Decision-Tree Classifiers for Computer Vision Applications

Avinash Kak
Purdue University

June 24, 2011
1:28am

An RVL Tutorial Presentation

(Presented in Fall 2010, Revised in Summer 2011)

©2011 Avinash Kak, Purdue University

CONTENTS

Section Title	*Page*
What do We Mean by a Computer Vision System That Learns on its Own How to Recognize Objects	3
Entropy	5
Conditional Entropy	10
Average Entropy	12
Using Class Entropy to Discover the Best Feature for Discriminating Between the Classes	13
Constructing a Decision Tree	19
The Perl Module `Algorithm::DecisionTree-1.41`	31
The Python Module `DecisionTree-1.5`	36
Decision Trees for Solving Computer Vision Problems	37
Converting a Decision-Tree Classifier into a Hash-Table Classifier Fast Object Recognition	49
For Digging Deeper	56

What do We Mean by a Computer Vision System that Learns on its Own How to Recognize Objects

- Consider the following computer vision experiment:
  - We show a number of different objects to a sensor system. These objects belong to M different classes.
  - For each object shown, all that we tell the computer is its class label. We do NOT tell the computer how to discriminate between the objects belonging to the different classes.

- We supply a large vocabulary of features to the computer and also provide the computer with tools to extract these features from the sensory information collected from each object. For image data, these features could be color, texture, etc. For range data, the features could be curvature, corners formed by the meeting of the planar surfaces, etc.

- The job given to the computer is: From the data thus collected, the computer must figure out on its how to best discriminate between the objects belonging to the different classes. In other words, the computer must learn on its own what features to use for discriminating between the classes and what features to ignore.

• What we have described above constitutes an exercise in a self-learning computer vision system.

Entropy

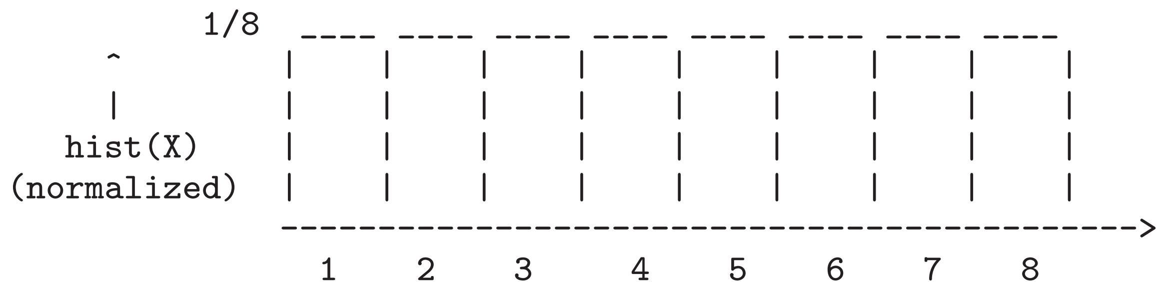
- As we will see later, entropy is a powerful tool that can be used by a computer to determine on its own own as to what features to use and how to carve up the feature space for achieving the best possible discrimination between the classes.

- What is entropy?

- If a random variable X can take N different values, the ith value with probability p_i, we can associate the following entropy with X:

$$H = -\sum_i p_i \log_2 p_i$$

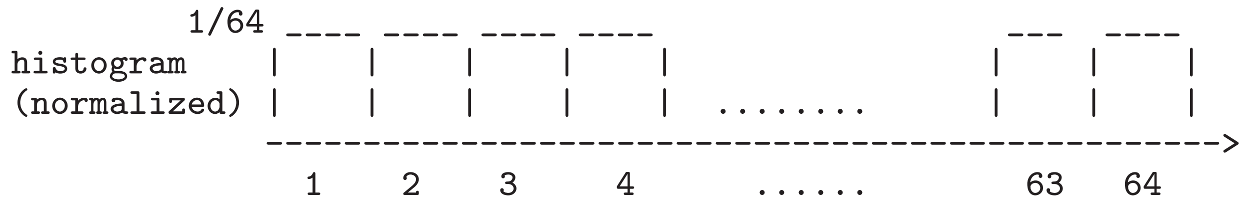
- To gain some insight into what H measures, consider the case when the normalized histogram of the values taken by the random variable X looks like



- In this case, X takes one of 8 possible values, each with a probability of $p_i = 1/8$. For a such a random variable, the entropy is given by

$$
\begin{aligned}
H &= -\sum_{i=1}^{8} \frac{1}{8} \log_2 \frac{1}{8} \\
&= -\sum_{i=1}^{8} \frac{1}{8} \log_2 2^{-3} \\
&= 3 \quad bits
\end{aligned}
$$

- Now consider the following example in which the uniformly distributed random variable X takes one of 64 possible values:

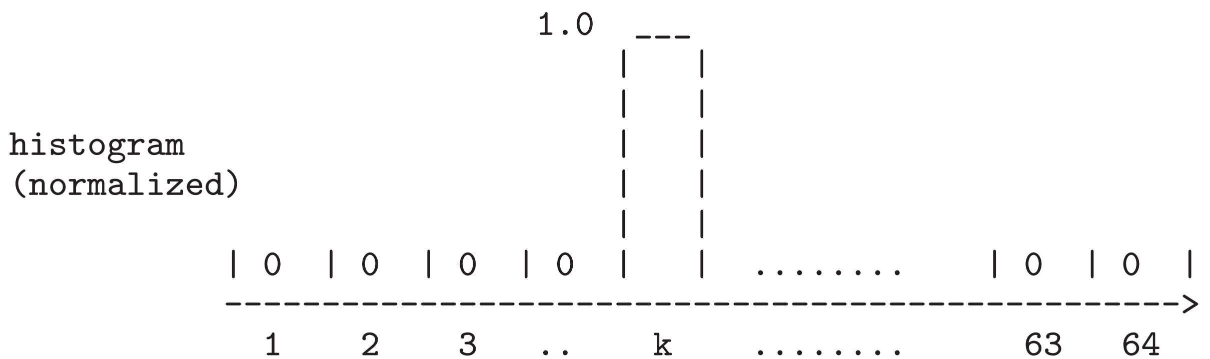


- In this case,

$$
\begin{aligned}
H &= -\sum_{i=1}^{64} \frac{1}{64} \log_2 \frac{1}{64} \\
&= -\sum_{i=1}^{8} \frac{1}{8} \log_2 2^{-6} \\
&= 6 \quad bits
\end{aligned}
$$

- So we see that the entropy, commonly measured in bits, has increased because now we have greater "chaos" in the values of X. It can now take one of 64 values with equal probability.

