CHAPTER 11

Voting and its Applications

11.1 THE HOUGH TRANSFORM

Given a set of N tokens, you must choose a collection of lines that represent those tokens. There may be more than one line. Reporting one line per pair of distinct points is not helpful. The *Hough transform* takes each token and casts a vote for every line that could pass through that token, then analyzes the votes to find the lines. This procedure can be applied to more complicated shapes, but the case of lines is enough to illustrate all issues.

Parametrize a line as the collection of points $\mathbf{x} = (x_1, x_2)^T$ such that

$$x_1 \cos\theta + x_2 \sin\theta + r = 0.$$

Now any pair of (θ, r) represents a unique line, where $r \geq 0$ is the perpendicular distance from the line to the origin and $0 \leq \theta < 2\pi$. Because the image has a known size, there is some R such that, if $r > R$, the line cannot have votes (these lines are too far away from the origin for any token to appear in the image). Call the set of acceptable (θ, r) *line space*. The lines that lie on the curve *in line space* given by $r = -x_1 \cos\theta + x_2 \sin\theta$ all pass through the point token at $\mathbf{x} = (x_1, x_2)^T$.

Discretize line space with some convenient grid, where each grid element is a bucket into which votes can be placed. This is the *accumulator array*. For the i'th point token at $\mathbf{x}_i = (x_{1,i}, x_{2,i})^T$, visit every bucket on the curve *in line space* given by $r = -x_{1,i} \cos\theta + x_{2,i} \sin\theta$ and add one to the count of votes in that bucket. Now analyze the accumulator array. If there are many point tokens that are collinear, there should be many votes in the grid element corresponding to that line (Figure 11.1).

Notice this is an extremely general procedure, and could apply to (say) fitting circles, planes or spheres (**exercises**). Practical issues make the Hough transform as described difficult to use, even for finding lines. To my knowledge, the Hough transform has not been used to fit lines in practice for some time. The obstacles are worth understanding. Assume there is only one line, and all tokens lie near it. Noise means tokens are not necessarily on the line. This noise has a nasty effect on the accumulator array. When noise moves a token in the image, the set of lines it will vote for in the accumulator array will move too. The bucket corresponding to the right line will lose votes, and some other buckets gain votes. If there is enough noise, the bucket with the largest number of votes may not correspond to the right line.

Even if there is only one line, you should not expect all tokens lie near it. Think about an image that is dark-ish on one side of a line and light-ish on the other. Texture or even image noise may generate tokens on either side that have nothing to do with the line. These tokens tend to result in phantom lines – buckets with many votes in them that do not correspond to actual lines (Figure 11.1).

