DECISION TREES: How to Construct Them and How to Use Them for Classifying New Data

Avinash Kak
Purdue University

December 6, 2016
3:58pm

An RVL Tutorial Presentation

(First presented in Fall 2010; updated December 2016)

CONTENTS

1. Introduction

- Let's say your problem involves making a decision based on N pieces of information. Let's further say that you can organize the N pieces of information and the corresponding decision as follows:

```
  f_1     f_2       f_3    ......     f_N      =>   DECISION
-------------------------------------------------------------------

val_1    val_2     val_3    .....    val_N     =>     d1
val_1    val_2     val_3    .....    val_N     =>     d2
val_1    val_2     val_3    .....    val_N     =>     d1
val_1    val_2     val_3    .....    val_N     =>     d1
val_1    val_2     val_3    .....    val_N     =>     d3
....
....
```

  For convenience, we refer to each column of the table as representing a feature `f_i` whose value goes into your decision making process. Each row of the table represents a set of values for all the features and the corresponding decision.

- As to what specifically the features `f_i` shown on the previous slide would be, that would obviously depend on your application. [In a medical context, each feature `f_i` could represent a laboratory test on a patient, the value `val_i` the result of the test, and the decision `d_i` the diagnosis. In drug discovery, each feature `f_i` could represent the name of an ingredient in the drug, the value `val_i` the proportion of that ingredient, and the decision `d_i` the effectiveness of the drug in a drug trial. In a Wall Street sort of an application, each feature could represent a criterion (such as the price-to-earnings ratio) for making a buy/sell investment decision, and so on.]

- If the different rows of the training data, arranged in the form of a table shown on the previous slide, capture adequately the statistical variability of the feature values as they occur in the real world, you may be able to use a decision tree for automating the decision making process on any new data. [As to what I mean by "capturing adequately the statistical variability of feature values", see Section 12 of this tutorial.]

- Let's say that your new data record for which you need to make a decision looks like:

```
new_val_1    new_val_2    new_val_2    ....   new_val_N
```

  the decision tree will spit out the best possible decision to make for this new data record given the statistical distribution of the feature values for all the decisions in the training data supplied through the table on Slide 3. The "quality" of this decision would obviously depend on the quality of the training data, as explained in Section 12.

- This tutorial will demonstrate how the notion of entropy can be used to construct a decision tree in which the feature tests for making a decision on a new data record are organized optimally in the form of a tree of decision nodes.

- In the decision tree that is constructed from your training data, the feature test that is selected for the root node causes maximal disambiguation of the different possible decisions for a new data record. [In terms of information content as measured by entropy, the feature test at the root would cause maximum reduction in the decision entropy in going from all the training data taken together to the data as partitioned by the feature test.]

- One then drops from the root node a set of child nodes, one for each value of the feature tested at the root node for the case of symbolic features. For the case when a numeric feature is tested at the root node, one drops from the root node two child nodes, one for the case when the value of the feature tested at the root is less than the decision threshold chosen at the root and the other for the opposite case.

- Subsequently, at each child node, you pose the same question you posed at the root node when you selected the best feature to test at that node: Which feature test at the child node in question would maximally disambiguate the decisions for the training data associated with the child node in question?

- In the rest of this Introduction, let's see how a decision-tree based classifier can be used by a computer vision system to automatically figure out which features work the best in order to distinguish between a set of objects. We assume that the vision system has been supplied with a very large number of elementary features (we could refer to these as the vocabulary of a computer vision system) and how to extract them from images. But the vision system has NOT been told in advance as to which of these elementary features are relevant to the objects.

- Here is how we could create such a self-learning computer vision system:

  - We show a number of different objects to a sensor system consisting of cameras, 3D vision sensors (such as the Microsoft Kinect sensor), and so on. Let's say 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 and texture attributes and the presence or absence of shape primitives. [For depth data, the features could be different types of curvatures of the object surfaces and junctions formed by the joins between the surfaces, etc.]

  - The job given to the computer: From the data thus collected, it must figure out on its own how to best discriminate between the objects belonging to the different classes. [That is, 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.

- As mentioned in Section 19 of this tutorial, such a computer vision system was successfully constructed and tested in my laboratory at Purdue as a part of a Ph.D thesis.

2. Entropy

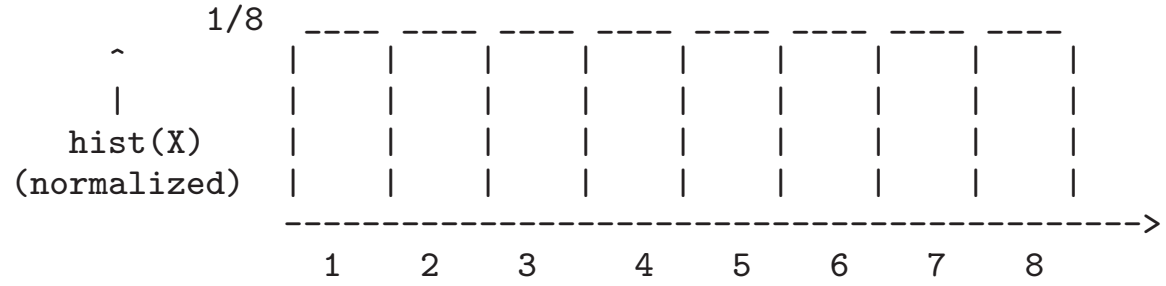
- Entropy is a powerful tool that can be used by a computer to determine on its own own as to what features to use and how to carve up the feature space for achieving the best possible discrimination between the classes. [You can think of each decision of a certain type in the last column of the table on Slide 3 as defining a class. If, in the context of computer vision, all the entries in the last column boil down to one of "apple," "orange," and "pear," then your training data has a total of three classes.]

- What is entropy?

- If a random variable X can take N different values, the i^{th} value x_i with probability $p(x_i)$, we can associate the following entropy with X:

$$H(X) \quad = \quad -\sum_{i=1}^{N} p(x_i) \log_2 p(x_i)$$

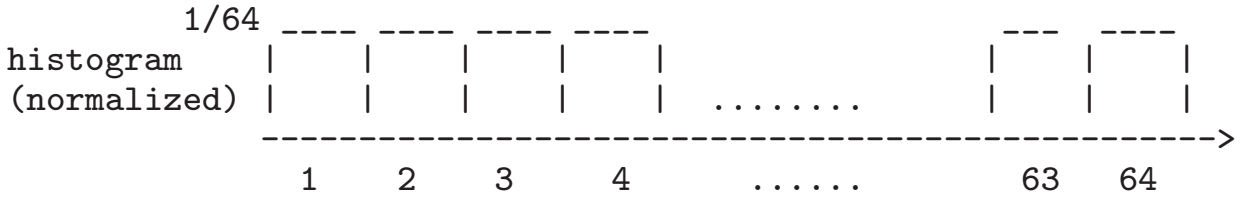
- To gain some insight into what H measures, consider the case when the normalized histogram of the values taken by the random variable X looks like



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

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

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

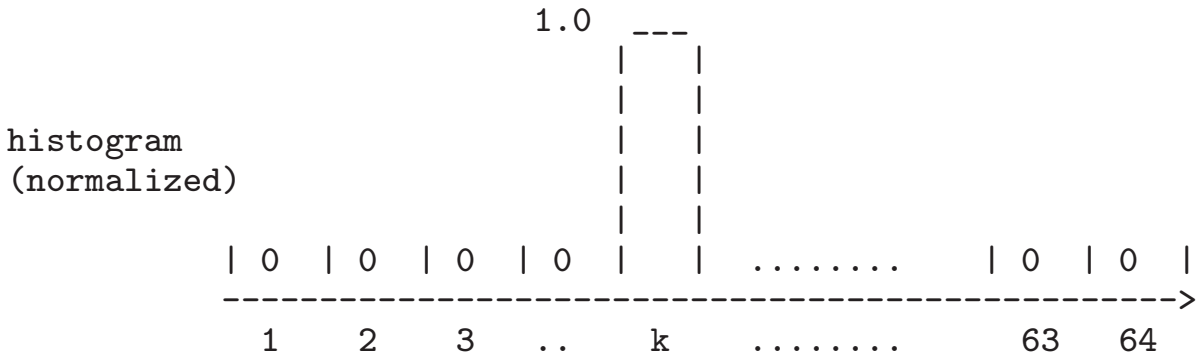


- In this case,

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

- So we see that the entropy, measured in bits because of 2 being the base of the logarithm, has increased because now we have greater uncertainty or "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 the "chaos": We will consider an X that is always known to take on a particular value:



- In this case, we obviously have

$$\begin{aligned} p(x_i) &= 1 \qquad x_i = k \\ &= 0 \qquad otherwise \end{aligned}$$

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

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

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

$$= \quad 0 \quad bits$$

where we use the fact that as $p \rightarrow 0_+$, $p \log p \rightarrow 0$ 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.**

3. Conditional Entropy

- Given two interdependent random variables X and Y, 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,

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

  The entropy contained in both variables when taken together is $H(X,Y)$. The above definition says that, if X and Y are interdependent, and if we know X, we can reduce our measure of chaos in Y by the chaos that is attributable to X. [For independent X and Y, one can easily show that $H(X,Y) = H(X) + H(Y)$.]

- But what do we mean by knowing X in the context of X and Y being interdependent? Before we answer this question, let's first look at the formula for the joint entropy $H(X,Y)$, which is given by

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

- When we say we know X, in general what we mean is that we know that the variable X has taken on a particular value. Let's say that X has taken on a specific value a. The entropy associated with Y would now be given by:

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

- The formula shown for $H(Y|X)$ on the previous page is the average of $H\Big(Y\Big|X=a\Big)$ over all possible instantiations a for X.

4. Average Entropies

- Given N independent random variables X_1, X_2, ... X_N, we can associate an average entropy with all N variables by

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

- For another kind of an average, the conditional entropy $H(Y|X)$ is also an average, in the sense that the right hand side shown below is an average with respect to all of the different ways the conditioning variable can be instantiated:

$$H\big(Y\big|X\big) \quad = \quad \sum_{a} H\big(Y\big|X=a\big) \;\times\; p\big(X=a\big)$$

  where $H\big(Y\big|X=a\big)$ is given by the formula at the bottom of the previous slide.

