Voting and its applications

Recall:

Idea: vector quantization and voting

There are a variety of uses for voting

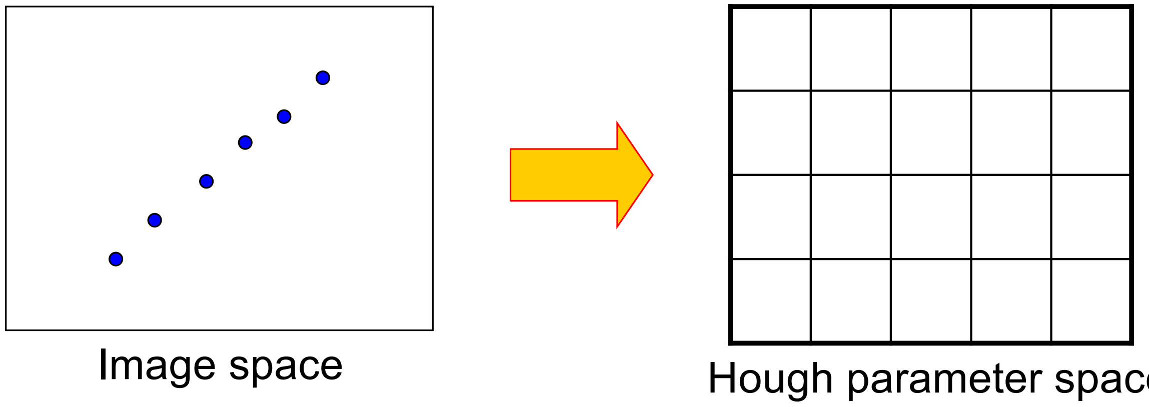
Voting for lines

- Data: $(x_1, y_1), \ldots, (x_n, y_n)$
- Line equation: $y = m\ x + b$

Parameters of line

Hough transform

- Possibly the earliest voting scheme – but still useful!
  - Discretize parameter space into bins
  - For each feature point in the image, put a vote in every bin in the parameter space that could have generated this point
  - Find bins that have the most votes



P.V.C. Hough, *Machine Analysis of Bubble Chamber Pictures*, Proc. Int. Conf. High Energy Accelerators and Instrumentation, 1959

Hough transform

- What does a **line** in the image space correspond to?
  - A **point** in the parameter space

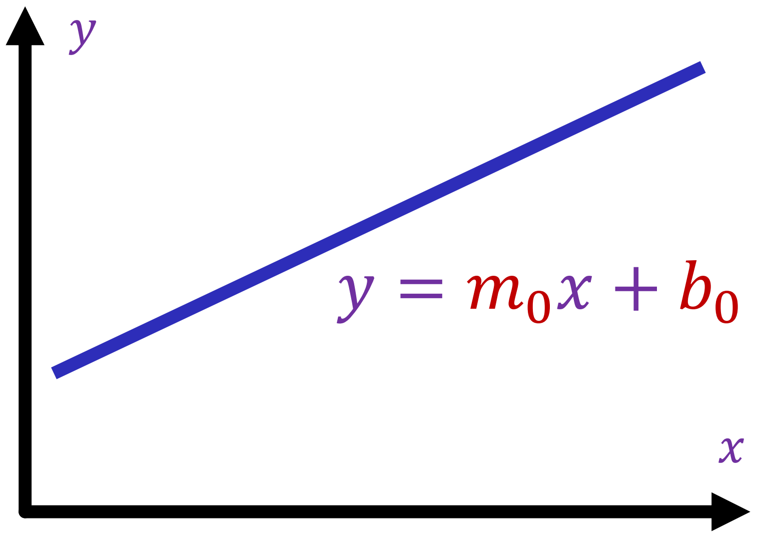
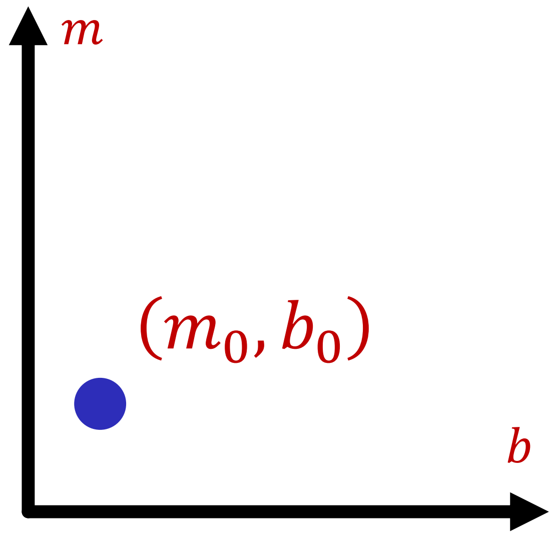


Image space



Parameter space

Hough transform

- What does a **point** in the image space correspond to?
  - A **line** in the parameter space: all (m, b) that satisfy $-b = x_0 m - y_0$

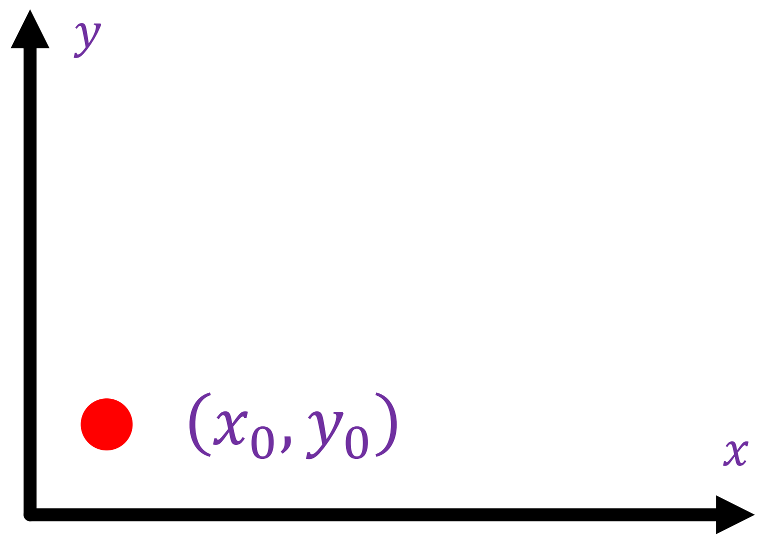
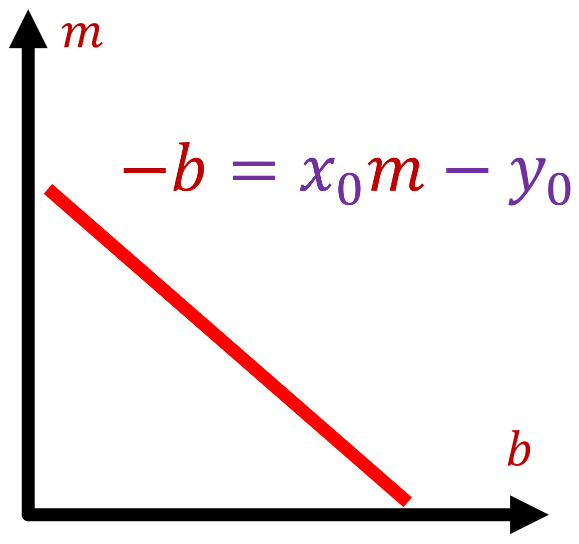


Image space



Parameter space

Hough transform

- What about **two points** in the image space?
  - A **point** in the parameter space, corresponding to the unique line that passes through both points

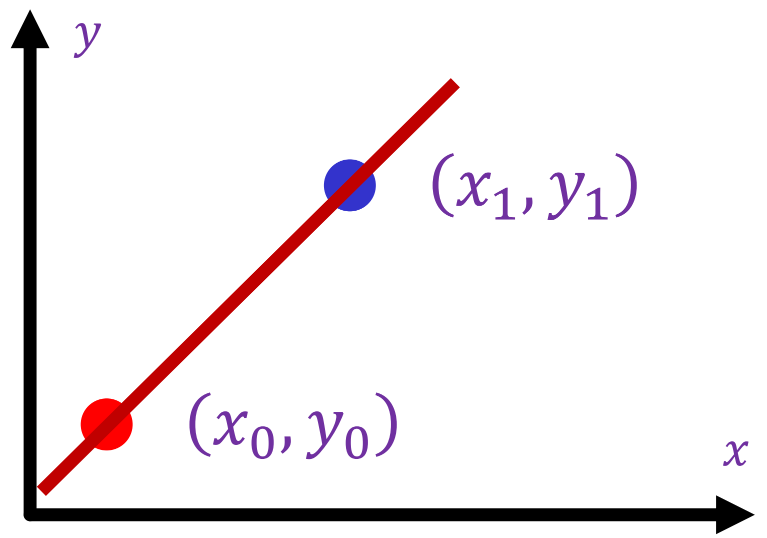
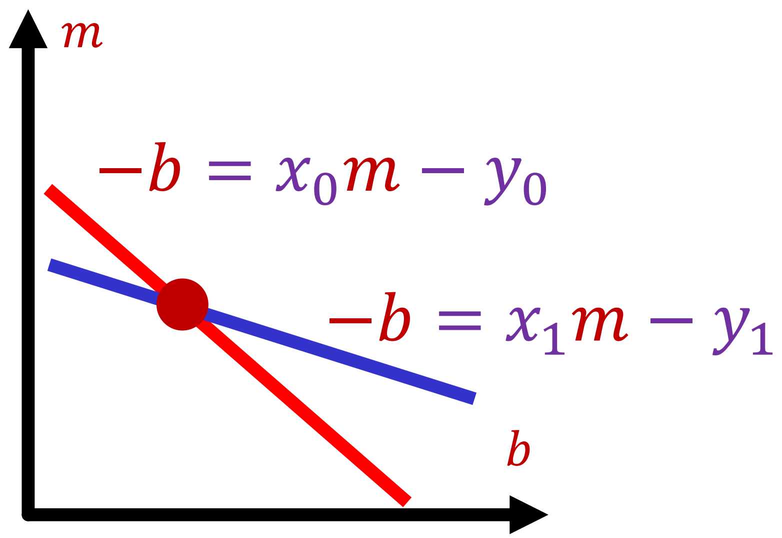


Image space



Parameter space

Hough transform

- What about **many points** in the image space?
  - Plot all the lines in the parameter space and try to find a spot where a large number of them intersect