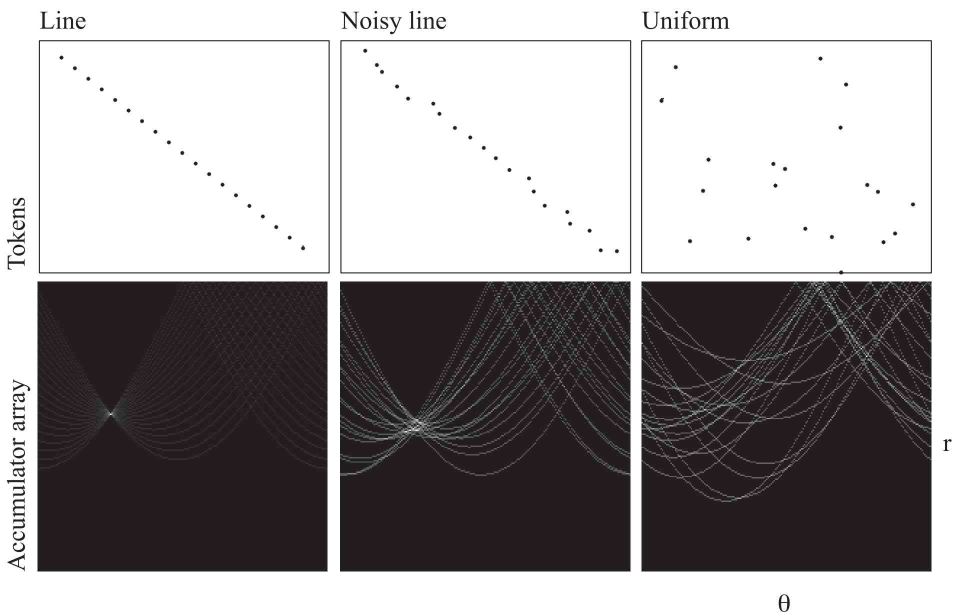

FIGURE 11.1: *The Hough transform maps each point like token to a curve of possible lines (or other parametric curves) through that point. These figures illustrate the Hough transform for lines. The* **top row** *shows tokens, and the* **bottom row** *shows the corresponding accumulator arrays (the number of votes is indicated by the gray level; largest number of votes is full bright).* **Left***: 20 points drawn from a line (largest number of votes is 20).* **Center***: the same points, offset by a small random vector (largest number of votes is 6).* **Right***: both coordinates of each data point are uniform random numbers in the range* $[0, 1]$*) (largest number of votes is 4).*

Changing the quantization of the accumulator array might look as though it could control noise effects. Votes that appeared in the same bucket in a coarsely quantized accumulator array tend to miss one another in a finer array, so buckets with large numbers of votes due to noise should break up. But in a finely quantized accumulator array, votes from tokens on the actual line will tend to miss one another, meaning you may miss lines.

11.2 INSTANCE CLASSIFICATION BY VOTING

Instance level classification is the problem of determining whether a particular object is present in an image. If it is there (wherever it appears) the image is labelled with that object. Instance classification is rather different than *category level categorization*, where one must determine whether any instance of a particular category is present. So, for example, if you have to tell whether your two-year old tabby cat is in a picture, you are doing instance level classification. If you have to tell whether there is a cat in the image, you are doing category level categorization.

Instance level classification is important and useful in applications. I will use

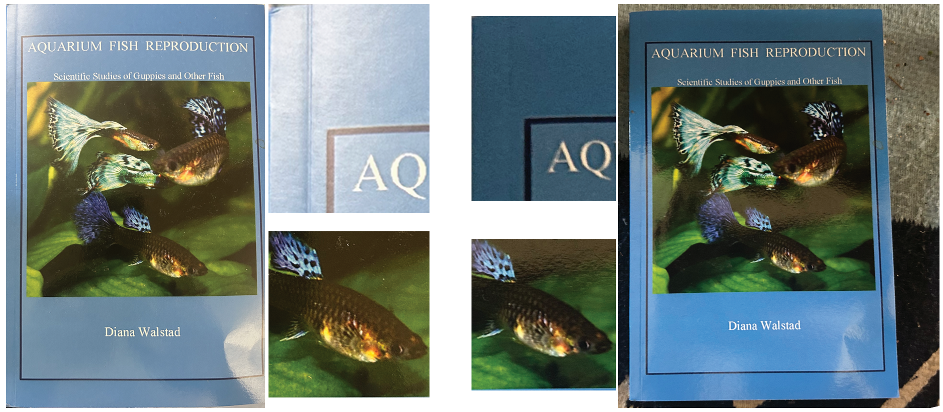

FIGURE 11.2:

the following problem as a running example. Assume you have a large collection of book cover images (*example images*), each with associated *metadata* (say, the name of the book, the author, the publisher, the edition and the publication date). A user holds a book cover in front of a camera. Assume the resulting *query image* is reasonable – there are no fingers obscuring the book cover, it is shown at a reasonable angle, it is shown in reasonable lighting, it happens to be the same size as in the example image, and so on. You wish to use the example images to determine what book appears in the query image *or* that the book is unknown (to you, anyway!). Notice that, because this is an instance level classification problem, two different editions would actually be two different instances if they looked different – so they might have different cover pictures.

