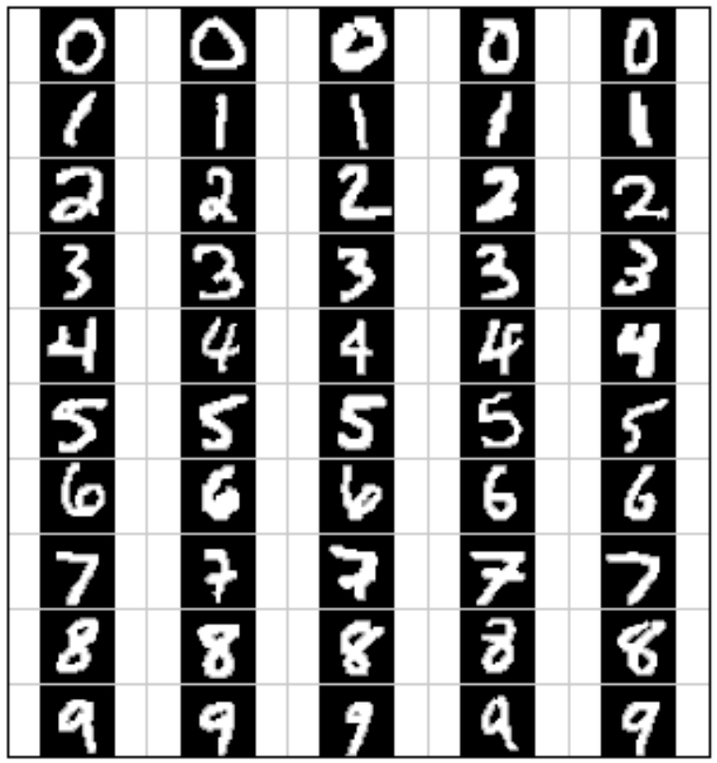
Random Forests and Ferns

David Capel

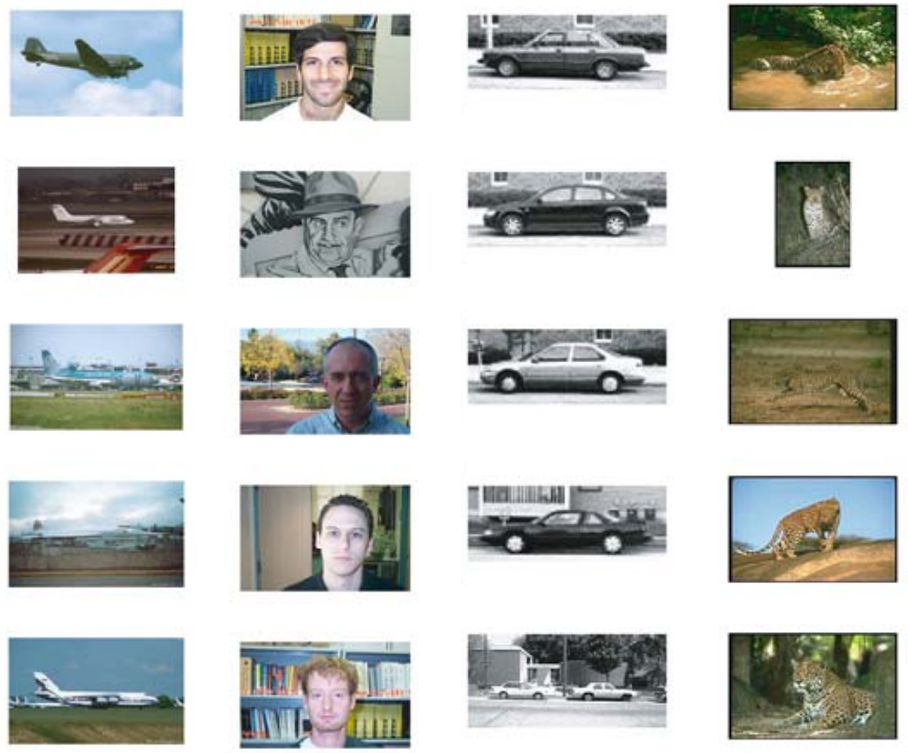
The Multi-class Classification Problem

Training: Labelled exemplars representing multiple classes

Handwritten digits



Planes Faces Cars Cats



Classifying: to which class does this new example belong?

 ?

 ?

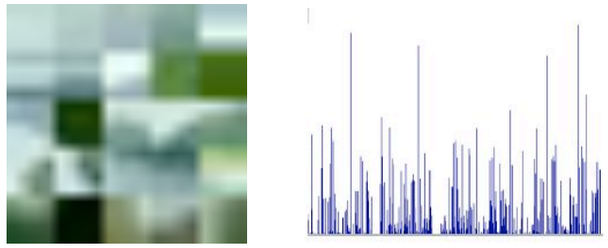
The Multi-class Classification Problem

- A classifier is a mapping **H** from feature vectors **f** to discrete class labels **C**

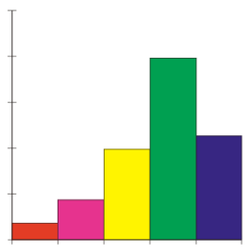
$$\mathbf{f} = (f_1, f_2, ..., f_N)$$
$$C \in \{c_1, c_2, ..., c_K\}$$
$$H : \mathbf{f} \rightarrow C$$

- Numerous choices of feature space **F** are possible, e.g.

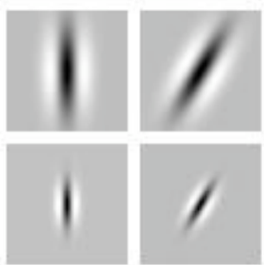
Raw pixel values



Texton histograms



Color histograms



Oriented filter banks

The Multi-class Classification Problem

- We have a (large) database of labelled exemplars

$$D^m = (\mathbf{f}^m, C^m) \quad \text{for } m = 1...M$$

- *Problem:* Given such training data, learn the mapping H

$$H : \mathbf{f} \rightarrow C$$

- *Even better:* learn the posterior distribution over class label conditioned on the features:

$$P(C = c_k | f_1, f_2, ..., f_N)$$

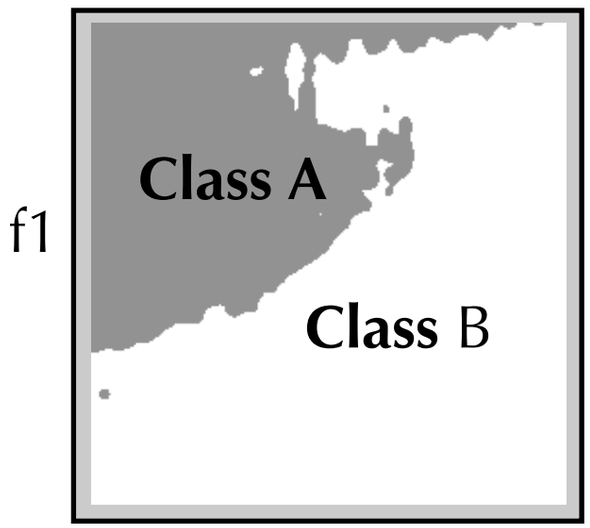
..and obtain classifier **H** as the mode of the posterior:

$$H(\mathbf{f}) = \underset{k}{\text{argmax}}\ P(C = c_k | f_1, f_2, ..., f_N)$$

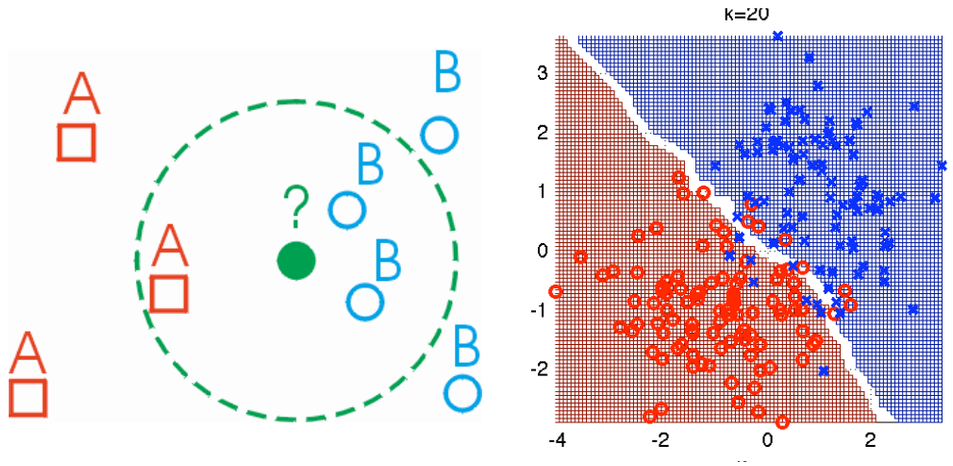
How do we represent and learn the mapping?

In Bishop's book, we saw numerous ways to build multi-class classifiers, e.g.

- Direct, non-parametric learning of class posterior (histograms)
- K-Nearest Neighbours
- Fisher Linear Discriminants
- Relevance Vector Machines
- Multi-class SVMs



