CHAPTER 13

Voting and its Applications

Again and again, one needs to choose the "best" model to explain or represent some tokens. One version of this problem is choosing a line to represent some tokens. Another version is deciding which instance – for example, which book cover – is present in an image. Now imagine you are given a set of points $\mathbf{x}_i$ in one image and a set of points $\mathbf{y}_i$ in another, and must find a transformation $\mathcal{T}$ so that $\mathcal{T}(\mathbf{x}_i)$ is close to $\mathbf{y}_i$. To see this as choosing a model, think of pairs $(\mathbf{x}_i, \mathbf{y}_i)$ as tokens and the transformation as the model.

There are two master recipes to solve versions of this problem. In one recipe, each token votes in some way, and the model with the most votes is chosen. In another, you set up an optimization problem describing the "best" model, then search for a solution. This chapter describes voting methods. These methods are now mostly not used in isolation, but can be quick and efficient solutions on occasion.

13.1 REPRESENTING LINES AND PLANES

There are a number of representations of lines and planes. I will describe a set of useful constructions and facts for lines; extending these to planes is straightforward **exercises** . It is usual to write (x_1, x_2) for points in 2D, because you can then extend to 3D without much difficulty.

Useful Fact: *Do not represent a line as the set of points(x, y) where $y = ax + b$ or a plane as the set of points (x, y, z) where $z = ax + by + c$. Vertical lines – where x is constant – and vertical planes – where $ax+by+c$ is a constant – cannot be represented in this form.*

Useful Fact: *Represent a line as the set of points $\mathbf{x} = (x_1, x_2)^T$ where $ax_1 + bx_2 + c = 0 = \mathbf{a}^T\mathbf{x} + c$ and a plane as the set of points (x_1, x_2, x_3) where $ax_1 + bx_2 + cx_3 + d = 0 = \mathbf{a}^T\mathbf{x} + d$. A tuple (a, b, c) corresponds to a line as long as not all elements are zero. A tuple (a, b, c, d) corresponds to a plane, as long as not all elements are zero.*

Useful Fact: *The line represented by* $(\lambda a, \lambda b, \lambda c) = (\lambda \mathbf{a}, \lambda c)$ *is the same as the line represented by* $(a, b, c) = (\mathbf{a}, c)$ *for* $\lambda \neq 0$*. This means that many tuples represent the same line. Choosing to avoid this ambiguity by requiring one element of the tuple to 1 means that you cannot represent some collection of lines. For example, the family* $(u, v, 1)$ *omits any line through the origin* **exercises**

Useful Fact: *The perpendicular distance from a point* $\mathbf{x}$ *to a line* $(\mathbf{a}, c)$ *is given by*

$$abs(\mathbf{a}^T\mathbf{x} + c) \qquad \text{if} \qquad \mathbf{a}^T\mathbf{a} = 1.$$

In my experience, this fact is useful enough to be worth memorizing.

Useful Fact: *The normal of a line represented as* $(a, b, c) = (\mathbf{a}, c)$ *is given by*

$$\frac{\mathbf{a}}{\sqrt{\mathbf{a}^T\mathbf{a}}}$$

Useful Fact: *If you represent a line by the tuple* $(\cos\theta, \sin\theta, r)$*, where* $0 \leq r$ *and* $0 \leq \theta < \pi$*, then all lines are represented, and there is exactly one* (θ, r) *that corresponding to a given line***exercises** *. For this representation,* r *is the perpendicular distance from the line to the origin and* θ *is the orientation of the line (meaning that the vector* $(\sin\theta, -\cos\theta)^T$ *points along the line).*

13.2 THE HOUGH TRANSFORM

The *Hough transform* is a general voting procedure that applies to a wide range of problems.

Procedure: 13.1 *Hough transform: Master recipe*

This recipe applies when you want to find a structure in a set of tokens. Allow each token to vote for *all* the structures that it could support. Then the structure with the most votes is the one you want.

Making this recipe concrete requires a little work (below). This idea is very seldom used directly for any problem, but it lies at the root of a wide range of ideas and is so worth understanding with an example.

13.2.1 Finding a Line with a 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, but there are many tokens on each line (so just 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.