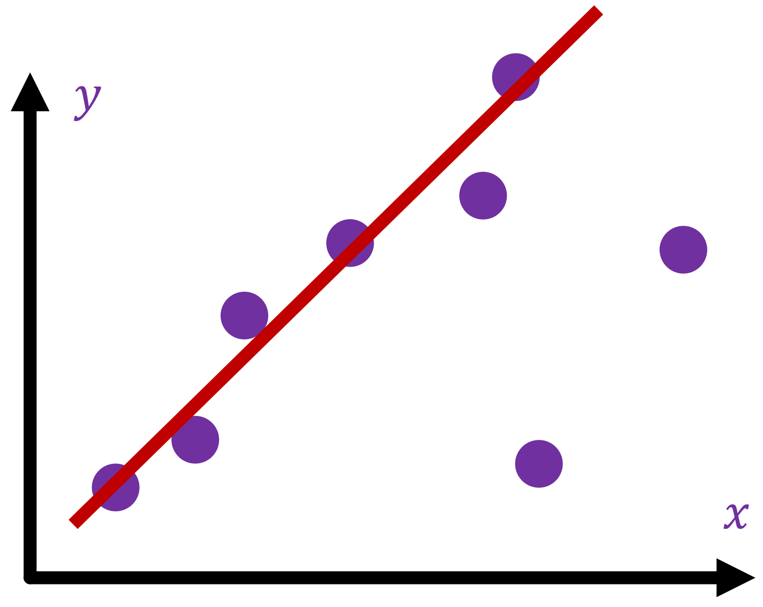
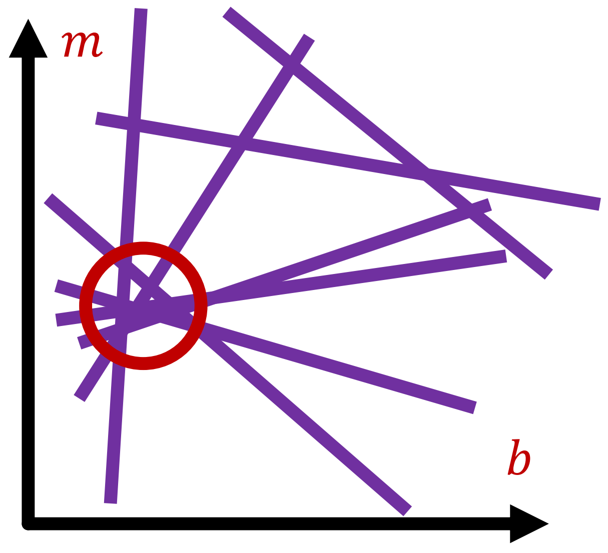


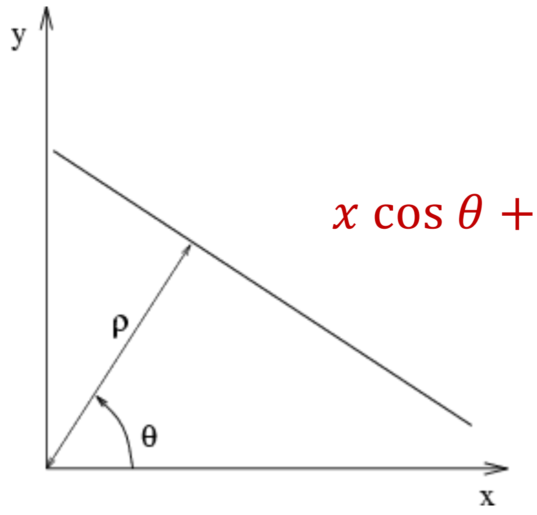
Image space



Parameter space

Parameter space representation

- In practice, we don't want to use the (m, b) space!
  - Unbounded parameter domains
  - Vertical lines require infinite m
- Alternative: *polar representation*
  - Each image point (x, y) yields a *sinusoid* in the (θ, ρ) parameter space



$$x \cos\theta + y \sin\theta = \rho$$

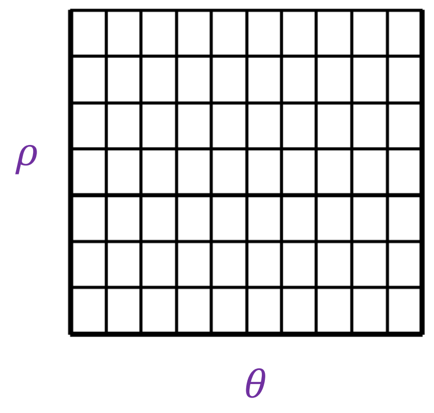
Algorithm outline

- Initialize accumulator H to all zeros
- For each feature point (x, y)
  - For $\theta = 0$ to 180

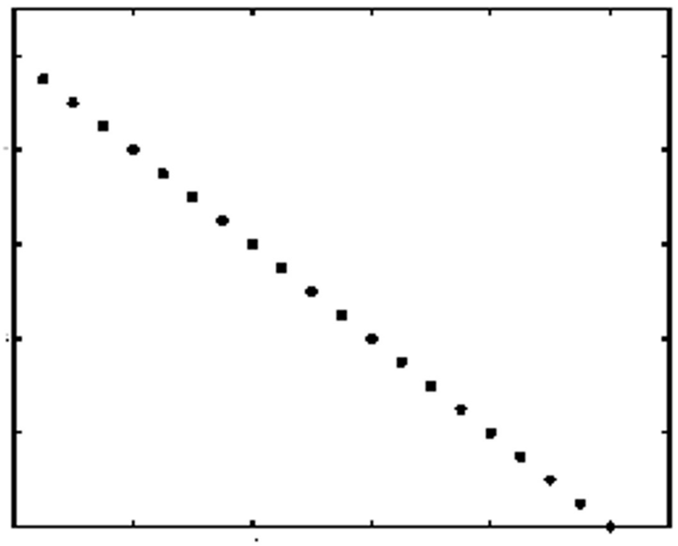
    $\rho = x \cos\theta + y \sin\theta$
    $H(\theta, \rho) \mathrel{+}= 1$

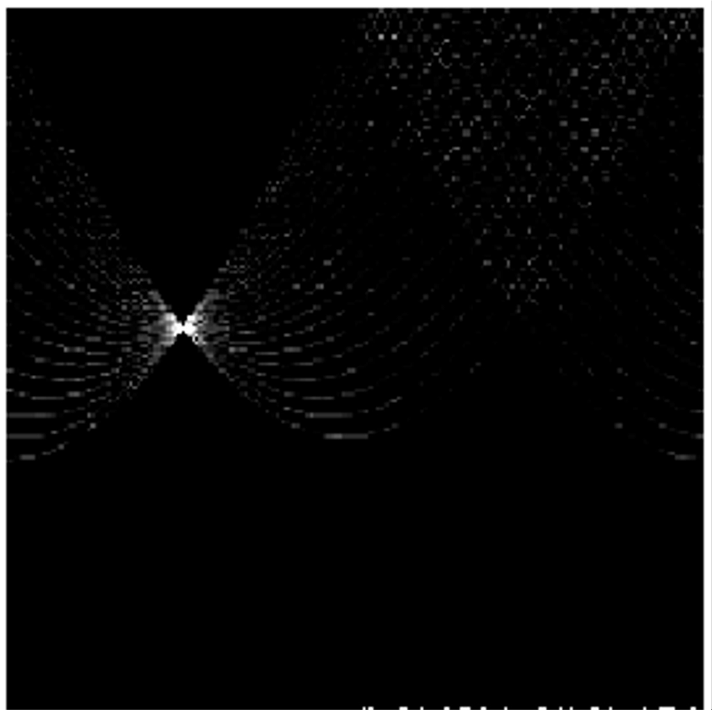
- Find the value(s) of (θ, ρ) where $H(\theta, \rho)$ is a local maximum (perform NMS on the accumulator array)
  - The detected line in the image is given by $\rho = x \cos\theta + y \sin\theta$

H: accumulator array (votes)