- Let's now consider an example at the other end of spectrum: We will consider an X that is always known to take on a particular value:



- In this case, we obviously have

$$
\begin{aligned}
p_i &= 1 \qquad\qquad i = k \\
&= 0 \qquad\qquad otherwise
\end{aligned}
$$

- The entropy for such an X would be given by:

$$
\begin{aligned}
H &= -\sum_{i=1}^{N} p_i \log_2 p_i \\
&= -\left[p_1 \log_2 p_1 + ...p_k \log_2 p_k + ... + p_N \log_2 p_N\right]
\end{aligned}
$$

$$= \quad -1 \times \log_2 1 \quad bits$$

$$= \quad 0 \quad bits$$

where we use the fact that as $x \rightarrow 0$, $x \log x \rightarrow 0$ in all of the terms of the summation except when $i = k$.

- So we see that the entropy becomes zero when X has zero chaos.

- **In general, the more nonuniform the probability distribution for an entity, the smaller the entropy associated with the entity.**

Conditional Entropy

- The conditional entropy $H(Y|X)$ measures how much entropy (chaos) remains in Y if we already know the value of the random variable X.

- In general,

$$H(Y|X) \quad = \quad H(X,Y) \; - \; H(X)$$

Note that this definition treats X at a purely symbolic level; that is, without instantiating it to any specific value. Given two random variables X and Y, the entropy contained in both when taken together is $H(X,Y)$. The above definition says that if we know X completely, we have obviously gained $H(X)$ bits of information.

- The formula for $H(X,Y)$ is given by

$$H(X,Y) = -\sum_{i,j} p(x_i, y_j) \log_2 p(x_i, y_j)$$

- As mentioned earlier, the formula for $H(Y|X)$ shown on the previous slide treats X purely symbolically.

- We may now raise the following question: What is the entropy associated with Y if we know that X has taken on a specific value a? The answer to that is:

$$H(Y|X=a) = -\sum_{i} p(y_i|X=a) \times \log_2 p(y_i|X=a)$$

Average Entropy

- Given N random variables X_1, X_2, $\ldots X_N$, we can associate an average entropy with all N random variables by

$$H_{av} = -\sum_{1}^{N} H(X_i) \times p(X_i)$$

Using Class Entropy to Discover the Best Feature for Discriminating Between the Classes

- Consider the following question: Let us say that we are given the measurement data as described on Slides 3 and 4. Let the exhaustive set of features known to the computer be $\{f_1, f_2, ..., f_K\}$.

- Now the computer wants to know as to which of these features is best in the sense of being the most class discriminative.

- How does the computer do that?

- To discover the best feature, all that the computer has to do is to compute the class entropy

$$H(C|f) \quad = \quad \sum_i H(C|f=i) \ \times \ p(f=i)$$

where the notation $f = i$ means that the feature f takes on a value of i. *So, remember, in such notation, i is NOT a particular choice of a feature, but a value taken by the feature f.* The computer selects that features for which $H(C|f)$ is the smallest values.

- Let's now focus on the calculation of the right hand side in the equation shown above.

- The entropy in each term on the right hand side in the equation shown on the previous slide can be calculated by

$$H(C|f=i) \quad = \quad -\sum_m p(C_m|f=i) \times \log_2 p(C_m|f=i)$$

- But how do we figure out $p(C_m|f = i)$? Note that C_m is the name of the mth class and the above summation is over all the classes.

- We will next present two different ways for calculating $p(C_m|f = i)$. The first approach works if we can assume that the objects shown to the sensor system are drawn uniformly from the different classes. If that is not the case, one must use the second approach.

- Our first approach for calculating $p(C_m|f = i)$ is count-based: Let's say we have M classes of objects that we show to a sensor system. We pick objects randomly from the population of all objects belonging to all M classes. Say the sensor system makes K feature measurements, $f_1, f_2, ..., f_K$, on each object. For each feature f_k, the sensor system keeps a count of the number of objects that gave rise to the $f_k = i$ value. Now we estimate $p(C_m|f = i)$ by counting off the number of objects from class C_m that exhibited the $f_k = i$ measurement.

- Our second approach for estimating $p(C_m|f = i)$ uses the Bayes' Theorem:

$$p(C_m|f = i) \quad = \quad \frac{p(f = i|C_m) \ \times \ p(C_m)}{p(f = i)}$$

This formula also allows us to carry out separate measurement experiments for objects belonging to different classes.

- Another advantage of the formula shown at the bottom of the previous slide is that it is no longer a problem if only a small number of objects are available for some of the classes — such non-uniformities in object populations are taken care of by the $p(C_m)$ term.

- The denominator in the formula at the bottom of the previous slide can be taken care of by the required normalization:

$$\sum_m p(C_m|f = i) \quad = \quad 1$$

- What's interesting is that if we do obtain $p(f = i)$ through the normalization mentioned above, we can also use it in the formula for calculating $H(C|f)$ as shown at the top in Slide 14. Otherwise, $p(f = i)$ would need to be estimated directly from the raw experimental data.

- So now we have all the information that is needed to estimate the class entropy $H(C|f)$ for any given feature f by using the formula shown at the top in Slide 14.

- It follows from the nature of entropy (See Slides 5 through 9) that the smaller the value for $H(C|f)$, especially in relation to the value of $H(C)$, the greater the class discriminatory power of f.