Represent a line by the tuple $(\cos\theta, \sin\theta, r)$, where $0 \leq r$ and $0 \leq \theta < \pi$. Because the image has a known size, there is some R such that, if $r > R$, the lines are too far away from the origin for any token to appear in the image. *Line space* is the set of (θ, r) such that $0 \leq r \leq R$ and $0 \leq \theta < \pi$. A point in 2D given by $(x_1, x_2)^T$ could lie on any line such that $r = -x_1 \cos\theta + x_2 \sin\theta$. Equivalently, a point in 2D corresponds to a curve in line space.

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 13.1).

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 13.1).

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

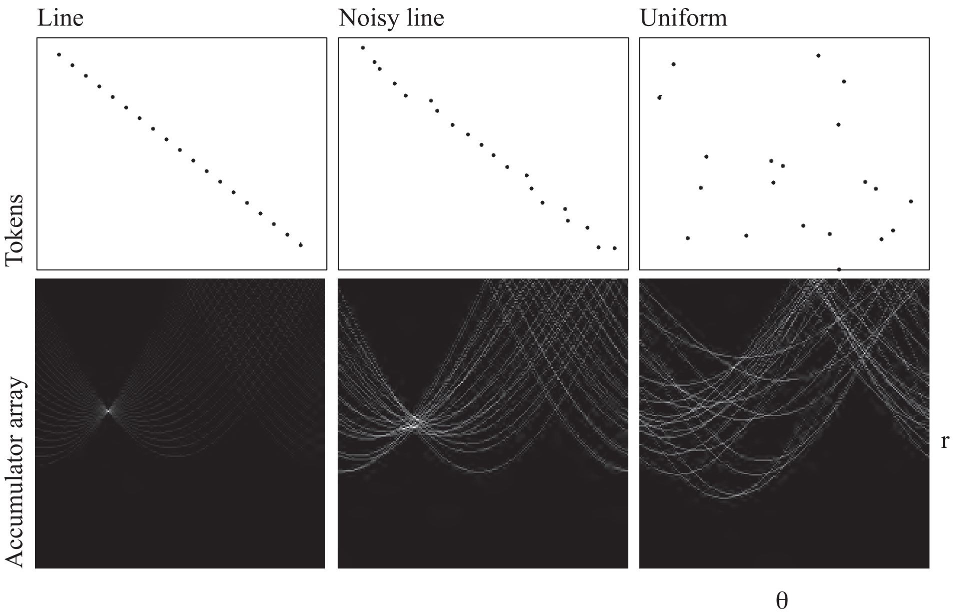


FIGURE 13.1: *The Hough transform maps each point like token to a curve of possible lines (or other parametric curves) through that point. These figures illustrate the Hough transform for lines. The* **top row** *shows tokens, and the* **bottom row** *shows the corresponding accumulator arrays (the number 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).*

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.

Fitting circles, planes or spheres following this recipe is just a matter of how one sets up the accumulator array and how one votes (**exercises**). However, issues of dimension become a serious problem **exercises** .

Remember this: *The Hough transform identifies a structure by allowing each token to vote for all possible structures associated with the token, then choosing the structure with the most votes. This recipe is easily made concrete for the case of lines; circles, planes, ellipses, spheres and so on follow, but present difficulties with dimension. Noise creates serious problems finding the structure with most votes. The general idea – that noisy or ambiguous predictions can be improved by voting – is extremely influential.*

13.3 CLASSIFICATION AND DETECTION 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 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 4.3.3, but results would be poor. The same book covers will look different when viewed in slightly different lighting (Figure 13.2), so just matching part of an image with the costs of Section 4.3.1 is unlikely to work well (when the chicken of Section 4.3.3 got darker or changed position, the match became worse).

13.3.1 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. The interest point construction of Section 8.2 is a powerful tool for dealing with these problems. The interest point representations were constructed to be stable under changes of lighting, scale, and orientation. Further, the interest points must be at least somewhat distinctive.