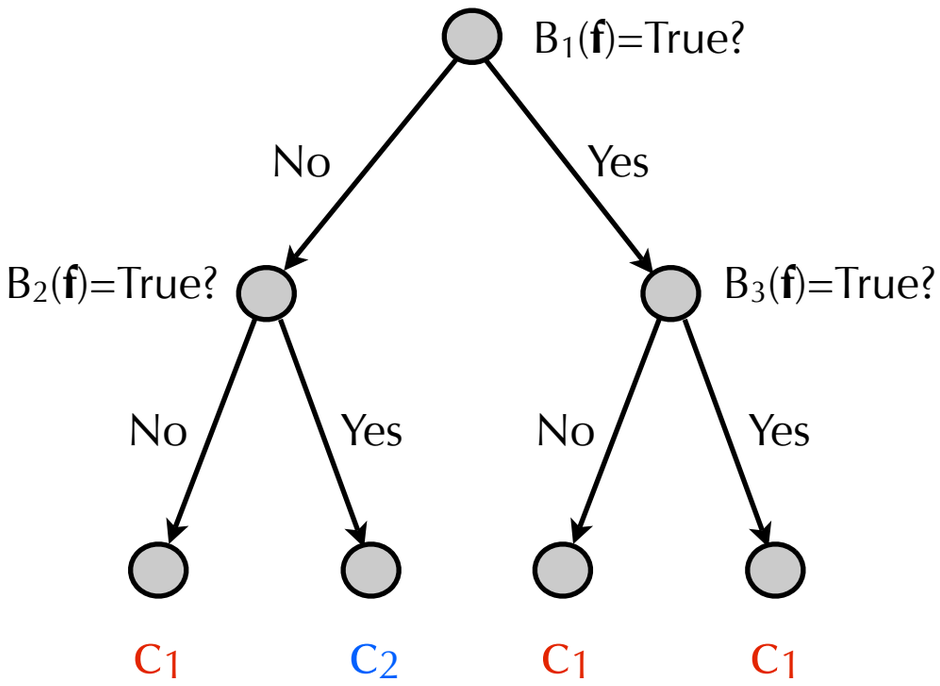
Direct, non-parametric



k-Nearest Neighbours

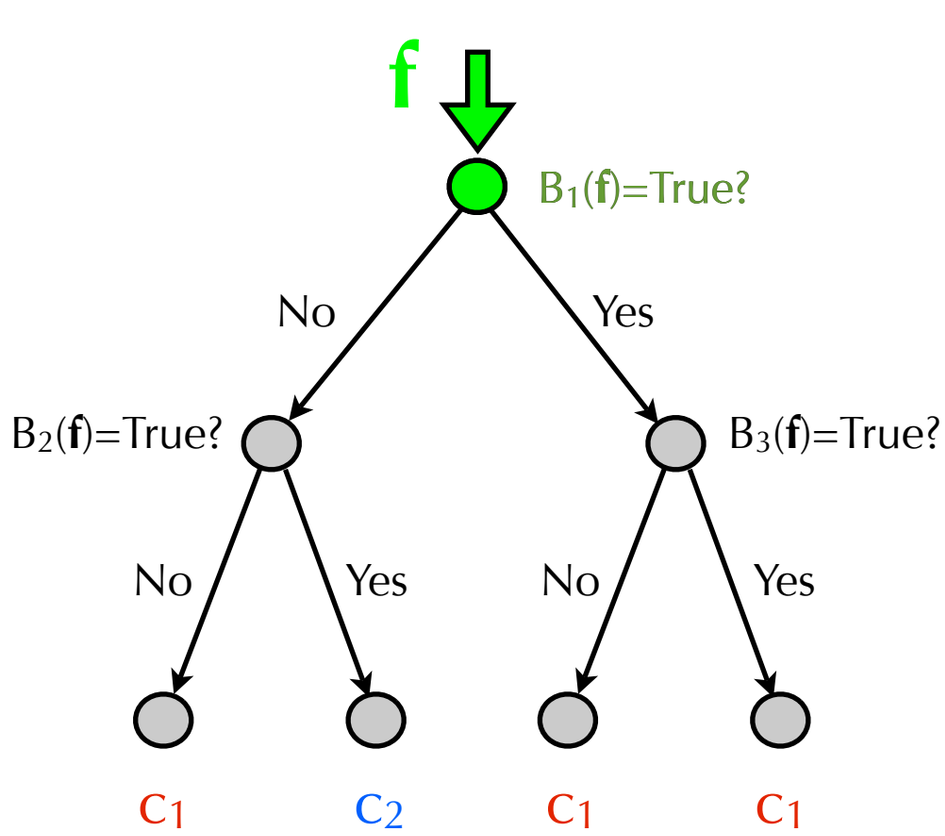
Binary Decision Trees

- Decision trees classify features by a series of Yes/No questions
- At each node, the feature space is split according to the outcome of some binary decision criterion
- The leaves are labelled with the class $\mathbf{C}$ corresponding the feature reached via that path through the tree



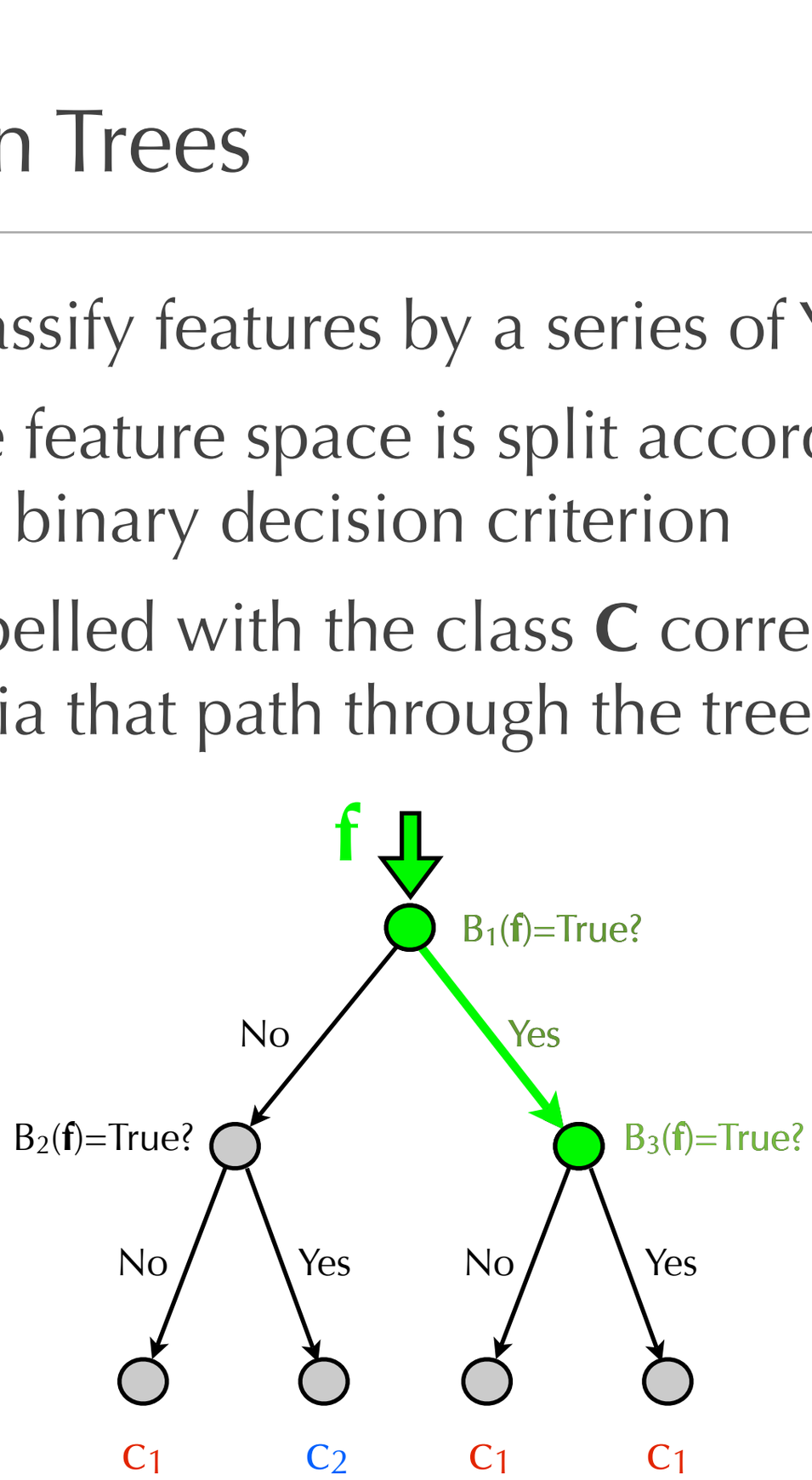
Binary Decision Trees

- Decision trees classify features by a series of Yes/No questions
- At each node, the feature space is split according to the outcome of some binary decision criterion
- The leaves are labelled with the class **C** corresponding the feature reached via that path through the tree



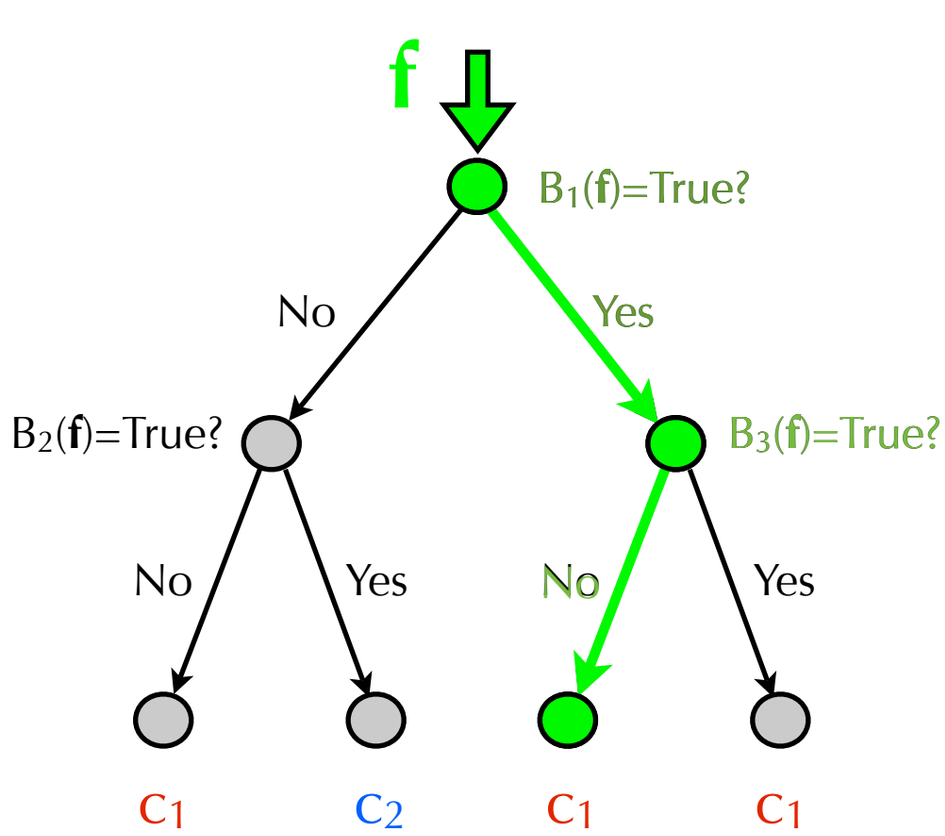
Binary Decision Trees

- Decision trees classify features by a series of Yes/No questions
- At each node, the feature space is split according to the outcome of some binary decision criterion
- The leaves are labelled with the class **C** corresponding the feature reached via that path through the tree



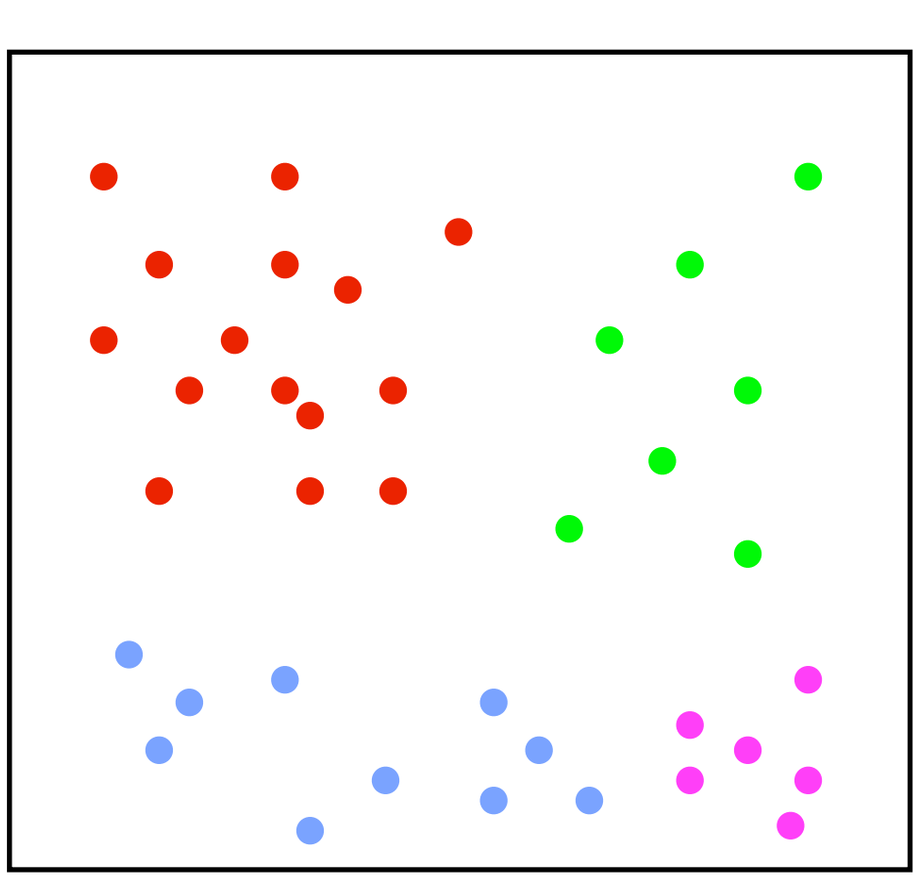
Binary Decision Trees

- Decision trees classify features by a series of Yes/No questions
- At each node, the feature space is split according to the outcome of some binary decision criterion
- The leaves are labelled with the class **C** corresponding the feature reached via that path through the tree



