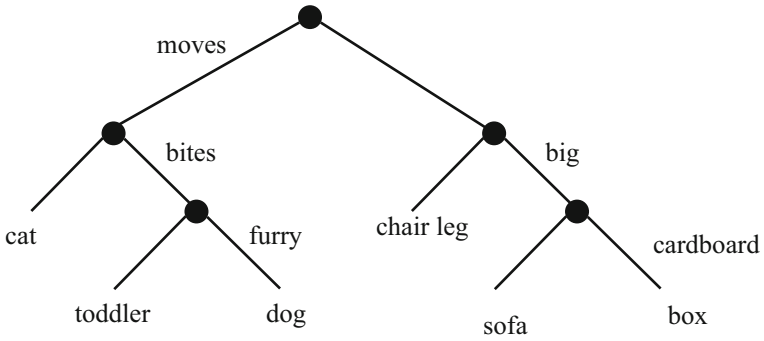

Fig. 11.5 This—the household robot's guide to obstacles—is a typical decision tree. I have labelled only one of the outgoing branches, because the other is the negation. So if the obstacle moves, bites, but isn't furry, then it's a toddler. In general, an item is passed down the tree until it hits a leaf. It is then labelled with the leaf's label

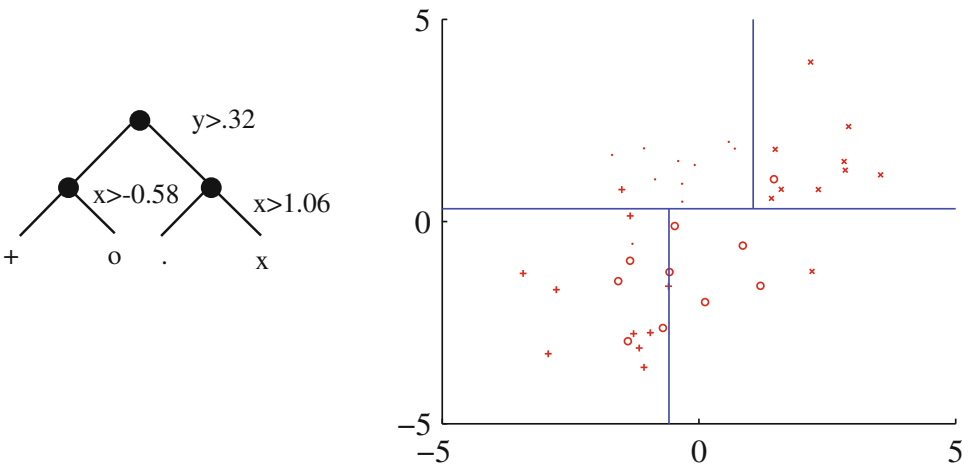

Fig. 11.6 A straightforward decision tree, illustrated in two ways. On the *left*, I have given the rules at each split; on the *right*, I have shown the data points in two dimensions, and the structure that the tree produces in the feature space

Figure 11.6 shows a simple 2D dataset with four classes, next to a decision tree that will correctly classify at least the training data. Actually classifying data with a tree like this is straightforward. We take the data item, and pass it down the tree. Notice it can't go both left and right, because of the way the tests work. This means each data item arrives at a single leaf. We take the most common label at the leaf, and give that to the test item. In turn, this means we can build a geometric structure on the feature space that corresponds to the decision tree. I have illustrated that structure in Fig. 11.6, where the first decision splits the feature space in half (which is why the term split is used so often), and then the next decisions split each of those halves into two.

The important question is how to get the tree from data. We will always use a binary tree, because it's easier to describe, because it's usual, and because it doesn't change anything important in the algorithm. Each non-leaf node has a **decision function**, which takes data items and returns either 1 or -1. We train the tree by thinking about its effect on the training data. We pass the whole pool of training data into the root. The decision function at any non-leaf node splits incoming data into two pools, one for the left child (all the data that the decision function labels 1) and the other for the right child (ditto, -1). This goes on recursively until finally, each leaf contains a pool of data, which it can't split because it is a leaf.

To classify a data item, we pass it down the tree, applying decision functions to choose left or right, until we reach a leaf. Any data item that reaches a particular leaf will get that leaf's label. This means that we would like all the items in the training data pool at a given leaf to agree on a label. But it matters how we achieve this. For example, a very large tree with one data item in each leaf will have excellent training accuracy, but is likely to have poor test accuracy. Intuition should suggest that good test accuracy can be obtained by a tree where each leaf has a large pool of data that all has one label.