- Should it happen that $H(C|f) = 0$ for some feature f, that implies that feature f can be used to identify objects belonging to at least one of the M classes with 100% accuracy.

Constructing a Decision Tree

- Now that you know how to use the class entropy to find the best feature that will discriminate between the classes, we will now extend this idea and show how you can construct a decision tree. Subsequently the tree may be used to classify future samples of data.

- But what is a decision tree?

- For those not familiar with decision tree ideas, the traditional way to classify multidimensional data is to start with a feature space whose dimensionality is the same as that of the data.

- In the traditional approach, each feature in the space would correspond to the attribute that each dimension of the data measures. You then use the training data to carve up the feature space into different regions, each corresponding to a different class. Subsequently, when you are trying to classify a new data sample, you locate it in the feature space and find the class label of the region to which it belongs. One can also give the data point the same class label as that of the nearest training sample. (This is referred to as the nearest neighbor classification.)

- A decision tree classifier works differently.

- When you construct a decision tree, you select for the root node a feature test that can be expected to maximally disambiguate the class labels that could be associated with the data you are trying to classify.

- You then attach to the root node a set of child nodes, one for each value of the feature you chose at the root node. Now at each child node you pose the same question that you posed when you found the best feature to use at the root node: What feature at the child node in question would maximally disambiguate the class labels to be associated with a given data vector assuming that the data vector passed the root node on the branch that corresponds to the child node in question. The feature that is best at each node is the one that causes the maximal reduction in class entropy at that node.

- Based on the discussion in the previous section, you already know how to find the best feature at the root node of a decision tree. Now the question is: How we do construct the rest of the decision tree?

- What we obviously need is a child node for every possible value of the feature test that was selected at the root node of the tree.

- Assume that the feature selected at the root node is f_k and that we are now on one of the child nodes hanging from the root. So the question now is how do we select the best feature to use at any of these child nodes.

- The root node feature was selected as that f which minimized $H(C \mid f)$. With this choice, we ended up with the feature f_k at the root. The feature to use at the child on the branch $f_k = i$ will be selected as that $f \neq f_k$ which minimizes $H(C \mid f_k = i, f)$.

- That is, for any feature f not previously used at the root, we calculate the entropy when we are in the $f_k = i$ branch:

$$H(C \mid f, f_k = i) \;=\; \sum_j H(C|f = j, f_k = i) \;\times\; p(f = j, f_k = i)$$

Whichever feature f yields the smallest value for the above entropy on the left will become the feature test of choice at the children of the root.

- Strictly speaking, the entropy formula shown above for the calculation of average entropy not correct. For the summation shown on the right to yield a true average, the formula shown would need to be expressed as*

$$H(C \mid f, f_k = i) \;=\; \sum_j H(C|f = j, f_k = i) \times \frac{p(f = j, f_k = i)}{\sum_j p(f = j, f_k = i)}$$

*Some folks refer to such normalizations in the calculation of average entropy as "JZ Normalization"— after Padmini Jaikumar and Josh Zapf.

- The component entropies in the above summation on the right would be given by

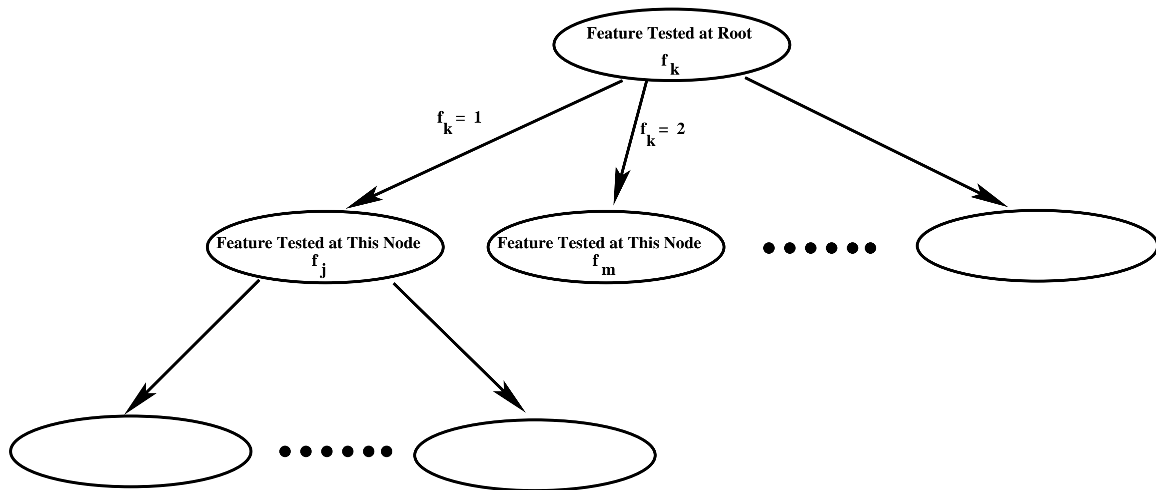
$$H(C \mid f = j, f_k = i) \;=\; -\sum_m p(C_m \mid f = j, f_k = i) \times \log_2 p(C_m \mid f = j, f_k = i)$$

for any given feature $f \neq f_k$.

- The conditional probability needed above is estimated using Bayes Theorem:

$$\begin{aligned} p(C_m \mid f = j, f_k = i) &= \frac{p(f = j, f_k = i | C_m) \;\times\; p(C_m)}{p(f = j, f_k = i)} \\ &= \frac{p(f = j | C_m) \times p(f_k = i | C_m) \times p(C_m)}{p(f = j) \times p(f_k = i)} \end{aligned}$$

where the second equality is based on the assumption that the features are statistically independent.



- You will add other child nodes to the root in the same manner, with one child node for each value that can be taken by the feature f_k.

- This process can be continued to extend the tree further to result in a structure that will look like what is shown in the figure on the next slide.

- Now we will address the very important issue of the stopping rule for growing the tree. That is, when does a node get a feature test so that it can be split further and when does it not?

- A node N is assigned the entropy that resulted in its creation. For example, the root gets the entropy $H(C)$ computed from the class priors.