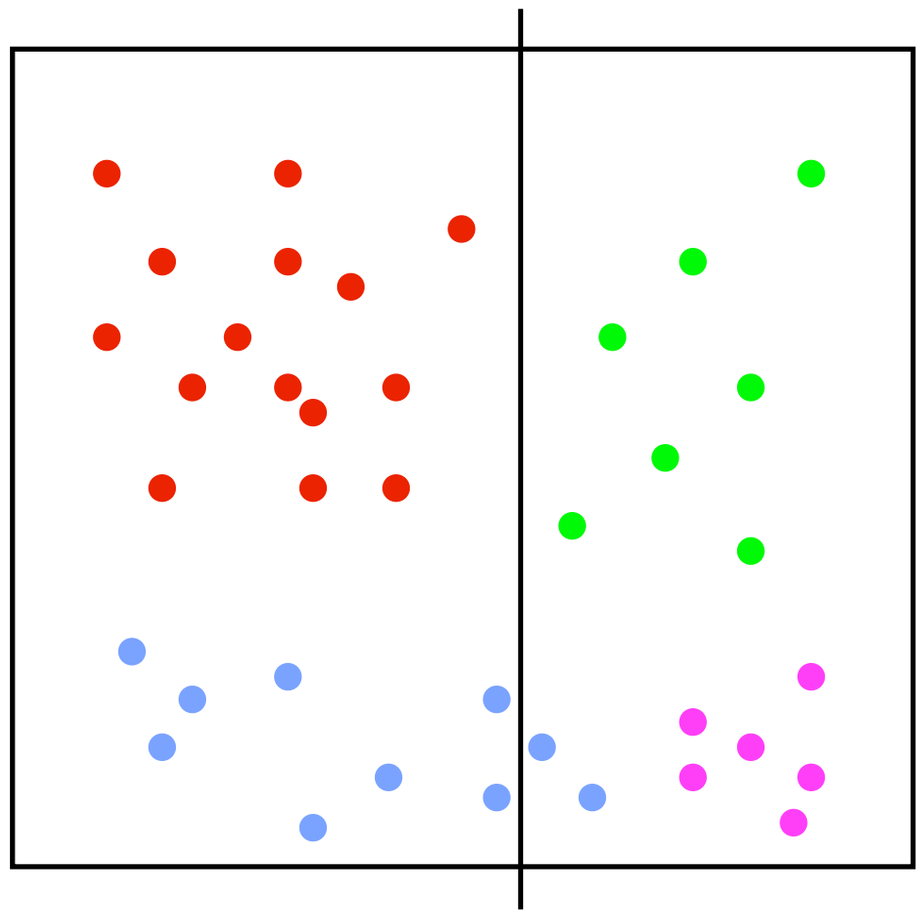
Binary Decision Trees: Training

- To train, recursively partition the training data into subsets according to some Yes/No tests on the feature vectors
- Partitioning continues until each subset contains features of a single class



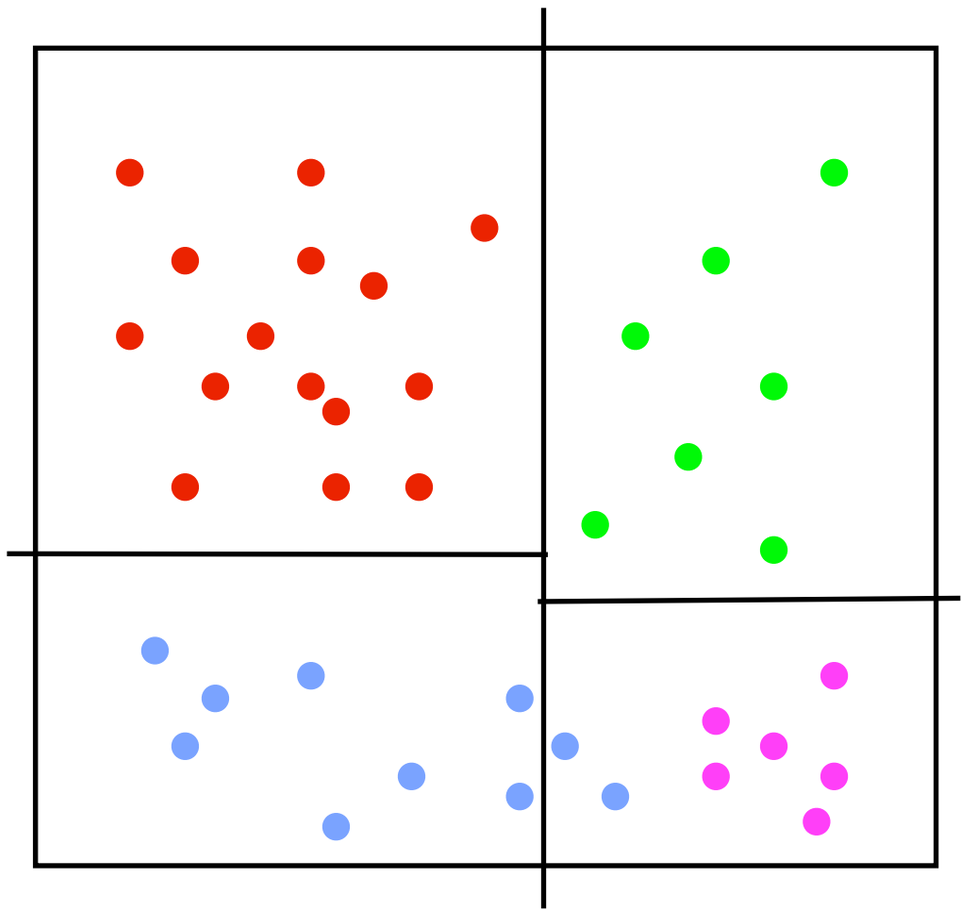
Binary Decision Trees: Training

- To train, recursively partition the training data into subsets according to some Yes/No tests on the feature vectors
- Partitioning continues until each subset contains features of a single class



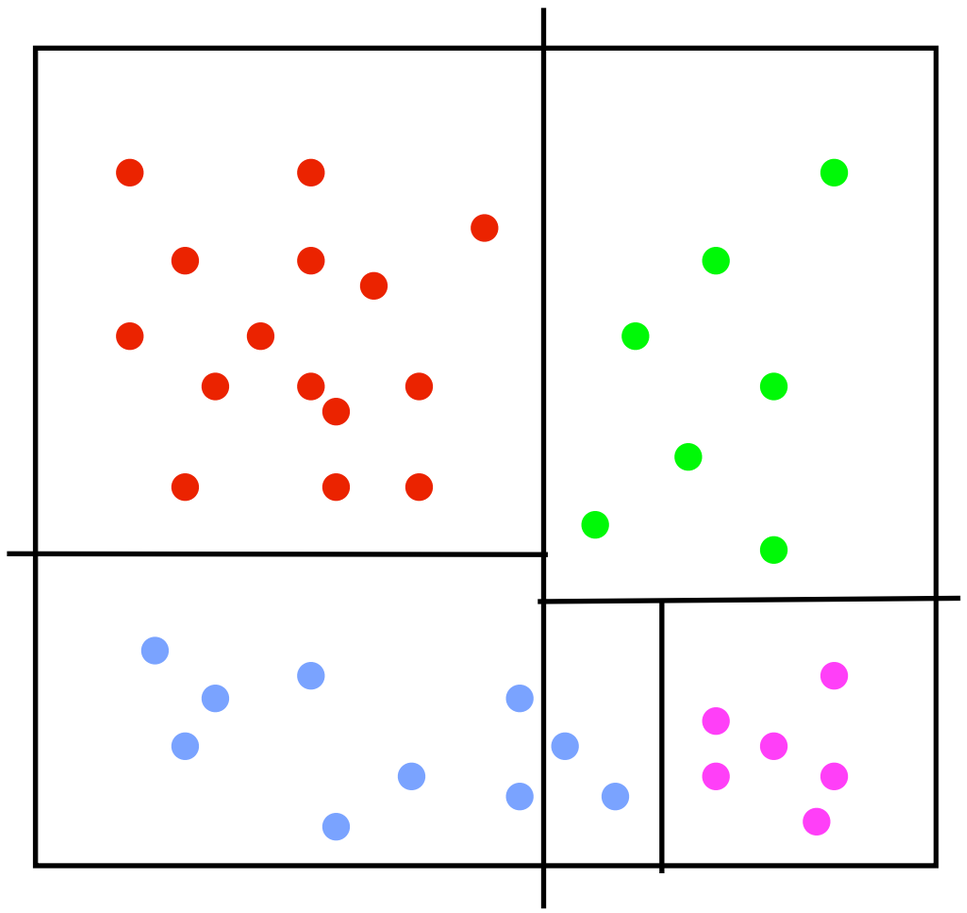
Binary Decision Trees: Training

- To train, recursively partition the training data into subsets according to some Yes/No tests on the feature vectors
- Partitioning continues until each subset contains features of a single class



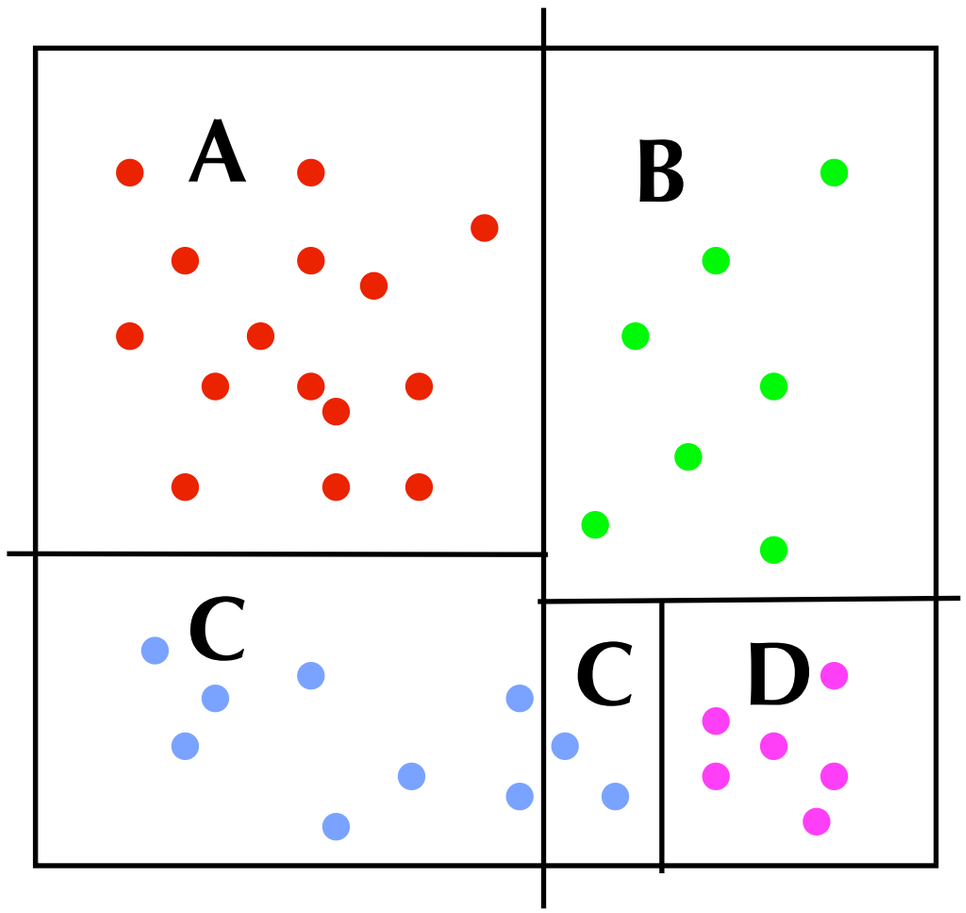
Binary Decision Trees: Training

- To train, recursively partition the training data into subsets according to some Yes/No tests on the feature vectors
- Partitioning continues until each subset contains features of a single class