All this means that it is quite difficult to determine the best tree. A really powerful alternative is to build a many simple trees using algorithms that incorporate a great deal of randomness. The algorithms ensure that we get a different tree each time we train a tree on a dataset. None of the individual trees will be particularly good (they are often referred to as "weak learners"). But with many such trees (a **decision forest**), we can allow each to vote; the class that gets the most votes, wins. This strategy is extremely effective.

11.5.1 Building a Decision Tree: General Algorithm

There are many algorithms for building decision trees. I will sketch an approach chosen for simplicity and effectiveness; be aware there are others. I will leave out enough detail that you probably couldn't implement a program from my description. I've used this approach because most people never need to implement a program (there are excellent packages available), and need only a moderate understanding of how one could be built. For the people who want more detail, there is more in the mathematical material in the end (Sect. 15.3).

Training the tree uses a straightforward algorithm. First, we choose a class of decision functions to use at each node. It turns out that a very effective decision function is to choose a single feature at random, then test whether its value is larger than, or smaller than a threshold (some minor adjustments are required if the feature chosen isn't ordinal). For this approach to work, one needs to be quite careful about the choice of threshold, which is what we describe in the next section. Surprisingly, being clever about the choice of *feature* doesn't seem add a great deal of value. We won't spend more time on other kinds of decision function, though there are lots.

Now assume we use a decision function as described, and we know how to choose a threshold. We start with the root node, then recursively either split the pool of data at that node, passing the left pool left and the right pool right, or stop splitting and return. Splitting involves choosing a decision function from the class to give the "best" split for a leaf. The main questions are how to choose the best split (next section), and when to stop.

Stopping is relatively straightforward. Quite simple strategies for stopping are very good. It is hard to choose a decision function with very little data, so we must stop splitting when there is too little data at a node. We can tell this is the case by testing the amount of data against a threshold, chosen by experiment. If all the data at a node belongs to a single class, there is no point in splitting. Finally, constructing a tree that is too deep tends to result in generalization problems, so we usually allow no more than a fixed depth D of splits. In applications, D can be quite small. It is quite common to use $D = 1$ (when the chopped down tree is, rather unhappily, known as a decision stump).

11.5.2 Building a Decision Tree: Choosing a Split

Choosing the best splitting threshold is more complicated. Figure 11.7 shows two possible splits of a pool of training data. One is quite obviously a lot better than the other. In the good case, the split separates the pool into positives and negatives. In the bad case, each side of the split has the same number of positives and negatives. We cannot usually produce splits as good as the good case here. What we are looking for is a split that will make the proper label more certain.

Figure 11.8 shows a more subtle case to illustrate this. The splits in this figure are obtained by testing the horizontal feature against a threshold. In one case, the left and the right pools contain about the same fraction of positive ('x') and negative ('o') examples. In the other, the left pool is all positive, and the right pool is mostly negative. This is the better choice of threshold. If we were to label any item on the left side positive and any item on the right side negative, the error rate would be fairly small. If you count, the best error rate for the informative split is 20% on the training data, and for the uninformative split it is 40% on the training data.

But we need some way to score the splits, so we can tell which threshold is best. Notice that, in the uninformative case, knowing that a data item is on the left (or the right) does not tell me much more about the data than I already knew before I had the split. In this case, we have that $p(1|\text{left pool, uninformative}) = 2/3 \approx 3/5 = p(1|\text{parent pool})$ and $p(1|\text{right pool, uninformative}) = 1/2 \approx 3/5 = p(1|\text{parent pool})$. For the informative split, knowing a data item is on the left classifies it completely, and knowing that it is on the right allows us to classify it an error rate of $1/3$. The informative split means that my uncertainty about what class the data item belongs to is significantly reduced if I know whether it goes left or right. To choose a good threshold, we need a score of how informative a split is. The score is known as the **information gain** (where larger is better). The details of how to compute information gain are involved, and of little interest unless you need to implement a decision tree; I've put this in the mathematical material at the end (Sect. 15.3.2).

We now have a relatively straightforward blueprint for an algorithm, which I have put in a box. It's a blueprint, because there are a variety of ways in which it can be revised and changed.