You might attack this problem with the elementary detector of Section 3.4.3, but results would be poor. The same book covers will look different when viewed in slightly different lighting, so just matching part of an image with the costs of Section 3.4.2 is unlikely to work well (when the chicken of Section 3.4.3 got darker or changed position, the match became worse).

Although the whole image of the book cover might not match because of lighting effects, patches might match (Figure 11.2). Give each different book a unique number. Take each known book cover image, and cut it into patches. Cluster all the patches, and build a tree as in Section **??**. Ensure that each leaf of the tree contains relatively few patches (hundreds rather than millions). At each leaf, record the number of the book that has the most patches in the leaf. To find the most likely book for a new cover image, cut that image into patches; pass each patch down the tree and record a vote for the book in its leaf; now choose the book with the largest number of votes. If that book has enough votes, decide the query image contains that book; if it doesn't, decide the query image contains an unknown book. This procedure is a manifestation of the underlying principle of the Hough transform: if many simple local measurements agree on something, they're likely right.

There are easy ways to enhance this recipe. You could build the tree using patches from multiple images of the book cover. For example, rotated by 0^o, 90^o,

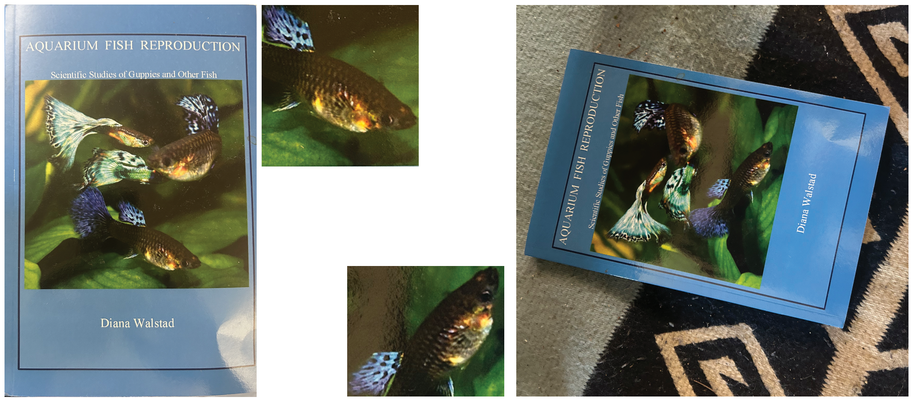

FIGURE 11.3:

180^0 and 270^0 and scaled by a range of scales); this is explored in the exercises (**exercises**). In the original recipe, each patch votes for the book that has the most patches in its leaf. Instead, the patch could record one vote for each book present in its leaf. Alternatively, a patch could record a vote only when the margin in its leaf is large enough – that is, the book that has the most patches in the leaf has substantially more votes than the book with the second largest number. Similarly, you could build multiple trees – each of which yields somewhat different behaviors, because of the random starts in the k-means step and the random subsampling in the hierarchical process – and accumulate votes over trees. These will yield improvements, but most powerful is to use interest points and modify the tree construction.

11.2.1 Voting Using Interest Points

The example images of the book covers are obtained under different circumstances – camera position, lighting, and so on – than the query images. In turn, a patch in a query image may look rather different from the corresponding patch in the right example image. The tree construction of Section **??** finds an approximate nearest neighbor *using the SSD image metric*, so the change in appearance could have serious consequences – the closest patch to the query patch may not be the right match, because the query patch has changed appearance. Further, some patches should likely not vote, because they are not distinctive. Very many books will have, say, patches of uniform dark grey on their covers, so voting with such a patch is likely unhelpful and might even lead to errors.