- The children of the root are assigned the entropy $H(C \mid f_k)$ that resulted in their creation.

- A child N of the root that is on the branch $f_k = i$ will get its own feature test (and will be split further) if and only if we can find a feature f_j such that $H(C \mid f_j, f_k = i)$ is less than the entropy $H(C|f_k)$ at N.

- If the condition $H(C \mid f_k = i, f) \ < \ H(C \mid f_k)$ cannot be satisfied at the child node $f_k = i$ of the root for any feature $f \neq f_k$, the child node remains without a feature test and becomes a leaf node of the decision tree.

- Another reason for a node to become a leaf node is that we have used up all the features along that branch up to that node.

- That brings us to the last important issue related to the construction of a decision tree: associating class probabilities with each node of the tree.

- As to why we need to associate class probabilities with the nodes in the decision tree, let us say we are given for classification a new data vector consisting of features and their corresponding values.

- For the classification of the new data vector mentioned above, we will first subject this data vector to the feature test at the root. We will then take the branch that corresponds to the value in the data vector for the root feature.

- Next, we will subject the data vector to the feature test at the child node on that branch. We will continue this process until we have used up all the feature values in the data vector. That should put us at one of the nodes, possibly a leaf node.

- Now we wish to know what the residual class probabilities are at that node. These class probabilities will represent our classification of the new data vector.

- If the feature tests along a path to a node in the tree are $f_k = i, f_m = l, \ldots$, we will associate the following class probability with the node:

$$p(C_m \mid f_k = i, f_m = l, \ldots)$$

for $m = 1, 2, \ldots, M$ where M is the number of classes.

- The above probability may be estimated with Bayes Theorem:

$$p(C_m \mid f_k = i, f_m = l, \ldots) = \frac{p(f_k = i, f_m = l, \ldots \mid C_m) \times p(C_m)}{p(f_k = i, f_m = l, \ldots)}$$

- If we again use the notion of statistical independence between the features both when they are considered on their own and when considered conditioned on a given class, we can write:

$$p(f_k = i, f_m = l, \ldots) = \prod_{f \ along \ branch} p(f = v)$$

$$p(f_k = i, f_m = l, \ldots \mid C_m) = \prod_{f \ along \ branch} p(f = v \mid C_m)$$

The Perl Module `Algorithm::DecisionTree-1.41` for Decision-Tree Induction and Classification

- The goal of this section is to introduce the reader to some of the more important functions in my Perl module **Algorithm::DecisionTree** that can be downloaded from

  http://search.cpan.org/~avikak/Algorithm-DecisionTree-1.41/lib/Algorithm/DecisionTree.pm

  The above URL is supposed to be one continuous string. If you cannot copy-and-paste it in your browser, just do a Google search on "Algorithm::DecisionTree" and go to **Version 1.41** when you get to the CPAN page for the module.

- Looking at the formulas in the previous section, we obviously need to compute the following sort of marginal, joint, and conditional probabilities:

$$
\begin{array}{l}
p(C_m) \\
p(f = v) \\
p(f = v \mid C_m) \\
p(C_m \mid f = v) \\
p(f_k = v_k, f_m = v_m \ldots) \\
p(f_k = v_k, f_m = v_m \ldots \mid C_m) \\
p(C_m \mid f_k = v_k, f_m = v_m \ldots)
\end{array}
$$

- And the following sorts of entropies:

$$
\begin{array}{l}
H(C \mid f) \\
H(C \mid f = v) \\
H(C \mid f_k = v_k, f_m = v_m \ldots)
\end{array}
$$

- We will now familiarize the reader with the functions in the Algorithm::DecisionTree module that compute the entities listed above.

- In the module Algorithm::DecisionTree, the following functions compute the required probabilities:

$$
\begin{array}{l}
prior_probability_for_class(class_name) \\
probability_for_feature_value(feature, value) \\
probability_for_feature_value_given_class(feature, value, class) \\
probability_for_a_class_given_feature_value(class, feature, value) \\
probability_of_a_sequence_of_features_and_values(\\
\qquad array_features_and_values) \\
probability_of_a_sequence_of_features_and_values_given_class(\\
\qquad class, array_of_features_and_values) \\
probability_for_a_class_given_sequence_of_features_and_values(\\
\qquad class, array_of_features_and_values)
\end{array}
$$

- And the following functions compute the required entropies:

$$
\begin{array}{l}
class_entropy_for_a_given_feature(feature) \\
class_entropy_for_a_given_feature_and_given_value(feature, value) \\
class_entropy_for_a_given_sequence_of_feature_values(\\
\qquad array_features_and_values)
\end{array}
$$

- The following functions in the module compute the decision tree from the probabilities and entropies listed above:

$$construct_decision_tree_classifier()$$
$$recursive_descent(root_node)$$

where the first function returns an instance of type Node that serves as the root node of the decision tree. This node is then supplied as the argument to the second function named above.

- A new data vector is classified with the help of the following functions:

$$classify()$$
$$recursive_descent_for_classification(root_node)$$

where the argument $root_node$ in the call to the second function represents the root node of the decision tree.

- Before the module can invoke any of the functionality described above, you must supply it with a training datafile that must be formatted as described in the documentation page of the module.

- The module also allows you to generate your own training and testing datasets.

- To generate your own data requires that you supply a parameter file to the module that contains the names of the classes, the names you wish to use for the features, and the different possible values taken by each of the future. Another critical part of the parameter file is the biasing information which tells the training data generator how you want the training samples to be biased (probabilistically speaking) for the different classes.

The Python Module `DecisionTree-1.5` for Decision-Tree Induction and Classification

- If you would rather use Python, you might want to check out my Python module `DecisionTree-1.5`. You can download it from

  http://pypi.python.org/pypi/DecisionTree/1.5

- The Python version should work faster for large decision trees since it uses probability and entropy caching much more extensively than the Perl module.

- Apart from the speedup achieved by caching, the overall functionality of the Python module is the same as that of the Perl module.