You can now exploit the tree construction of Section **??** to vote. Voting will be by passing an interest point down a tree built using the following steps. Give each different book a unique number. Build a collection of interest points by taking each known book cover image, and finding its interest points. Build a tree from the interest point descriptors as in Section **??**. This works because an interest point descriptor is a vector of fixed length, which is all the tree-building procedure requires. Build the tree so that each leaf of the tree contains relatively few interest points (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, find the interest points for the image and compute their descriptors. Pass each descriptor down the tree and record a vote for the book in its leaf. Choose the book with the largest number of votes. If that book has enough votes, decide the query image contains that book; if

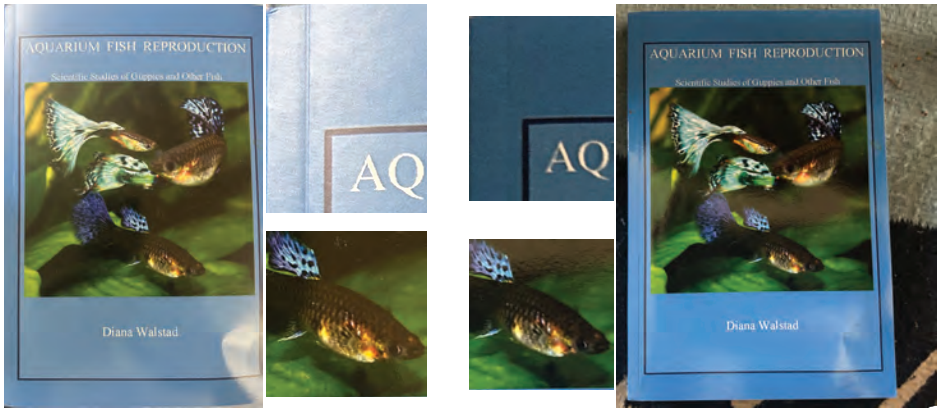

AQUARIUM FISH REPRODUCTION
Scientific Studies of Guppies and Other Fish
Diana Walstad
AQ
AQ
AQUARIUM FISH REPRODUCTION
Scientific Studies of Guppies and Other Fish
Diana Walstad

FIGURE 13.2: *On the* **left***, an example image of a book cover, with two patches cropped from the example. On the* **right***, a (cropped) image of the book cover, with instances of those two patches cropped from the image. Notice how the color has changed in the book images. This effect is caused (at least!) by glossy reflections from the plastic coating on the surface of the book in the example image. One patch has had a significant color change, and likely won't match at the pixel level; the other might well match. A SIFT feature representation of the patches would certainly match.* Image credit: *Cover of Diana Walstad's fascinating book on reproduction in aquarium fish, mostly guppies.*

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.

The main question here is how you vote. When an interest point arrives at a leaf, you could record one vote for the book that is most common in that leaf. Alternatively, you could record a fraction of a vote each for each book present in the leaf, 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. It is helpful if the fractions add up to one so that common interest points do not dominate the voting **exercises** . 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.

As another alternative, an interest point could record a vote only when the margin in its leaf is large enough – that is, the book that has the most interest points 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 modify the tree construction.

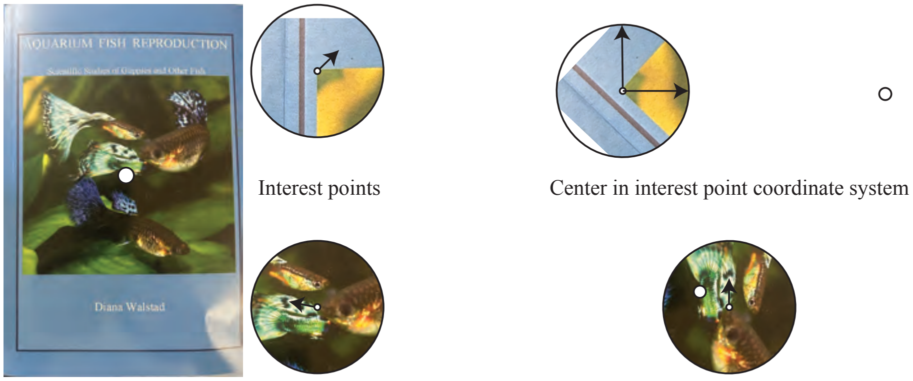

AQUARIUM FISH REPRODUCTION
Scientific Studies of Guppies and Other Fish
Diana Walstad
Interest points
Center in interest point coordinate system

FIGURE 13.3: *Interest points can vote on the location of the center of the book, because each interest point carries its own coordinate system. On the* **left**, *the cover image, with the center of the cover marked as an empty circle.* **Center** *shows two interest points, with their coordinate systems attached. I have marked the origin of the coordinate system with another circle, and the orientation with an arrow. On the* **right**, *I show each interest point rotated so that the coordinate systems line up, and mark the location of the center of the book in each point's coordinate system. For the* **top** *interest point, the book center is quite far away from the origin, but for the* **bottom** *interest point, it is quite close.* Image credit: *Cover of Diana Walstad's fascinating book on reproduction in aquarium fish, mostly guppies.*

13.3.2 Voting on Centers

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.