The interest point construction of Section 8.2 is a powerful tool for dealing with these problems. Rather than using patches, find interest points in the example images, compute their representations, and build a tree using the representations. This is straightforward – each interest point representation is a vector of fixed size, just like each patch. Now query that tree using the representations of the interest points in the query image. The interest point representations were constructed to be stable under changes of lighting, scale, and orientation. Further, the interest

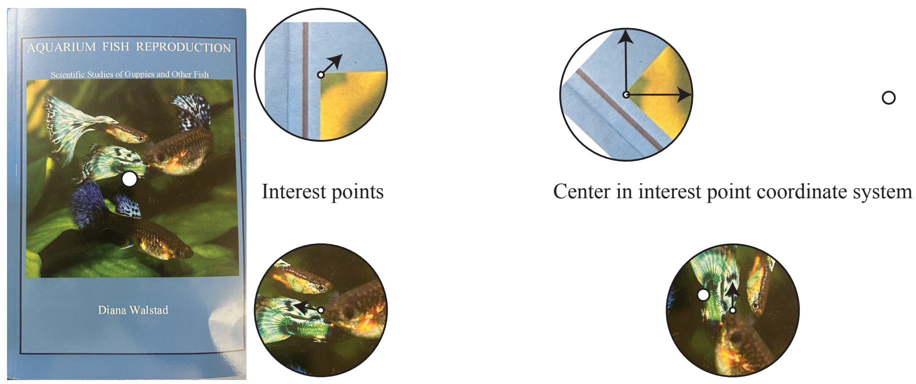

FIGURE 11.4:

points must be at least somewhat distinctive.

Each interest point in the input image could vote for more than one book. Pass the interest point down the tree. Now look at the leaf it lies in. The original strategy recorded one vote for the book that appeared most often in the leaf. This helps suppress the influence of interest points that are common, but means that only quite distinctive interest points can vote. Alternatively, you could divide the vote into fractions proportional to the number of times each book appears, and record fractional votes. So if "Decline and Fall" appears once, "Scoop" appears once, and "Put out more Flags" appears three times, then "Decline and Fall" and "Scoop" each get $1/5$ of a vote, and "Put out more Flags" gets $3/5$ of a vote. Now you may get the identity of a book right even if there is nothing particularly distinctive on its cover. Less helpfully, many titles will get small numbers of votes, and there is a bigger prospect of the wrong title getting too many votes.

11.3 ELEMENTARY DETECTION BY VOTING

The elementary classifier above is a classifier because it tells whether a book cover is present in an image. It can be improved into a detector, because the interest point construction yields *where* the book cover is.

If you build the instance detector using interest points, you will find it can be inaccurate. Part of the difficulty is that the same interest point can appear on many different book covers. For example, each large letter on a cover is likely to produce some interest points – in the worst case, an interest point on a query image might match every book with a 'T" on its cover, which isn't helpful. The current voting scheme looks only at what interest points are on the cover, but does not account for *where* they are. It is quite straightforward to do so by further voting, and the result is an elementary detector – the system can tell what book cover is present *and* where it is.

FIGURE 11.5:

11.3.1 Voting on Centers

Assume that each example image is cropped to the cover of the book, and contains nothing else, so the center of the cover is easy to find in the example images. When you construct an interest point, you construct a local image coordinate system (origin at corner, Section 8.2.1; scale from Section 8.2.2; and orientation from Section 8.2.3). For each interest point in the example image, you can record the location of the books center *in this interest point's coordinate system*, and insert this information in the tree with the interest point.

Now think about a query image of the book cover. Find an interest point in that query image, and match it using the tree. You can recover a predicted location of the center of the book from that interest point. It is just the location recorded in the tree, but now in the coordinate system of the interest point in the query image. Different interest points in the query image that agree on the name of the book should also agree on the location of the center of the book.

This observation increases the scope of voting considerably. A simple and very effective strategy is to censor votes. Collect all votes for a particular book. For each predicted center, check that there is another prediction (or two other predictions, and so on) of the center nearby. If there is, record a vote for that book. If there is not, the interest point does not vote. Finally, take the book with the largest number of votes. Notice that this reduces the chance that you misidentify the book, but might increase the chance that you label the book as "unknown".