Decision Trees for Solving Computer Vision Problems

- The approach presented so for the induction of a decision tree works well when features take on values that are symbolic, as would be the case for an application in which the class names, the feature names, and the values for the features are something along the following lines:

```
class names:    malignant    benign
class priors:   0.4          0.6

feature: smoking
values:  heavy    medium    light    never

feature: exercising
values:  never    occasionally   regularly

feature: fatIntake
values:  low    medium   heavy

feature: videoAddiction
values:  none   low   medium   heavy
```

- This does not mean that the feature values are not allowed to be numerical, but any numbers would be treated purely symbolically.

- The fact that features values are only allowed to be symbolic prevents the types of decision trees constructed by the Algorithm::DecisionTree module from being useful in computer vision applications.

- In computer vision, a feature may take on a discrete value drawn from a set of very large cardinality, or, for that matter, a value drawn from a continuous interval.

- Consider, for example, using color as a feature in object recognition.

- If we consider all three primary color components (R,G,B) together and assume that each is quantized to 256 levels, the value of the color feature will be one out of 16,777,216 ($= 256 \times 256 \times 256$) values.

- An alternative would be to use three separate color features, one for each primary color. So the three features could be labeled $colorR$, $colorG$ and $colorB$. Now each feature would take on only one out of 256 values.

- In either representation, we have the following problem: Too large and much too fine-grained a fan-out at the nodes of the decision tree that use these features.

- The fine-grained fan-out is particularly troublesome because of the ever-present measurement noise in these types of features.

- To choose a path in the decision tree on the basis of, say, the value of $colorR$ being 123 may or may not make sense considering that the decision paths for the same feature being, say, 122 or 124 (or any of the other nearby values) may be just as applicable, if not more.

- For computer vision applications, the fan-out logic at each node is much better structured along the following lines:
  - Let's say that our root node tests for feature f_k and the range of values that this feature can take is from v_{min} to v_{max}. (We choose f_k for the root since $f = f_k$ gives us the minimum value for $H(C|f)$. This entropy can be calculated as earlier.)
  - Our goal now is to choose a decision point v_d in the interval $[v_{min}, v_{max}]$ so that we get the smallest value for the average of the class entropies calculated separately over the intervals $[v_{min}, v_d)$ and $[v_d, v_{max})$, especially so in relation to the class entropy calculated over the entire interval $[v_{min}, v_{max}]$.

– That is, we first want to calculate

$$H_< = H(C \mid f_k < v_d)$$

$$H_> = H(C \mid f_k \geq v_d)$$

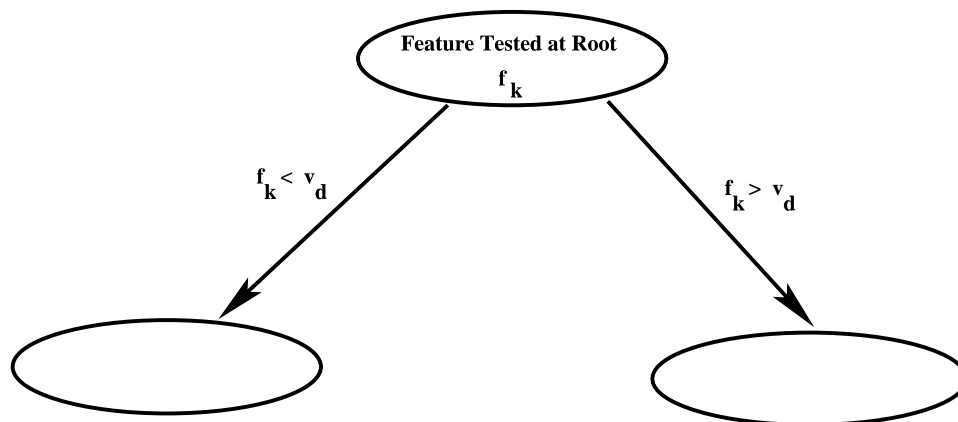
and find the average of the two:

$$H_{av} = H_< \times p(f_k < v_d) \; + \; H_> \times p(f_k \geq v_d)$$

– We choose for v_d that point which yields the smallest value for H_{av}.

– If H_{av} is not less than $H(C \mid f_k)$ we do NOT expand the node with f_k as the feature test. So on this decision path, this is where the decision tree will stop.

– However, if H_{av} is less than $H(C \mid f_k)$, we now create two child nodes and the decision tree will look like what is shown in the figure on the next slide (assuming that root node tested for feature f_k).



- The entropy $H(C \mid f_k < v_d)$ needed for the calculations at the root node can be estimated by

$$H(C \mid f_k < v_d) \;=\; -\sum_m p(C_m \mid f_k < v_d) \times \log_2 p(C_m \mid f_k < v_d)$$

- The probability $p(C_m \mid f_k < v_d)$ may be estimated by culling from the training data all those object instances whose f_k feature measurements satisfy the condition $f_k < v_d$ and then counting the number of instances in which the object identity corresponds to the class label C_m.

- However, for greater theoretical validity, it is better to estimate the probability $p(C_m \mid f_k < v_d)$ by using the Bayes Theorem:

$$p(C_m \mid f_k < v_d) \ = \ \frac{p(f_k < v_d \mid C_m) \times p(C_m)}{p(f_k < v_d)}$$

where the denominator can be treated merely as a normalization constant that may be estimated from the constraint

$$\sum_m p(C_m \mid f_k < v_d) \quad = \quad 1$$

- Obviously, we can calculate the entropy $H(C \mid f_k \geq v_d)$ needed at the root in a similar manner.