Procedure 11.4 (Building a Decision Tree: Overall) We have a dataset containing N pairs $(\mathbf{x}_i, y_i)$. Each x_i is a d-dimensional feature vector, and each y_i is a label. Call this dataset a **pool**. Now recursively apply the following procedure:

(continued)

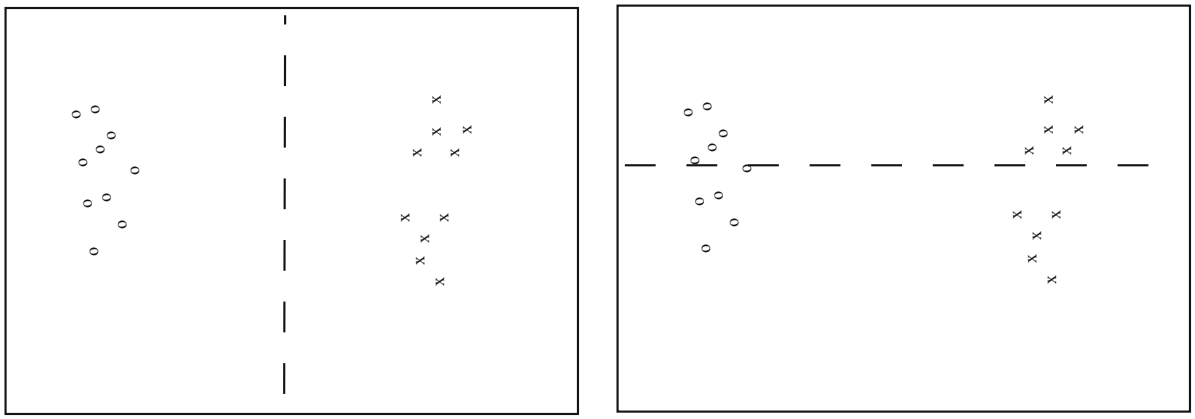

Fig. 11.7 Two possible splits of a pool of training data. Positive data is represented with an 'x', negative data with a 'o'. Notice that if we split this pool with the informative line, all the points on the left are 'o's, and all the points on the right are 'x's. This is an excellent choice of split—once we have arrived in a leaf, everything has the same label. Compare this with the less informative split. We started with a node that was half 'x' and half 'o', and now have two nodes each of which is half 'x' and half 'o'—this isn't an improvement, because we do not know more about the label as a result of the split

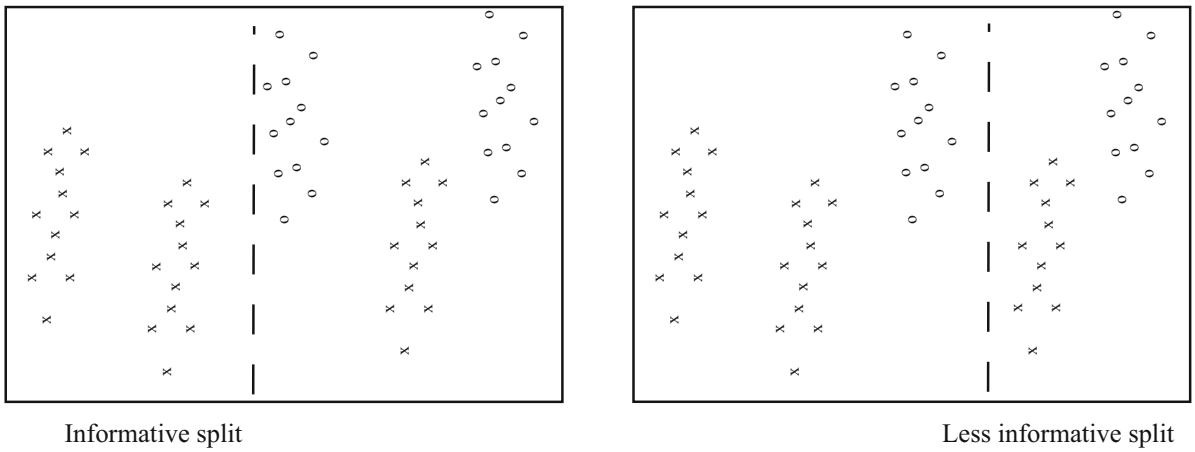

Fig. 11.8 Two possible splits of a pool of training data. Positive data is represented with an 'x', negative data with a 'o'. Notice that if we split this pool with the informative line, all the points on the left are 'x's, and two-thirds of the points on the right are 'o's. This means that knowing which side of the split a point lies would give us a good basis for estimating the label. In the less informative case, about two-thirds of the points on the left are 'x's and about half on the right are 'x's—knowing which side of the split a point lies is much less useful in deciding what the label is

- If the pool is too small, or if all items in the pool have the same label, or if the depth of the recursion has reached a limit, stop.
- Otherwise, search the features for a good split that divides the pool into two, then apply this procedure to each child.