Binary Decision Trees: Training

- To train, recursively partition the training data into subsets according to some Yes/No tests on the feature vectors
- Partitioning continues until each subset contains features of a single class



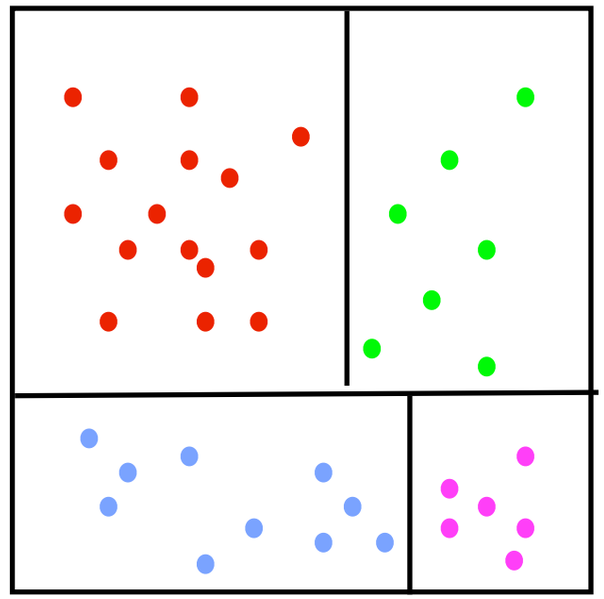
Common decision rules

- **Axis-aligned splitting**

  Threshold on a single feature at each node

  Very fast to evaluate

$$f_n > T$$

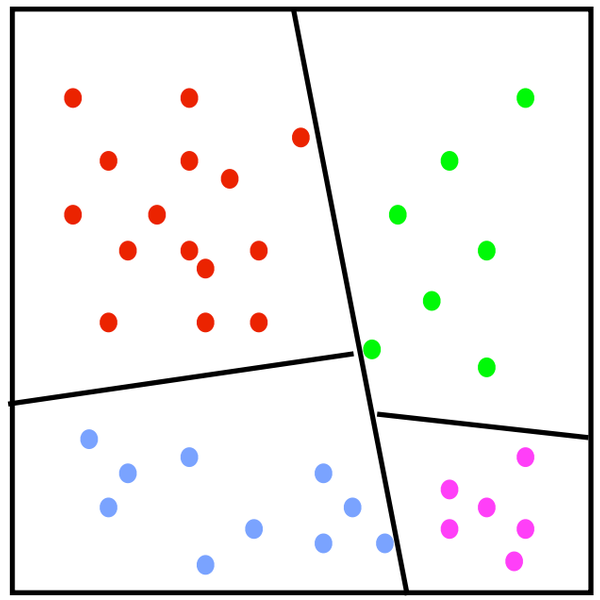


- **General plane splitting**

  Threshold on a linear combination of features

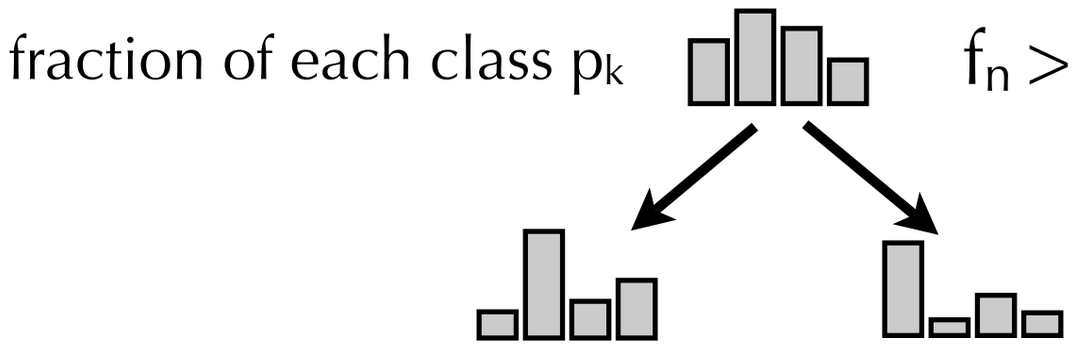
  More expensive but produces smaller trees

$$\mathbf{w}^\top \mathbf{f} > T$$



Choosing the right split

- The split location may be chosen to optimize some measure of classification performance of the child subsets
- Encourage child subsets with lower entropy (confusion)



- Gini impurity

$$I_G = \sum_k p_k^{\text{left}}(1 - p_k^{\text{left}}) + \sum_k p_k^{\text{right}}(1 - p_k^{\text{right}})$$

- Information gain (entropy reduction)

$$I_E = -\sum_k p_k^{\text{left}} \log p_k^{\text{left}} - \sum_k p_k^{\text{right}} \log p_k^{\text{right}}$$

Pros and Cons of Decision Trees

Advantages

- Training can be fast and easy to implement
- Easily handles a large number of input variables

Drawbacks

- Clearly, it is always possible to construct a decision tree that scores 100% classification accuracy on the *training* set
- But they tend to over-fit and do not generalize very well
- Hence, performance on the *testing* set will be far less impressive

So, what can be done to overcome these problems?

Random Forests

Ho, Tin (1995) "Random Decision Forests" ***Int'l Conf. Document Analysis and Recognition***

Breiman, Leo (2001) "Random Forests" ***Machine Learning***

Basic idea

- Somehow introduce randomness into the tree-learning process
- Build multiple, independent trees based on the training set
- When classifying an input, each tree votes for a class label
- The forest output is the consensus of all the tree votes
- *If the trees really are independent, the performance should improve with more trees*

How to introduce randomness?

- **Bagging**

  Generate randomized training sets by sampling with replacement from the full training set (bootstrap sampling)

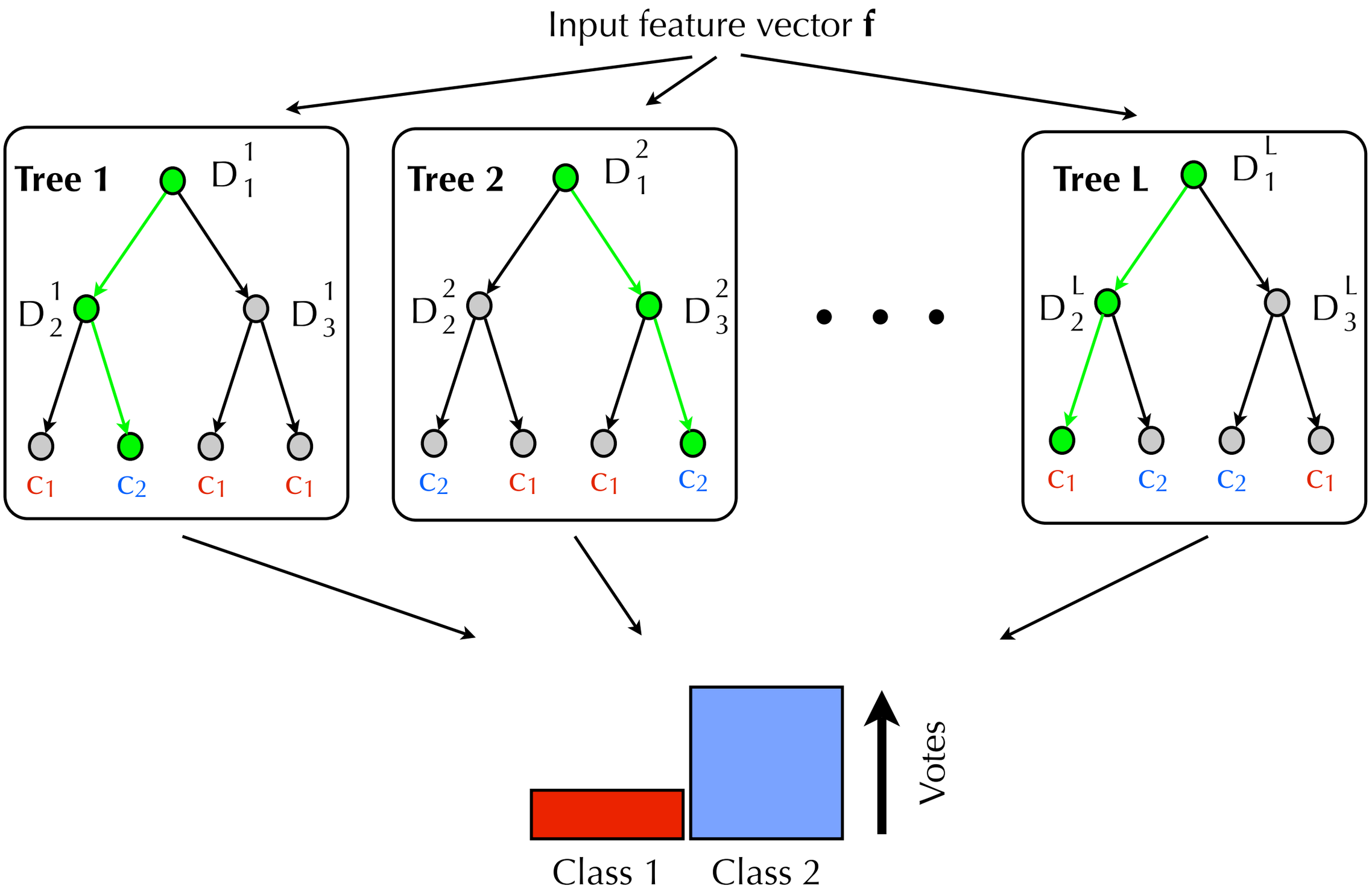
Full training set	D_1 D_2 D_3 D_4 D_5 D_6 D_7 D_8 D_9 D_{10} D_{11} D_{12}
Random "bag"	D_4 D_9 D_3 D_4 D_{12} D_{10} D_{10} D_7 D_3 D_1 D_6 D_1

- **Feature subset selection**

  Choose different random subsets of the full feature vector to generate each tree

Full feature vector	f_1 f_2 f_3 f_4 f_5 f_6 f_7 f_8 f_9 f_{10}
Feature subset	f_4 f_6 f_7 f_9 f_{10}