Basic illustration

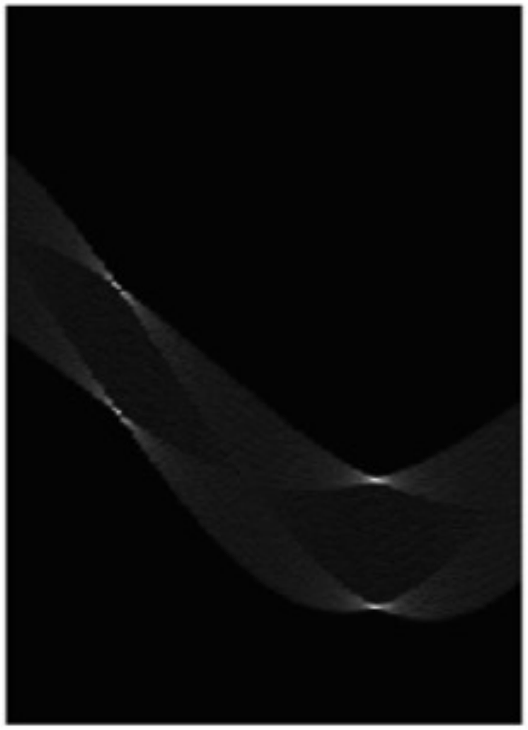

features


votes

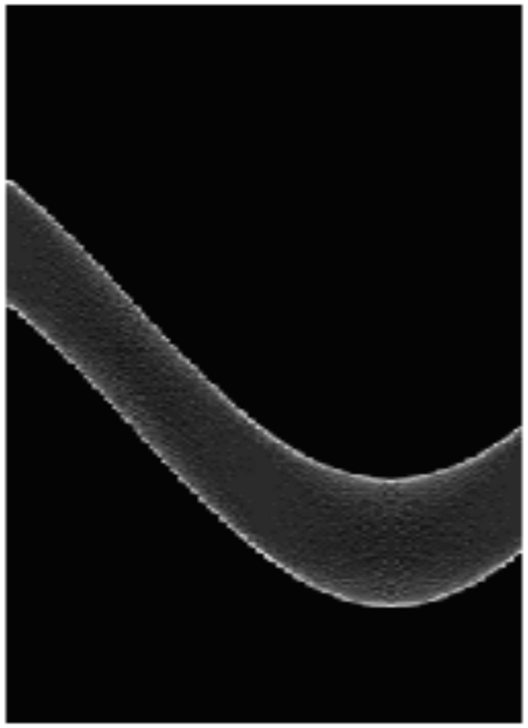
Hough transform demo

Other shapes

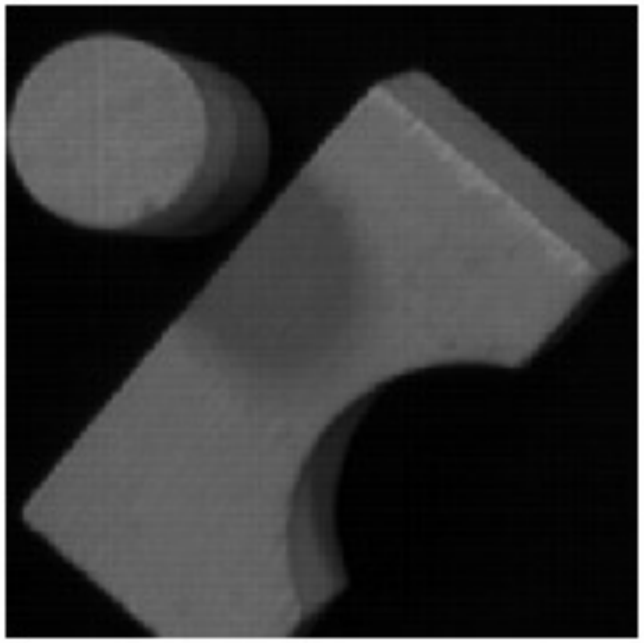
Square

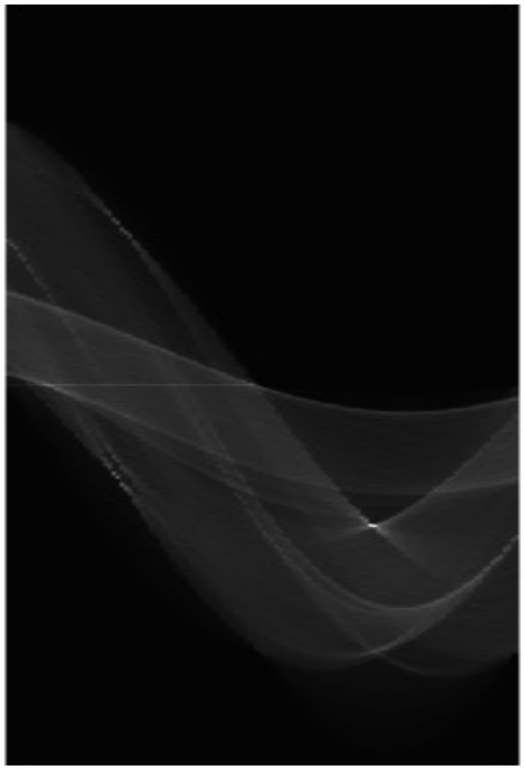


Circle

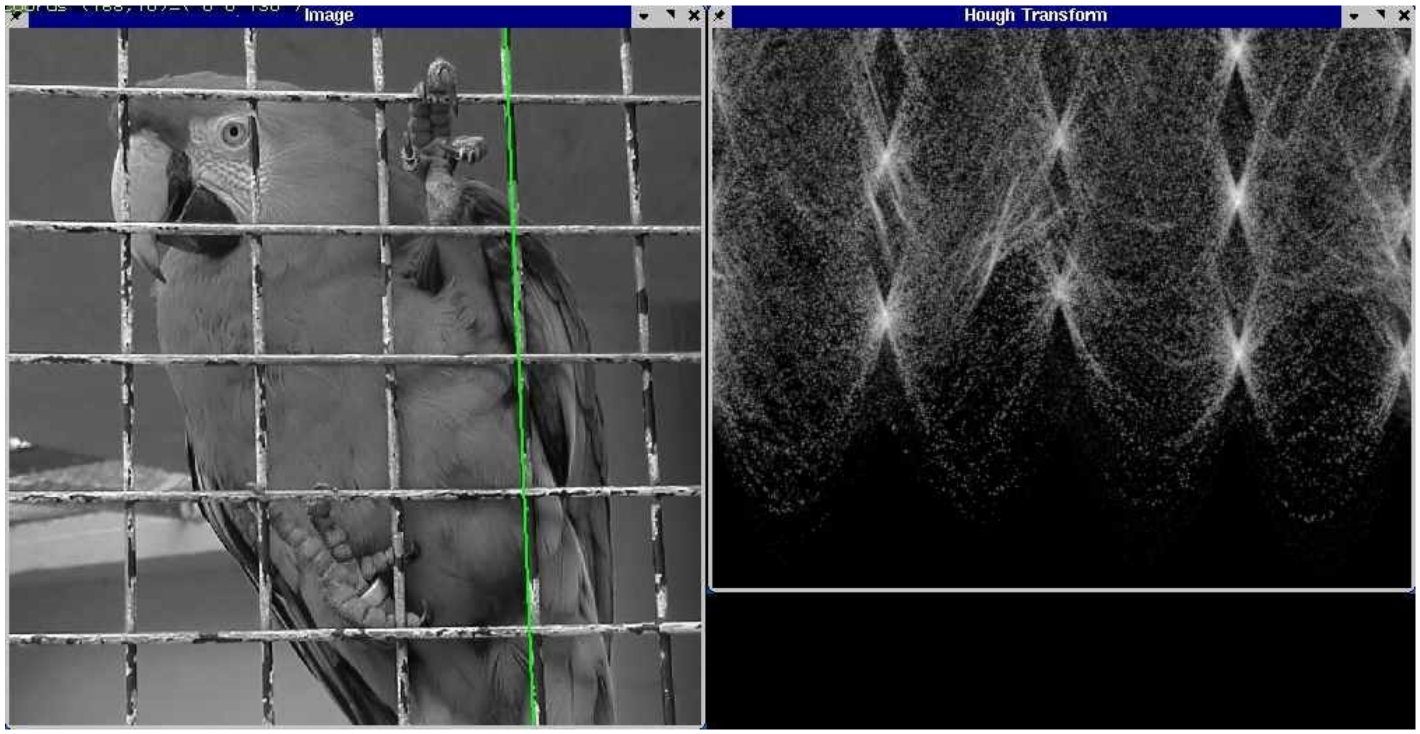


Several lines



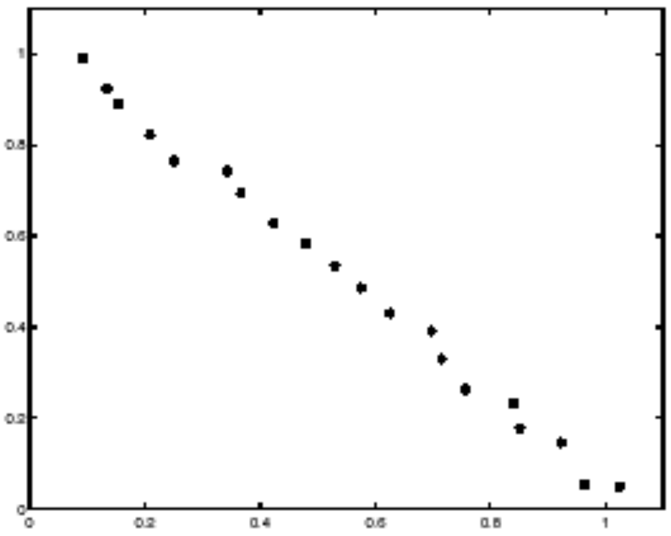


A more complicated image



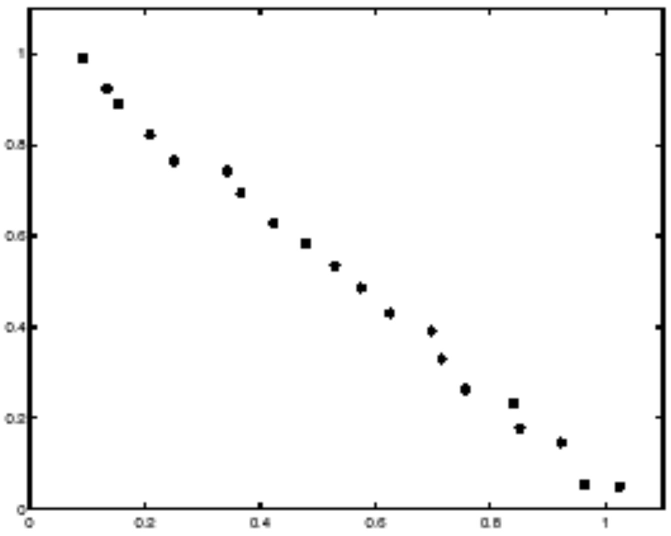
Source

Effect of noise

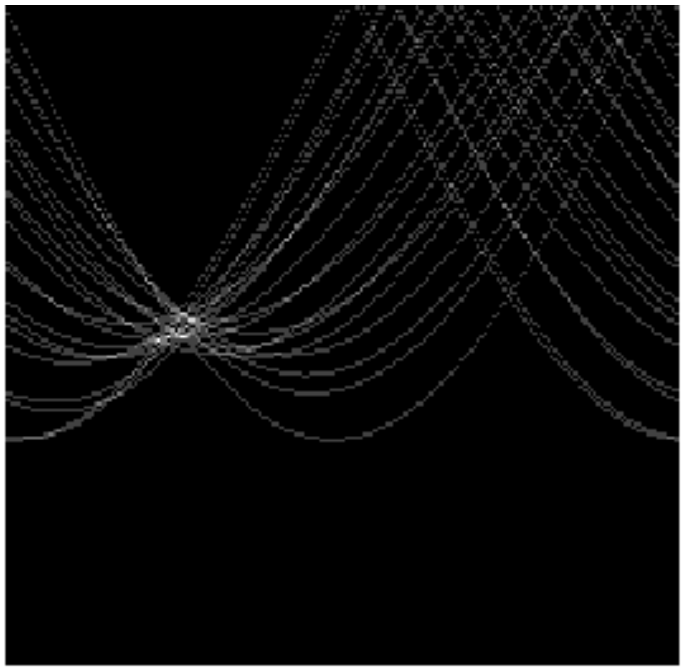


features

Effect of noise



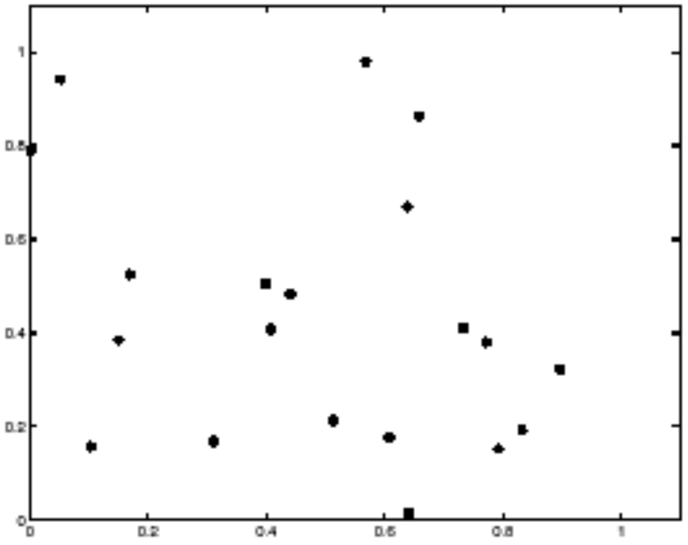
features



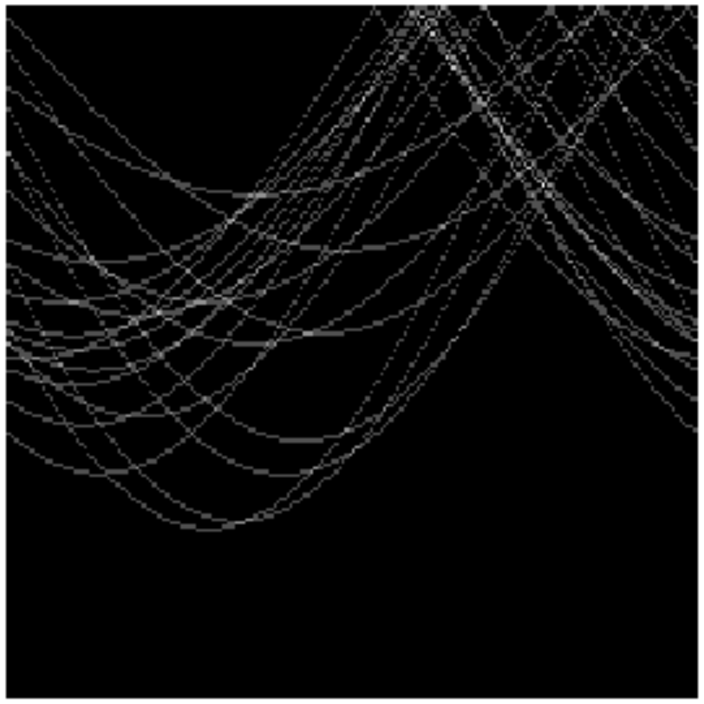
votes

Peak gets fuzzy and hard to locate

Effect of outliers



features



votes

Uniform noise can lead to spurious peaks in the array

Dealing with noise

- How to choose a good grid discretization?
  - **Too coarse:** large votes obtained when too many different lines correspond to a single bucket
  - **Too fine:** miss lines because some points that are not exactly collinear cast votes for different buckets
- Increment neighboring bins (smoothing in accumulator array)
- Try to get rid of irrelevant features
  - E.g., take only edge points with significant gradient magnitude