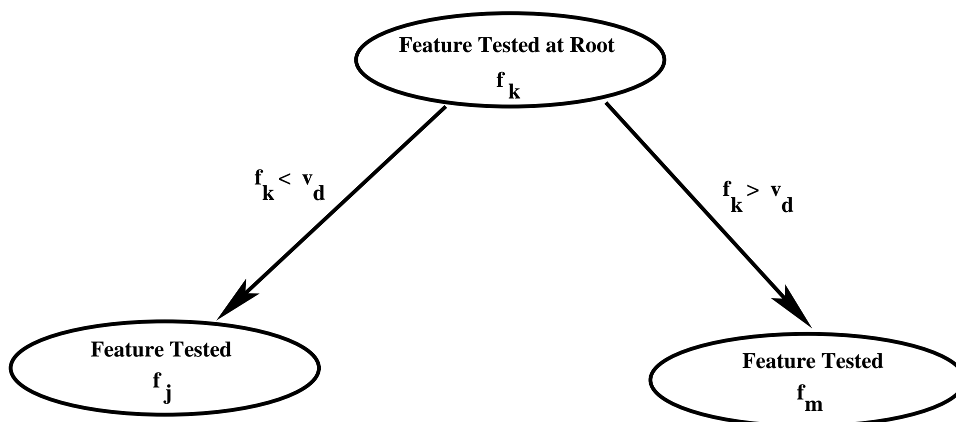
- Basically what we have done so far is to bipartition the range of values that can be taken on by the f_k, the bifurcation point being a decision threshold v_d that yields the greatest reduction in the entropy.

- Having expanded the root node with feature test on f_k, let's now consider the branch $f_k < v_d$ of the decision tree.

- For the next feature test on the left branch emanating from the root, we now search through all features other than f_k and, for each such feature f, we compute $H(C \mid f_{k<}, f)$ where $f_{k<}$ means $f_k < v_d$. The feature that gives us the smallest value for this entropy is chosen as the feature test for the child node.

- Let us say that the above reasoning has given us f_j as the feature to test on in the left child of the root.

- We go through similar reasoning for the right branch emanating from the root. We calculate $H(C \mid f_{k>}, f)$ where $f_{k>}$ means $f_k \geq v_d$ for all features $f \neq f_k$ and we choose for the feature test that f which yields the smallest value for the entropy. Let this feature be f_l. The decision tree will now look like what is shown in the figure below.



- To extend the tree further, we now bipartition the range of values at each of the two second-level nodes.

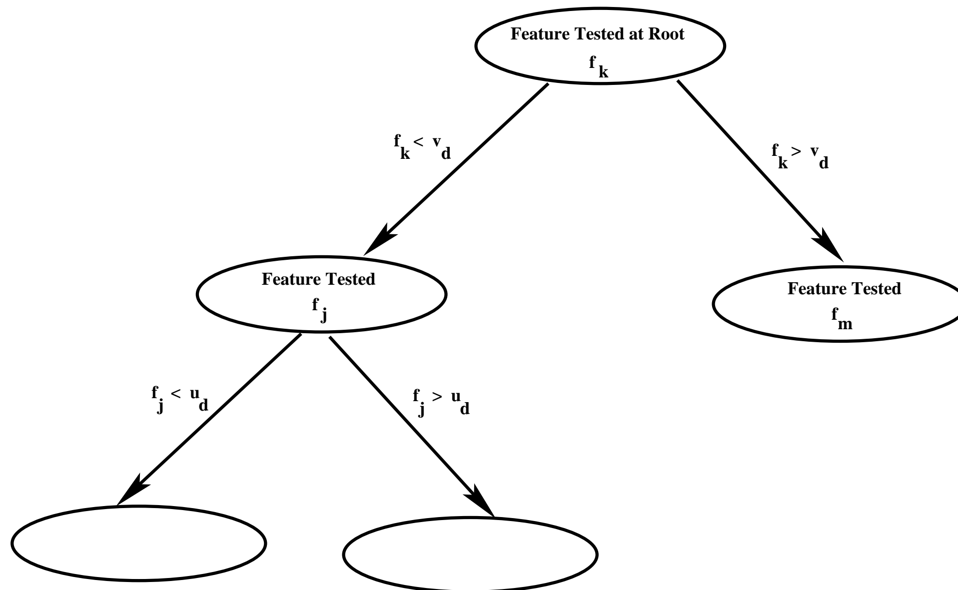
- Focusing on the feature test f_j in the left node, let the range of values for this feature be $[u_{min}, u_{max}]$. We now find a decision threshold u_d in this interval that yields the largest reduction in the average of the entropies calculated as follows:

$$
\begin{aligned}
H_< &= H(C \mid f_k < v_d, f_j < u_d) \\
H_> &= H(C \mid f_k \geq v_d) \\
\\
H_{av} &= H_< \times p(f_k < v_d, f_j < u_d) \ + \ H_> \times p(f_k \geq v_d, f_d \geq u_d)
\end{aligned}
$$

  In the above calculations, we choose for the decision point u_d that value which minimizes H_{av}.

- If it turns out that this H_{av} is not smaller than the value of $H(C \mid f_{k<}, f_j)$ calculated previously, we do not bifurcate this node and stop growing the decision tree on this branch.

- At this point, our decision tree will look like what is shown below:



- As we continue growing the decision tree in this manner, an interesting point of difference arises between the traditional decision trees we talked about earlier and the decision trees needed for computer vision applications When we consider the features for the feature tests to use at the children of the node where we just used the f_j feature for our feature test, we throw the parent node's feature f_k back into contention.

- In general, this difference between the traditional decision trees and the decision trees needed for computer vision is more illusory than real. That is because when considering the root node feature f_k at the the third-level nodes in the tree, the values of f_k will be limited to the interval $[v_{min}, v_d)$ in the left children of the root and to the interval $[v_d, v_{max}$ in the right children of the root. Testing for whether the value of the feature f_k is in, say, the interval $[v_{min}, v_d)$ is not the same feature test as testing for whether this value is in the interval $[v_{min}, v_{max}]$.

Converting a Decision-Tree Classifier into a Hash-Table Classifier for Fast Object Recognition

- Once you have constructed a decision tree for classification, it can be converted into a hash table for fast object recognition.