- To establish the claim made in the previous bullet, note that

$$
\begin{aligned}
H\Big(Y\Big|X\Big) &= \sum_a H\Big(Y\Big|X=a\Big) \;\times\; p(X=a) \\
&= -\sum_a \Big\{ \sum_j p\Big(y_j\Big|X=a\Big) \;\log_2 p\Big(y_j\Big|X=a\Big)\Big\}\; p(X=a) \\
&= -\sum_i \sum_j p\Big(y_j\Big|x_i\Big) \;\log_2 p\Big(y_j\Big|x_i\Big)\; p(x_i) \\
&= -\sum_i \sum_j \frac{p(x_i,y_j)}{p(x_i)} \;\log_2 \frac{p(x_i,y_j)}{p(x_i)}\; p(x_i) \\
&= -\sum_i \sum_j p(x_i,y_j)\Big[\log_2 p(x_i,y_j) - \log_2 p(x_i)\Big] \\
&= H(X,Y) \;+\; \sum_i \sum_j p(x_i,y_j) \log_2 p(x_i) \\
&= H(X,Y) \;+\; \sum_i p(x_i) \log_2 p(x_i) \\
&= H(X,Y) \;-\; H(X)
\end{aligned}
$$

The 3^{rd} expression is a rewrite of the 2^{nd} with a more compact notation. [In the 7^{th}, we note that we get a marginal probability when a joint probability is summed with respect to its free variable.]

5. 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 as conditioned on each specific feature f separately as follows:

$$H(C|f) \quad = \quad \sum_{a} H\Big(C\Big|v(f)=a\Big) \;\times\; p\Big(v(f)=a\Big)$$

  where the notation $v(f) = a$ means that the value of feature f is some specific value a. The computer selects that feature f for which $H(C|f)$ is the smallest value. [In the formula shown above, the averaging carried out over the values of the feature f is the same type of averaging as shown at the bottom of Slide 17.] **NOTATION:** Note that C is a random variable over the class labels. If your training data mentions the following three classes: "apple," "orange," and "pear," then C as a random variable takes one of these labels as its value.

- 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

$$H\Big(C\Big|v(f)=a\Big) \;\; = \;\; -\sum_{m} p\Big(C_m\Big|v(f)=a\Big) \times \log_2 p\Big(C_m\Big|v(f)=a\Big)$$

  where C_m is the name of the m^{th} class and the summation is over all the classes.

- But how do we figure out $p\Big(C_m\Big|v(f)=a\Big)$ that is needed on the right hand side?

- We will next present two different ways for calculating $p\Big(C_m\Big|v(f)=a\Big)$. 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 $p(C_m \big| v(f) = a)$ is count-based: Given M classes of objects that we show to a sensor system, we pick objects randomly from the population of all objects belonging to all classes. Say the sensor system is allowed to measure K different kinds of features: $f_1, f_2, ..., f_K$. For each feature f_k, the sensor system keeps a count of the number of objects that gave rise to the $v(f_k) = a$ value. Now we estimate $p(C_m \big| v(f) = a)$ for any choice of $f = f_k$ simply by counting off the number of objects from class C_m that exhibited the $v(f_k) = a$ measurement.

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

$$p\Big(C_m \Big| v(f) = a\Big) \;\; = \;\; \frac{p\Big(v(f) = a \Big| C_m\Big) \;\times\; p(C_m)}{p\Big(v(f) = a\Big)}$$

  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:

$$\sum_m p\Big(C_m \Big| v(f) = a\Big) \quad = \quad 1$$

- What's interesting is that if we do obtain $p\Big(v(f) = a\Big)$ 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 20. Otherwise, $p\Big(v(f) = a\Big)$ 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 20.

- It follows from the nature of entropy (See Slides 10 through 14) 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.

6. 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_j and that we are now at one of the child nodes hanging from the root. So the question now is how do we select the best feature to use at the child node.

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

  [**REMINDER:** Whereas $v(f_j)$ stands for the "value of feature f_j," the notation a_j stands for a specific value taken by that feature.]

- That is, for any feature f not previously used at the root, we find the conditional entropy (with respect to our choice for f) when we are on the $v(f_j) = a_j$ branch:

$$H\Big(C\Big|f, v(f_j) = a_j\Big) \;=\; \sum_b H\Big(C\Big|v(f)=b, v(f_j)=a_j\Big) \times p\Big(v(f)=b, v(f_j)=a_j\Big)$$

  Whichever feature f yields the smallest value for the entropy mentioned on the left hand side of the above equation will become the feature test of choice at the branch in question.

- Strictly speaking, the entropy formula shown above for the calculation of average entropy is not correct since it does not reflect the fact that the probabilistic averaging on the right hand side is only with respect to the values taken on by the feature f.

- In the equation on the previous slide, for the summation shown on the right to yield a true average with respect to different possible values for the feature f, the formula would need to be expressed as*

$$H\Big(C\Big|f, v(f_j)=a_j\Big) \;=$$

$$\sum_b H\Big(C\Big|v(f)=b, v(f_j)=a_j\Big) \;\times\; \frac{p\Big(v(f)=b, v(f_j)=a_j\Big)}{\sum_{a_j} p\Big(v(f)=b, v(f_j)=a_j\Big)}$$

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

$$H\Big(C\Big|v(f)=b, v(f_j)=a_j\Big) \;=$$

$$-\sum_m p\Big(C_m\Big|v(f)=b, v(f_j)=a_j\Big) \times \log_2 p\Big(C_m\Big|v(f)=b, v(f_j)=a_j\Big)$$

for any given feature $f \neq f_j$.

*In my lab at Purdue, we refer to such normalizations in the calculation of average entropy as "JZ Normalization"— after Padmini Jaikumar and Josh Zapf.

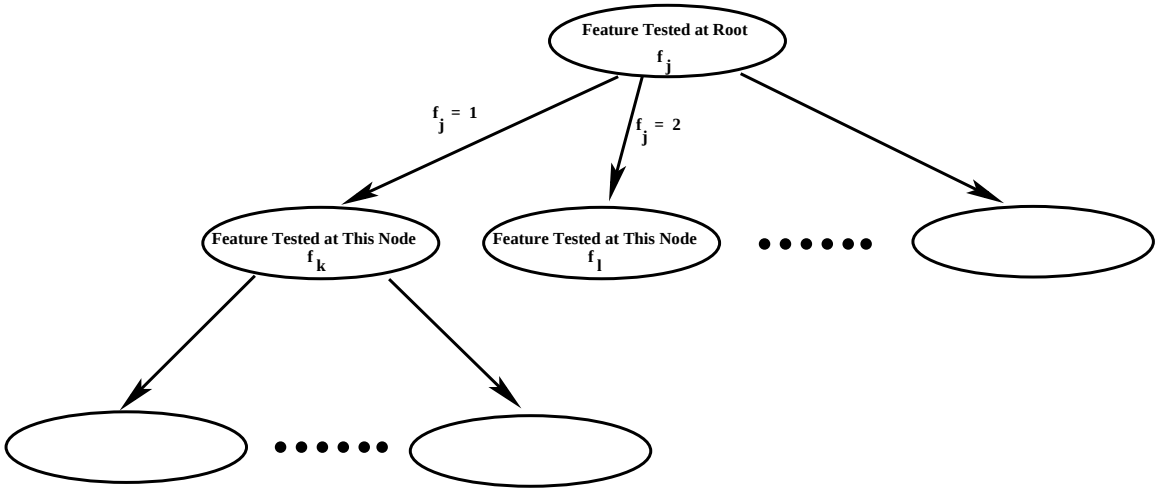
- The conditional probability needed in the previous formula is estimated using Bayes Theorem:

$$p\Big(C_m \Big| v(f)=b, v(f_j)=a_j\Big) \;\; =$$

$$= \;\; \frac{p\Big(v(f)=b, v(f_j)=a_j \Big| C_m\Big) \;\times\; p\Big(C_m\Big)}{p\Big(v(f)=b, v(f_j)=a_j\Big)}$$

$$= \;\; \frac{p\Big(v(f)=b \Big| C_m\Big) \times p\Big(v(f_j)=a_j \Big| C_m\Big) \times p\Big(C_m\Big)}{p\Big(v(f)=b\Big) \times p\Big(v(f_j)=a_j\Big)}$$

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_j.

- 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 above.

- 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 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 $H(C|f_j)$ that resulted in their creation.

- A child node of the root that is on the branch $v(f_j) = a_j$ gets its own feature test (and is split further) if and only if we can find a feature f_k such that $H\Big(C|f_k, v(f_j) = a_j\Big)$ is less than the entropy $H(C|f_j)$ inherited by the child from the root.

- If the condition $H\Big(C\Big|f, v(f_j)=v_j\Big) < H\Big(C\Big|f_j)\Big)$ cannot be satisfied at the child node on the branch $v(f_j) = a_j$ of the root for any feature $f \neq f_j$, 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 $v(f_j) = a_j,\ v(f_k) = b_k, \ldots$, we will associate the following class probability with the node:

$$p\Big(C_m \Big| v(f_j) = a_j,\ v(f_k) = b_k,\ \ \ldots\Big)$$

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

- The above probability may be estimated with Bayes Theorem:

$$p\Big(C_m \Big| v(f_j) = a_j,\ v(f_k) = b_k,\ \ \ldots\Big) \ \ = \ \ \frac{p\Big(v(f_j) = a_j,\ v(f_k) = b_k, \ldots \Big| C_m\Big) \times p\Big(C_m\Big)}{p\Big(v(f_j) = a_j,\ v(f_k) = b_k,\ \ \ldots\Big)}$$

- 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:

$$p\Big(v(f_j)=a_j,\ v(f_k)=b_k,\ \ldots\Big) \quad = \quad \prod_{f\ \ on\ \ branch} p\Big(v(f)=value\Big)$$

$$p\Big(v(f_j)=a_j,\ v(f_k)=b_k,\ \ldots\Big|C_m\Big) \quad = \quad \prod_{f\ \ on\ \ branch} p\Big(v(f)=value\Big|C_m\Big)$$

7. Incorporating Numeric Features

- A feature is numeric if it can take any floating-point value from a continuum of values. The sort of reasoning we have described so far for choosing the best feature at a node and constructing a decision tree cannot be applied directly to the case of numeric features.