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

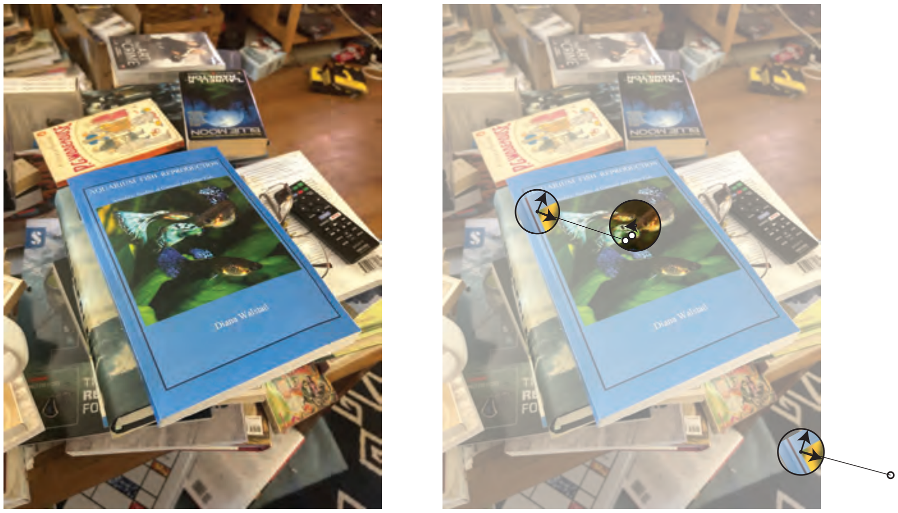


FIGURE 13.4: *Matching to a book cover is conceptually straightforward. Find each interest point in the scene. Pass each down the tree to find what covers it might match and where it places the center of the book* in the scene image. *If enough interest points agree as to a center location, allow them to vote on their titles. Here, three interest points have been detected; two agree on the location and title, and the third is ignored because nothing agrees with it.* Image credit: *Cover of Diana Walstad's fascinating book on reproduction in aquarium fish, mostly guppies.*

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 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

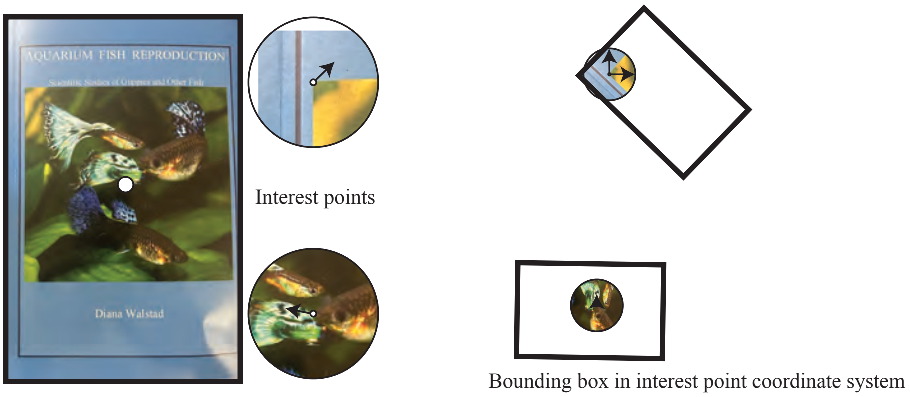

AQUARIUM FISH REPRODUCTION
Scientific Studies of Guppies and Other Fish
Diana Walstad
Interest points
Bounding box in interest point coordinate system

FIGURE 13.5: *Interest points can vote on center location, orientation and scale of the book cover. This information yields the outline of the book, so you can think of interest points each carrying a vote as to the book cover outline. On the* **left**, *the cover image, with the outline marked in heavy lines.* **Center** *shows two interest points, with their coordinate systems attached. On the* **right**, *I show each interest point rotated so that the coordinate systems line up. I have marked the outline of the cover,* in the interest point's coordinate system *with heavy lines.* Image credit: *Cover of Diana Walstad's fascinating book on reproduction in aquarium fish, mostly guppies.*

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 21.3 offers much more efficient constructions of comparable features.

13.3.3 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. As another alternative, 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** .