We search for a good split by the following procedure:

- Choose a subset of the feature components at random. Typically, one uses a subset whose size is about the square root of the feature dimension.
- For each component of this subset, search for a good split. If the component is ordinal, do so using the procedure of box 11.5, otherwise use the procedure of box 11.6.

Procedure 11.5 (Splitting an Ordinal Feature) We search for a good split on a given ordinal feature by the following procedure:

- Select a set of possible values for the threshold.
- For each value split the dataset (every data item with a value of the component below the threshold goes left, others go right), and compute the information gain for the split.

Keep the threshold that has the largest information gain.

A good set of possible values for the threshold will contain values that separate the data "reasonably". If the pool of data is small, you can project the data onto the feature component (i.e. look at the values of that component alone), then choose the $N - 1$ distinct values that lie between two data points. If it is big, you can randomly select a subset of the data, then project that subset on the feature component and choose from the values between data points.

Procedure 11.6 (Splitting a Non-Ordinal Feature) Split the values this feature takes into sets pools by flipping an unbiased coin for each value—if the coin comes up H, any data point with that value goes left, and if it comes up T, any data point with that value goes right. Repeating this procedure F times, computing the information gain for each split, then keep the split that has the best information gain. We choose F in advance, and it usually depends on the number of values the categorical variable can take.

11.5.3 Forests

Rather than build the best possible tree, we have built a tree efficiently, but with a number of random choices. If we were to rebuild the tree, we would obtain a different result. This suggests the following extremely effective strategy: build many trees, and classify by merging their results.

There are two important strategies for building and evaluating decision forests. I am not aware of evidence strongly favoring one over the other, but different software packages use different strategies, and you should be aware of the options. In one strategy, we separate labelled data into a training and a test set. We then build multiple decision trees, training each using the whole training set. Finally, we evaluate the forest on the test set. In this approach, the forest has not seen some fraction of the available labelled data, because we used it to test. However, each tree has seen every training data item.

Procedure 11.7 (Building a Decision Forest) We have a dataset containing N pairs $(\mathbf{x}_i, y_i)$. Each $\mathbf{x}_i$ is a d-dimensional feature vector, and each y_i is a label. Separate the dataset into a test set and a training set. Train multiple distinct decision trees on the training set, recalling that the use of a random set of components to find a good split means you will obtain a distinct tree each time.

In the other strategy, sometimes called **bagging**, each time we train a tree we randomly subsample the labelled data with replacement, to yield a training set the same size as the original set of labelled data. Notice that there will be duplicates in this training set, which is like a bootstrap replicate. This training set is often called a **bag**. We keep a record of the examples that do not appear in the bag (the "out of bag" examples). Now to evaluate the forest, we evaluate each tree on its out of bag examples. Each example gets votes for labels from each tree for which it is out of bag. Now classify each example using these votes, and compute the error for every example. In this approach, the entire forest has seen all labelled data, yet we get a good estimate of error, because no tree in the forest has been evaluated on data used to train it.

Procedure 11.8 (Building a Decision Forest Using Bagging) We have a dataset containing N pairs $(\mathbf{x}_i, y_i)$. Each $\mathbf{x}_i$ is a d-dimensional feature vector, and each y_i is a label. Now build k bootstrap replicates of the training data set. Train one decision tree on each replicate.

Once we have a forest, we must classify test data items. There are two major strategies. The simplest is to classify the item with each tree in the forest, then take the class with the most votes. This is effective, but discounts some evidence that might be important. For example, imagine one of the trees in the forest has a leaf with many data items with the same class label; another tree has a leaf with exactly one data item in it. One might not want each leaf to have the same vote.

Procedure 11.9 (Classification with a Decision Forest) Given a test example $\mathbf{x}$, pass it down each tree of the forest. Now choose one of the following strategies.

- Each time the example arrives at a leaf, record one vote for the label that occurs most often at the leaf. Now choose the label with the most votes.
- Each time the example arrives at a leaf, record N_l votes for each of the labels that occur at the leaf, where N_l is the number of times the label appears in the training data at the leaf. Now choose the label with the most votes.