- However, numeric features lend themselves to recursive partitioning that eventually results in the same sort of a decision tree you have seen so far.

- When we talked about symbolic features in Section 5, we calculated the class entropy with respect to a feature by constructing a probabilistic average of the class entropies with respect to knowing each value separately for the feature.

- Let's say f is a numeric feature. For a numeric feature, a better approach consists of calculating the class entropy vis-a-vis a decision threshold on the feature values:

$$H\Big(C\Big|v_{th}(f)=\theta\Big) \quad = \quad H\Big(C\Big|v(f)\leq\theta\Big) \;\times\; p\Big(v(f)\leq\theta\Big) \;+\; H\Big(C\Big|v(f)>\theta\Big) \;\times\; p\Big(v(f)>\theta\Big)$$

  where $v_{th}(f)=\theta$ means that we have set the decision threshold for the values of the feature f at θ for the purpose of partitioning the data into parts, one for which $v(f)\leq\theta$ and the other for which $v(f)>\theta$.

- The left side in the equation shown above is the average entropy for the two parts considered separately on the right hand side. The threshold for which this average entropy is the minimum is the best threshold to use for the numeric feature f.

- To illustrate the usefulness of minimizing this average entropy for discovering the best threshold, consider the case when we have only two classes, one for which all values of f are less than θ and the other for which all values of f are greater than θ. For this case, the left hand side above would be zero.

- The components entropies on the right hand side in the previous equation can be calculated by

$$H\Big(C\Big|v(f) \leq \theta\Big) \;\; = \;\; -\sum_{m} p\Big(C_m\Big|v(f) \leq \theta\Big) \times \log_2 p\Big(C_m\Big|v(f) \leq \theta\Big)$$

and

$$H\Big(C\Big|v(f) > \theta\Big) \;\; = \;\; -\sum_{m} p\Big(C_m\Big|v(f) > \theta\Big) \times \log_2 p\Big(C_m\Big|v(f) > \theta\Big)$$

- We can estimate $p\big(C_m\big|v(f)\leq\theta\big)$ and $p\big(C_m\big|v(f)>\theta\big)$ by using the Bayes' Theorem:

$$p\Big(C_m\Big|v(f)\leq\theta\Big) \;=\; \frac{p\Big(v(f)\leq\theta\Big|C_m\Big)\;\times\;p(C_m)}{p\Big(v(f)\leq\theta\Big)}$$

and

$$p\Big(C_m\Big|v(f)>\theta\Big) \;=\; \frac{p\Big(v(f)>\theta\Big|C_m\Big)\;\times\;p(C_m)}{p\Big(v(f)>\theta\Big)}$$

The various terms on the right sides in the two equations shown above can be estimated directly from the training data.

- However, in practice, you are better off using the normalization shown on the next page for estimating the denominator in the equations shown above.

- Although the denominator in the equations on the previous slide can be estimated directly from the training data, you are likely to achieve superior results if you calculate this denominator directly from (or, at least, adjust its calculated value with) the following normalization constraint on the probabilities on the left:

$$\sum_m p\Big(C_m \;\Big|\; v(f) \leq \theta\Big) \quad = \quad 1$$

and

$$\sum_m p\Big(C_m \;\Big|\; v(f) > \theta\Big) \quad = \quad 1$$

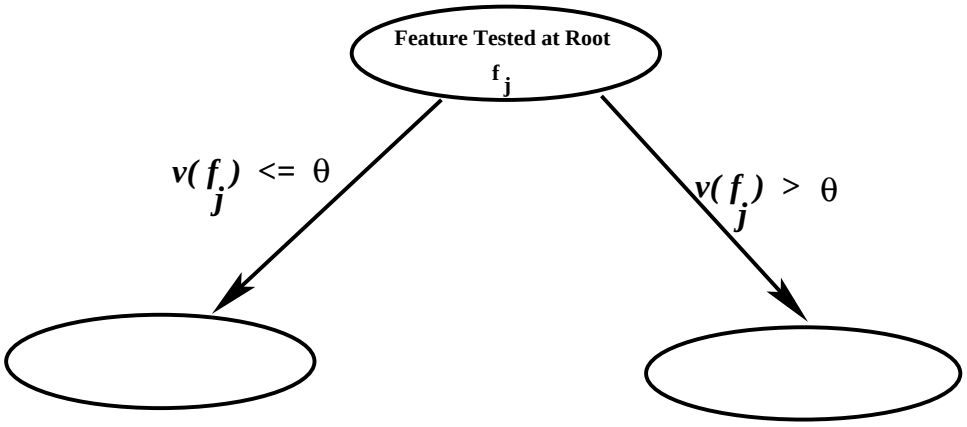
- Now we are all set to use this partitioning logic to choose the best feature for the root node of our decision tree. We proceed as explained on the next slide.

- Given a set of numeric features and a training data file, we seek that numeric feature for which the average entropy over the two parts created by the thresholding partition is the least.

- For each numeric feature, we scan through all possible partitioning points (these would obviously be the sampling points over the interval corresponding to the values taken by that feature), and we choose that partitioning point which minimizes the average entropy of the two parts. We consider this partitioning point as the best decision threshold to use vis-a-vis that feature.

- Given a set of numeric features, their associated best decision thresholds, and the corresponding average entropies over the partitions obtained, we select for our best feature that feature that has the least average entropy associated with it at its best decision threshold.

- After finding the best feature for the root node in the manner described above, we can drop two branches from it, one for the training samples for which $v(f) \leq \theta$ and the other for the samples for which $v(f) > \theta$ as shown in the figure below:



- The argument stated above can obviously be extended to a mixture of numeric and symbolic features as explained on the next slide.

- Given a mixture or symbolic and numeric features, we associate with each symbolic feature the best possible entropy calculated in the manner explained in Section 5. And, we associate with each numeric feature the best entropy that corresponds to the best threshold choice for that feature. Given all the features and their associated best class entropies, we choose that feature for the root node of our decision tree for which the class entropy is the minimum.

- Now that you know how to construct the root node for the case when you have just numeric features or a mixture of numeric and symbolic features, the next question is how to branch out from the root node.

- If the best feature selected for the root node is symbolic, we proceed in the same way as described in Section 5 in order to grow the tree to the next level.

- On the other hand, if the best feature f selected at the root node is numeric and the best decision threshold for the feature is θ, we must obviously construct two child nodes at the root, one for which $v(f) \leq \theta$ and the other for which $v(f) > \theta$.

- To extend the tree further, we now select the best features to use at the child nodes of the root. Let's assume for a moment that the best feature chosen for a child node also turns out to be numeric.

- Let's say we used the numeric feature f_j, along with its decision threshold θ_j, at the root and that the choice of the best feature to use at the left child turns out to be f_k and that its best decision threshold is θ_k.

- The choice (f_k, θ_k) at the left child of the root must be the best possible among all possible features and all possible thresholds for those features so that following average entropy is minimized:

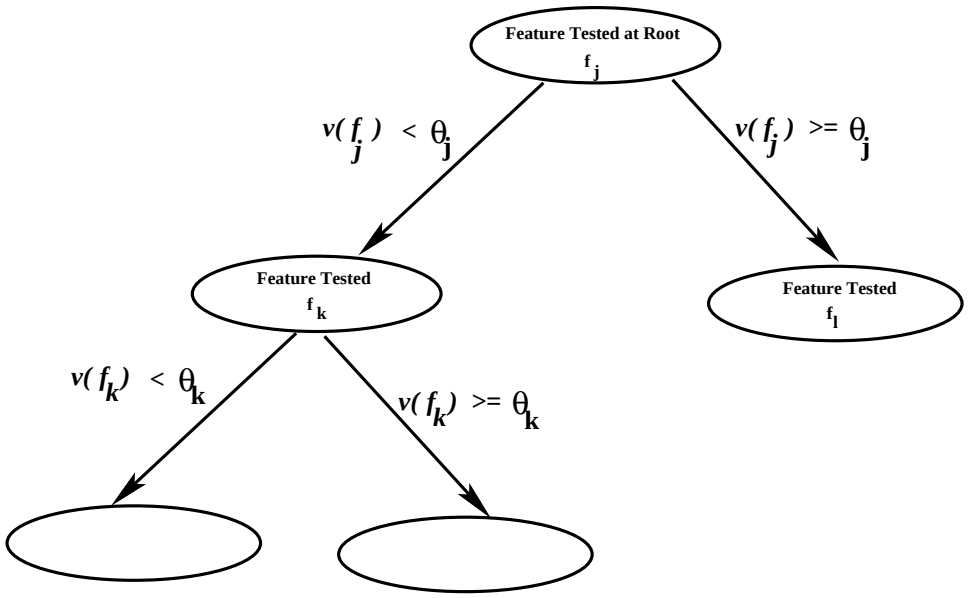
$$H\Big(C \;\Big|\; v(f_j) \leq \theta_j,\; v(f_k) \leq \theta_k\Big) \times p\Big(v(f_j) \leq \theta_j,\; v(f_k) \leq \theta_k\Big)$$

$$+$$

$$H\Big(C \;\Big|\; v(f_j) \leq \theta_j,\; v(f_k) > \theta_k\Big) \times p\Big(v(f_j) \leq \theta_k,\; v(f_k) > \theta_k\Big)$$

- For the purpose of our explanations, we assume that the left child at a node with a numeric test feature always corresponds to the "less than or equal to the threshold" case and the right child to the "greater than the threshold" case.

- 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 previous case when we had purely symbolic features and when we also have numeric features. 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 decision trees for the purely symbolic case and the decision trees needed when you must also deal with numeric features is more illusory than real. That is because when considering the root node feature f_l at the third-level nodes in the tree, the values of f_l will be limited to the interval $[v_{min}(f_l), \theta_l)$ in the left children of the root and to the interval $[\theta_l, v_{max}(f_l))$ in the right children of the root. Testing for whether the value of the feature f_l is in, say, the interval $[v(f_{min}(f_l), \theta_l)$ is not the same feature test as testing for whether this value is in the interval $[v_{min}(f_l), v_{max}(f_l))$.

- Once we arrive at a child node, we carry out at the child node the same reasoning that we carried out at the root for the selection of the best feature at the child node and to then grow the tree accordingly.