Hough transform: Pros and cons

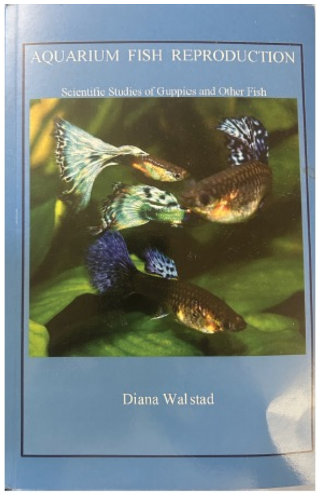
- Pros
  - Can deal with non-locality and occlusion
  - Can detect multiple instances of a model
  - Some robustness to noise: noise points unlikely to contribute consistently to any single bin
  - Leads to a surprisingly general strategy for shape localization (more on this next)
- Cons
  - Complexity increases exponentially with the number of model parameters – in practice, not used beyond three or four dimensions
  - Non-target shapes can produce spurious peaks in parameter space
  - It’s hard to pick a good grid size

Instance level classification (and detection)

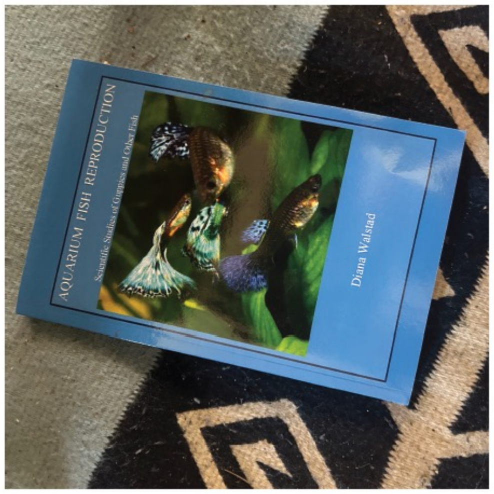
Instance level classification is the problem of determining whether a particular object is present in an image. If it is there (wherever it appears) the image is labelled with that object. Instance classification is rather different than *category level categorization*, where one must determine whether 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.

Example: Find the book

Get pictures of book covers, and attach book name

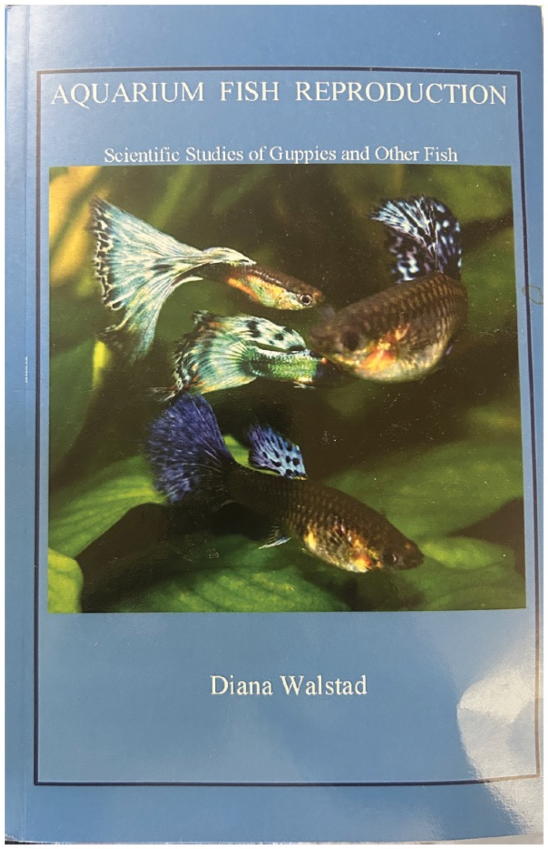
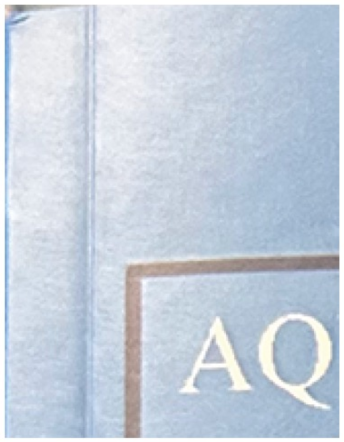

AQUARIUM FISH REPRODUCTION
Scientific Studies of Guppies and Other Fish
Diana Walstad

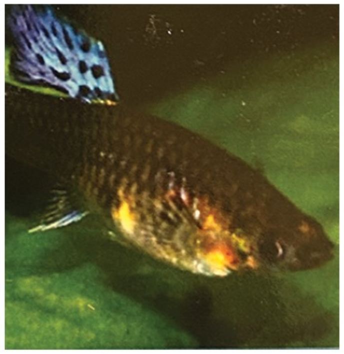
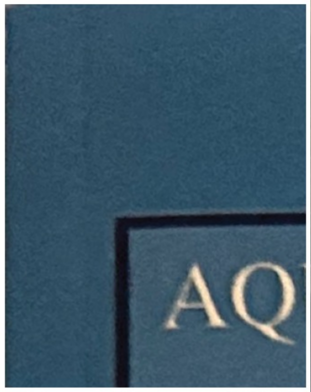
Now find the book cover in some other image

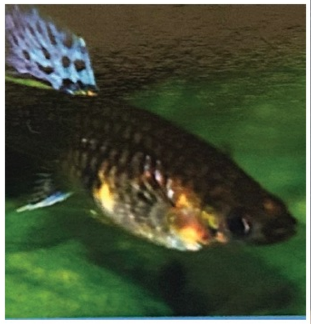
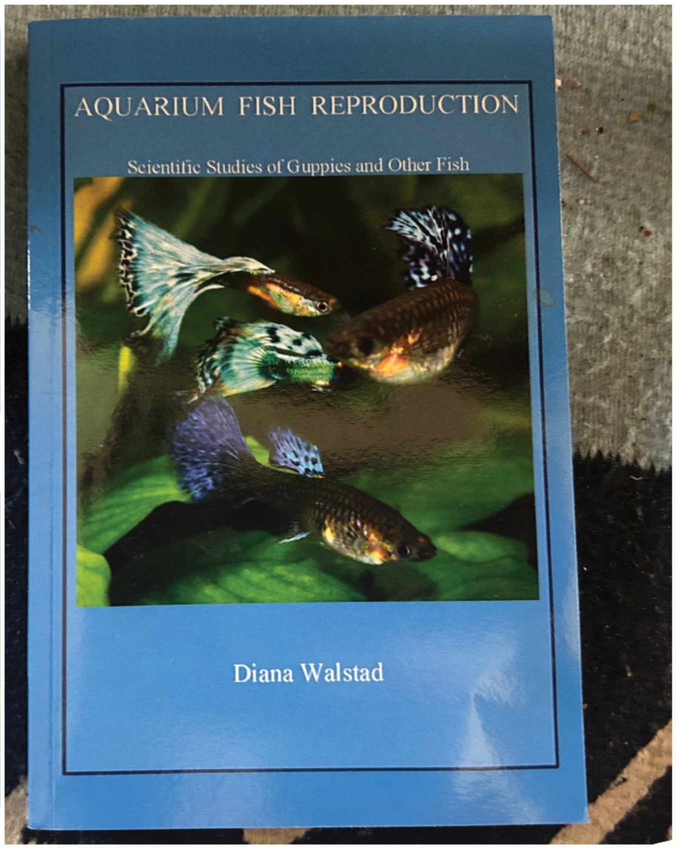

AQUARIUM FISH REPRODUCTION
Scientific Studies of Guppies and Other Fish
Diana Walstad

Can’t do this with SSD; patches might work

Notice changes in color caused by changes in lighting

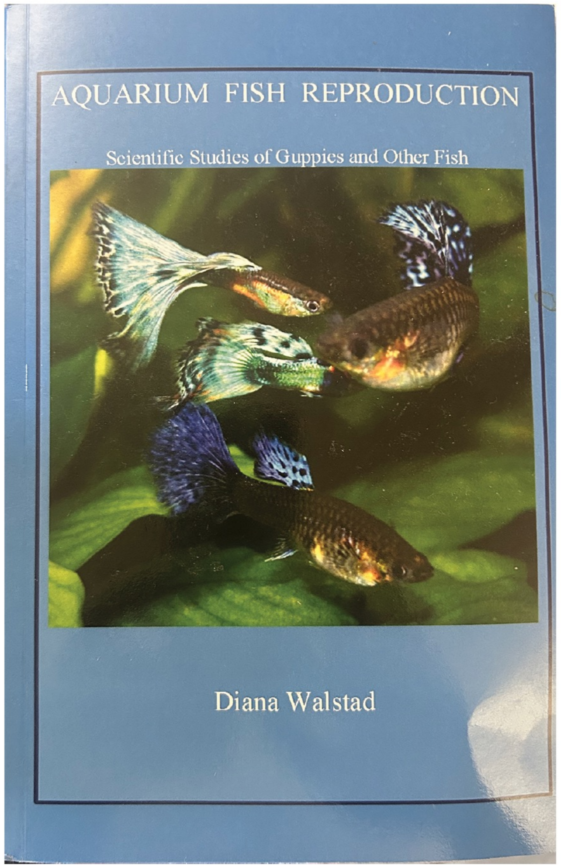

AQUARIUM FISH REPRODUCTION
Scientific Studies of Guppies and Other Fish
Diana Walstad


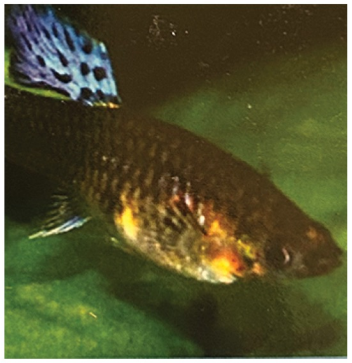



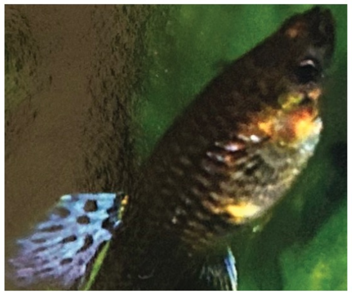


AQUARIUM FISH REPRODUCTION
Scientific Studies of Guppies and Other Fish
Diana Walstad