An alternative strategy that takes this observation into account is to pass the test data item down each tree. When it arrives at a leaf, we record one vote for each of the training data items in that leaf. The vote goes to the class of the training data item. Finally, we take the class with the most votes. This approach allows big, accurate leaves to dominate the voting process. Both strategies are in use, and I am not aware of compelling evidence that one is always better than the other. This may be because the randomness in the training process makes big, accurate leaves uncommon in practice.

Worked example 11.3 (Classifying Heart Disease Data) Build a random forest classifier to classify the "heart" dataset from the UC Irvine machine learning repository. The dataset is at http://archive.ics.uci.edu/ml/datasets/Heart+Disease. There are several versions. You should look at the processed Cleveland data, which is in the file "processed.cleveland.data.txt".

Solution I used the R random forest package. This uses a bagging strategy. This package makes it quite simple to fit a random forest, as you can see. In this dataset, variable 14 (V14) takes the value 0, 1, 2, 3 or 4 depending on the severity of the narrowing of the arteries. Other variables are physiological and physical measurements pertaining to the patient (read the details on the website). I tried to predict all five levels of variable 14, using the random forest as a multivariate classifier. This works rather poorly, as the out-of-bag class confusion matrix below shows. The total out-of-bag error rate was 45%.

True	Predict					
	0	1	2	3	4	Class error
0	151	7	2	3	1	7.9%
1	32	5	9	9	0	91%
2	10	9	7	9	1	81%
3	6	13	9	5	2	86%
4	2	3	2	6	0	100%

This is the example of a class confusion matrix from Table 11.1. Fairly clearly, one can predict narrowing or no narrowing from the features, but not the degree of narrowing (at least, not with a random forest). So it is natural to quantize variable 14 to two levels, 0 (meaning no narrowing), and 1 (meaning any narrowing, so the

(continued)

original value could have been 1, 2, or 3). I then built a random forest to predict this quantized variable from the other variables. The total out-of-bag error rate was 19%, and I obtained the following out-of-bag class confusion matrix

True	Predict		
	0	1	Class error
0	138	26	16%
1	31	108	22%

Notice that the false positive rate (16%, from 26/164) is rather better than the false negative rate (22%). You might wonder whether it is better to train on and predict $0, \ldots, 4$, then quantize the predicted value. If you do this, you will find you get a false positive rate of 7.9%, but a false negative rate that is much higher (36%, from 50/139). In this application, a false negative is likely more of a problem than a false positive, so the tradeoff is unattractive.

Remember this: *Random forests are straightforward to build, and very effective. They can predict any kind of label. Good software implementations are easily available.*

11.6 You Should

11.6.1 Remember These Definitions

11.6.2 Remember These Terms

11.6.3 Remember These Facts

11.6.4 Use These Procedures

11.6.5 Be Able to

- build a nearest neighbors classifier using your preferred software package, and produce a cross-validated estimate of its error rate or its accuracy;
- build a naive bayes classifier using your preferred software package, and produce a cross-validated estimate of its error rate or its accuracy;
- build an SVM using your preferred software package, and produce a cross-validated estimate of its error rate or its accuracy;
- write code to train an SVM using stochastic gradient descent, and produce a cross-validated estimate of its error rate or its accuracy;
- and build a decision forest using your preferred software package, and produce a cross-validated estimate of its error rate or its accuracy.

Programming Exercises

11.1 The UC Irvine machine learning data repository hosts a famous collection of data on whether a patient has diabetes (the Pima Indians dataset), originally owned by the National Institute of Diabetes and Digestive and Kidney Diseases and donated by Vincent Sigillito. This can be found at http://archive.ics.uci.edu/ml/datasets/Pima+Indians+Diabetes. This data has a set of attributes of patients, and a categorical variable telling whether the patient is diabetic or not. This is an exercise oriented to users of R, because you can use some packages to help.

(a) Build a simple naive Bayes classifier to classify this data set. You should hold out 20% of the data for evaluation, and use the other 80% for training. You should use a normal distribution to model each of the class-conditional distributions. You should write this classifier yourself.

(b) Now use the `caret` and `klaR` packages to build a naive bayes classifier for this data. The `caret` package does cross-validation (look at `train`) and can be used to hold out data. The `klaR` package can estimate class-conditional densities using a density estimation procedure that I will describe much later in the course. Use the cross-validation mechanisms in `caret` to estimate the accuracy of your classifier.

(c) Now install SVMLight, which you can find at http://svmlight.joachims.org, via the interface in `klaR` (look for `svmlight` in the manual) to train and evaluate an SVM to classify this data. You don't need to understand much about SVM's to do this—we'll do that in following exercises. You should hold out 20% of the data for evaluation, and use the other 80% for training.

