CHAPTER 6

Classifying with Random Forests

I described a classifier as a rule that takes a feature, and produces a class. One way to build such a rule is with 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 (Figure **??**), 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**.

Figure **??** 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.

The important question is how to get the tree from data. It turns out that the best approach for building a tree incorporates a great deal of randomness. As a result, we will 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"). The natural thing to do is to produce many such trees (a **decision forest**), and allow each to vote; the class that gets the most votes, wins. This strategy is extremely effective.

6.1 BUILDING A DECISION TREE

There are many algorithms for building decision trees. We will use an approach chosen for simplicity and effectiveness; be aware there are others. We will always use a binary tree, because it's easier to describe and because that's usual (it doesn't change anything important, though). Each 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. Any node splits its incoming data into two pools, left (all the data that the decision function labels 1) and right (ditto, -1). Finally, each leaf contains a pool of data, which it can't split because it is a leaf.

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 algorithm is to choose a single feature at random, then test whether its value is larger than, or smaller than a threshold. 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. Some minor adjustments, described below, are required if the feature chosen isn't ordinal.

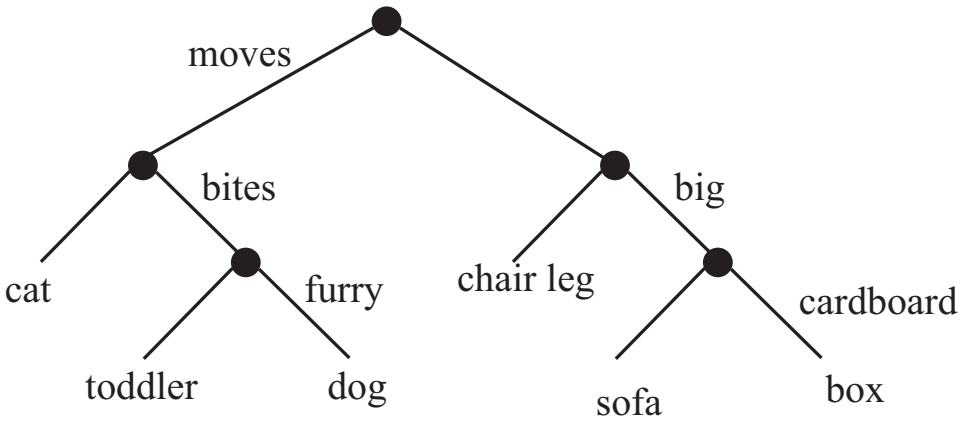

FIGURE 6.1: *This — the household robot's guide to obstacles — is a typical decision tree. I have labelled only one of the outgoing branches, because the other is the negation. So if the obstacle moves, bites, but isn't furry, then it's a toddler. In general, an item is passed down the tree until it hits a leaf. It is then labelled with the leaf's label.*

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. Choosing the best splitting threshold is more complicated.

6.1.1 Entropy and Information Gain

Figure 6.2 shows two possible splits of a pool of training data. These splits 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. Clearly this is the better choice of threshold. But we need some way to score what has happened, so we can tell which threshold is best. Notice that, in

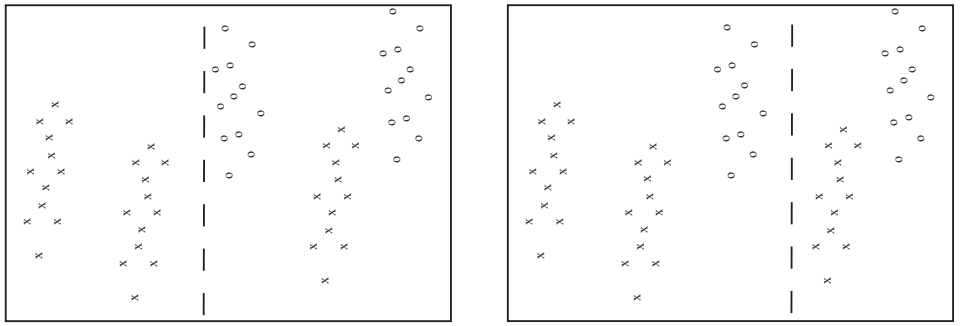

FIGURE 6.2: *Two possible splits of a pool of training data. Positive data is represented with an 'x', negative data with a 'o'. Notice that if we split this pool with the informative line, all the points on the left are 'x's, and two-thirds of the points on the right are 'o's. This means that knowing 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.*

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. This is because

$$p(1|\text{left pool}) \approx p(1|\text{parent pool}).$$

In the second case, knowing a data item is on the left classifies it completely. In this case, 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 to keep track of how informative the split is.