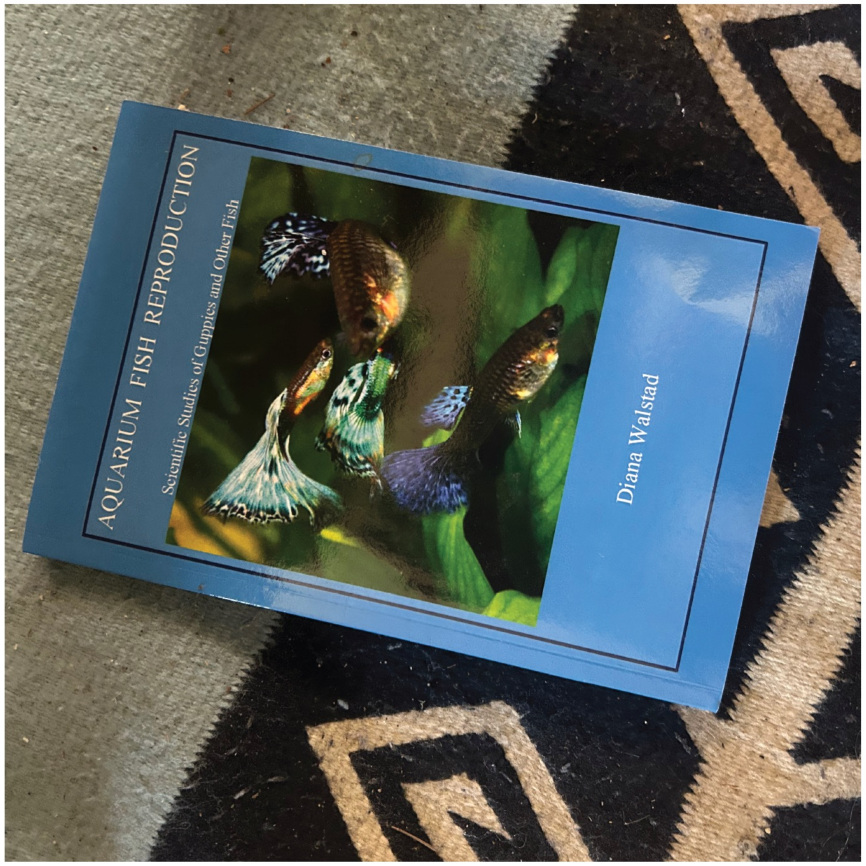
Can't do this with SSD; patches might work

Notice changes in appearance caused by rotation


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






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

Simplest patch algorithm (not much good)

Make a tree of cover image patches (with labels)

For test image, pass patches down tree, vote

Vote how?:

- most common book in leaf
- one vote for each book in leaf
- vote for ANN patches label

Issues:

- every patch votes, so there must be junk votes (easily fixed)
- tree doesn’t know you’re using it for labelling, may not be efficient
- rotation (perhaps insert patches at many rotations?)

Using interest points

Make a tree of cover image interest points (with labels)

You can do this cause interest point descriptors are vectors of fixed length. They're nearby when interest points are similar.

For test image, pass interest points down tree, vote

Vote how?:

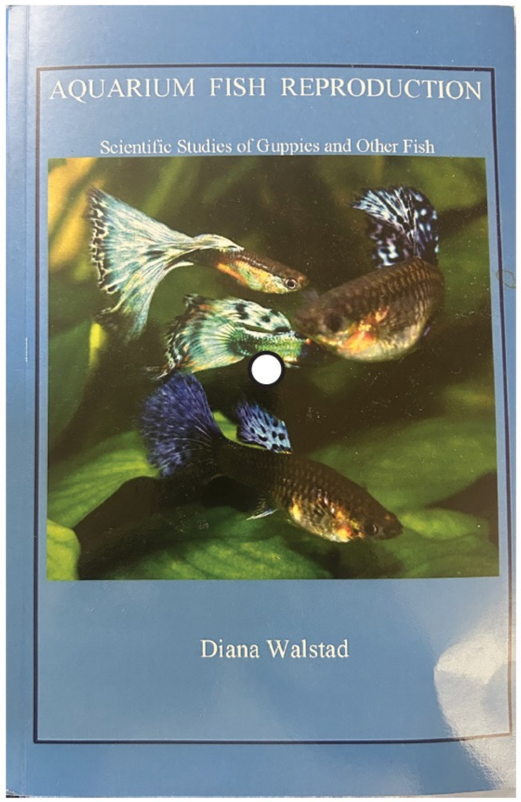
- most common book in leaf
- one vote for each book in leaf
- vote for ANN interest point label

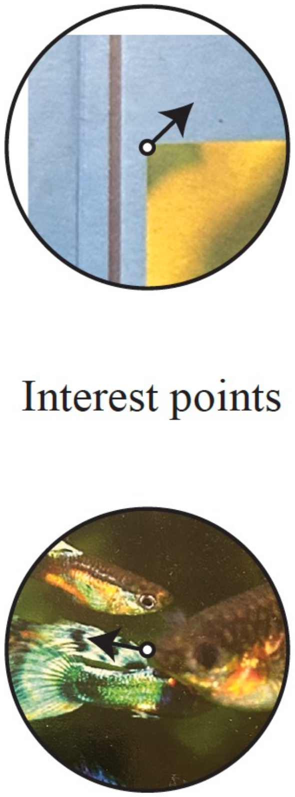
Issues:

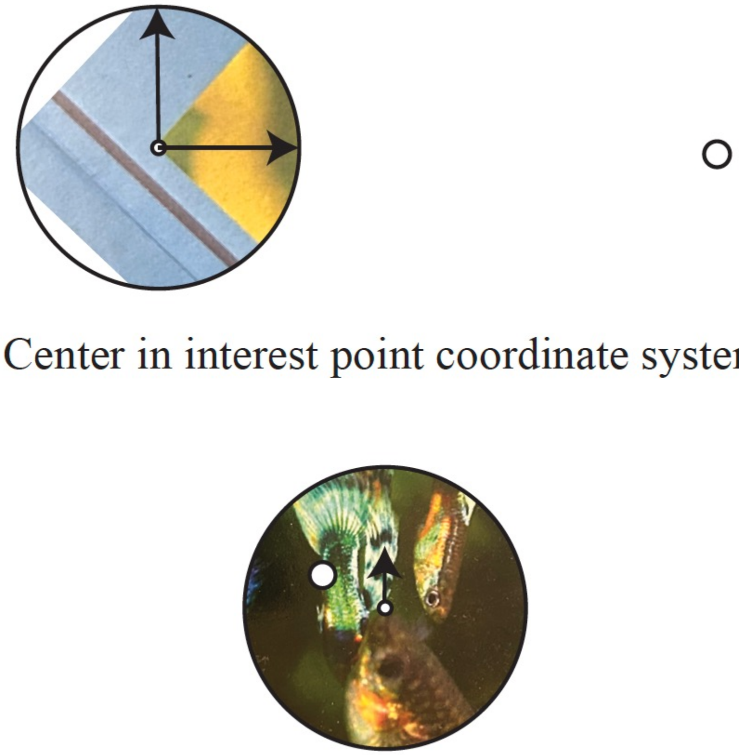
- every patch votes, so there must be junk votes (easily fixed)
- tree doesn't know you're using it for labelling, may not be efficient
- ~~rotation (perhaps insert patches at many rotations?)~~

Every patch votes – generalized Hough transform

Each interest point has a coordinate system,
so it “knows” where the center of the book is

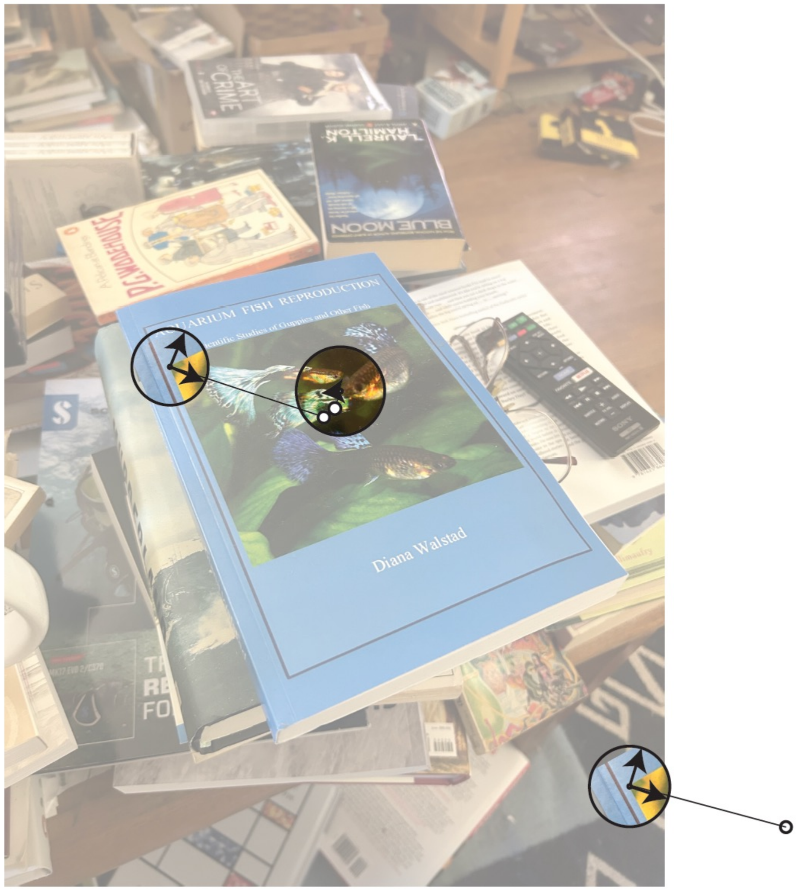

AQUARIUM FISH REPRODUCTION
Scientific Studies of Guppies and Other Fish
Diana Walstad


Interest points


Center in interest point coordinate system

Every patch votes – generalized Hough transform

Allow an interest point to vote for a book only if another one agrees with center



Every patch votes – generalized Hough transform

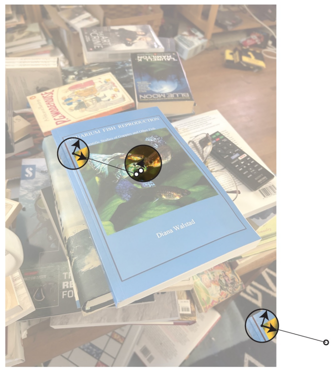
Allow an interest point to vote for a book only if another one agrees with center

How?