Alternatively, you could think about voting in terms of an accumulator array (the practical obstacles to actually doing this should be obvious and should deter you). In principle, you could have a 3D accumulator array. Two dimensions are spatial, and the third is the identity of the book. You would pass every interest point detected in the image through the tree and vote for the book and location

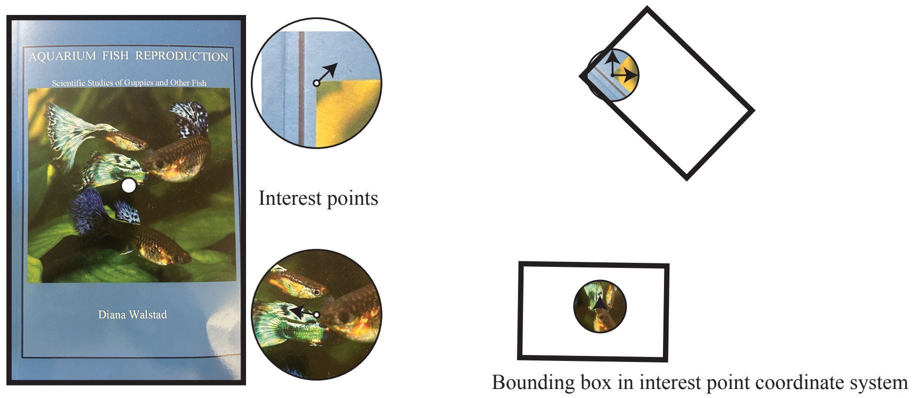

FIGURE 11.6:

associated with it. You would then analyze the accumulator array – this is like the voting procedure above; the votes are censored because votes for the wrong location of the center won't find one another in the accumulator array. You should think of this accumulator array as very complicated feature that describes the image. Rather than trying to build this, you should think of it as an example of the kind of image features that *could* be constructed. Chapter 41.2 offers much more efficient constructions of comparable features.

11.3.2 Recovering Location, Scale and Orientation

Matching an interest point tells you more than just where the center of the book might be. Take an interest point in an example image of a book cover. You could record the orientation and the bounding box of the book cover in the interest point's coordinate system – a total of five parameters, and use that. You could not use the accumulator (too many buckets, **exercises**), but you could use this information to censor votes. Alternatively, ignore the bounding box until you have determined what book is present. Now use the interest points that were allowed to vote for that book to determine the bounding box of the book present in the image. Alternatively, you could record the rotation, scale and aspect ratio of the book cover as well as the location of the center of the book (this is equivalent to the bounding box, **exercises**). Quite a useful detector can be built like this **exercises** .

11.4 MODIFYING THE TREE

A tree constructed using hierachical k-means may not be particularly good for these classification and detection tasks. The hierarchical k-means construction tends to split the data up so that leaves contain interest points that are similar to one another. A better tree would exploit the labels during construction. For example, it might have leaves that contain interest points that agree on label.