Random Forests



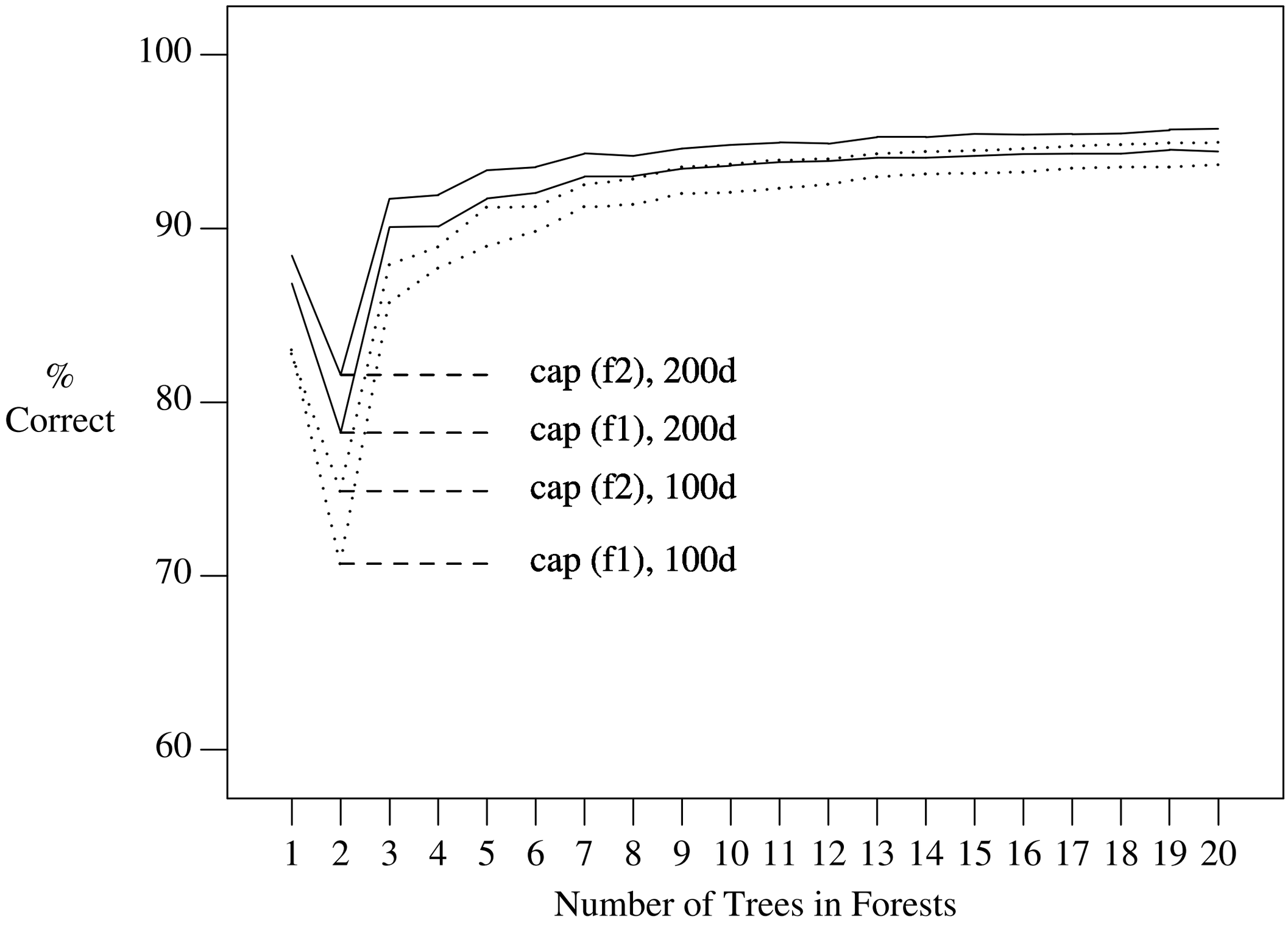
Performance comparison

- Results from Ho 1995
- Handwritten digits (10 classes)
- 20x20 pixel binary images
- 60000 training, 10000 testing samples

- Feature set **f1** : raw pixel values (400 features)
- Feature set **f2** : **f1** plus gradients (852 features in total)

- Uses full training set for every tree (no bagging)
- Compares different features subset sizes (100 and 200 resp)

Performance comparison



Revision: Naive Bayes Classifiers

- We would like to evaluate the posterior probability over class label

$$\underset{k}{\text{argmax}}\ P(C_k|f_1, f_2, ..., f_N)$$

- Bayes' rules tells us that this is equivalent to

$$\underset{k}{\text{argmax}}\ P(f_1, f_2, ..., f_N|C_k)P(C_k) \qquad \text{(likelihood x prior)}$$

- But learning the joint likelihood distributions over all features is most likely intractable!
- Naive Bayes makes the simplifying assumption that features are conditionally independent given the class label

$$P(f_1, f_2, ..., f_N|C_k) = \prod_{i=1}^{N} P(f_i|C_k)$$

Revision: Naive Bayes Classifiers

$$\text{Class}(\mathbf{f}) \equiv \underset{k}{\text{argmax}}\ P(C_k) \prod_{n=1}^{N} P(f_n|C_k)$$

- This independence assumption is usually false!
- The resulting approximation tends to grossly underestimate the true posterior probabilities

However ...

- It is usually easy to learn the 1-d conditional densities $P(f_i|C_k)$
- It often works rather well in practice!
- ***Can we do better without sacrificing simplicity?***

Ferns : “Semi-Naive” Bayes

M. Özuysal et al. “Fast Keypoint Recognition in 10 Lines of Code” ***CVPR 2007***

- Group of features into L small sets of size S (called Ferns)

$$Fl = \{f_{l,1}, f_{l,2}, ..., f_{l,S}\}$$

  where features f_n are the outcome of a binary test on the input vector, such that $f_n : \{0,1\}$

- Assume **groups** are conditionally independent, hence

$$P(f_1, f_2, ..., f_N|C_k) = \prod_{l=1}^{L} P(F_l|C_k)$$

- Learn the class-conditional distributions for each group and apply Bayes rule to obtain posterior,

$$\text{Class}(\mathbf{f}) \equiv \underset{k}{\text{argmax}}\ P(C_k) \prod_{l=1}^{L} P(F_l|C_k)$$

Naive vs Semi-Naive Bayes

- Full joint class-conditional distribution: $P(f_1, f_2, ..., f_N|C_k)$

  Intractable to estimate

- Naive approximation: $P(f_1, f_2, ..., f_N|C_k) = \prod_{i=1}^{N} P(f_i|C_k)$

  Too simplistic

  Poor approximation to true posterior

- Semi-Naive: $P(f_1, f_2, ..., f_N|C_k) = \prod_{l=1}^{L} P(F_l|C_k)$

  Balance of complexity and tractability

  Trade complexity/performance by choice of Fern size (S), NumFerns (L)

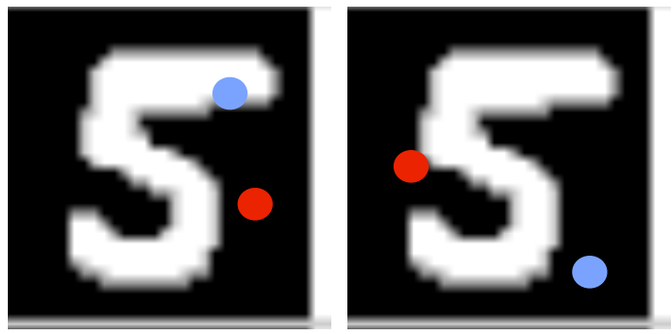
So how does a single Fern work?

- A Fern applies a series of S binary tests to the input vector **I**

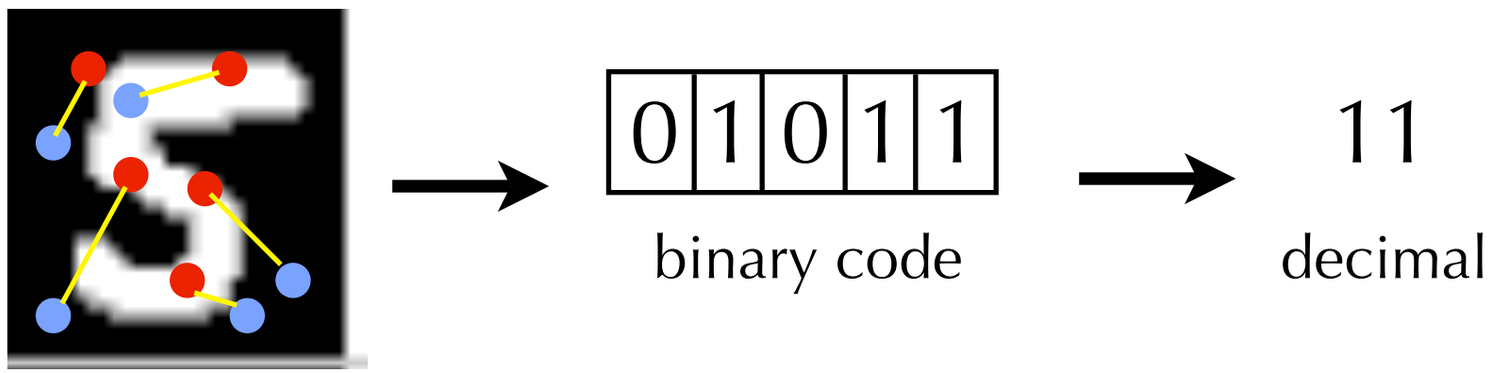
e.g relative intensities of a pair of pixels:

$f_1(\mathbf{I}) = I(x_a,y_a) > I(x_b,y_b)$ -> true

$f_2(\mathbf{I}) = I(x_c,y_c) > I(x_d,y_d)$ -> false



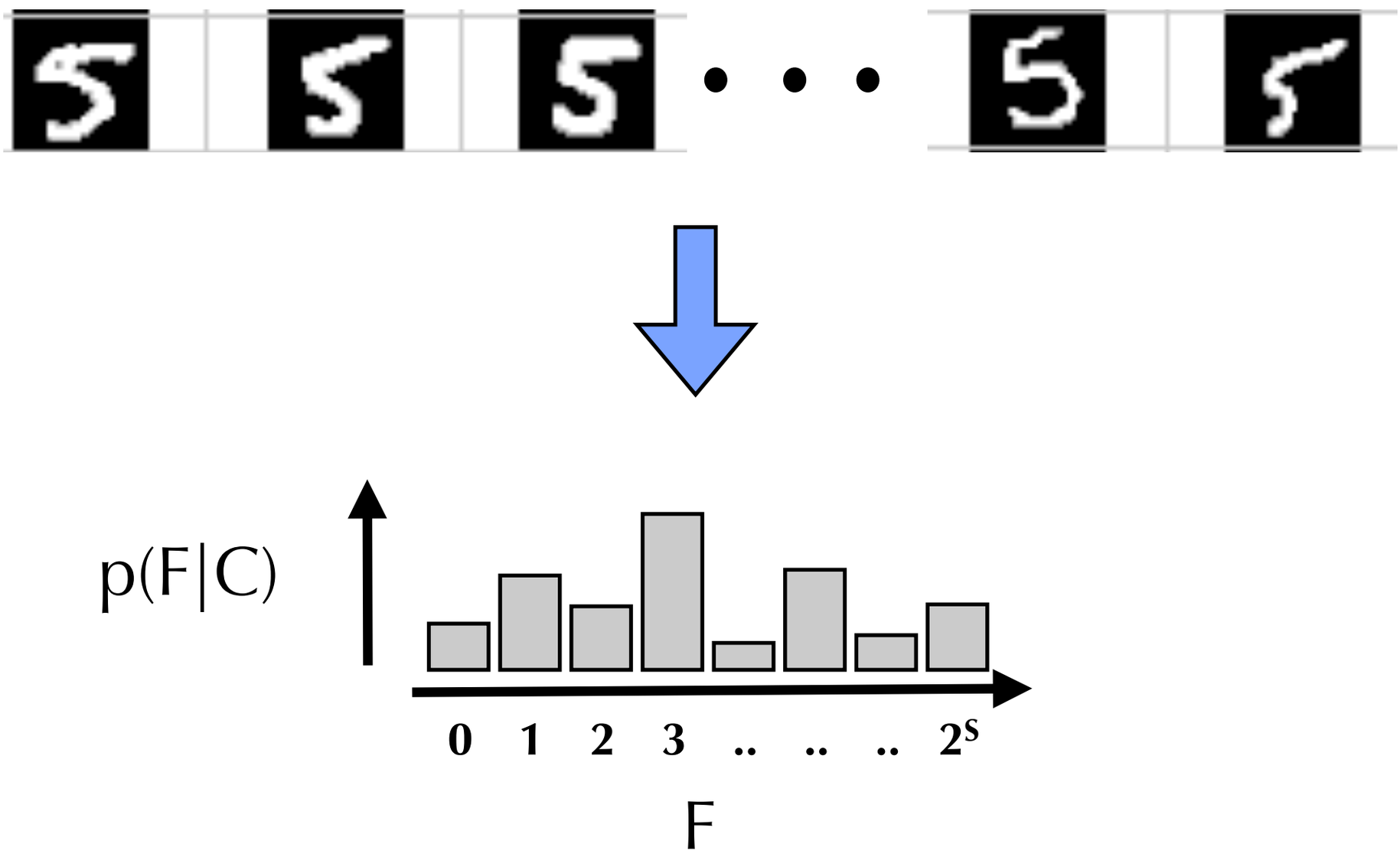
- This gives an S-digit binary code for the feature which may be interpreted as an integer in the range [0 ... 2^S-1]