Remember this: *Deciding whether an image contains an instance of an object is instance classification; instance detection is reporting where that instance is in the image. Voting on interest points can be used for both instance classification and instance detection. You can determine whether: interest points agree on what is in the image; interest points agree on where its center is; interest points agree on what its orientation is; and interest points agree on the scale of the object. There are a variety of voting procedures. It can be particularly useful to censor votes.*

13.4 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. But interest point descriptors are constructed so they do not change much between different images of the same thing. This means that small changes in descriptor that result in a change of label are important and reliable. You could expect to build a more useful tree using the labels. Such trees are typically called *decision trees*.

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

You can see how to build a different tree by thinking about how to walk a point down a tree. Go to the 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 just a matter of changing the procedure to choose the child that a point should lie in.

You want a tree where most of the data items in each leaf have the same label, and where there are not too many leaves. If there are many different labels in each leaf, the voting may be indecisive. If there are too many leaves, a new interest point may not arrive at the right leaf – a failure of generalization, a topic discussed in greater detail in Chapter 20. It is far too much trouble to build an optimal tree. Instead, a powerful approach for building a tree incorporates a great

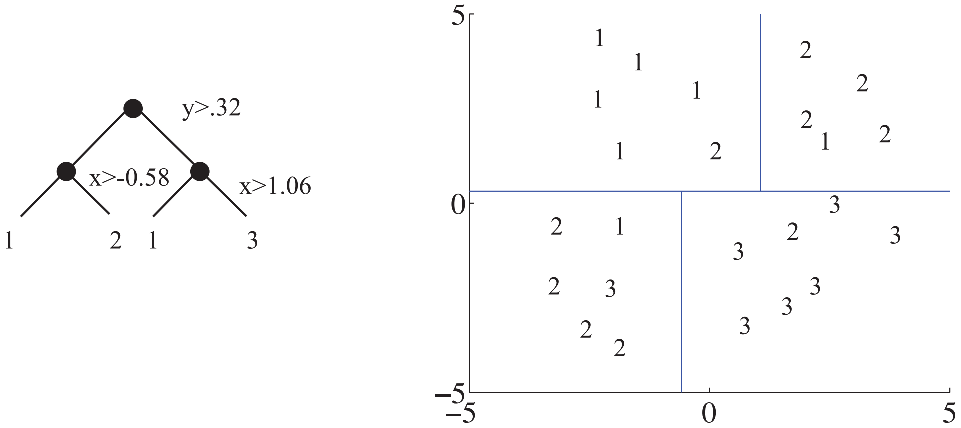


FIGURE 13.6: *The tree of Figure 10.9 divides feature space by choosing which of a set of centers is closest to a query point. This figure shows a straightforward decision tree, illustrated in two ways. The test is now a test of one of the dimensions against a threshold. The data points belong to three 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.*

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.

13.4.1 Building 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 is easier to describe and because it is usual (it doesn't change anything important, though). In the binary case, each node that isn't a leaf has a *decision function*, which takes data items and returns either 1 or -1 (for left child or right child).

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.

Building the tree uses a straightforward algorithm. First, choose a class of decision functions to use at each node. 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

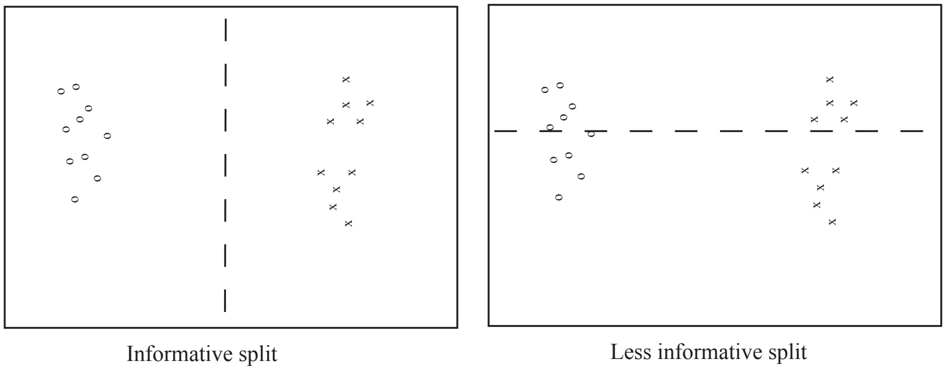


FIGURE 13.7: *Two possible splits of a pool of training data. Positive data is represented with an 'x', negative data with a 'o'. Notice that if we split this pool with the informative line, all the points on the left are 'o's, and all the points on the right are 'x's. This is an excellent choice of 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.*

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.