11.2 The UC Irvine machine learning data repository hosts a collection of data on student performance in Portugal, donated by Paulo Cortez, University of Minho, in Portugal. You can find this data at https://archive.ics.uci.edu/ml/datasets/Student+Performance. It is described in P. Cortez and A. Silva. "Using Data Mining to Predict Secondary School Student Performance," In A. Brito and J. Teixeira Eds., *Proceedings of 5th FUture BUsiness TEChnology Conference (FUBUTEC 2008)* pp. 5-12, Porto, Portugal, April, 2008,

There are two datasets (for grades in mathematics and for grades in Portugese). There are 30 attributes each for 649 students, and 3 values that can be predicted (G1, G2 and G3). Of these, ignore G1 and G2.

(a) Use the mathematics dataset. Take the G3 attribute, and quantize this into two classes, $G3 > 12$ and $G3 \leq 12$. Build and evaluate a naive bayes classifier that predicts G3 from all attributes except G1 and G2. You should build this classifier from scratch (i.e. DON'T use the packages described in the code snippets). For binary attributes, you should use a binomial model. For the attributes described as "numeric", which take a small set of values, you should use a multinomial model. For the attributes described as "nominal", which take a small set of values, you should again use a multinomial model. Ignore the "absence" attribute. Estimate accuracy by cross-validation. You should use at least tenfolds, excluding 15% of the data at random to serve as test data, and average the accuracy over those folds. Report the mean and standard deviation of the accuracy over the folds.

(b) Now revise your classifier of the previous part so that, for the attributes described as "numeric", which take a small set of values, you use a multinomial model. For the attributes described as "nominal", which take a small set of values, you should still use a multinomial model. Ignore the "absence" attribute. Estimate accuracy by cross-validation. You should

use at least tenfolds, excluding 15% of the data at random to serve as test data, and average the accuracy over those folds. Report the mean and standard deviation of the accuracy over the folds.

(c) Which classifier do you believe is more accurate and why?

11.3 The UC Irvine machine learning data repository hosts a collection of data on heart disease. The data was collected and supplied by Andras Janosi, M.D., of the Hungarian Institute of Cardiology, Budapest; William Steinbrunn, M.D., of the University Hospital, Zurich, Switzerland; Matthias Pfisterer, M.D., of the University Hospital, Basel, Switzerland; and Robert Detrano, M.D., Ph.D., of the V.A. Medical Center, Long Beach and Cleveland Clinic Foundation. You can find this data at https://archive.ics.uci.edu/ml/datasets/Heart+Disease.

Use the processed Cleveland dataset, where there are a total of 303 instances with 14 attributes each. The irrelevant attributes described in the text have been removed in these. The 14'th attribute is the disease diagnosis. There are records with missing attributes, and you should drop these.

(a) Take the disease attribute, and quantize this into two classes, num $= 0$ and num > 0. Build and evaluate a naive bayes classifier that predicts the class from all other attributes Estimate accuracy by cross-validation. You should use at least tenfolds, excluding 15% of the data at random to serve as test data, and average the accuracy over those folds. Report the mean and standard deviation of the accuracy over the folds.

(b) Now revise your classifier to predict each of the possible values of the disease attribute (0–4 as I recall). Estimate accuracy by cross-validation. You should use at least tenfolds, excluding 15% of the data at random to serve as test data, and average the accuracy over those folds. Report the mean and standard deviation of the accuracy over the folds.

11.4 The UC Irvine machine learning data repository hosts a collection of data on breast cancer diagnostics, donated by Olvi Mangasarian, Nick Street, and William H. Wolberg. You can find this data at http://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Diagnostic). For each record, there is an id number, 10 continuous variables, and a class (benign or malignant). There are 569 examples. Separate this dataset randomly into 100 validation, 100 test, and 369 training examples.

Write a program to train a support vector machine on this data using stochastic gradient descent. You should not use a package to train the classifier (you don't really need one), but your own code. You should ignore the id number, and use the continuous variables as a feature vector. You should scale these variables so each has unit variance. You should search for an appropriate value of the regularization constant, trying at least the values $\lambda = [1e-3, 1e-2, 1e-1, 1]$. Use the validation set for this search.

You should use at least 50 epochs of at least 100 steps each. In each epoch, you should separate out 50 training examples at random for evaluation. You should compute the accuracy of the current classifier on the set held out for the epoch every 10 steps. You should produce:

(a) A plot of the accuracy every 10 steps, for each value of the regularization constant.