- It's really just a simple hashing scheme that drops input vectors into one of 2^S "buckets"

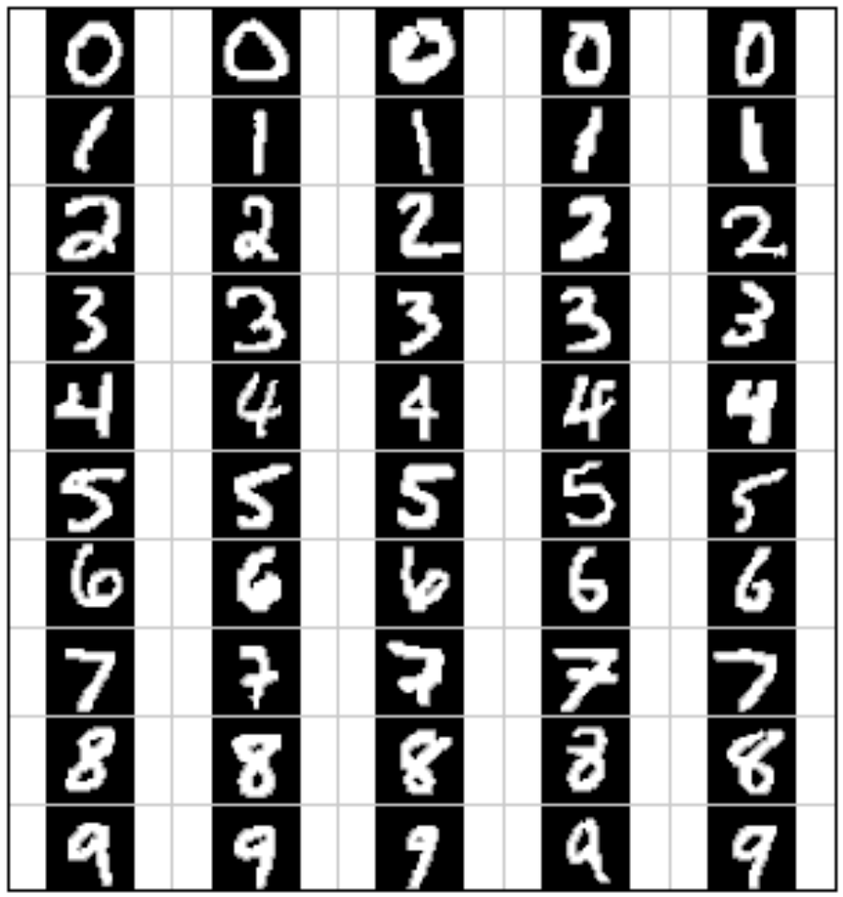
So how does a single Fern work?

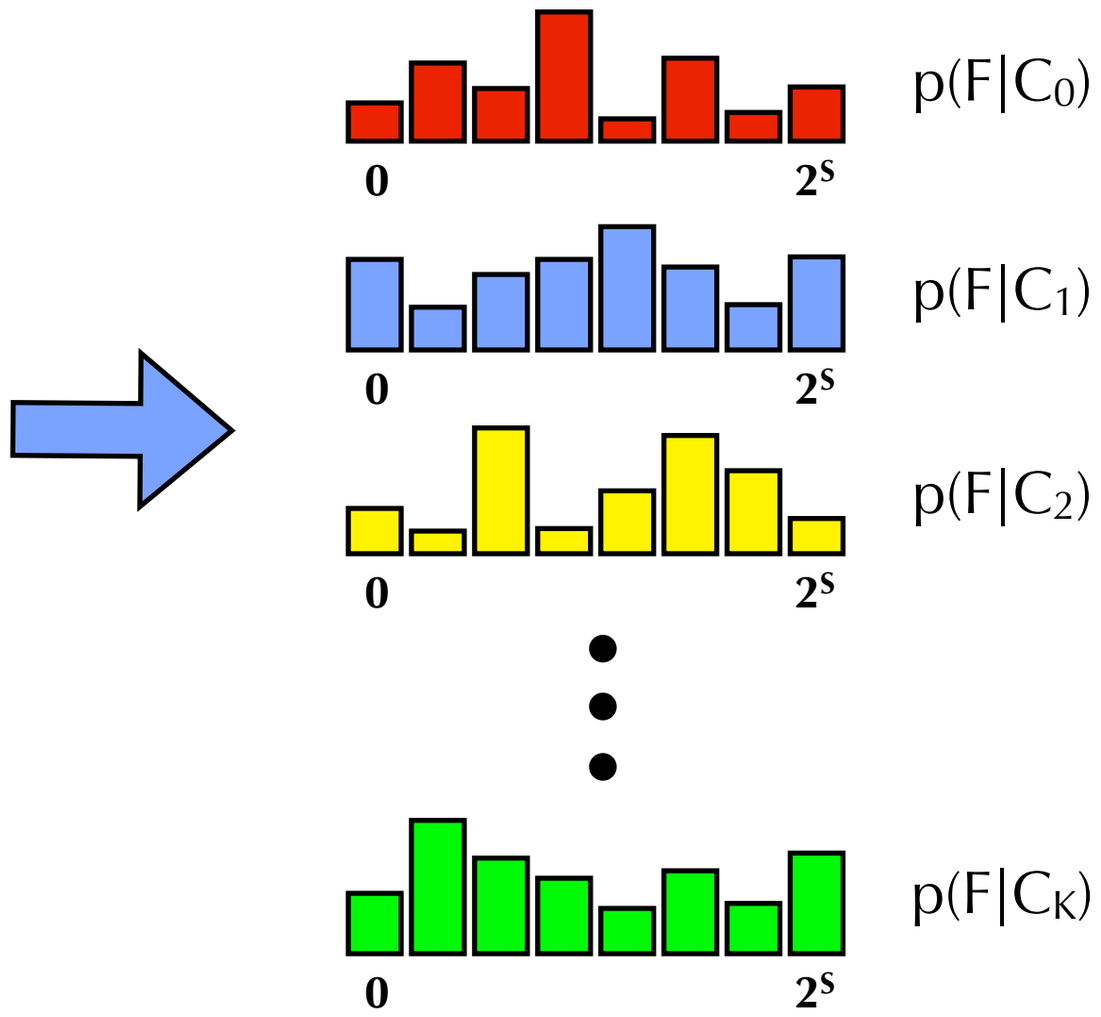
- The output of a fern when applied to a large number of input vectors of the same class is a **multinomial distribution**



Training a Fern

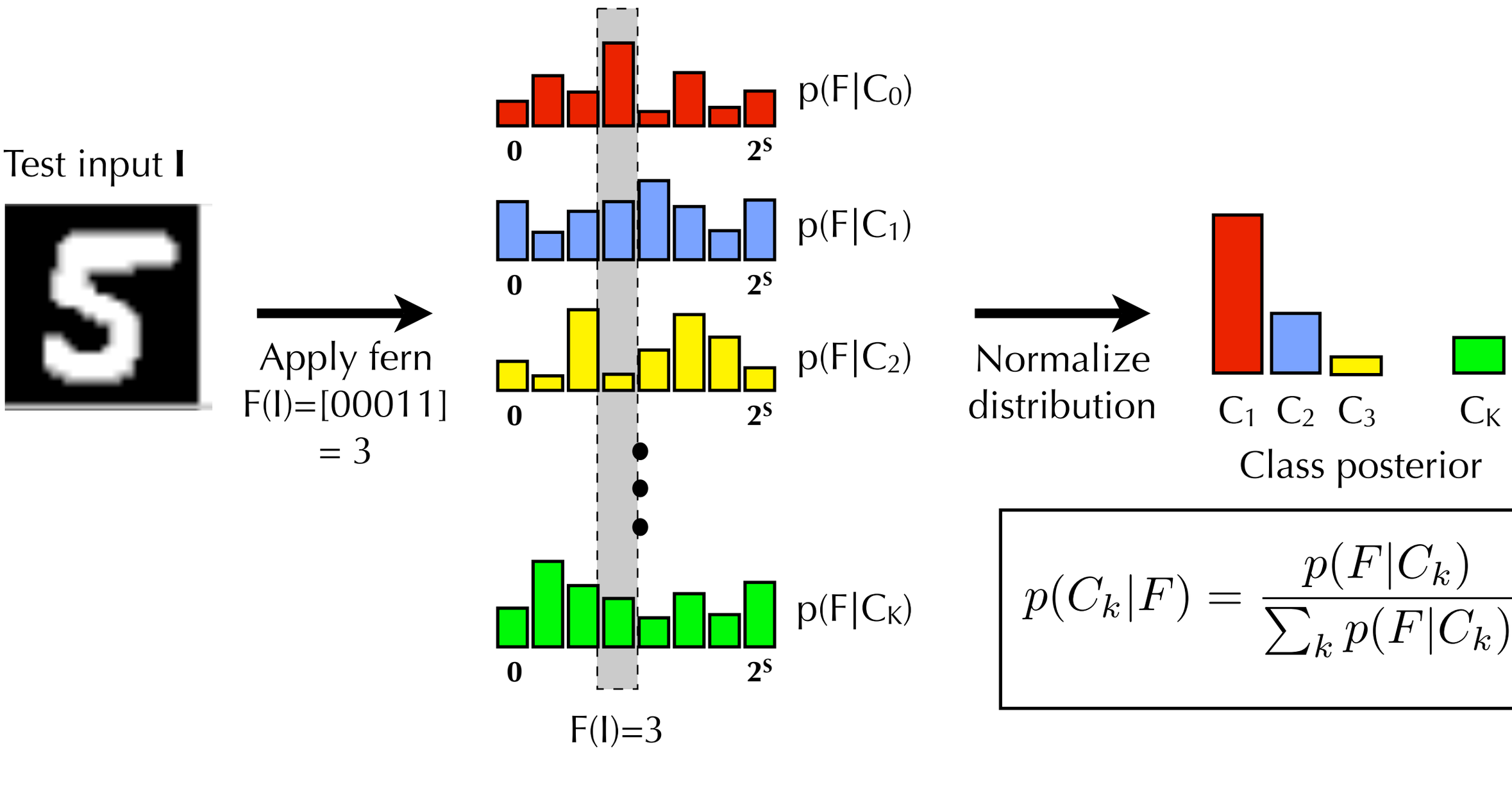
- Apply the fern to each labelled training example $D_{m=}(I_m, c_m)$ and compute its output $F(D_m)$
- Learn multinomial densities $p(F|C_k)$ as histograms of fern output for each class