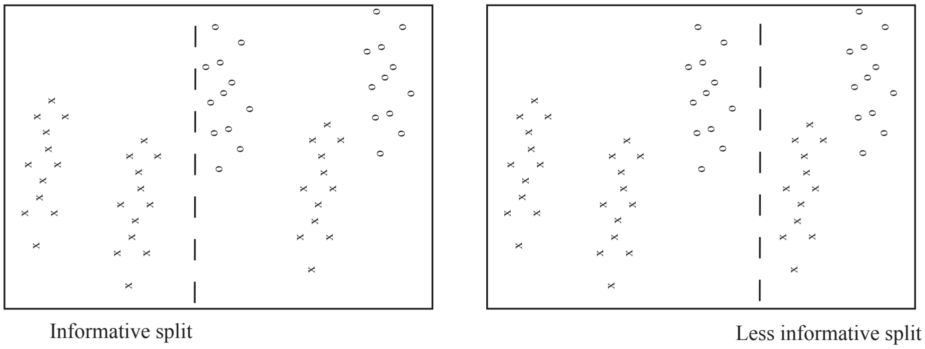
Constructing the tree is a matter of starting with the whole dataset at the root, then recursively either splitting the dataset at a node or stopping and returning. If the node is split, the dataset arriving at the node is split too, with points in the left side going left and those in the right going 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.

13.4.2 Choosing a Split

Figure 13.7 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



Informative split

Less informative split

FIGURE 13.8: *Two possible splits of a pool of training data. Positive data is represented with an 'x', negative data with a 'o'. Notice that if 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.*

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 13.8 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 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.

13.4.3 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 the entropy $H(\mathcal{C})$ of the pool $\mathcal{C}$ is

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

(where you should interpret $0 \log_2 0 = 0$). 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: 13.2 *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 13.3.

Procedure: 13.3 *Splitting a 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.

13.4.4 Building and Evaluating a Decision Forest

A single decision tree can 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.

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: 13.4 *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, 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.

Procedure: 13.5 *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.

13.4.5 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.

Procedure: 13.6 *Classification with a decision forest*

Given a test example **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.

Resources: Simple classification with decision forests

- **Simple image classification datasets:**
  - **MNIST** is a dataset of 60,000 training and 10, 000 test examples of isolated handwritten digits, originally as binary images. You can find it in a number of places (search!); one is `https://www.kaggle.com/datasets/hojjatk/mnist-dataset`. It was originally constructed and popularized by Yann Le Cun, and has had fantastic influence. It is now mostly used for checking methods (if something won't work on MNIST, it probably won't work at all).
  - **Fashion-MNIST** is a dataset of 60, 000 28x28 grayscale images in 10 classes, with 6000 images per class. There are 50,000 training and 10, 000 test images. The images depict fashion items, and the dataset is intended to replace MNIST as resource for checking methods. It was created by *****
  - **CIFAR-10** is a dataset of 60, 000 32x32 color images in 10 classes, with 6000 images per class. There are 50, 000 training images and 10, 000 test images. The dataset was created by Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. Find it at `https://www.cs.toronto.edu/~kriz/cifar.html`.
  - **CIFAR-100** is a dataset of 60, 000 32x32 color images in 100 classes, with 600 images per class. There are 50, 000 training images and 10, 000 test images. The dataset was created by Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. Find it at `https://www.cs.toronto.edu/~kriz/cifar.html`.
  - **Food-101** is a dataset of *** color images of food items in 101 classes, with *** images per class. The dataset was created by Lukas Bossard, Matthieu Guillaumin and Luc Van Gool. Find it at `https://www.kaggle.com/datasets/dansbecker/food-101`.
- **Building decision forests:**
  - OpenCV supports decision trees and decision forests, but I haven't found a tutorial. There is a very good worked example, linked to the book "Machine Learning for OpenCV" by Michael Beyeler at `https://github.com/mbeyeler/opencv-machine-learning/blob/master/notebooks/10.03-Using-Random-Forests-for-Face-Recognition.ipynb`.
  - **Scikit-learn** will build decision forests for you; see `https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html` for the function and examples at `https://scikit-learn.org/stable/modules/ensemble.html#forest`.
  - **XGboost** is a large and very efficient library for building decision forests trained using a procedure I have not described (gradient boosting). However, XGboost will train a decision forest (largely) as described in the text, see `https://xgboost.readthedocs.io/en/stable/tutorials/rf.html`. XGboost has a well-deserved reputation for speed and efficiency.