To illustrate the difference, think about a collection of books that all have a

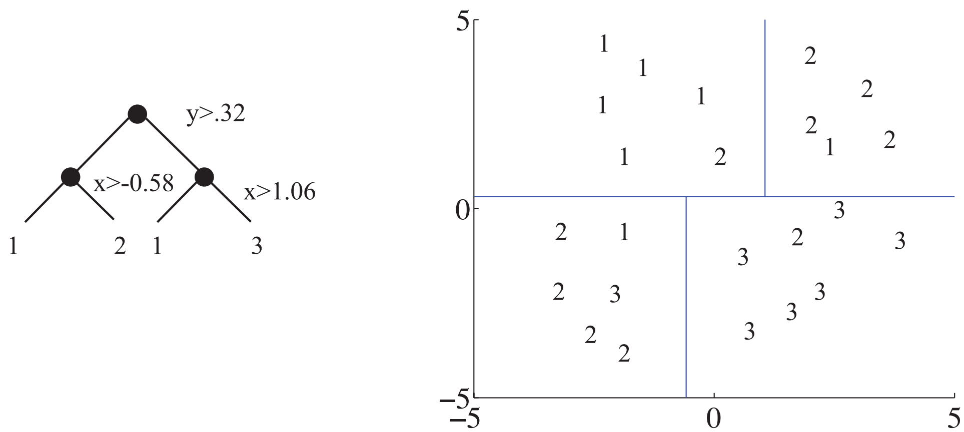

FIGURE 11.7: *The tree of Figure 10.3 divides feature space by choosing which of a set of centers is closest to a query point. This figure shows a straightforward decision tree, illustrated in two ways. The test is now a test of one of the dimensions against a threshold. The data points belong to three classes. On the* **left***, I have given the rules at each split, and labelled each leaf with the most common class in the leaf. On the* **right***, I have shown the data points in two dimensions, and the structure that the tree produces in the feature space.*

large face on the cover. Each will have an interest point at the inside and outside corner of each eye (say). These interest points will mostly look quite similar to one another, and might all end up in the same leaf using a hierarchical k-means tree. But some differences are more important than others. For example, eyelashes might have quite a small effect on the description of the interest point, and so interest points at eyes with small lashes may appear in the same leaf as interest points at eyes with large lashes. Ideally, the tree is constructed to exploit this small difference in appearance, because it has a big effect on identity. Ensuring that eyes with small eyelashes appear in different leaves than eyes with large eyelashes should help improve the accuracy of the tree.

It turns out that a powerful approach for building a tree incorporates a great deal of randomness. As a result, you get a different tree each time you 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.

11.4.1 Building a Decision Tree

Walking a data point down a tree is straightforward. Go to root and apply the following recursive procedure: if the node the point is at is a leaf, stop and report the leaf; otherwise, decide which child the point lies in, then recur. For the tree built with hierarchical k-means, each child has a center associated with it, and the point lies in the child whose center is closest. Building a different kind of tree is

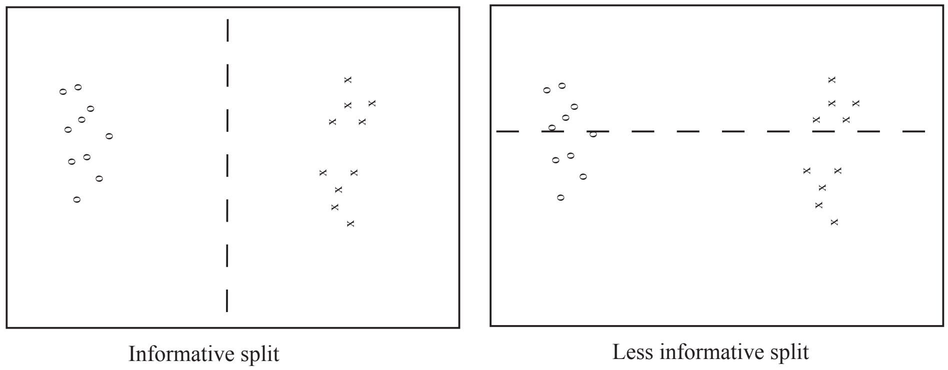

FIGURE 11.8: *Two possible splits of a pool of training data. Positive data is represented with an 'x', negative data with a 'o'. Notice that if we split this pool with the informative line, all the points on the left are 'o's, and all the points on the right are 'x's. This is an excellent choice of 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.*

just a matter of changing the procedure to choose the child. If the leaf is associated with a label, the tree is typically called a *decision tree*.

There are many algorithms for building decision trees. I will describe an approach chosen for simplicity and effectiveness; be aware there are others. I will always use a binary tree, because it's easier to describe and because that's usual (it doesn't change anything important, though). In the binary case, each node has a *decision function*, which takes data items and returns either 1 or -1.

Now think about the tree's effect on the training data. 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. Ideally, all the data items in each leaf have the same label, and there are not too many leaves.