(b) Your estimate of the best value of the regularization constant, together with a brief description of why you believe that is a good value.

(c) Your estimate of the accuracy of the best classifier on held out data

11.5 The UC Irvine machine learning data repository hosts a collection of data on adult income, donated by Ronny Kohavi and Barry Becker. You can find this data at https://archive.ics.uci.edu/ml/datasets/Adult For each record, there is a set of continuous attributes, and a class $\geq$50 K or $<$50 K. There are 48,842 examples. You should use only the continous attributes (see the description on the web page) and drop examples where there are missing values of the continuous attributes. Separate the resulting dataset randomly into 10% validation, 10% test, and 80% training examples.

Write a program to train a support vector machine on this data using stochastic gradient descent. You should not use a package to train the classifier (you don't really need one), but your own code. You should ignore the id number, and use the continuous variables as a feature vector. You should scale these variables so that each has unit variance. You should search for an appropriate value of the regularization constant, trying at least the values $\lambda = [1e-3, 1e-2, 1e-1, 1]$. Use the validation set for this search

You should use at least 50 epochs of at least 300 steps each. In each epoch, you should separate out 50 training examples at random for evaluation. You should compute the accuracy of the current classifier on the set held out for the epoch every 30 steps. You should produce:

(a) A plot of the accuracy every 30 steps, for each value of the regularization constant.
(b) Your estimate of the best value of the regularization constant, together with a brief description of why you believe that is a good value.
(c) Your estimate of the accuracy of the best classifier on held out data

11.6 The UC Irvine machine learning data repository hosts a collection of data on the whether p53 expression is active or inactive. You can find out what this means, and more information about the dataset, by reading: Danziger, S.A., Baronio, R., Ho, L., Hall, L., Salmon, K., Hatfield, G.W., Kaiser, P., and Lathrop, R.H. "Predicting Positive p53 Cancer Rescue Regions Using Most Informative Positive (MIP) Active Learning," *PLOS Computational Biology*, 5(9), 2009; Danziger, S.A., Zeng, J., Wang, Y., Brachmann, R.K. and Lathrop, R.H. "Choosing where to look next in a mutation sequence space: Active Learning of informative p53 cancer rescue mutants", *Bioinformatics*, 23(13), 104–114, 2007; and Danziger, S.A., Swamidass, S.J., Zeng, J., Dearth, L.R., Lu, Q., Chen, J.H., Cheng, J., Hoang, V.P., Saigo, H., Luo, R., Baldi, P., Brachmann, R.K. and Lathrop, R.H. "Functional census of mutation sequence spaces: the example of p53 cancer rescue mutants," *IEEE/ACM transactions on computational biology and bioinformatics*, 3, 114–125, 2006.

You can find this data at https://archive.ics.uci.edu/ml/datasets/p53+Mutants. There are a total of 16,772 instances, with 5409 attributes per instance. Attribute 5409 is the class attribute, which is either active or inactive. There are several versions of this dataset. You should use the version `K8.data`.

(a) Train an SVM to classify this data, using stochastic gradient descent. You will need to drop data items with missing values. You should estimate a regularization constant using cross-validation, trying at least three values. Your training method should touch at least 50% of the training set data. You should produce an estimate of the accuracy of this classifier on held out data consisting of 10% of the dataset, chosen at random.
(b) Now train a naive bayes classifier to classify this data. You should produce an estimate of the accuracy of this classifier on held out data consisting of 10% of the dataset, chosen at random.
(c) Compare your classifiers. Which one is better? why?

11.7 The UC Irvine machine learning data repository hosts a collection of data on whether a mushroom is edible, donated by Jeff Schlimmer and to be found at http://archive.ics.uci.edu/ml/datasets/Mushroom. This data has a set of categorical attributes of the mushroom, together with two labels (poisonous or edible). Use the R random forest package (as in the example in the chapter) to build a random forest to classify a mushroom as edible or poisonous based on its attributes.

(a) Produce a class-confusion matrix for this problem. If you eat a mushroom based on your classifier's prediction it is edible, what is the probability of being poisoned?