Classifying using a single Fern

- Given a test input, simply apply the fern and "look-up" the posterior distribution over class label



$$p(C_k|F) = \frac{p(F|C_k)}{\sum_k p(F|C_k)}$$

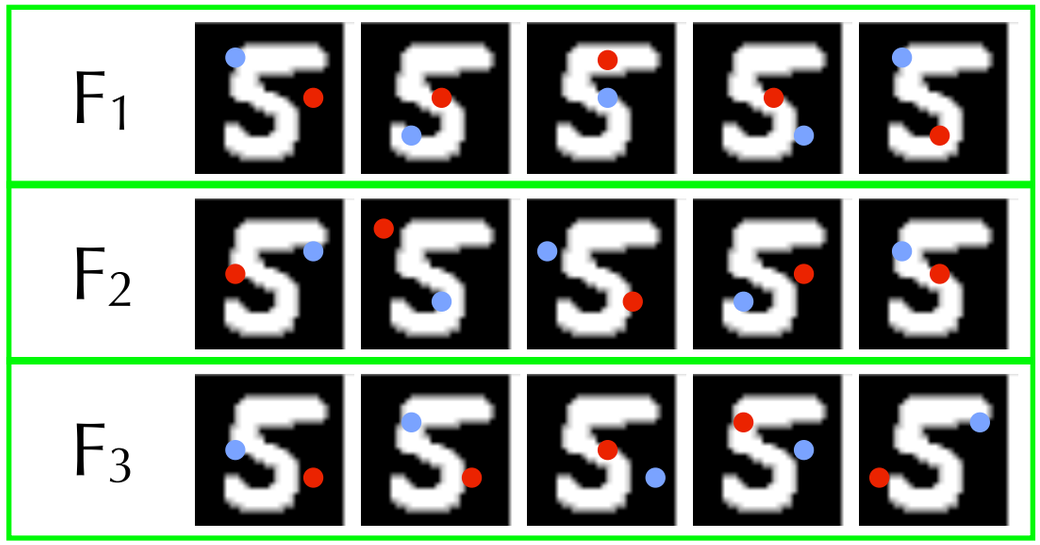
Adding randomness: an ensemble of ferns

- A single fern does not give great classification performance
- **But ..** we can build an ensemble of "independent" ferns by randomly choosing different subsets of features

e.g. $F_1 = \{f_2, f_7, f_{22}, f_5, f_9\}$

$F_2 = \{f_4, f_1, f_{11}, f_8, f_3\}$

$F_3 = \{f_6, f_{31}, f_{28}, f_{11}, f_2\}$

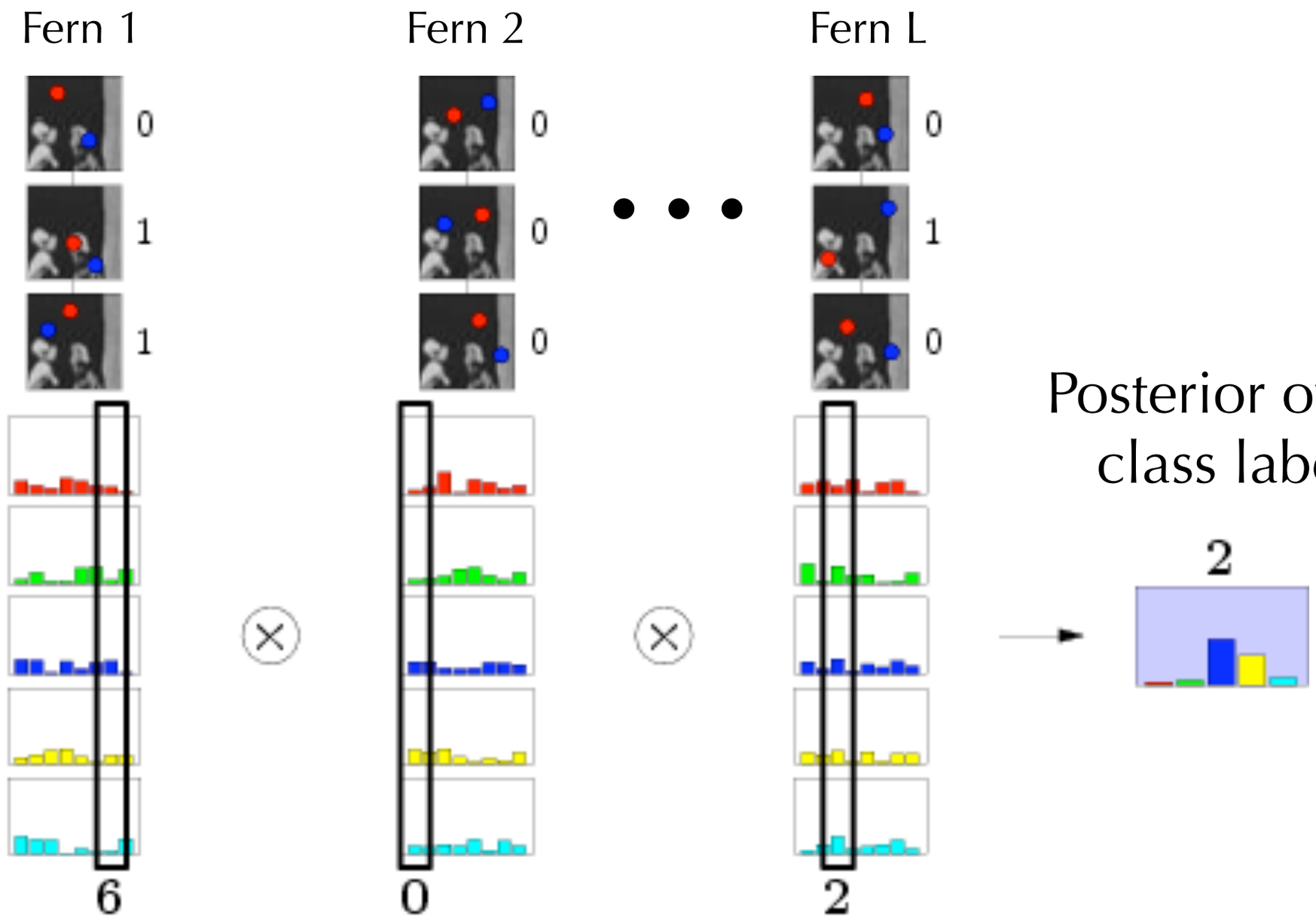


- Finally, combine their outputs using Semi-Naive Bayes:

$$\text{Class}(\mathbf{f}) \equiv \underset{k}{\text{argmax}}\ P(C_k) \prod_{l=1}^{L} P(F_l|C_k)$$

Classifying using Random Ferns

- Having randomly selected and trained a collection of ferns, classifying new inputs involves only simple look-up operations

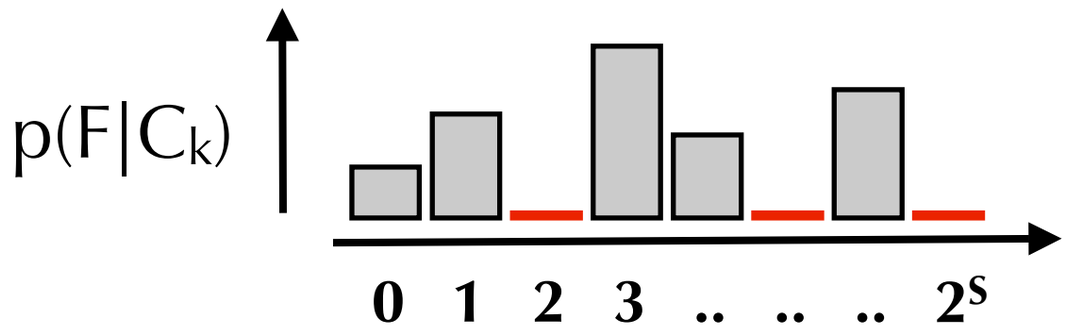


One small subtlety...

- Even for moderate size ferns, the output range can be large

  e.g. fern size S=10, output range = [0 ... 2^{10}] = [0 ... 1024]

- Even with large training set, many "buckets" may see no samples



These zero-probabilities will be problematic for our Semi-Naive Bayes!

- **So ..** assume a Dirichlet prior on the distributions P(F|C_k)
- Assigns a baseline low probability to all possible fern outputs:

$$p(F = z|C_k) = \frac{N(F = z|C_k) + 1}{\sum_{z=0}^{2^S-1}(N(F = z|C_k) + 1)}$$

where N(F=z|C_k) is the number of times we observe fern output equal to z in the training set for class C_k

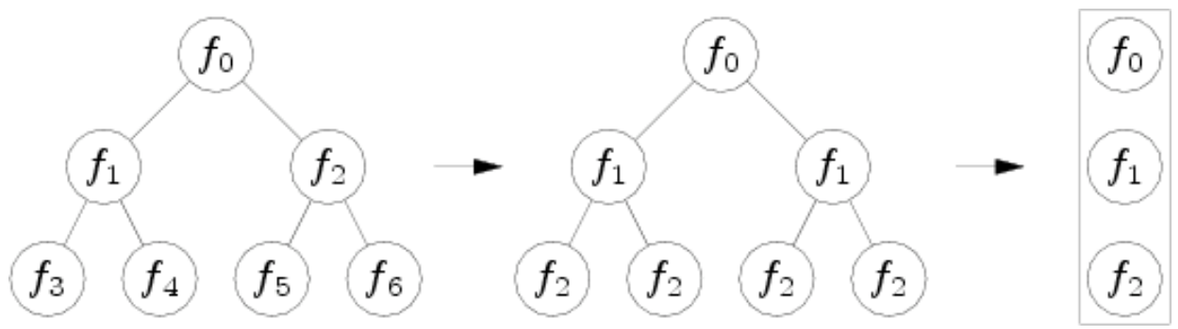
Comparing Forests and Ferns

Forests

- Decision trees directly learn the posterior $P(C_k|F)$
- Applies different sequence of tests in each child node
- Training time grows exponentially with tree depth
- Combine tree hypotheses by averaging

Ferns

- Learn class-conditional distributions $P(F|C_k)$
- Applies the same sequence of tests to every input vector
- Training time grows linearly with fern size S
- Combine hypothesis using Bayes rule (multiplication)