Training the tree uses a straightforward algorithm. First, 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 (by a gross extension of metaphor, this is sometimes known as a *decision stump*). For this approach to work, one needs to be quite careful about the choice of threshold (next section). Surprisingly, being clever about the choice of *feature* doesn't seem add a great deal of value. I won't spend more time on other kinds of decision function, though there are lots.

Constructing the tree is a matter of starting with the whole dataset at the root, then recursively either splitting the dataset at that node or stopping and returning. If the node is split, the dataset arriving at the node is split too, with points in the left node going left and those in the right going right. The main questions are how to choose a split (next section), and when to stop splitting.

Stopping is relatively straightforward, and simple strategies for stopping work. It is hard to choose a decision function with very little data, so splitting must stop when there is too little data at a node. 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 stop anyhow at a fixed depth D of splits.

Here is a strategy for choosing a split. For some number of attempts, choose a single feature uniformly and at random. Set up a range of threshold values for that feature. Each represents a possible decision function (i.e. test the chosen feature against the chosen threshold). Now compute some measure of goodness for each of the decision functions, and keep the best. Experience shows this strategy is effective, with an appropriate measure of goodness.

Figure 11.8 shows two possible splits of a pool of training data. There are two classes ("positives" or 1 and "negatives" or -1). One split 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. Assume you know which child a data point lies in. The good case is good because you then require no more information to tell what its label is. The bad case is bad because you do require quite a lot more information to predict the point's label.

Figure **??** 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 you 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.

All this suggests a procedure to score how good the split is. In the uninformative case, knowing that a data item is on the left (or the right) does not tell you much more about the data than you already knew. This is because $p(1|\text{left pool, uninformative}) = 2/3 \approx 3/5 = p(1|\text{parent pool})$ and $p(1|\text{right pool, uninformative}) = 1/2 \approx 3/5 = p(1|\text{parent pool})$. For the informative pool, 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 your uncertainty about what class the data item belongs to is significantly reduced if you know whether it goes left or right. To choose a good threshold, you need to keep track of how informative the split is.

11.4.2 Choosing a Split with Information Gain

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 $\mathcal{C}$ 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

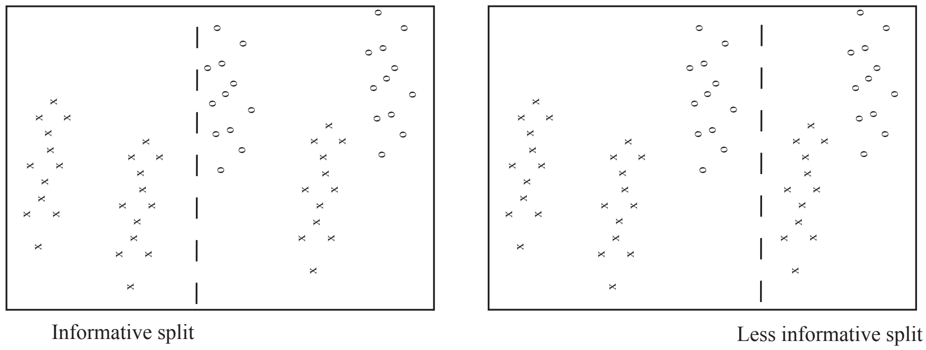

FIGURE 11.9: *Two possible splits of a pool of training data. Positive data is represented with an 'x', negative data with a 'o'. Notice that if you split this pool with the informative line, all the points on the left are 'x's, and two-thirds of the points on the right are 'o's. This means that knowing which side of the split a point lies would give you 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 entropy $H(\mathcal{C})$ 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, $H(\mathcal{P}_l)$ bits are needed; for an item in the right pool, $H(\mathcal{P}_r)$ bits are needed. If the parent pool is split, you will 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, you 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 the parent pool is split. A good split is one that results in left and right pools that are informative. In turn, you should need fewer bits to classify once you have split than before the split. You can see the difference

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

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. All this yields 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: 11.1 *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:

- 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 using the procedure of box 11.2.

Procedure: 11.2 *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 compue 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.

11.4.3 Forests