8. The Perl Module

Algorithm::DecisionTree-3.42

NOTE: *Versions 2.0 and higher of this module can handle simultaneously the numeric and the symbolic features. Even for the purely symbolic case, you are likely to get superior results with the latest version of the module than with the older versions.*

- 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-3.42/lib/Algorithm/DecisionTree.pm`

  Please read the documentation at the CPAN site for the API of this software package. [The URL shown above is supposed to be one continuous string. You can also just do a Google search on "Algorithm::DecisionTree" and go to Version 3.42 when you get to the CPAN page for the module.]

- To use the Perl module, you first need to construct an instance of the `Algorithm::DecisionTree` class as shown below:

```
my $training_datafile = "stage3cancer.csv";

my $dt = Algorithm::DecisionTree->new(
                training_datafile => $training_datafile,
                csv_class_column_index => 2,
                csv_columns_for_features => [3,4,5,6,7,8],
                entropy_threshold => 0.01,
                max_depth_desired => 8,
                symbolic_to_numeric_cardinality_threshold => 10,
                csv_cleanup_needed => 1,
        );
```

- The constructor option `csv_class_column_index` informs the module as to which column of your CSV file contains the class labels for the data records. THE COLUMN INDEXING IS ZERO BASED. The constructor option `csv_columns_for_features` specifies which columns are to be used for feature values. The first row of the CSV file must specify the names of the features. See examples of CSV files in the `Examples` subdirectory of the module.

- The option `symbolic_to_numeric_cardinality_threshold` in the constructor is also important. For the example shown above, if an ostensibly numeric feature takes on only 10 or fewer different values in your training data file, it will be treated like a symbolic features. The option `entropy_threshold` determines the granularity with which the entropies are sampled for the purpose of calculating entropy gain with a particular choice of decision threshold for a numeric feature or a feature value for a symbolic feature.

- The option `csv_cleanup_needed` is important for extracting data from "messy" CSV files. That is, CSV files that use double-quoted strings for either the field names or the field values and that allow for commas to be used inside the double-quoted strings.

- After you have constructed an instance of the DecisionTree module, you read in the training data file and initialize the probability cache by calling:

```
$dt->get_training_data();
$dt->calculate_first_order_probabilities();
$dt->calculate_class_priors();
```

- Now you are ready to construct a decision tree for your training data by calling:

```
$root_node = $dt->construct_decision_tree_classifier();
```

  where `$root_node` is an instance of the `DTNode` class that is also defined in the module file.

- With that, you are ready to start classifying new data samples — as I show on the next slide.

- Let's say that your data record looks like:

```
my @test_sample  = qw /  g2=4.2
                         grade=2.3
                         gleason=4
                         eet=1.7
                         age=55.0
                         ploidy=diploid /;
```

  you can classify it by calling:

```
my $classification = $dt->classify($root_node, \@test_sample);
```

- The call to `classify()` returns a reference to a hash whose keys are the class names and the values the associated classification probabilities. This hash also includes another key-value pair for the solution path from the root node to the leaf node at which the final classification was carried out.

- The module also allows you to generate your own training datasets for experimenting with decision trees classifiers. For that, the module file contains the following classes:
  (1) `TrainingDataGeneratorNumeric`, and
  (2) `TrainingDataGeneratorSymbolic`

- The class `TrainingDataGeneratorNumeric` outputs a CSV training data file for experimenting with numeric features.

- The numeric values are generated using a multivariate Gaussian distribution whose mean and covariance are specified in a parameter file. See the file `param_numeric.txt` in the `Examples` directory for an example of such a parameter file. Note that the dimensionality of the data is inferred from the information you place in the parameter file.

- The class `TrainingDataGeneratorSymbolic` generates synthetic training data for the purely symbolic case. It also places its output in a '.csv' file. The relative frequencies of the different possible values for the features is controlled by the biasing information you place in a parameter file. See `param_symbolic.txt` for an example of such a file.

- See the web page at

  `http://search.cpan.org/~avikak/Algorithm-DecisionTree-3.42/`

  for a full description of the API of this Perl module.

- Additionally, for large test datasets, see Section 10 of this Tutorial for the demonstration scripts in the Perl module that show you how you can classify all your data records in a CSV file in one go.

9. The Python Module

DecisionTree-3.4.3

NOTE: *Versions prior to 2.0 could only handle symbolic training data. Versions 2.0 and higher can handle both symbolic and numeric training data.*

- Version 2.0 was a major re-write of the module for incorporating numeric features.

- Version 2.1 was a cleaned up version of v. 2.0. Version 2.2 introduced the functionality to evaluate the quality of training data. The latest version is 3.4.3. To download Version 3.4.3:

  http://pypi.python.org/pypi/DecisionTree/3.4.3

  The API of this software package is at the HTML link near the top of the web page at the above URL.

- The module makes the following two assumptions about the training data in a '.csv' file: that the first column (meaning the column with index 0) contains a unique integer identifier for each data record, and that the first row contains the names to be used for the features.

- Shown below is a typical call to the constructor of the module:

```
training_datafile = "stage3cancer.csv"

dt = DecisionTree.DecisionTree(
                 training_datafile = training_datafile,
                 csv_class_column_index = 2,
                 csv_columns_for_features = [3,4,5,6,7,8],
                 entropy_threshold = 0.01,
                 max_depth_desired = 3,
                 symbolic_to_numeric_cardinality_threshold = 10,
                 csv_cleanup_needed = 1,
     )
```

  In this call to the `DecisionTree` constructor, the option `csv_class_column_index` is used to tell the module that the class label is in the column indexed 2 (meaning the third column) of the '.csv' training data file.

- The constructor option `csv_columns_for_features` is used to tell the module that the columns indexed 3 through 8 are to be used as features.

- To explain the role of the constructor option `symbolic_to_numeric_cardinality_threshold` in the call shown on the previous slide, note that the module can treat those numeric looking features symbolically if the different numerical values taken by the feature are small in number. In the call shown, if a numeric feature takes 10 or fewer unique values, it will be treated like a symbolic feature.

- If the module can treat certain numeric features symbolically, you might ask as to what happens if the value for such a feature in a test sample is not exactly the same as one of the values in the training data.

- When a numeric feature is treated symbolically, a value in a test sample is "snapped" to the closest value in the training data.

- No matter whether you construct a decision tree from purely symbolic data, or purely numeric data, or a mixture of the two, the two constructor parameters that determine the number of nodes in the decision are `entropy_threshold` and `max_depth_desired`. More on these on the next slide.

- As for the role of `entropy_threshold`, recall that a child node is created only if the difference between the entropy at the current node and the child node exceeds a threshold. This parameter sets that threshold.

- Regarding the parameter `max_depth_desired`, note that the tree is grown in a depth-first manner to the maximum depth set by this parameter.

- The option `csv_cleanup_needed` is important for extracting data from "messy" CSV files. That is, CSV files that use double-quoted strings for either the field names or the field values and that allow for commas to be used inside the double-quoted strings.

- After the call to the constructor, the following three methods **must** be called to initialize the probabilities:

```
dt.get_training_data()
dt.calculate_first_order_probabilities_for_numeric_features()
dt.calculate_class_priors()
```

- The tree itself is constructed and, if so desired, displayed by the following calls:

```
root_node = dt.construct_decision_tree_classifier()
root_node.display_decision_tree("      ")
```

  where the "white-space" string supplied as the argument to the display method is used to offset the display of the child nodes in relation to the display of the parent nodes.

- After you have constructed a decision tree, it is time classify a test sample.

- Here is an example of the syntax used for a test sample and the call you need to make to classify it:

```
test_sample  = ['g2 = 4.2',
                'grade = 2.3',
                'gleason = 4',
                'eet = 1.7',
                'age = 55.0',
                'ploidy = diploid']

classification = dt.classify(root_node, test_sample)
```

- The `classification` returned by the call to `classify()` as shown on the previous slide is a dictionary whose keys are the class names and whose values are the classification probabilities associated with the classes.

- For further information, see the various example scripts in the Examples subdirectory of the module.

- The module also allows you to generate your own synthetic symbolic and numeric data files for experimenting with decision trees.

- For large test datasets, see the next section for the demonstration scripts that show you how you can classify all your data records in a CSV file in one go.

- See the web page at

  `https://engineering.purdue.edu/kak/distDT/DecisionTree-3.4.3.html`

  for a full description of the API of this Python module.

10. Bulk Classification of All Test Data Records in a CSV File

- For large test datasets, you would obviously want to process an entire file of test data records in one go.

- The `Examples` directory of both the Perl and the Python versions of the module include demonstration scripts that show you how you can classify all your data records in one fell swoop if the records are in a CSV file.

- For the case of Perl, see the following scripts in the `Examples` directory of the module for bulk classification of data records:

  `classify_test_data_in_a_file.pl`

- And for the case of Python, check out the following script in the `Examples` directory for doing the same things:

  `classify_test_data_in_a_file.py`

- All scripts mentioned on this section require three command-line arguments, the first argument names the training datafile, the second the test datafile, and the third the file in which the classification results will be deposited.

- The other examples directories, `ExamplesBagging`, `ExamplesBoosting`, and `ExamplesRandomizedTrees`, also contain scripts that illustrate how to carry out bulk classification of data records when you wish to take advantage of bagging, boosting, or tree randomization. In their respective directories, these scripts are named:

```
bagging_for_bulk_classification.pl
boosting_for_bulk_classification.pl
classify_database_records.pl

bagging_for_bulk_classification.py
boosting_for_bulk_classification.py
classify_database_records.py
```

11. Dealing with Large Dynamic-Range and Heavy-tailed Features

- For the purpose of estimating the probabilities, it is necessary to sample the range of values taken on by a numerical feature. For features with "nice" statistical properties, this sampling interval is set to the median of the differences between the successive feature values in the training data. (Obviously, as you would expect, you first sort all the values for a feature before computing the successive differences.) This logic will not work for the sort of a feature described below.

- Consider a feature whose values are heavy-tailed, and, at the same time, the values span a million to one range.

- What I mean by heavy-tailed is that rare values can occur with significant probabilities. It could happen that most of the values for such a feature are clustered at one of the two ends of the range. At the same time, there may exist a significant number of values near the end of the range that is less populated.

- Typically, features related to human economic activities — such as wealth, incomes, etc. — are of this type.