Remember this: *A decision tree is built using labels as well as features. Splitting nodes by testing a feature against a threshold yields effective decision trees. The feature and threshold are chosen by a small search over a set of randomly chosen features. Choosing a split using information gain is effective. Each time you build a tree like this, you will get a different tree. Individual trees tend not to be great classifiers, but voting across a forest can produce a very good classifier. All the voting procedures described for a tree built using hierachical k-means apply to a decision forest, with minimal changes*

13.5 YOU SHOULD

13.5.1 remember these definitions:

13.5.2 remember these facts:

Do not represent lines or planes like this 224
Represent lines and planes like this 224
Representing lines by triples involves an ambiguity 225
A formula for perpendicular distance from point to line 225
The normal of a line. 225
Representing a line by angle and distance can be convenient. 225
Hough transform summary. 227
Voting on interest points can be used for both instance classification and instance detection. 233
All the voting procedures described apply to a decision tree built using labels and features. 242

13.5.3 remember these procedures:

Hough transform: Master recipe 226
Building a decision tree: overall 238
Splitting a feature 238
Building a decision forest 239
Building a decision forest using bagging 239
Classification with a decision forest 240

13.5.4 use these resources:

Resources for decision forests 242

13.5.5 be able to:

- Apply a Hough transform to find lines.
- Apply a simple voting method for instance classification and detection.
- Explain why voting is helpful and why censoring votes can be useful.

EXERCISES

QUICK CHECKS

13.1. What is the normal of a plane represented by $(\mathbf{a}, d)$?

13.2. Why does the family of lines given by $(u, v, 1)$ omit any line through the origin?

13.3. What lines are missing from the family of lines given by $(1, v, w)$?

13.4. You are given a line (a, b, c). Show there is exactly one (θ, r) in $0 \leq r < 0$ and $0 \leq \theta < \pi$ so that the line can be represented as $(\cos\theta, \sin\theta, r)$.

13.5. You are given a line (a, b, c). Show there are exactly two (θ, r) in $0 \leq r < 0$ and $0 \leq \theta < 2\pi$ so that the line can be represented as $(\cos\theta, \sin\theta, r)$. Interpret these two solutions.

13.6. Imagine you wish to fit circles of fixed, known radius with a hough transform. What is the dimension of the accumulator array?

13.7. Imagine you wish to fit ellipses with a hough transform. What is the dimension of the accumulator array?

13.8. Imagine you wish to fit spheres with a hough transform. What is the dimension of the accumulator array?

13.9. You wish to fit curves with a hough transform. The accumulator array is d dimensional. You want to have n bins along each axis. How many bins are there in total? What problems might occur if d is big?

13.10. Section 13.3.1 has: "It is helpful if the fractions add up to one so that common interest points do not dominate the voting." Explain.

13.11. Section 13.3.1 suggests you could censor votes using the orientation and the bounding box of the book cover in the interest point's coordinate system. How would this work?

13.12. Compute the entropy for a pool of data with eight classes and the same number of data items in each class.

13.13. Compute the entropy for a pool of data with eight classes, where all the data is in one class.

13.14. What is the information gain for each of the splits of Figure 13.7?

LONGER PROBLEMS

13.15. Section 13.4.3 says: "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." Prove this.

PROGRAMMING EXERCISES

13.16. Build various simple image classifiers using interest points and the ideas of Section **??** and Section 13.4. Use the Food-101 dataset (because the images are reasonably sized – interest points for a 28x28 image aren't that helpful). Use 80% of the data for each class to build the forest, and evaluate the classification using the remaining 20% of the data. To evaluate, use the error rate – the fraction of classification attempts that are wrong (lower is better!).

(a) Build a single decision tree using interest points and hierarchical k-means to build the tree. How well can you get this to work? Is your procedure better than simply labelling the test data at random?

(b) Replace your decision tree with one built using the procedures of Section 13.4. Did the error rate improve?

(c) Replace your decision tree with a decision forest built using a package (I recommend using XGboost if you can). How well can you get it to work?