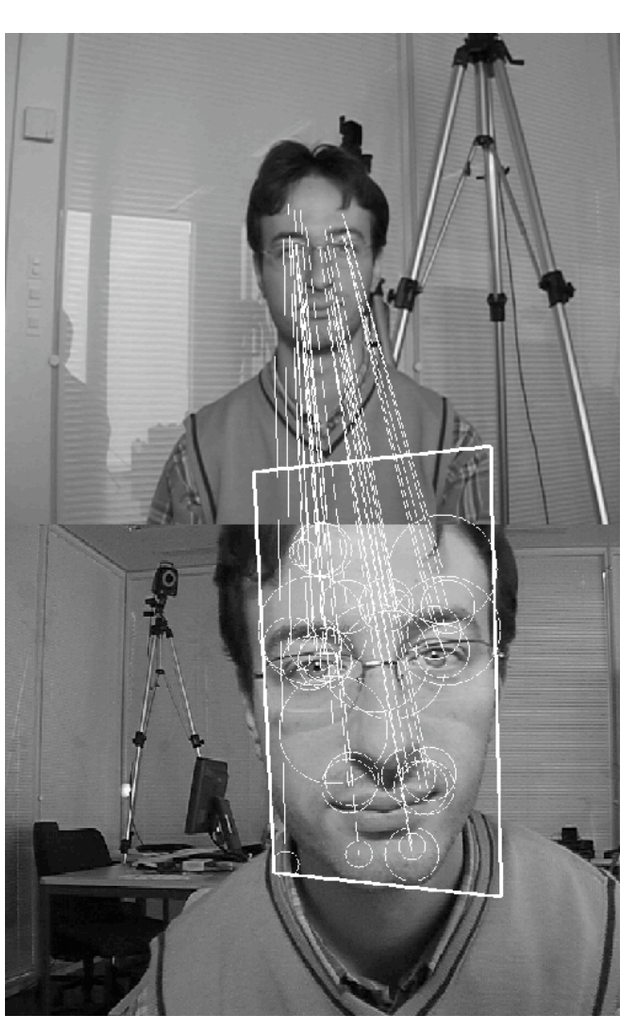


Tree classifier

Fern classifier

Application: Fast Keypoint Matching

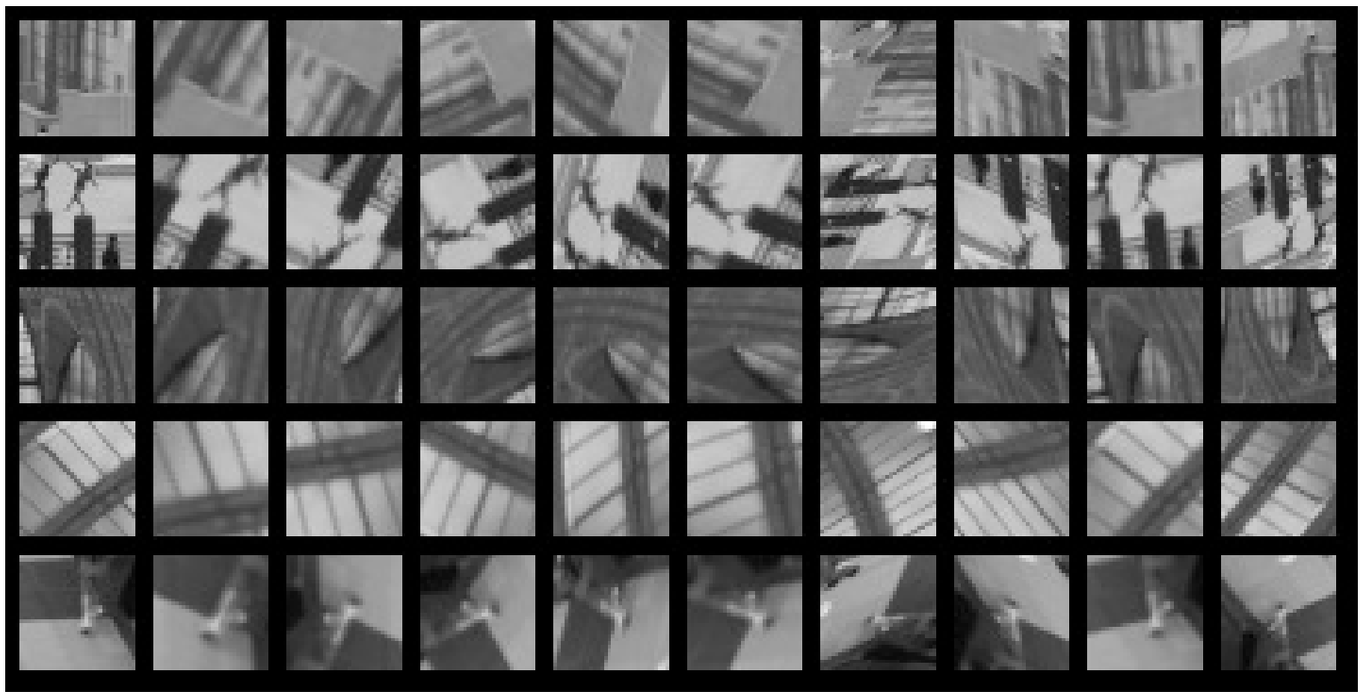
- Ozuysal et al. use Random Ferns for keypoint recognition
- Similar to typical application of SIFT descriptors
- Very robust to affine viewing deformations
- Very fast to evaluate (13.5 microsec per keypoint)



Training

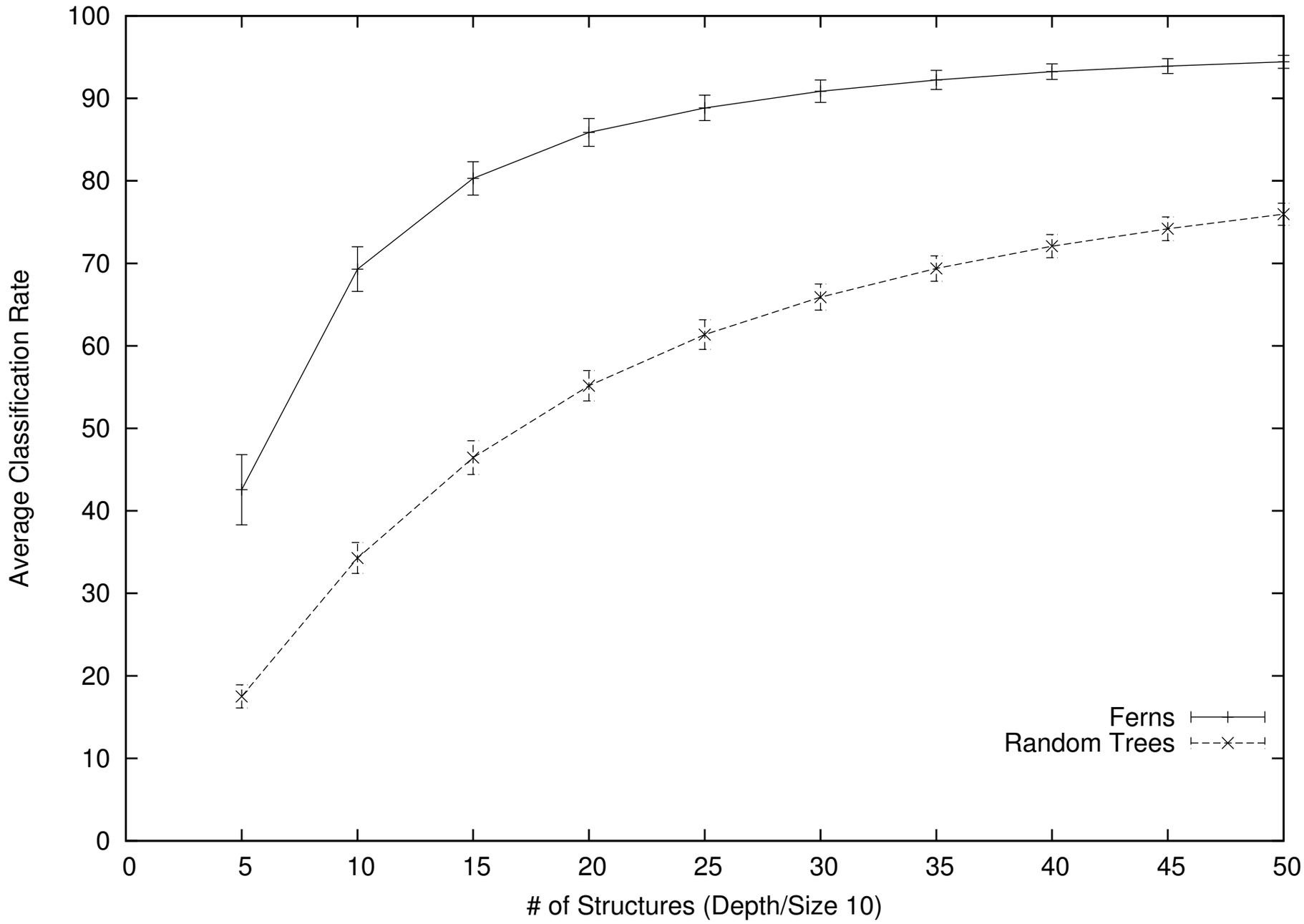
- Each keypoint to be recognized is a separate class
- Training sets are generated by synthesizing random affine deformations of the image patch (10000 samples)

Synthesized viewpoint variation



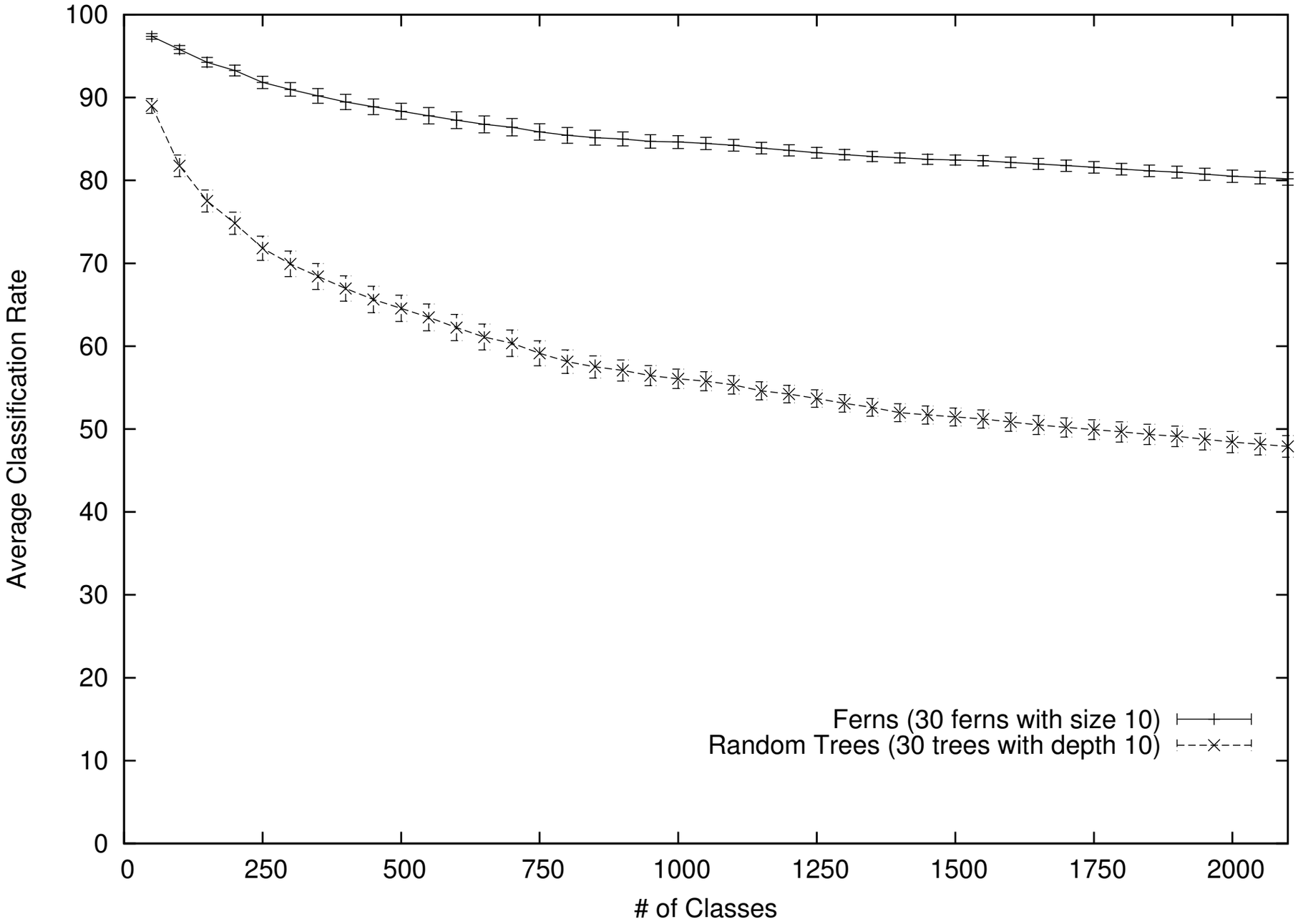
- Features are pairwise intensity comparisons: $f_n(\mathbf{I}) = I(x_a,y_a) > I(x_b,y_b)$
- Fern size S=10 (randomly selected pixel pairs)
- Ensembles of 5 to 50 ferns are tested

Classification Rate vs Number of Ferns



- 300 classes were used
- Also compares to Random Trees of equivalent size

Classification Rate vs Number of Classes (Keypoints)



- 30 random ferns of size 10 were used

Drawback: Memory requirements

- **Fern classifiers can be very memory hungry, e.g**

Fern size = 11

Number of ferns = 50

Number of classes = 1000

RAM = $2^{\text{fern size}}$ * sizeof(float) * NumFerns * NumClasses

= 2048 * 4 * 50 * 1000

= 400 MBytes!

Conclusions?

- Random Ferns are easy to understand and easy to train
- Very fast to perform classification once trained
- Provide a probabilistic output (how accurate though?)
- Appear to outperform Random Forests
- Can be very memory hungry!