- With the median-based method of setting the sampling interval as described on the previous slide, you could end up with a sampling interval that is much too small. That could potentially result in millions of sampling points for the feature if you are not careful.

- Beginning with Version 2.22 of the Perl module and Version 2.2.4 of the Python module, you have two options for dealing with such features. You can choose to go with the default behavior of the module, which is to sample the value range for such a feature over a maximum of 500 points.

- Or, you can supply an additional option to the constructor that sets a user-defined value for the number of points to use. The name of the option is `number_of_histogram_bins`. The following script

  ```
  construct_dt_for_heavytailed.pl
  ```

  in the "examples" directory shows an example of how to call the constructor of the module with the `number_of_histogram_bins` option.

12. Testing the Quality of the Training Data

- Even if you have a great algorithm for constructing a decision tree, its ability to correctly classify a new data sample would depend ultimately on the quality of the training data.

- Here are the four most important reasons for why a given training data file may be of poor quality: (1) Insufficient number of data samples to adequately capture the statistical distributions of the feature values as they occur in the real world; (2) The distributions of the feature values in the training file not reflecting the distribution as it occurs in the real world; (3) The number of the training samples for the different classes not being in proportion to the real-world prior probabilities of the classes; and (4) The features not being statistically independent.

- A quick way to evaluate the quality of your training data is to run an N-fold cross-validation test on the data. This test divides all of the training data into N parts, with $N-1$ parts used for training a decision tree and one part used for testing the ability of the tree to classify correctly. This selection of $N-1$ parts for training and one part for testing is carried out in all of the N different possible ways. Typically, $N = 10$.

- You can run a 10-fold cross-validation test on your training data with version 2.2 **or higher** of the Python Decision Tree module and version 2.1 **or higher** of the Perl version of the same.

- The next slide presents a word of caution in using the output of a cross-validation to either trust or not trust your training data file.

- **Strictly speaking, a cross-validation test is statistically meaningful only if the training data does NOT suffer from any of the four shortcomings I mentioned at the beginning of this section.** [The real purpose of a cross-validation test is to estimate the Bayes classification error — meaning the classification error that can be attributed to the overlap between the class probability distributions in the feature space.]

- Therefore, one must bear in mind the following when interpreting the results of a cross-validation test: If the cross-validation test says that your training data is of poor quality, then there is no point in using a decision tree constructed with this data for classifying future data samples. On the other hand, if the test says that your data is of good quality, your tree may still be a poor classifier of the future data samples on account of the four data shortcomings mentioned at the beginning of this section.

- Both the Perl and the Python Decision-Tree modules contain a special subclass `EvalTrainingData` that is derived from the main `DecisionTree` class. The purpose of this subclass is to run a 10-fold cross-validation test on the training data file you specify.

- The code fragment shown below illustrates how you invoke the testing function of the `EvalTrainingData` class in the Python version of the module:

```
training_datafile = "training3.csv"
eval_data = DecisionTree.EvalTrainingData(
                    training_datafile = training_datafile,
                    csv_class_column_index = 1,
                    csv_columns_for_features = [2,3],
                    entropy_threshold = 0.01,
                    max_depth_desired = 3,
                    symbolic_to_numeric_cardinality_threshold = 10,
                    csv_cleanup_needed = 1,
              )
eval_data.get_training_data()
eval_data.evaluate_training_data()
```

  In this case, we obviously want to evaluate the quality of the training data in the file `training3.csv`.

- The last statement in the code shown on the previous slide prints out a Confusion Matrix and the value of Training Data Quality Index on a scale of 0 to 100, with 100 designating perfect training data. The Confusion Matrix shows how the different classes were mis-identified in the 10-fold cross-validation test.

- The syntax for invoking the data testing functionality in Perl is the same:

```
my $training_datafile = "training3.csv";

my $eval_data = EvalTrainingData->new(
                  training_datafile => $training_datafile,
                  csv_class_column_index => 1,
                  csv_columns_for_features => [2,3],
                  entropy_threshold => 0.01,
                  max_depth_desired => 3,
                  symbolic_to_numeric_cardinality_threshold => 10,
                  csv_cleanup_needed => 1,
                );
$eval_data->get_training_data();
$eval_data->evaluate_training_data()
```

- This testing functionality can also be used to find the best values one should use for the constructor parameters `entropy_threshold`, `max_depth_desired`, and `symbolic_to_numeric_cardinality_threshold`.

- The following two scripts in the `Examples` directory of the Python version of the module:

```
evaluate_training_data1.py
evaluate_training_data2.py
```

  and the following two in the `Examples` directory of the Perl version

```
evaluate_training_data1.pl
evaluate_training_data2.pl
```

  illustrate the use of the EvalTrainingData class for testing the quality of your data.

13. Decision Tree Introspection

- Starting with Version 2.3.1 of the Python module and with Version 2.30 of the Perl module, you can ask the `DTIntrospection` class of the modules to explain the classification decisions made at the different nodes of the decision tree.

- Perhaps the most important bit of information you are likely to seek through DT introspection is the list of the training samples that fall directly in the portion of the feature space that is assigned to a node.

- However, note that, when training samples are non-uniformly distributed in the underlying feature space, it is possible for a node

to exist even when there are no training samples in the portion of the feature space assigned to the node. [That is because the decision tree is constructed from the probability densities estimated from the training data. When the training samples are non-uniformly distributed, it is entirely possible for the estimated probability densities to be non-zero in a small region around a point even when there are no training samples specifically in that region. (After you have created a statistical model for, say, the height distribution of people in a community, the model may return a non-zero probability for the height values in a small interval even if the community does not include a single individual whose height falls in that interval.)]

- That a decision-tree node can exist even where there are no training samples in that portion of the feature space that belongs to the node is an important indicator of the **generalization abilities** of a decision-tree-based classifier.

- In light of the explanation provided above, before the `DTIntrospection` class supplies any answers at all, it asks you to accept the fact that features can take on non-zero probabilities at a point in the feature space even though there are zero training samples at that point (or in a small region around that point). If you do not accept this rudimentary fact, the introspection class will not yield any answers (since you are not going to believe the answers anyway).

- The point made above implies that the path leading to a node in the decision tree may test a feature for a certain value or threshold despite the fact that the portion of the feature space assigned to that node is devoid of any training data.

- See the following three scripts in the `Examples` directory of Version 2.3.2 **or higher** of the Python module for how to carry out DT introspection:

```
introspection_in_a_loop_interactive.py
introspection_show_training_samples_at_all_nodes_direct_influence.py
introspection_show_training_samples_to_nodes_influence_propagation.py
```

  and the following three scripts in the `Examples` directory of Version 2.31 **or higher** of the Perl module

```
introspection_in_a_loop_interactive.pl
introspection_show_training_samples_at_all_nodes_direct_influence.pl
introspection_show_training_samples_to_nodes_influence_propagation.pl
```

- In both cases, the first script places you in an interactive session in which you will first be asked for the node number you are interested in.

- Subsequently, you will be asked for whether or not you are interested in specific questions that the introspection can provide answers for.

- The second of the three scripts listed on the previous slide descends down the decision tree and shows for each node the training samples that fall directly in the portion of the feature space assigned to that node.

- The last of the three script listed on the previous slide shows for each training sample how it affects the decision-tree nodes either directly or indirectly through the generalization achieved by the probabilistic modeling of the data.

- The output of the script `introspection_show_training_samples_at_all_nodes_direct_influence` looks like:

```
Node 0: the samples are: None
Node 1: the samples are: ['sample_46', 'sample_58']
Node 2: the samples are: ['sample_1', 'sample_4', 'sample_7', ...
Node 3: the samples are: []
Node 4: the samples are: []
...
...
```

- The nodes for which no samples are listed come into existence through the generalization achieved by the probabilistic modeling of the data.

- The output produced by the script `introspection_show_training_samples_to_nodes_influence_propagation` looks like what is shown on the next slide.

```
sample_1:
   nodes affected directly: [2, 5, 19, 23]
   nodes affected through probabilistic generalization:
        2=> [3, 4, 25]
            25=> [26]
        5=> [6]
            6=> [7, 13]
                7=> [8, 11]
                    8=> [9, 10]
                    11=> [12]
                13=> [14, 18]
                    14=> [15, 16]
                        16=> [17]
        19=> [20]
            20=> [21, 22]
        23=> [24]

sample_4:
   nodes affected directly: [2, 5, 6, 7, 11]
   nodes affected through probabilistic generalization:
        2=> [3, 4, 25]
            25=> [26]
        5=> [19]
            19=> [20, 23]
                20=> [21, 22]
                23=> [24]
        6=> [13]
            13=> [14, 18]
                14=> [15, 16]
                    16=> [17]
        7=> [8]
            8=> [9, 10]
        11=> [12]

...
...
...
```

- For each training sample, the display on the previous slide first presents the list of nodes that are directly affected by the sample. A node is affected directly by a sample if the latter falls in the portion of the feature space that belongs to the former. Subsequently, for each training sample, the display shows a subtree of the nodes that are affected indirectly by the sample through the generalization achieved by the probabilistic modeling of the data. In general, a node is affected indirectly by a sample if it is a descendant of another node that is affected directly.

- In the on-line documentation associated with the Perl and the Python modules, the section titled "The Introspection API" lists the methods you can invoke in your own code for carrying out DT introspection.

14. Incorporating Bagging

- Starting with Version 3.0 of the Python `DecisionTree` module and Version 3.0 of the Perl version of the same you can now carry out decision-tree based classification with bagging.

- Bagging means randomly extracting smaller datasets (we refer to them as bags of data) from the main training dataset and constructing a separate decision tree for each bag. Subsequently, given a test sample, you can classify it with each decision tree and base your final classification on, say, the majority vote from all the decision trees.

- (1) If your original training dataset is sufficiently large; (2) you have done a good job of catching in it all of the significant statistical variations for the different classes, and (3) assuming that no single feature is too dominant with regard to inter-class discriminations, bagging has the potential to reduce classification noise and bias.

- In both the Python and the Perl versions of the DecisionTree module, bagging is implemented through the `DecisionTreeWithBagging` class.

- When you construct an instance of this class, you specify the number of bags through the constructor parameter `how_many_bags` and the extent of overlap in the data in the bags through the parameter `bag_overlap_fraction`, as shown on the next slide.