It turns out to be straightforward to keep track of information, in simple cases. We will start with an example. Assume I have 4 classes. There are 8 examples in class 1, 4 in class 2, 2 in class 3, and 2 in class 4. How much information *on average* will you need to send me to tell me the class of a given example? Clearly, this depends on how you communicate the information. You could send me the complete works of Edward Gibbon to communicate class 1; the Encyclopaedia for class 2; and so on. But this would be redundant. The question is how little can you send me. Keeping track of the amount of information is easier if we encode it with bits (i.e. you can send me sequences of '0's and '1's).

Imagine the following scheme. If an example is in class 1, you send me a '1'. If it is in class 2, you send me '01'; if it is in class 3, you send me '001'; and in class 4, you send me '101'. Then the expected number of bits you will send me is

$$p(\text{class} = 1)1 + p(2)2 + p(3)3 + p(4)3 = \frac{1}{2}1 + \frac{1}{4}2 + \frac{1}{8}3 + \frac{1}{8}3$$

which is 1.75 bits. This number doesn't have to be an integer, because it's an expectation.

Notice that for the i'th class, you have sent me $-\log_2 p(i)$ bits. We can write the expected number of bits you need to send me as

$$-\sum_i p(i) \log_2 p(i).$$

This expression handles other simple cases correctly, too. You should try what happens if you have two classes, each with 8 examples in them; 256 classes, each with one example in them; and 5 classes, with 16 examples in class 1, 8 in class 2, etc. If you try other examples, you may find it hard to construct a scheme where you can send as few bits *on average* as this expression predicts. It turns out that, in general, the smallest number of bits you will need to send me is given by the expression

$$-\sum_i p(i) \log_2 p(i)$$

under all conditions, though it may be hard or impossible to determine what representation is required to achieve this number.

Now we return to the splits. Write $\mathcal{P}$ for the set of all data at the node. Write $\mathcal{P}_l$ for the left pool, and $\mathcal{P}_r$ for the right pool. The **entropy** of a pool $\mathcal{C}$ is a function $H(\mathcal{C})$ that scores how many bits would be required to represent the class of an item in that pool, on average. Write $n(i; \mathcal{C})$ for the number of items of class i in the pool, and $N(\mathcal{C})$ for the number of items in the pool. Then the entropy of the pool $\mathcal{C}$ is

$$-\sum_i \frac{n(i;\mathcal{C})}{N(\mathcal{C})} \log_2 \frac{n(i;\mathcal{C})}{N(\mathcal{C}}.$$

It is straightforward that $H(\mathcal{P})$ bits are required to classify an item in the parent pool $\mathcal{P}$. For an item in the left pool, we need $H(\mathcal{P}_l)$ bits; for an item in the right pool, we need $H(\mathcal{P}_r)$ bits. If we split the parent pool, we expect to encounter items in the left pool with probability

$$\frac{N(\mathcal{P}_l)}{N(\mathcal{P})}$$

and items in the right pool with probability

$$\frac{N(\mathcal{P}_r)}{N(\mathcal{P})}.$$

This means that, on average, we must supply

$$\frac{N(\mathcal{P}_l)}{N(\mathcal{P})} H(\mathcal{P}_l) + \frac{N(\mathcal{P}_r)}{N(\mathcal{P})} H(\mathcal{P}_r)$$

bits to classify data items if we split the parent pool. Now a good split is one that results in left and right pools that are informative. In turn, we should need fewer bits to classify once we have split than we need before the split. You can see the difference

$$I(\mathcal{P}_l, \mathcal{P}_r; \mathcal{P}) = H(\mathcal{P}) - \left(\frac{N(\mathcal{P}_l)}{N(\mathcal{P})} H(\mathcal{P}_l) + \frac{N(\mathcal{P}_r)}{N(\mathcal{P})} H(\mathcal{P}_r) \right)$$

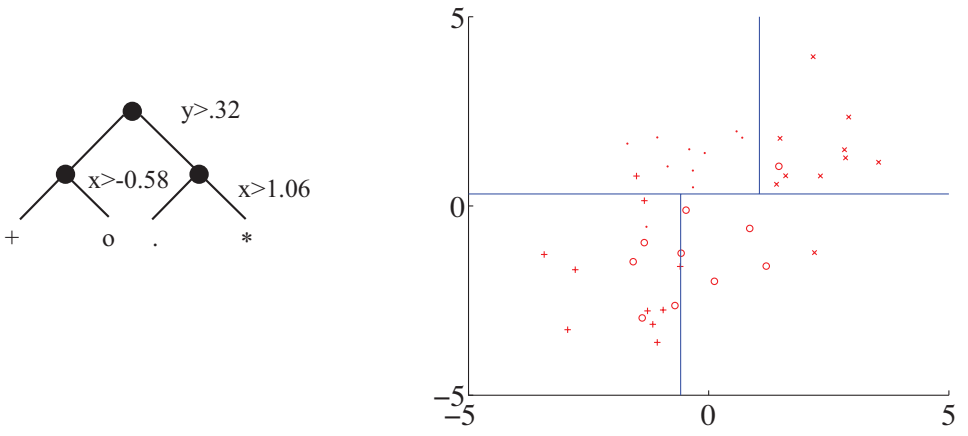

FIGURE 6.3: *A straightforward decision tree*

as the **information gain** caused by the split. This is the average number of bits that you *don't* have to supply if you know which side of the split an example lies. Better splits have larger information gain.