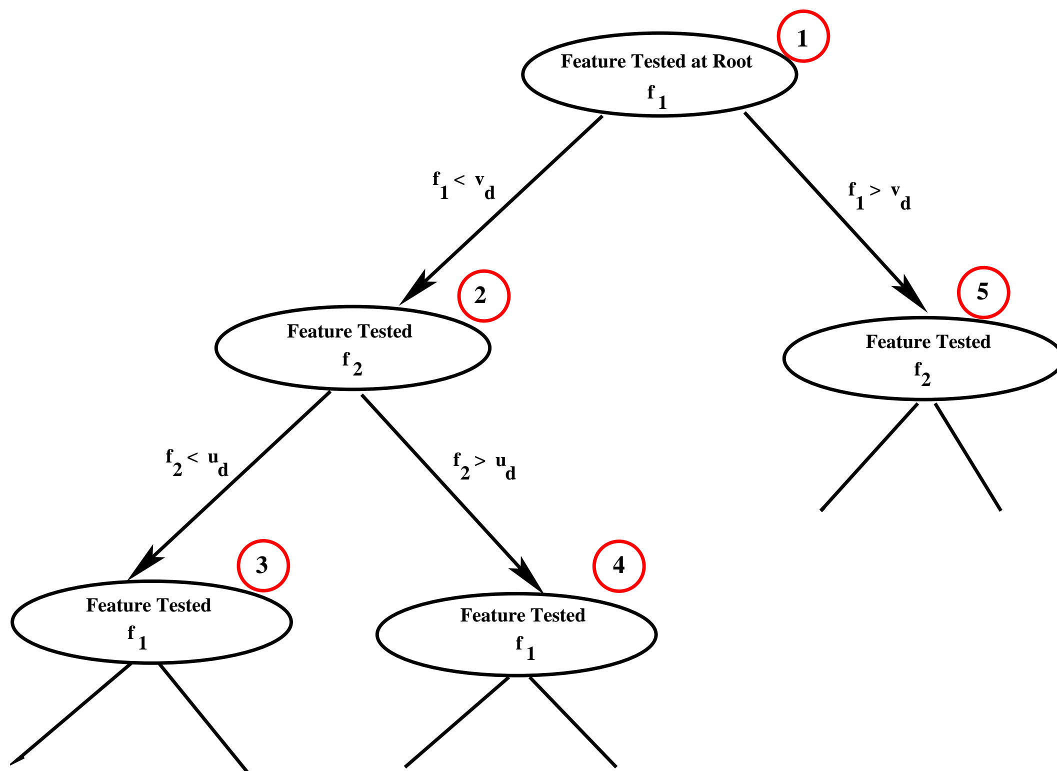
- Creating such hash tables is straightforward for the traditional decision trees in which the feature values are all symbolic (even when numeric, since they would be treated symbolically).

- So in the rest of this tutorial, I will focus on how such a hash table may be constructed for a decision tree of the sort described in the previous section.

- As you now know, a node in a decision trees for computer vision applications has only two children, unless the node is a leaf node, in which case it has no children. (Such trees are also called binary trees.)

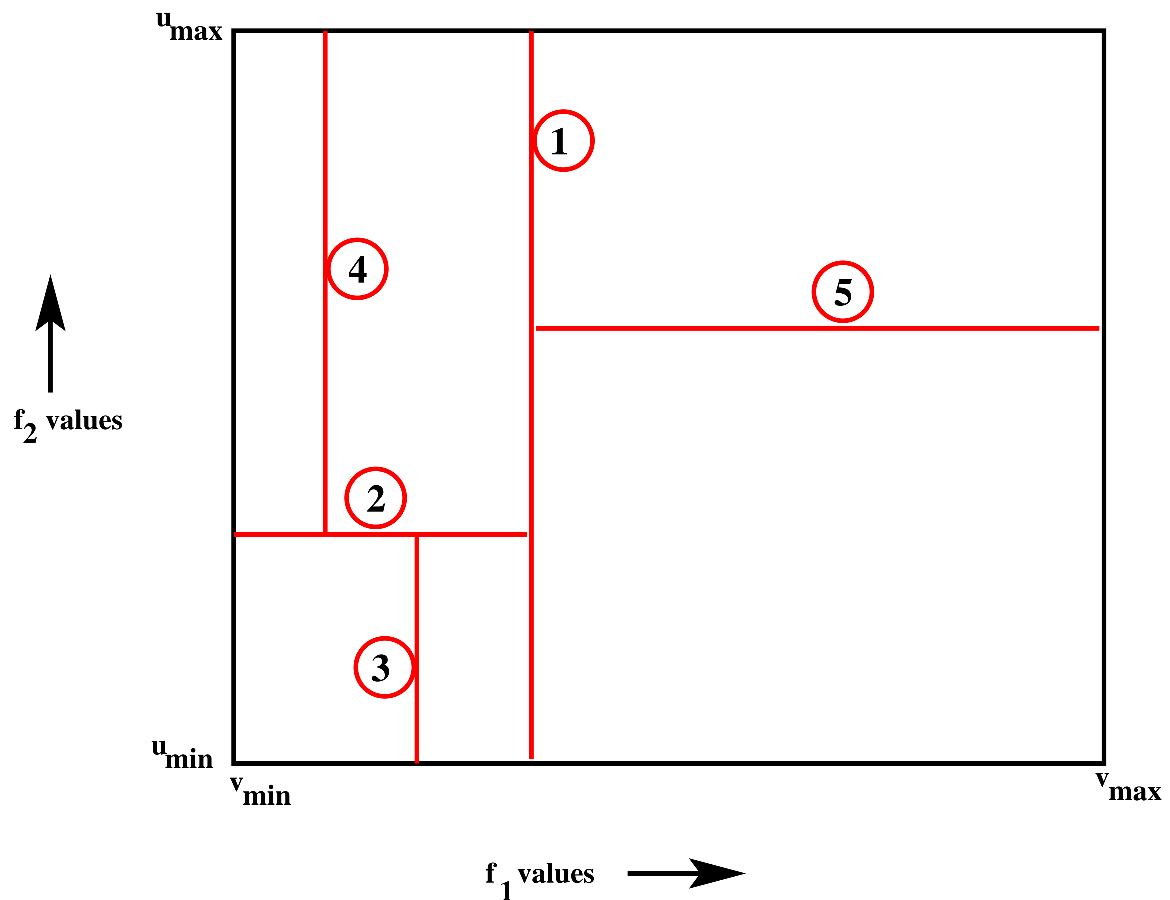
- For my explanation, I will assume that we have only two features f_1 and f_2 that we will use to recognize objects. The feature f_1 could stand for the color of the object and the feature f_2 could be a measure of its texture in the image.