- Here is an example of how you'd call `DecisionTreeWithBagging` class's constructor in Python:

```
import DecisionTreeWithBagging
dtbag = DecisionTreeWithBagging.DecisionTreeWithBagging(
            training_datafile = training_datafile,
            csv_class_column_index = 2,
            csv_columns_for_features = [3,4,5,6,7,8],
            entropy_threshold = 0.01,
            max_depth_desired = 8,
            symbolic_to_numeric_cardinality_threshold=10,
            how_many_bags = 4,
            bag_overlap_fraction = 0.20,
            csv_cleanup_needed = 1,
        )
```

- And here is how you would do it in Perl:

```
use Algorithm::DecisionTreeWithBagging;
my $training_datafile = "stage3cancer.csv";
my $dtbag = Algorithm::DecisionTreeWithBagging->new(
              training_datafile => $training_datafile,
              csv_class_column_index => 2,
              csv_columns_for_features => [3,4,5,6,7,8],
              entropy_threshold => 0.01,
              max_depth_desired => 8,
              symbolic_to_numeric_cardinality_threshold=>10,
              how_many_bags => 4,
              bag_overlap_fraction => 0.2,
              csv_cleanup_needed => 1,
            );
```

- As mentioned previously, the constructor parameters how_many_bags and bag_overlap_fraction determine how bagging is carried vis-a-vis your training dataset.

- As implied by the name of the parameter, the number of bags is set by how_many_bags. Initially, the entire training dataset is randomized and divided into how_many_bags non-overlapping partitions. Subsequently, we expand each such partition by a fraction equal to bag_overlap_fraction by drawing samples randomly from the other bags. For example, if how_many_bags is set to 4 and bag_overlap_fraction set to 0.2, we first divide the training dataset (after it is randomized) into 4 non-overlapping partitions and then add additional samples drawn from the other partitions to each partition.

- To illustrate, let's say that the initial non-overlapping partitioning of the training data yields 100 training samples in each bag. With `bag_overlap_fraction` set to 0.2, we next add to each bag 20 additional training samples that are drawn randomly from the other three bags.

- After you have constructed an instance of the `DecisionTreeWithBagging` class, you can call the following methods of this class for the bagging based decision-tree classification:

  get_training_data_for_bagging(): This method reads your training datafile, randomizes it, and then partitions it into the specified number of bags. Subsequently, if the constructor parameter `bag_overlap_fraction` is some positive fraction, it adds to each bag a number of additional samples drawn at random from the other bags. As to how many additional samples are added to each bag, suppose the parameter `bag_overlap_fraction` is set to 0.2, the size of each bag will grow by 20% with the samples drawn from the other bags.

show_training_data_in_bags(): Shows for each bag the name-tags of the training data samples in that bag.

calculate_first_order_probabilities(): Calls on the appropriate methods of the main DecisionTree class to estimate the first-order probabilities from the samples in each bag.

calculate_class_priors(): Calls on the appropriate method of the main DecisionTree class to estimate the class priors for the data classes found in each bag.

construct_decision_trees_for_bags(): Calls on the appropriate method of the main DecisionTree class to construct a decision tree from the training data in each bag.

display_decision_trees_for_bags(): Display separately the decision tree for each bag.

classify_with_bagging(test_sample): Calls on the appropriate methods of the main DecisionTree class to classify the argument test sample.

display_classification_results_for_each_bag(): Displays separately the classification decision made by each the decision tree constructed for each bag.

get_majority_vote_classification(): Using majority voting, this method aggregates the classification decisions made by the individual decision trees into a single decision.

See the example scripts in the directory ExamplesBagging for how to call these methods for classifying individual samples and for bulk classification when you place all your test samples in a single file.

- The `ExamplesBagging` subdirectory in the main installation directory of the modules contains the following scripts that illustrate how you can incorporate bagging in your decision tree based classification:

```
bagging_for_classifying_one_test_sample.py

bagging_for_bulk_classification.py
```

The same subdirectory in the Perl version of the module contains the following scripts:

```
bagging_for_classifying_one_test_sample.pl

bagging_for_bulk_classification.pl
```

- As the name of the script implies, the first Perl or Python script named on the previous slide shows how to call the different methods of the `DecisionTreeWithBagging` class for classifying a single test sample.

- When you are classifying a single test sample, as in the first of the two scripts named on the previous slide, you can also see how each bag is classifying the test sample. You can, for example, display the training data used in each bag, the decision tree constructed for each bag, etc.

- The second script named on the previous slide is for the case when you place all of the test samples in a single file. The demonstration script displays for each test sample a single aggregate classification decision that is obtained through majority voting by all the decision trees.

15. Incorporating Boosting

- Starting with Version 3.2.0 of the Python `DecisionTree` module and Version 3.20 of the Perl version of the same, you can now use boosting for decision-tree based classification.

- In both cases, the module includes a new class called `BoostedDecisionTree` that makes it easy to incorporate boosting in a decision-tree classifier. [**NOTE:** Boosting does not always result in superior classification performance. Ordinarily, the theoretical guarantees provided by boosting apply only to the case of binary classification. Additionally, your training dataset must capture all of the significant statistical variations in the classes represented therein.]

- If you are not familiar with boosting, you may want to first browse through my tutorial "AdaBoost for Learning Binary and Multiclass Discriminations" that is available at:

  `https://engineering.purdue.edu/kak/Tutorials/AdaBoost.pdf`

  Boosting for designing classifiers owes its origins to the now celebrated paper "A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting" by Yoav Freund and Robert Schapire that appeared in 1995 in the Proceedings of the 2nd European Conf. on Computational Learning Theory.

- A boosted decision-tree classifier consists of a cascade of decision trees in which each decision tree is constructed with samples that are mostly those that are misclassified by the previous decision tree.

- You specify a probability distribution over the training dataset for selecting samples for training each decision tree in the cascade. At the beginning, the distribution is uniform over all of the samples.

- Subsequently, this probability distribution changes according to the misclassifications by each tree in the cascade: if a sample is misclassified by a given tree in the cascade, the probability of its being selected for training the next tree increases significantly.

- You also associate a trust factor with each decision tree depending on its power to classify correctly all of the training data samples.

- After a cascade of decision trees is constructed in this manner, you construct a final classifier that calculates the class label for a test data sample by taking into account the classification decisions made by each individual tree in the cascade, the decisions being weighted by the trust factors associated with the individual classifiers.

- Here is an example of how you'd call the constructor of the `BoostedDecisionTree` class in Python:

```
import BoostedDecisionTree
training_datafile = "training6.csv"

boosted = BoostedDecisionTree.BoostedDecisionTree(
              training_datafile = training_datafile,
              csv_class_column_index = 1,
              csv_columns_for_features = [2,3],
              entropy_threshold = 0.01,
              max_depth_desired = 8,
              symbolic_to_numeric_cardinality_threshold =
              how_many_stages = 10,
              csv_cleanup_needed = 1,
)
```

- And here is an example of how you'd call the constructor of the `BoostedDecisionTree` class in Perl:

```
use Algorithm::BoostedDecisionTree;
my $training_datafile = "training6.csv";
my $boosted = Algorithm::BoostedDecisionTree->new(
              training_datafile => $training_datafile,
              csv_class_column_index => 1,
              csv_columns_for_features => [2,3],
              entropy_threshold => 0.01,
              max_depth_desired => 8,
              symbolic_to_numeric_cardinality_threshold=>10,
              how_many_stages => 4,
              csv_cleanup_needed => 1,
);
```

- In both constructor calls shown above, note the parameter `how_many_stages`. This parameter controls how many stages will be used in the boosted decision tree classifier. As mentioned earlier, a separate decision tree is constructed for each stage of boosting using a set of training samples drawn randomly through a probability distribution maintained over the entire training dataset.

- After you have constructed an instance of the `BoostedDecisionTree` class, you can call the following methods of this class for constructing the full cascade of decision trees and for boosted decision-tree classification of your test data:

- **get_training_data_for_base_tree():** In this method name, the string base tree refers to the first tree of the cascade. This is the tree for which the training samples are drawn assuming a uniform distribution over the entire dataset. This method reads your training datafile, creates the data structures from the data ingested for constructing the base decision tree.

  show_training_data_for_base_tree(): Shows the training data samples and some relevant properties of the features used in the training dataset.

  calculate_first_order_probabilities_and_class_priors(): This method calls on the appropriate methods of the main `DecisionTree` class to estimate the first-order probabilities and the class priors.

construct_base_decision_tree(): This method calls on the appropriate method of the main `DecisionTree` class to construct the base decision tree.

display_base_decision_tree(): As you would guess, this method displays the base decision tree.

construct_cascade_of_trees(): Uses the AdaBoost algorithm (described in the AdaBoost tutorial mentioned at the beginning of this section) to construct a cascade of decision trees. As mentioned earlier, the training samples for each tree in the cascade are drawn using a probability distribution over the entire training dataset. This probability distribution for any given tree in the cascade is heavily influenced by which training samples are misclassified by the previous tree.

display_decision_trees_for_different_stages(): This method displays separately the decision tree constructed for each stage of the cascade.

classify_with_boosting(test_sample): This method calls on each decision tree in the cascade to classify the argument test sample.

display_classification_results_for_each_stage() This method shows you the classification decisions made by each decision tree in the cascade. The method also prints out the trust factor associated with each decision tree. It is important to look simultaneously at the classification decision and the trust factor for each tree — since a classification decision made by a specific tree may appear bizarre for a given test sample. This method is useful primarily for debugging purposes.

show_class_labels_for_misclassified_samples_in_stage(stage_index): As for the previous method, this method is useful mostly for debugging. It returns class labels for the samples misclassified by the stage whose integer index is supplied as an argument to the method. Say you have 10 stages in your cascade. The value of the argument `stage_index` would run from 0 to 9, with 0 corresponding to the base tree.

trust_weighted_majority_vote_classifier(): Uses the "final classifier" formula of the AdaBoost algorithm to pool together the classification decisions made by the individual trees while taking into account the trust factors associated with the trees. As mentioned earlier, we associate with each tree of the cascade a trust factor that depends on the overall misclassification rate associated with that tree.