build an accumulator array for book centers,

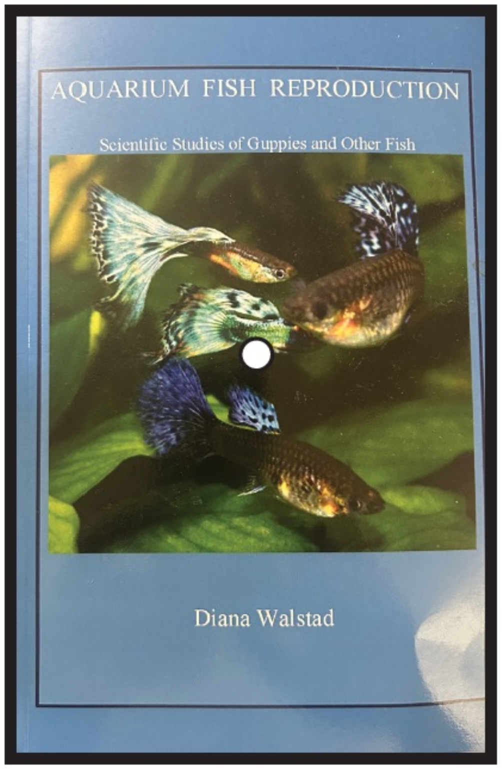
accumulate votes

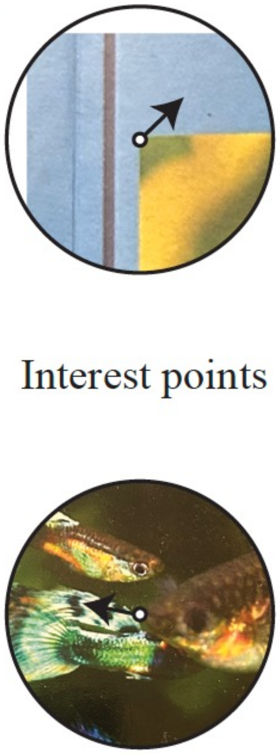
scan it to check for “big” votes – do they agree on book label?

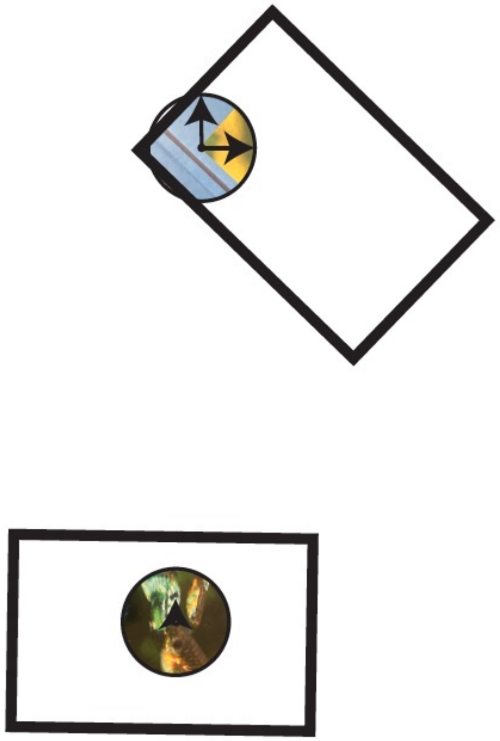


Every patch votes – generalized Hough transform

Interest points know other stuff – the outline of the book


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


Interest points


Bounding box in interest point coordinate system

We now have…

A primitive classifier

  What book is in this image?

A primitive detector

  What books are in this image, and where are they?

There are more accurate technologies,

  tend to be slower, require more compute

This one is extremely fast and very efficient, particularly with an improved tree

The tree

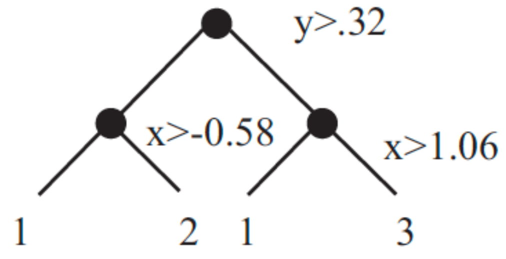
Current construction:

- Hierarchical k-means
  - This breaks up each node into leaves whose elements are close to one another.
  - But this isn't what we want
- Decision tree:
  - Break each node into leaves where leaves are "informative" about label
  - eg in binary tree, labels on the left are all different from labels on the right

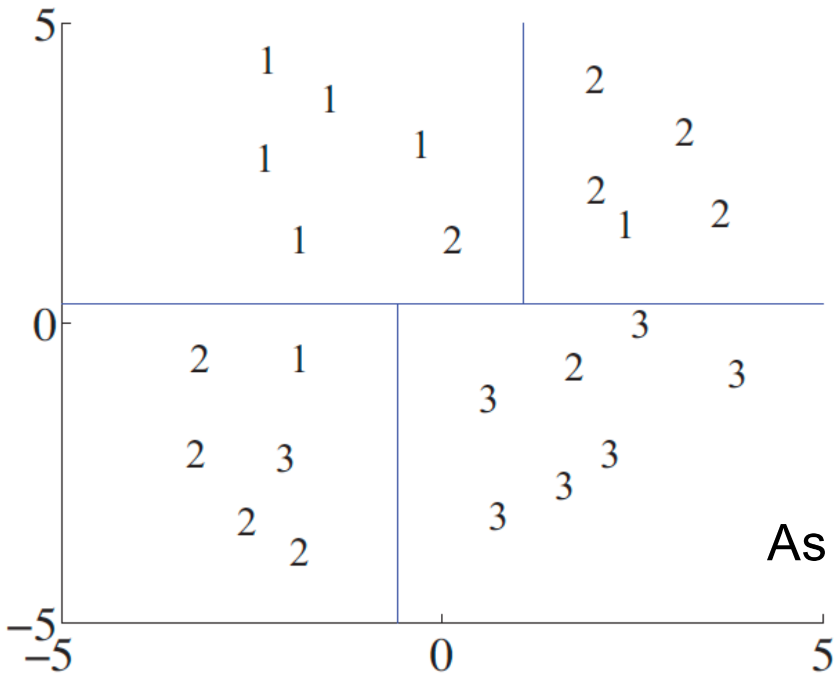
Decision trees

For us:

- tree is now binary
- split is test a single dimension against a threshold



As tree

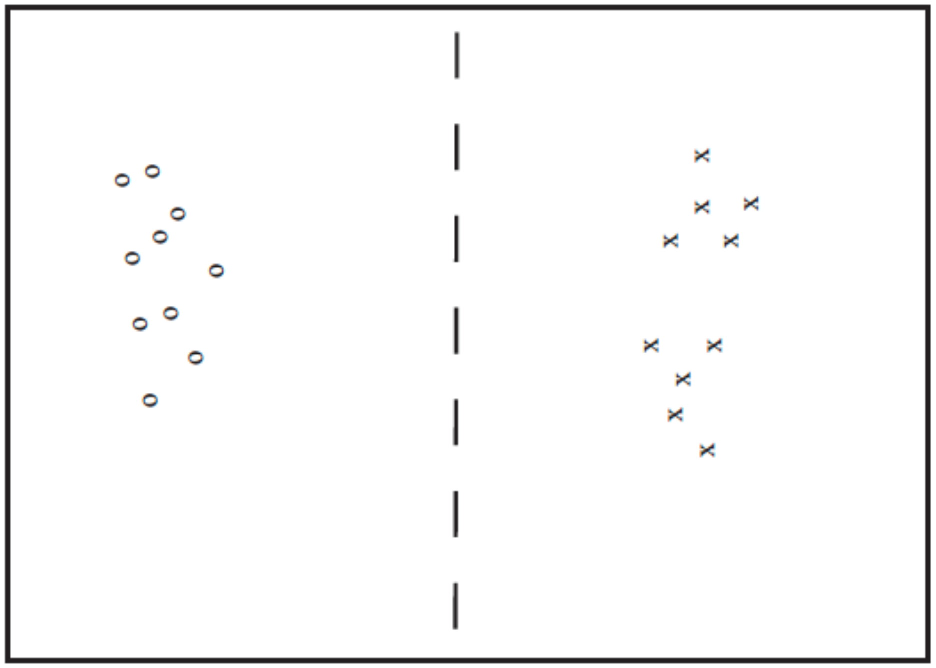


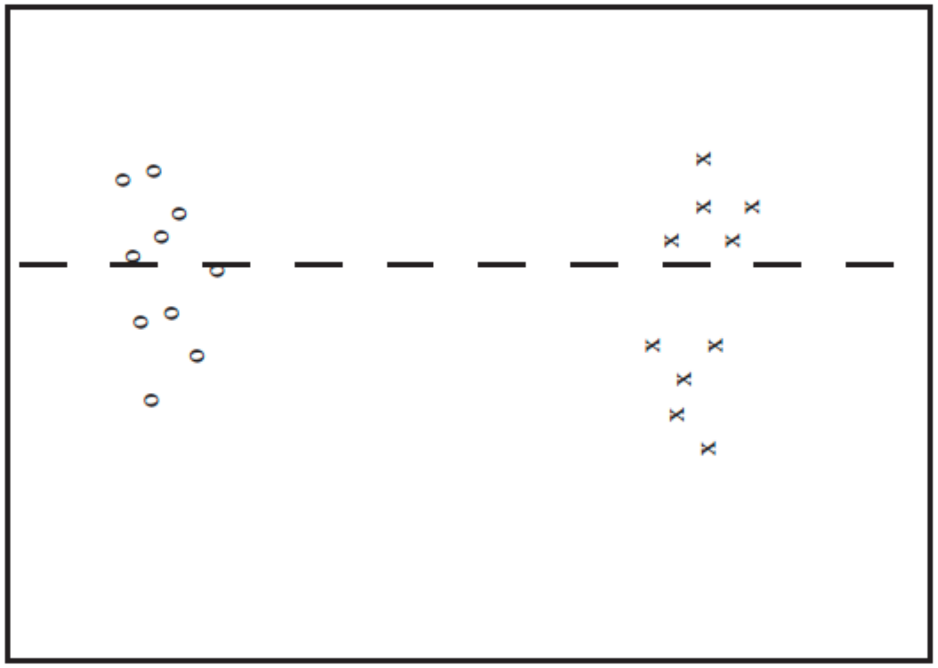
As cells in 2D

Idea: randomized search for informative splits

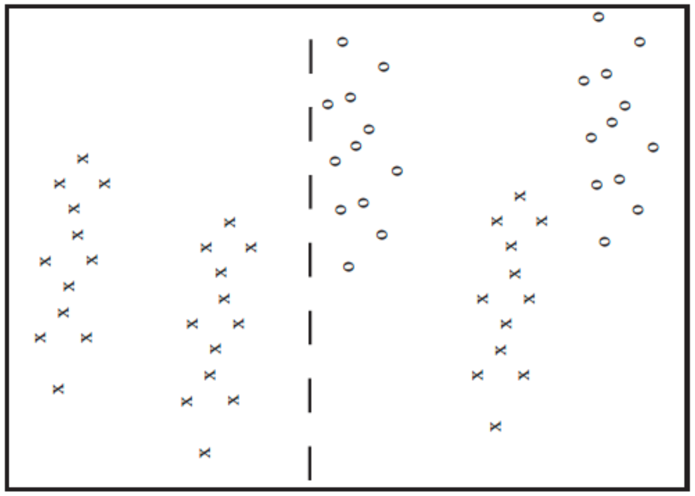
Informative split:

We know more about the likely label if we know which side we're on


Informative split


Less informative split

Harder case



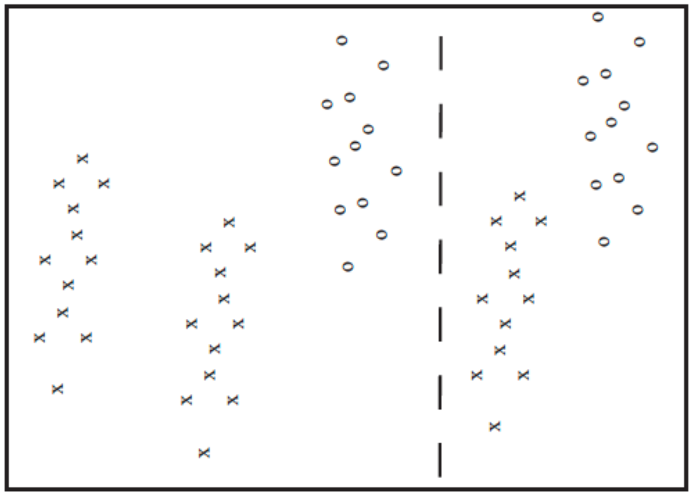
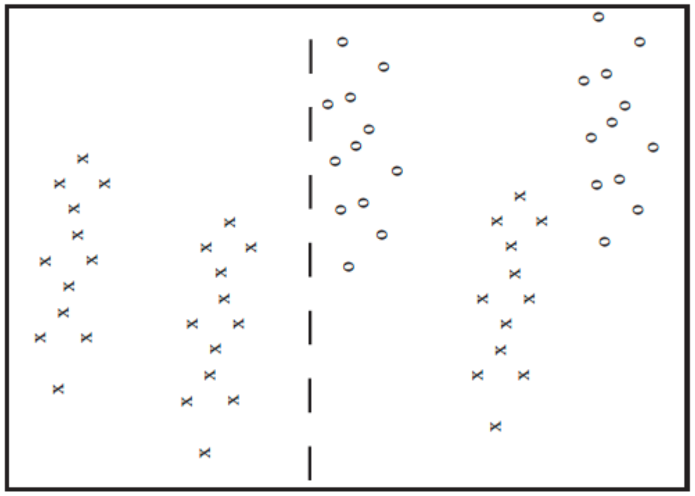


FIGURE 11.9: *Two possible splits of a pool of training data. Positive data is represented with an 'x', negative data with a 'o'. Notice that if you split this pool with the informative line, all the points on the left are 'x's, and two-thirds of the points on the right are 'o's. This means that knowing 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.*

Harder case


Informative split

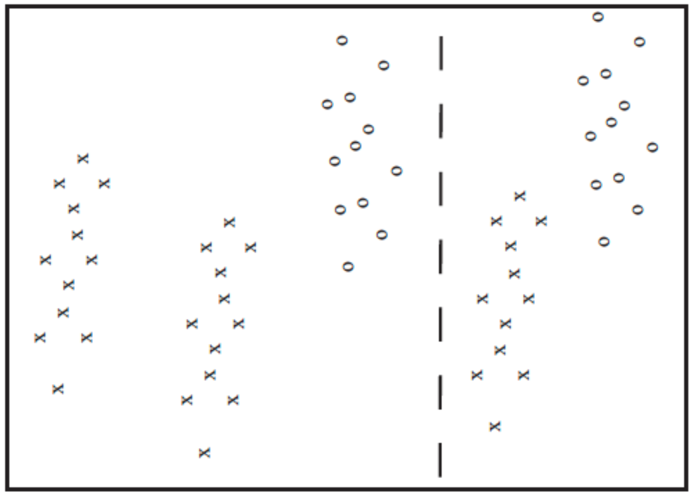
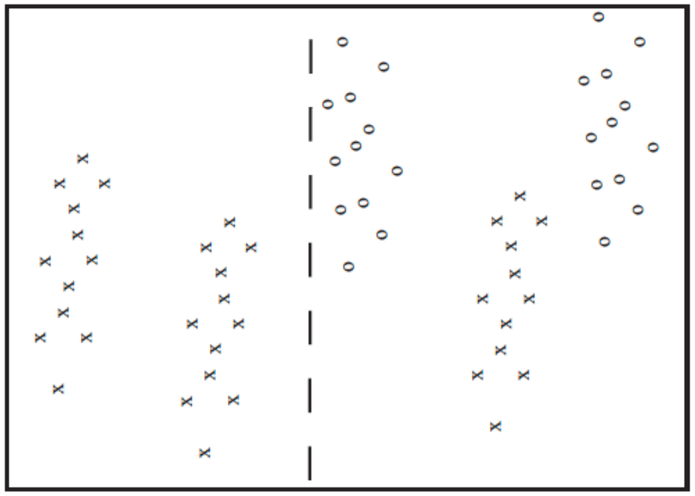
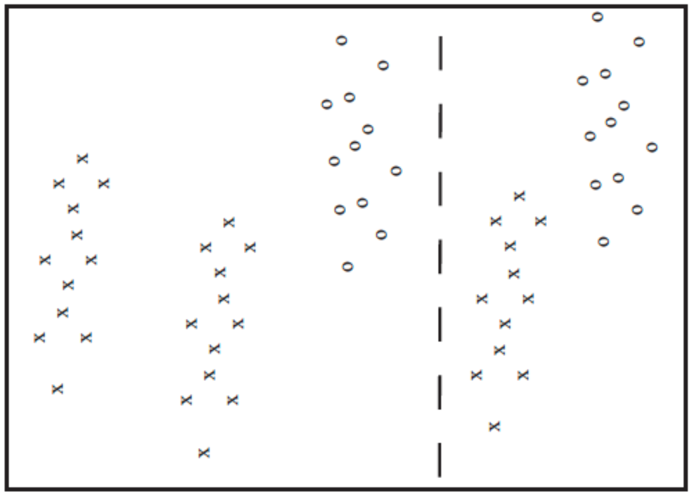

Less informative split

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.

Ha


Informative split


Less informative split

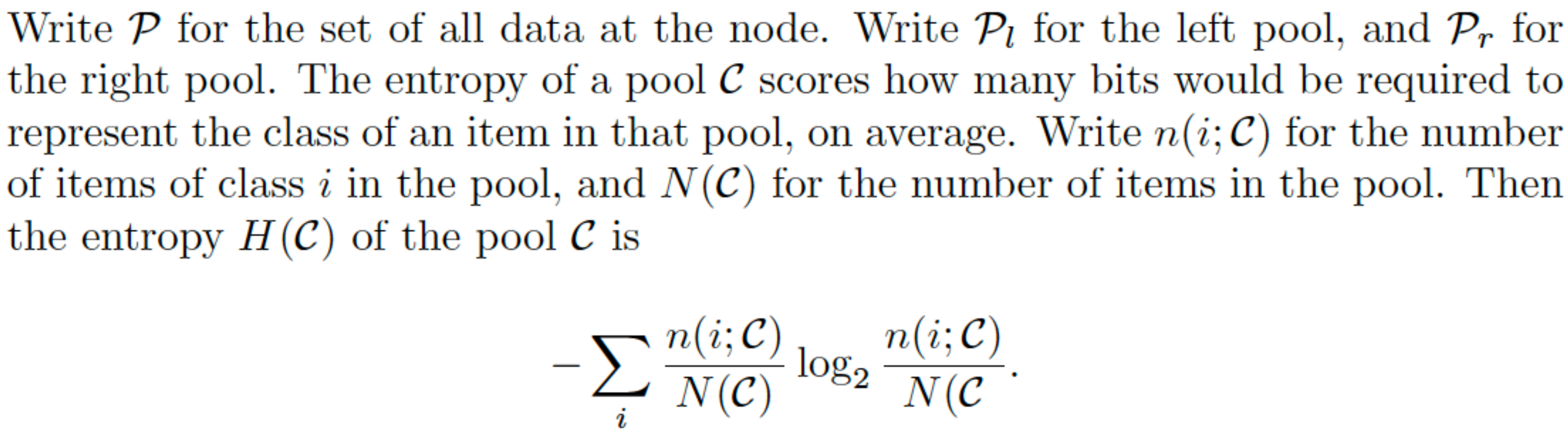
All this suggests a procedure to score how good the split is. In the uninformative case, knowing that a data item is on the left (or the right) does not tell you much more about the data than you already knew. This is because $p(1|\text{left pool, uninformative}) = 2/3 \approx 3/5 = p(1|\text{parent pool})$ 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.

The entropy of a labelled pool of data

Q: how much information do I need to supply to know the label of a randomly selected example?

A: Entropy

could be as small as 0 (all labels the same)



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

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

What does a split do?