- With just the two features f_1 and f_2, let say that our decision tree looks like what is shown in the figure on the next slide.



- The numbers in red circles in the decision tree shown above indicate the order in which the nodes were visited.

- Note that when we create two child nodes at any node in the tree, we are dividing up a portion of the underlying feature space, the portion that can be considered to be allocated to the node in question.



- Each node being in charge of a portion of the feature space and how it gets partitioned when we create two child nodes there is illustrated by the figure above. In this figure, the circled numbers next to the partitioning lines correspond to the numbers attached to the nodes in the decision tree on the previous slide.

- It is good for mental imagery to associate the entropies we talked about earlier with the different portions of the feature space. For example, the entropy $H(C \mid f_{1<})$ obviously corresponds to the portion of the feature space to the left of the vertical dividing line that has the number 1 in the figure. Similarly, the entropy $H(C \mid f_{1<}, f_{2<})$ corresponds to the portion that is to the left of the vertical dividing line numbered 1 and below the horizontal dividing line numbered 2.

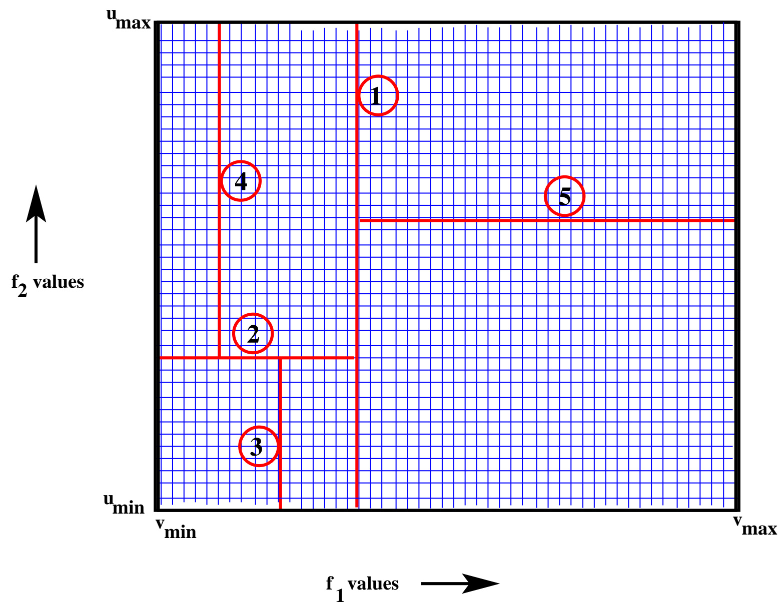
- As we grow the decision tree, our goal must be to reach the nodes that are pure or until there is no further reduction in the entropy in the sense we talked about earlier. A node is pure if it has zero entropy. Obviously, the classification made at that node will be unambiguous.

- After we have finished growing up the tree, we are ready to convert it into a hash table.

- We first create a sufficiently fine quantization of the underlying feature space so that the partitions created by the decision tree are to the maximum extent feasible on the quantization boundaries.

- We are allowed to use different quantization intervals along the different features to ensure the fulfillment of this condition.

- The resulting divisions in the feature space will look like what is shown in the figure on the next slide.



- The tabular structure shown above can now be linearized into a 1-D array of cells, with each cell pointing to the unique class label that corresponds to that point in the feature space (assuming that portion of the feature space is owned by a pure node). However, should it be the case that the portion of the feature space from which the cell is drawn is impure, the cell in our linearized structure can point to all of the applicable class labels and the associated probabilities.

For Digging Deeper ....

- During her Ph.D dissertation in the Robot Vision Lab at Purdue, Lynne Grewe created a full-blown implementation of a decision-tree/hashtable based classifier for recognizing 3D objects in a robotic workcell. It was a pretty amazing dissertation. She not only implemented the underlying theory, but also put together a sensor suite for collecting the data so that she could give actual demonstrations on a working robot.

- The learning phase in Lynne's demonstrations consisted of merely showing 3D objects to the sensor suite. For each object shown, the human would tell the computer what its identity and pose was.

- From the human supplied class labels and pose information, the computer constructed a decision tree in the manner described in the previous section. Subsequently, the decision tree was converted into a hash table for fast classification.

- The testing phase consisted of the robot utilizing the hash table constructed during the learning phase to recognize the objects and to estimate their poses. The proof of the pudding lay in the robot successfully manipulating the objects.

- The details of this system are published in

```
Lynne Grewe and Avinash Kak, "Interactive Learning
    of a Multi-Attribute Hash Table Classifier for
    Fast Object Recognition," Computer Vision and
    Image Undersgtanding, pp. 387-416, Vol. 61,
    No. 3, 1995.
```

Acknowledgment

In one form or another, decision trees have been around for the last fifty years. However, their popularity during the last decade is owing to the entropy-based method proposed by Ross Quinlan for their construction. Fundamental to Quinlan's approach is the notion that a decision node in a tree should be split only if the entropy at the ensuing child nodes taken together will be less than the entropy at the node in question. The algorithm presented in this tutorial is based on the same idea.

I have enjoyed several animated conversations with Josh Zapf and Padmini Jaikumar on the topic of decision tree induction. (As a matter of fact, this tutorial was prompted by some early conversations with Josh regarding decision trees, in general, and regarding Lynne Grewe's implementation of decision-tree induction for computer vision applications.) We are still in some disagreement regarding the computation of average entropies at the nodes of a decision tree. But then life would be very dull if people always agreed with one another all the time.