6.1.2 Choosing a Split with Information Gain

Recall that our decision function is to choose a feature at random, then test its value against a threshold. Any data point where the value is larger goes to the left pool; where the value is smaller goes to the right. This may sound much too simple to work, but it is actually effective and popular. Assume that we are at a node, which we will label k. We have the pool of training examples that have reached that node. The i'th example has a feature vector $\mathbf{x}_i$, and each of these feature vectors is a d dimensional vector.

We choose an integer j in the range $1 \ldots d$ uniformly and at random. We will split on this feature, and we store j in the node. Recall we write $x_i^{(j)}$ for the value of the j'th component of the i'th feature vector. We will choose a threshold t_k, and split by testing the sign of $x_i^{(j)} - t_k$. Choosing the value of t_k is easy. Assume there are N_k examples in the pool. Then there are $N_k - 1$ possible values of t_k that lead to different splits. To see this, sort the N_k examples by $x^{(j)}$, then choose values of t_k halfway between example values (Figure **??**). For each of these values, we compute the information gain of the split. We then keep the threshold with the best information gain.

We can elaborate this procedure in a useful way, by choosing m features at random, finding the best split for each, then keeping the feature and threshold value that is best. It is important that m is a lot smaller than the total number of features — a usual root of thumb is that m is about the square root of the total number of features. It is usual to choose a single m, and choose that for all the splits.

Now assume we happen to have chosen to work with a feature that isn't

ordinal, and so can't be tested against a threshold. A natural, and effective, strategy is as follows. We can split such a feature into two 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. We chose this split at random, so it might not be any good. We can come up with a good split by repeating this procedure F times, computing the information gain for each split, then keeping the one 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.

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: 6.1 *Building a decision tree*

Assume we have a data set

TODO: an algorithm block

6.2 FORESTS

A single decision tree tends to yield poor classifications. One reason is because the tree is not chosen to give the best classification of its training data. We used a random selection of splitting variables at each node, so the tree can't be the "best possible". Obtaining the best possible tree presents significant technical difficulties. It turns out that the tree that gives the best possible results on the training data can perform rather poorly on test data. The training data is a small subset of possible examples, and so must differ from the test data. The best possible tree on the training data might have a large number of small leaves, built using carefully chosen splits. But the choices that are best for training data might not be best for test data.

Rather than build the best possible tree, we have built a tree efficiently, but with 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.

6.2.1 Building and Evaluating a Decision Forest

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.

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, and average these error terms. In this approach, the entire forest has seen all labelled data, and we also get an estimate of error, but no tree has seen all the training data.

6.2.2 Classifying Data Items with a Decision Forest

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.

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.

Notice one of the major attractions of random forests. Our strategy doesn't depend on the number of classes we are dealing with (though the results might).

Worked example 6.1 *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. There is sample code in listing **??**. 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%.

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

This is the example of a class confusion matrix from table 5.1. 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 original value could have been 1, 2, or 3). I then built a random forest to predict this from the other variables. The total out-of-bag error rate was 19%, and I obtained the following out-of-bag class confusion matrix

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

Notice that the false positive rate (16%, from 26/164) is rather better than the false negative rate (22%). Looking at these class confusion matrices, you might wonder whether it is better to predict $0, \ldots, 4$, then quantize. But this is not a particularly good idea. While the false positive rate is 7.9%, the false negative rate 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.

Listing 6.1: R code used for the random forests of worked example 1

```
setwd('/users/daf/Current/courses/Probcourse/Trees/RCode');
install.packages('randomForest')
library(randomForest)
heart<-read.csv('processed.cleveland.data.txt', header=FALSE)
heart$levels<-as.factor(heart$V14)
heartforest.allvals<-
  randomForest(formula=levels~V1+V2+V3+V4+V5+V6
               +V7+V8+V9+V10+V11+V12+V13,
               data=heart, type='classification', mtry=5)
# this fits to all levels
# I got the CCM by typing
heartforest.allvals
heart$yesno<-cut(heart$V14, c(-Inf, 0.1, Inf))
heartforest<-
  randomForest(formula=yesno~V1+V2+V3+V4+V5+V6
               +V7+V8+V9+V10+V11+V12+V13,
               data=heart, type='classification', mtry=5)
# this fits to the quantized case
# I got the CCM by typing
heartforest
```