- The `ExamplesBoosting` subdirectory in the main installation directory contains the following three scripts:

```
boosting_for_classifying_one_test_sample_1.py
boosting_for_classifying_one_test_sample_2.py
boosting_for_bulk_classification.py
```

that illustrate how you can use boosting with the help of the `BoostedDecisionTree` class. The Perl version of the module contains the following similarly named scripts in its `ExamplesBoosting` subdirectory:

```
boosting_for_classifying_one_test_sample_1.pl
boosting_for_classifying_one_test_sample_2.pl
boosting_for_bulk_classification.pl
```

- As implied by the names of the first two scripts, these show how to call the different methods of the `BoostedDecisionTree` class

for classifying a single test sample. When you are classifying a single test sample, you can see how each stage of the cascade of decision trees is classifying the test sample. You can also view each decision tree separately and also see the trust factor associated with the tree.

- The third script listed on the previous slide is for the case when you place all of the test samples in a single file. The demonstration script displays for each test sample a single aggregate classification decision that is obtained through trust-factor weighted majority voting by all the decision trees.

16. WORKING WITH RANDOMIZED DECISION TREES

- Consider the following two situations that call for using randomized decision trees, meaning multiple decision trees that are trained using data extracted randomly from a large database of training samples:

  – Consider a two-class problem for which the training database is grossly imbalanced in how many majority-class samples it contains vis-a-vis the number of minority class samples. Let's assume for a moment that the ratio of majority class samples to minority class samples is 1000 to 1. Let's also assume that you have a test dataset that is drawn randomly from the same population mixture from which the training database

was created. Now consider a **stupid** data classification program that classifies everything as belonging to the majority class. If you measure the classification accuracy rate as the ratio of the number of samples correctly classified to the total number of test samples selected randomly from the population, this classifier would work with an accuracy of 99.99%.

– Let's now consider another situation in which we are faced with a huge training database but in which every class is equally well represented. Feeding all the data into a single decision tree would be akin to polling all of the population of the United States for measuring the Coke-versus-Pepsi preference in the country. You are likely to get better results if you construct multiple decision trees,

each trained with a collection of training samples drawn randomly from the training database. After you have created all the decision trees, your final classification decision could then be based on, say, majority voting by the trees.

- Both the data classification scenarios mentioned above can be tackled with ease through the programming interface provided by the new `RandomizedTreesForBigData` class that comes starting with Version 3.3.0 of the Python version and Version 3.42 of the Perl version of the `DecisionTree` module.

- If you want to use `RandomizedTreesForBigData` for classifying data that is overwhelmingly dominated by one class, you would call the constructor of this class in the following fashion for the Python version of the module:

```
import RandomizedTreesForBigData
training_datafile = "MyLargeDatabase.csv"
rt = RandomizedTreesForBigData.RandomizedTreesForBigData(
         training_datafile = training_datafile,
         csv_class_column_index = 48,
         csv_columns_for_features = [39,40,41,42],
         entropy_threshold = 0.01,
         max_depth_desired = 8,
         symbolic_to_numeric_cardinality_threshold = 10,
         looking_for_needles_in_haystack = 1,
         how_many_trees = 5,
         csv_cleanup_needed = 1,
     )
```

Except for obvious changes, the syntax is very similar for the Perl case also.

- Note in particular the constructor parameters:

```
looking_for_needles_in_haystack

how_many_trees
```

The parameter `looking_for_needles_in_haystack` invokes the logic for constructing an ensemble of decision trees, each based on a

training dataset that uses all of the minority class samples, and a random drawing from the majority class samples. The second parameter, `how_many_trees`, tells the system how many trees it should construct.

- With regard to the second data classification scenario presented at the beginning of this section, shown at the top of the next slide is how you would invoke the constructor of the `RandomizedTreesForBigData` class for constructing an ensemble of decision trees, with each tree trained with randomly drawn samples from a large database of training data (with no consideration given to any population imbalances between the different classes):

```
import RandomizedTreesForBigData
training_datafile = "MyLargeDatabase.csv"
rt = RandomizedTreesForBigData.RandomizedTreesForBigData(
         training_datafile = training_datafile,
         csv_class_column_index = 48,
         csv_columns_for_features = [39,40,41,42],
         entropy_threshold = 0.01,
         max_depth_desired = 8,
         symbolic_to_numeric_cardinality_threshold = 10,
         how_many_training_samples_per_tree = 50,
         how_many_trees = 17,
         csv_cleanup_needed = 1,
     )
```

Again, except for obvious changes, the syntax is very similar for the Perl version of the module.

- Note in particular the constructor parameters in this case:

```
how_many_training_samples_per_tree
how_many_trees
```

The first parameter will set the number of samples that will be drawn randomly from the training database and the second the

number of decision trees that will be constructed. **IMPORTANT:** When you set the `how_many_training_samples_per_tree` parameter, you are not allowed to also set the `looking_for_needles_in_haystack` parameter, and vice versa.

- After you have constructed an instance of the `RandomizedTreesForBigData` class, you can call the following methods of this class for constructing an ensemble of decision trees and for data classification with the ensemble:

  get_training_data_for_N_trees(): What this method does depends on which of the two constructor parameters, `looking_for_needles_in_haystack` or `how_many_training_samples_per_tree`, is set. When the former is set, it creates a collection of training datasets for `how_many_trees` number of decision trees, with each dataset being a mixture of the minority class and sample drawn randomly

from the majority class. However, when the latter option is set, all the datasets are drawn randomly from the training database with no particular attention given to the relative populations of the two classes.

show_training_data_for_all_trees(): As the name implies, this method shows the training data being used for all the decision trees. This method is useful for debugging purposes using small datasets.

calculate_class_priors(): Calls on the appropriate method of the main `DecisionTree` class to estimate the class priors for the training dataset to be used for each decision tree.

construct_all_decision_trees(): Calls on the appropriate method of the main `DecisionTree` class to construct the decision trees.

display_all_decision_trees(): Displays all the decision trees in your terminal window. (The textual form of the decision trees is written out to the standard output.)

classify_with_all_trees(): A test sample is sent to each decision tree for classification.

display_classification_results_for_all_trees(): The classification decisions returned by the individual decision trees are written out to the standard output.

get_majority_vote_classification(): This method aggregates the classification results returned by the individual decision trees and returns the majority decision.

- The `ExamplesRandomizedTrees` subdirectory in the main installation directory of the module shows example scripts that you can use to become more familiar with the `RandomizedTreesForBigData` class for solving needle-in-a-haystack and big-data data classification problems. These scripts are:

```
randomized_trees_for_classifying_one_test_sample_1.py

randomized_trees_for_classifying_one_test_sample_2.py

classify_database_records.py
```

- The first of the scripts listed above shows the constructor options to use for solving a needle-in-a-haystack problem — that is, a problem in which a vast majority of the training data belongs to just one class.

- The second script shows the constructor options for using randomized decision trees for the case when you have access to a very large database of training samples and you'd like to construct an ensemble of decision trees using training samples pulled randomly from the training database.

- The third script listed on the previous page illustrates how you can evaluate the classification power of an ensemble of decision trees as constructed by `RandomizedTreesForBigData` by classifying a large number of test samples extracted randomly from the training database.

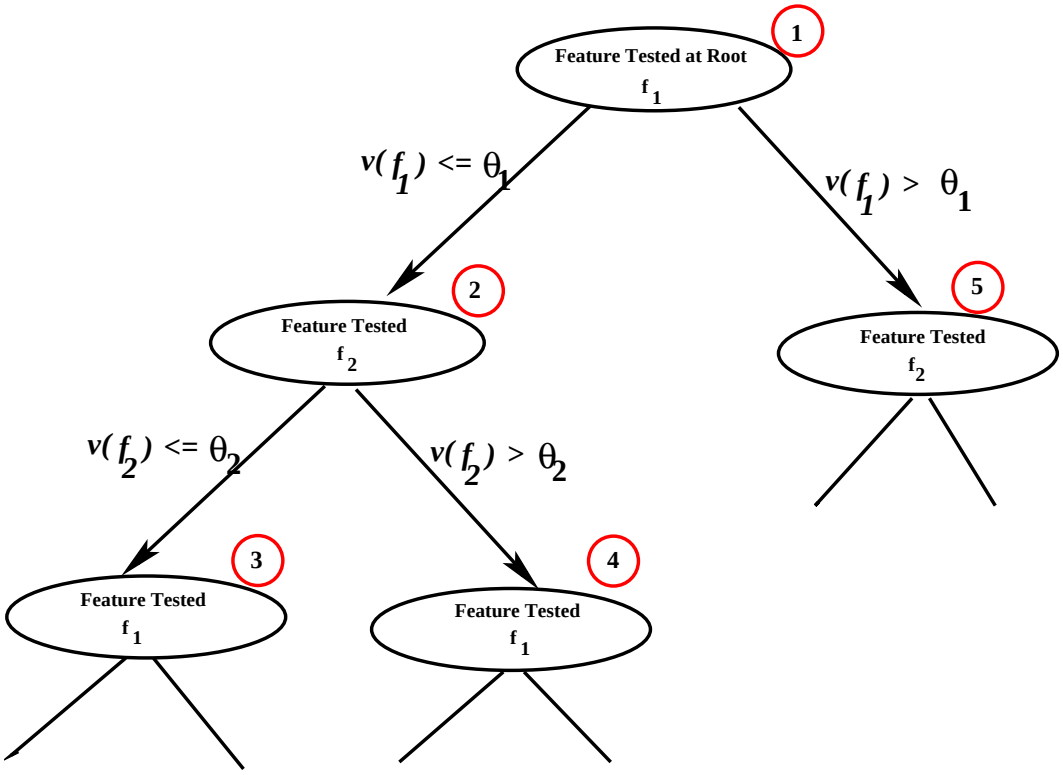
17. Speeding Up Decision Tree Based Classification with Hash Tables

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

- In this section, we will assume that we only have numeric features. As you now know, each interior node for such a decision tree has only two children, unless the node is a leaf node, in which case it has no children.