MNIST Exercises

The following exercises are elaborate, but rewarding. The MNIST dataset is a dataset of 60,000 training and 10,000 test examples of handwritten digits, originally constructed by Yann Lecun, Corinna Cortes, and Christopher J.C. Burges. It is very widely used to check simple methods. There are 10 classes in total ("0" to "9"). This dataset has been extensively studied, and there is a history of methods and feature constructions at https://en.wikipedia.org/wiki/MNIST_database and at http://yann.lecun.com/exdb/mnist/. You should notice that the best methods perform extremely well. The original dataset is at http://yann.lecun.com/exdb/mnist/. It is stored in an unusual format, described in detail on that website. Writing your own reader is pretty simple, but web search yields readers for standard packages. There is reader code in matlab available (at least) at http://ufldl.stanford.edu/wiki/index.php/Using_the_MNIST_Dataset. There is reader code for R available (at least) at https://stackoverflow.com/questions/21521571/how-to-read-mnist-database-in-r.

The dataset consists of 28×28 images. These were originally binary images, but appear to be grey level images as a result of some anti-aliasing. I will ignore mid grey pixels (there aren't many of them) and call dark pixels "ink pixels", and light pixels "paper pixels". The digit has been centered in the image by centering the center of gravity of the image pixels. Here are some options for re-centering the digits that I will refer to in the exercises.

- **Untouched:** do not re-center the digits, but use the images as is.
- **Bounding box:** construct an $b \times b$ bounding box so that the horizontal (resp. vertical) range of ink pixels is centered in the box.

- **Stretched bounding box:** construct an $b \times b$ bounding box so that the horizontal (resp. vertical) range of ink pixels runs the full horizontal (resp. vertical) range of the box. Obtaining this representation will involve rescaling image pixels: you find the horizontal and vertical ink range, cut that out of the original image, then resize the result to $b \times b$.

Once the image has been re-centered, you can compute features. Here are some options for constructing features that I will refer to in the exercises.

- **Raw pixels:** use the raw pixel values from images.
- **PCA:** project images onto the first d principal components computed for the entire dataset.
- **Local PCA:** first, compute the first d principal components for each digit class separately. Now for any image, compute a $10d$ dimensional feature vector by, for each class, subtracting that class mean from the image, then projecting the image onto the d principal components for that class. Finally, stack all 10 d dimensional features you get. This measures how much the difference between the image and the class mean looks like the difference between images of that class and the class mean.

11.8 Investigate classifying MNIST using naive bayes. Use the procedures of Sect. 11.3.1 to compare four cases on raw pixel image features. These cases are obtained by choosing either normal model or binomial model for every feature, and untouched images or stretched bounding box images.

(a) Which is the best case?
(b) How accurate is the best case? (remember, the answer to this is *not* obtained by taking the best accuracy from the previous subexercise—check Sect. 11.3.1 if you're vague on this point).

11.9 Investigate classifying MNIST using nearest neighbors. You will use approximate nearest neighbors. Obtain the FLANN package for approximate nearest neighbors from http://www.cs.ubc.ca/~mariusm/index.php/FLANN/FLANN. To use this package, you should consider first using a function that builds an index for the training dataset (`flann_build_index()`, or variants), then querying with your test points (`flann_find_nearest_neighbors_index()`, or variants). The alternative (`flann_find_nearest_neighbors()`, etc.) builds the index then throws it away, which can be inefficient if you don't use it correctly.

(a) Compare untouched raw pixels with bounding box raw pixels and with stretched bounding box raw pixels. Which works better? Why? Is there a difference in query times?
(b) Does rescaling each feature (i.e. each pixel value) so that it has unit variance improve either classifier from the previous subexercise?
(c) Plot accuracy against d for a variety of d values for stretched bounding box PCA. You should use some large values of d, reasonably close to 784 ($= 28 \times 28$). Compare this to the accuracy of stretched bounding box raw pixels (equivalent to $d = 784$).
(d) Does rescaling each feature (i.e. each projected direction) so that it has unit variance improve results from the previous subexercise?

11.10 Investigate classifying MNIST using an SVM. Compare the following four cases: untouched raw pixels; stretched bounding box raw pixels; stretched bounding box PCA; and stretched bounding box local PCA. Which works best? Why?

11.11 Investigate classifying MNIST using a decision forest. Using the same parameters for your forest construction (i.e. same depth of tree; same number of trees; etc.), compare the following four cases: untouched raw pixels; stretched bounding box raw pixels; stretched bounding box PCA; and stretched bounding box local PCA. Which works best? Why?

11.12 If you've done all four previous exercises, you're likely tired of MNIST, but very well informed. Compare your methods to the table of methods at http://yann.lecun.com/exdb/mnist/. What improvements could you make?