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

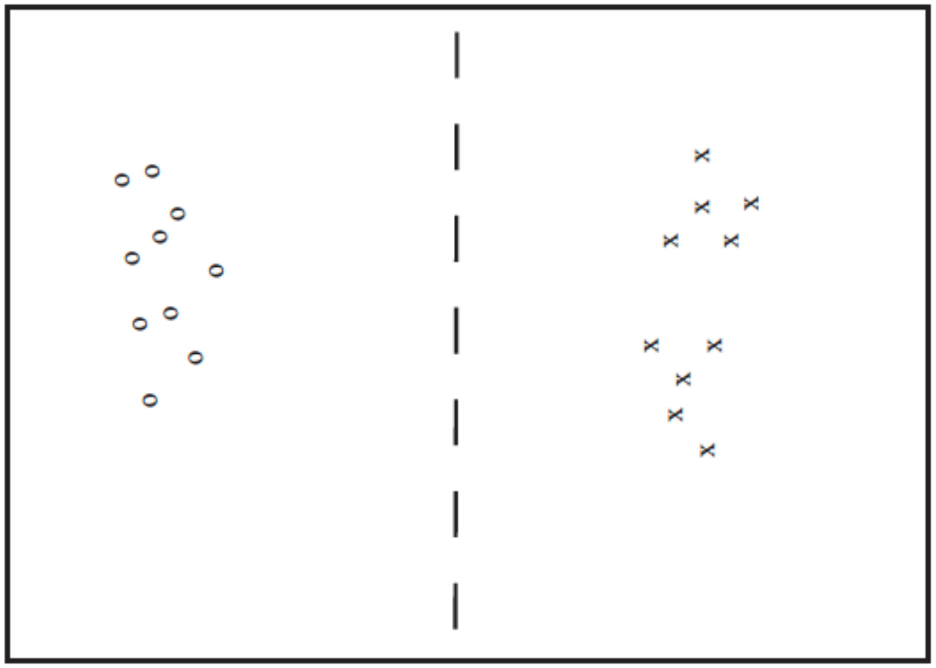
Scoring a split

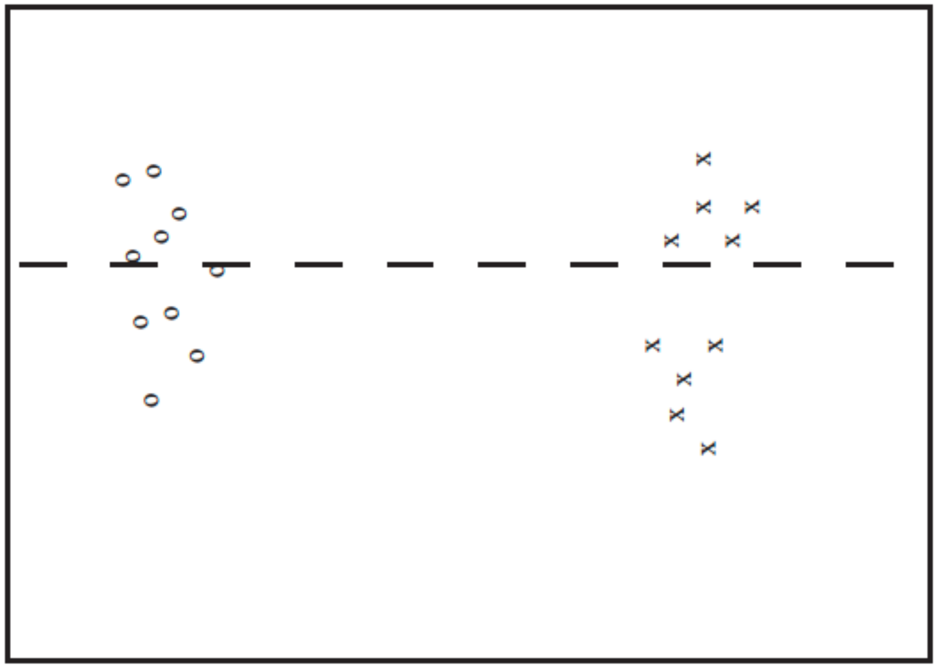
classify once you have split than before the split. You can see the difference

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

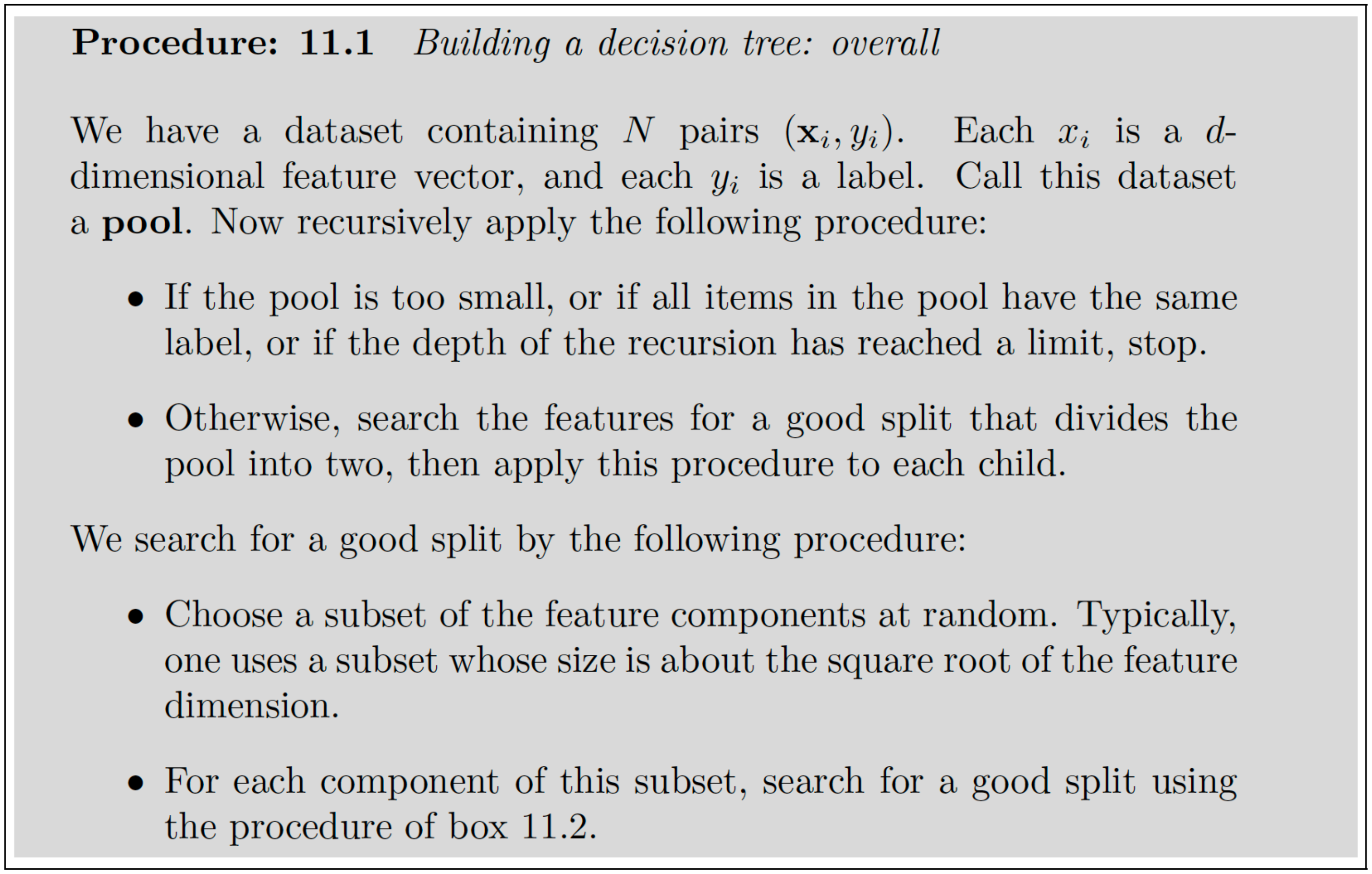
as the *information gain* caused by the split. This is the average number of bits that you *don't* have to supply if you know which side of the split an example lies. Better splits have larger information gain. All this yields a relatively straightforward blueprint for an algorithm, which I have put in a box. It's a blueprint, because there are a variety of ways in which it can be revised and changed.

Scoring a split


Informative split


Less informative split

Building a tree



Procedure: 11.1 *Building a decision tree: overall*

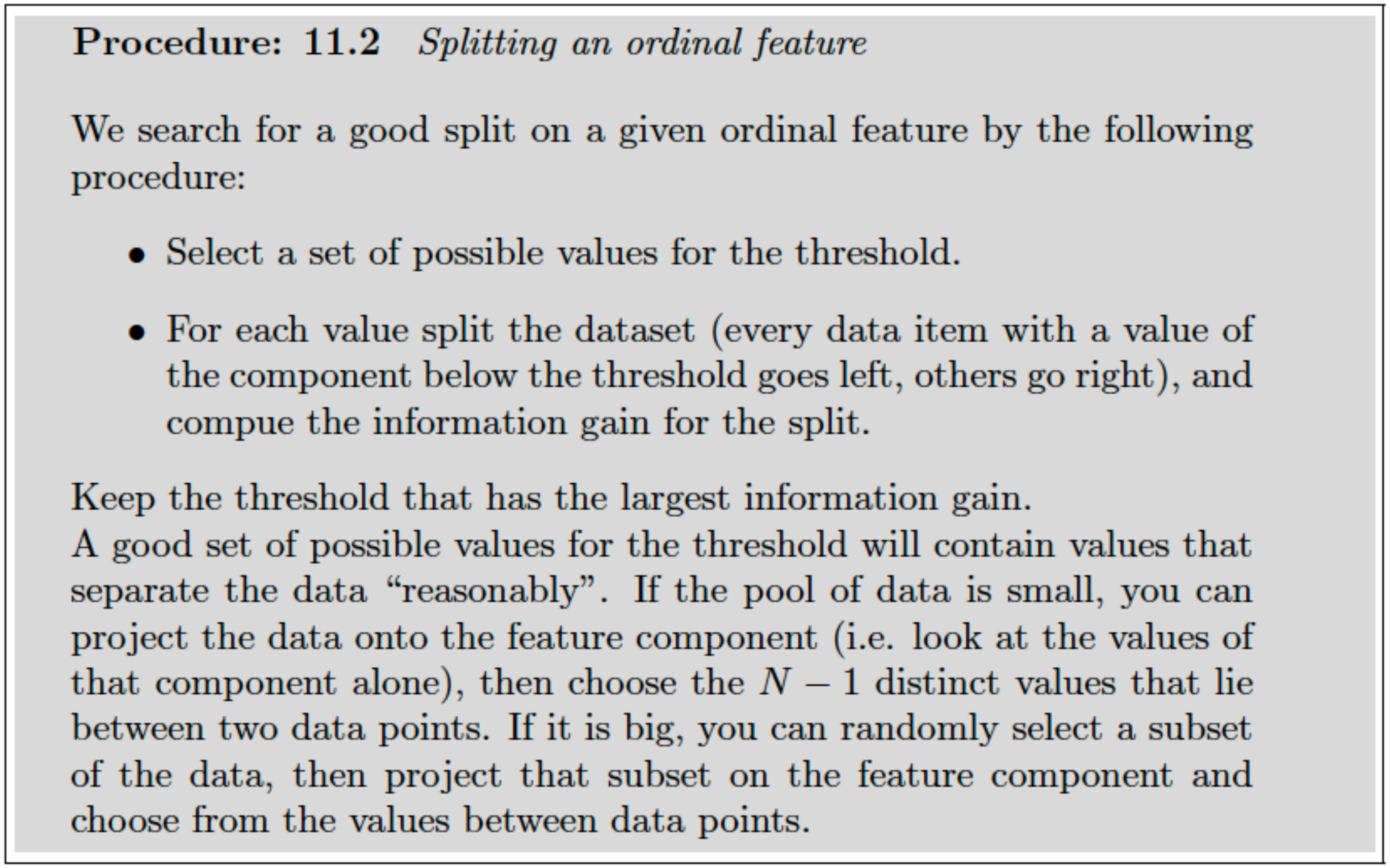
We have a dataset containing N pairs $(\mathbf{x}_i, y_i)$. Each x_i is a d-dimensional feature vector, and each y_i is a label. Call this dataset a **pool**. Now recursively apply the following procedure:

- 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.

Building a tree - II



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.

Forests

Use this tree the same way you'd use the HKM tree

- pass interest points down, pick up votes at leaves

It isn't spectacular, and it can't be the best tree

- random components in construction

Idea:

- build multiple trees and vote over them, too – a decision forest

Variants:

- subsample training dataset and train each tree on a different sample (bagging)

Why bother?

Quite accurate classifiers that are fantastically fast at run-time

Can use very little compute

Used in consumer devices
    eg Kinect body configuration detection (now gone)