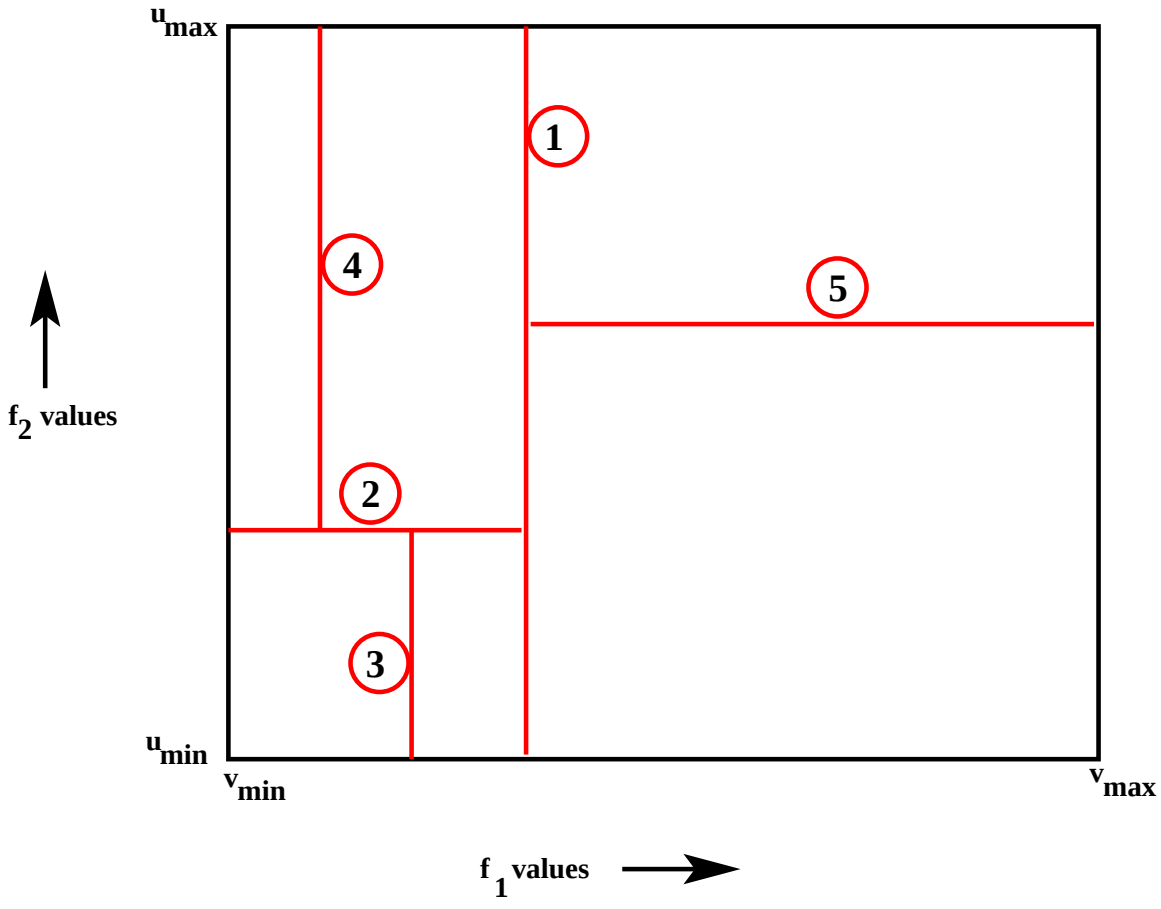
- For the purpose of explanation and pictorial depiction, let's assume that we are dealing with the case of just two numeric features. We will denote these features by f_1 and f_2.

- With just the two features f_1 and f_2, let's say that our decision tree looks like what is shown in the figure below:



- 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 on the previous page. In this figure, the circled numbers next to the partitioning lines correspond to the numbers attached to the nodes in the decision tree on page 65.

- 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.

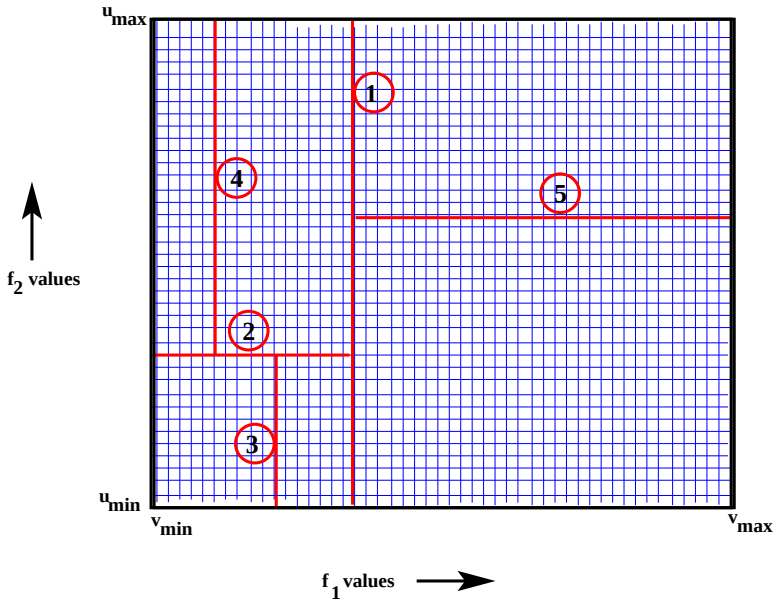
- As we grow the decision tree, our goal must be to reach the nodes that are pure or until there is no further reduction in the entropy in the sense we talked about earlier. A node is pure if it has zero entropy. Obviously, the classification made at that node will be unambiguous.

- 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.

- The resulting one-dimensional array of cells lends itself straightforwardly to being stored as an associative list in the form of a hash table. For example, if you are using Perl, you would use the built-in hash data structure for creating such a hash table. You can do the same in Python with a dictionary.

18. Constructing Regression Trees

- So far we have focused exclusively on decision trees. As you should know by this time, decision tree based modeling requires that the class labels be distinct. That is, the training dataset must contain a relatively small number of discrete class labels for all of your data records if you want to model the data with one or more decision trees.

- However, when one is trying to understand all of the associational relationships that exist in a large database, one often runs into situations where, instead of discrete class labels, you have a continuously valued variable as a dependent variable whose values are predicated on a set of feature values.

- It is for such situations that you will find useful the new class `RegressionTree` that is now a part of the `DecisionTree` module. If interested in regression, look for the `RegressionTree` class in Version 3.4.3 of the Python module and in Version 3.42 of the Perl module.

- For both the Perl and the Python cases, the `RegressionTree` class has been programmed as a subclass of the main `DecisionTree` class. The `RegressionTree` calls on the `DecisionTree` class for several record keeping and some key low-level data processing steps.

- **You can think of regression with a regression tree as a powerful generalization of the very commonly used linear regression algorithms.**

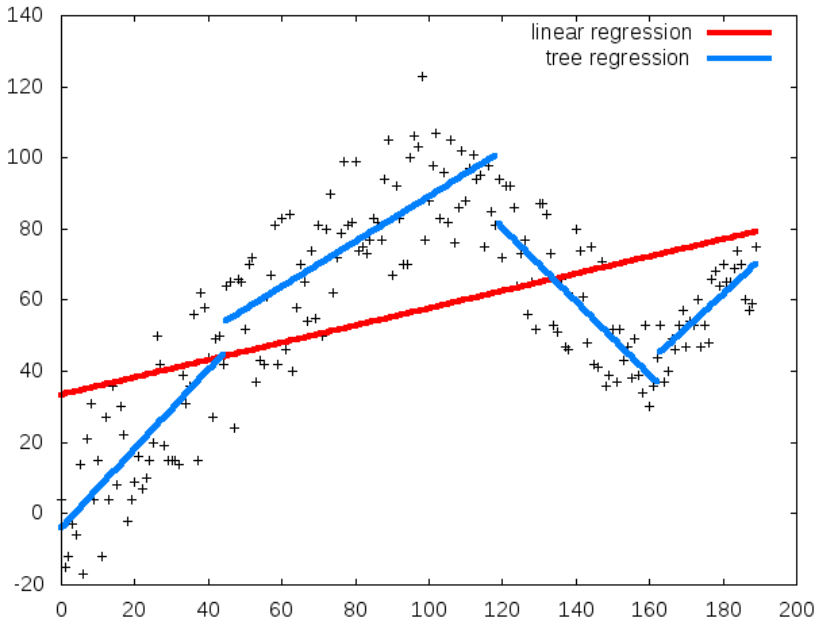
- Although you can certainly carry out polynomial regression with run-of-the-mill linear regression algorithms for modeling nonlinearities between the predictor variables and the dependent variable, specifying the degree of a polynomial is often tricky. Additionally, a polynomial can inject continuities between the predictor and the predicted variables that may not actually exist in the real data.

- Regression trees, on the other hand, give you a piecewise linear relationship between the predictor and the predicted variables that is freed from the constraints of superimposed continuities at the joins between the different segments.

- See the following tutorial for further information regarding the standard linear regression approach and the regression that can be achieved with the `RegressionTree` class:

  `https://engineering.purdue.edu/kak/Tutorials/RegressionTree.pdf`

- While linear regression has sufficed for many applications, there are many others where it fails to perform adequately. Just to illustrate this point with a simple example, shown below is some noisy data for which the linear regression yields the line shown in red. The blue line is the output of the tree regression algorithm as implemented in the `RegressionTree` class:



- You will find the `RegressionTree` class easy to use in your own scripts. See my Regression Tree tutorial at:

  `https://engineering.purdue.edu/kak/Tutorials/RegressionTree.pdf`

  for how to call the constructor of this class and how to invoke the functionality incorporated in it.

- You will also find example scripts in the `ExamplesRegression` subdirectories of the main installation directory that you can use to become more familiar with tree regression.

19. Historical Antecedents of Decision Tree Classification in Purdue RVL

- 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 sections of this tutorial. Subsequently, the decision tree was converted into a hash table for fast classification.

- The testing phase consisted of the robot using the hash table constructed during the learning phase to recognize the objects and to estimate their poses. The fact that the robot successfully manipulated the objects established for us the viability of using decision-tree based learning in the context of robot vision.

- 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 Understanding, pp. 387-416, Vol. 61,
    No. 3, 1995.
```

20. Acknowledgments

In one form or another, decision trees have been around for over fifty years. From a statistical perspective, they are closely related to classification and regression by recursive partitioning of multidimensional data. Early work that demonstrated the usefulness of such partitioning of data for classification and regression can be traced, in the statistics community, to the work done by Terry Therneau in the early 1980's, and, in the machine learning community, to the work of Ross Quinlan in the mid 1990's.

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.