<table>
<thead>
<tr>
<th>Section Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>3</td>
</tr>
<tr>
<td>Entropy</td>
<td>10</td>
</tr>
<tr>
<td>Conditional Entropy</td>
<td>15</td>
</tr>
<tr>
<td>Average Entropy</td>
<td>17</td>
</tr>
<tr>
<td>Using Class Entropy to Discover the Best Feature for Discriminating Between the Classes</td>
<td>18</td>
</tr>
<tr>
<td>Constructing a Decision Tree</td>
<td>24</td>
</tr>
<tr>
<td>Incorporating Numeric Features</td>
<td>37</td>
</tr>
<tr>
<td>The Perl Module  Algorithm::DecisionTree-2.30</td>
<td>49</td>
</tr>
<tr>
<td>The Python Module  DecisionTree-2.3.1</td>
<td>57</td>
</tr>
<tr>
<td>Bulk Classification of All Test Data Records in a CSV or a <code>.dat</code> File</td>
<td>66</td>
</tr>
<tr>
<td>Dealing with Large Dynamic-Range and Heavy-tailed Features</td>
<td>69</td>
</tr>
<tr>
<td>Decision Tree Introspection</td>
<td>72</td>
</tr>
<tr>
<td>Speeding Up Decision Tree Based Classification with Hash Tables</td>
<td>76</td>
</tr>
<tr>
<td>Testing the Quality of the Training Data</td>
<td>83</td>
</tr>
<tr>
<td>For Digging Deeper</td>
<td>89</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>91</td>
</tr>
</tbody>
</table>
1. Introduction

- Let’s say that 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 as follows:

\[
\begin{array}{cccccc}
  f_1 & f_2 & f_3 & \ldots & f_N & \Rightarrow \text{DECISION} \\
\end{array}
\]

<table>
<thead>
<tr>
<th>val_1</th>
<th>val_2</th>
<th>val_3</th>
<th>\ldots</th>
<th>val_N</th>
<th>\Rightarrow</th>
<th>d1</th>
</tr>
</thead>
<tbody>
<tr>
<td>val_1</td>
<td>val_2</td>
<td>val_3</td>
<td>\ldots</td>
<td>val_N</td>
<td>\Rightarrow</td>
<td>d2</td>
</tr>
<tr>
<td>val_1</td>
<td>val_2</td>
<td>val_3</td>
<td>\ldots</td>
<td>val_N</td>
<td>\Rightarrow</td>
<td>d1</td>
</tr>
<tr>
<td>val_1</td>
<td>val_2</td>
<td>val_3</td>
<td>\ldots</td>
<td>val_N</td>
<td>\Rightarrow</td>
<td>d1</td>
</tr>
<tr>
<td>\ldots</td>
<td>\ldots</td>
<td>\ldots</td>
<td>\ldots</td>
<td>\ldots</td>
<td>\ldots</td>
<td>\ldots</td>
</tr>
</tbody>
</table>

where, as you’d expect, each value \( \text{val}_i \) would be specific to each row of the table shown above. For convenience, we refer to each column of the table as representing a feature \( f_i \) whose value goes into your decision making process.
• 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 11 of this tutorial.]
Let’s say that your new data record for which you need to make a decision looks like:

\[ \text{new}_{-1} \text{val}_1 \text{ new}_{-1} \text{val}_2 \text{ new}_{-1} \text{val}_2 \text{ .... } \text{new}_{-1} \text{val}_N \]

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 11.

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 could be used by a computer vision system to automatically figure out which features work 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 make any sense in a given context.
• Consider therefore the following computer vision experiment that could be used for creating a system that figures out on its own how to best distinguish between a set of object on the basis of the training examples supplied by a human. [As mentioned in Section 14 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.]

– We show a number of different objects to a sensor system consisting of cameras, 3D vision systems, etc. These objects belong to $M$ different classes.

– For each object shown, all that we tell the computer is its class label. We do NOT tell the computer how to discriminate between the objects belonging to the different classes.
We supply a large vocabulary of features to the computer and also provide the computer with tools to extract these features from the sensory information collected from each object. [For image data, these features could be color, texture, etc. For range data, the features could be curvature, corners formed by the meeting of the planar surfaces, etc.]

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

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

- Entropy is a powerful tool that can be used by a computer to determine on its 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 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 involves 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 $p_i$, we can associate the following entropy with $X$:

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

\[
\begin{array}{cccccccc}
\text{hist}(X) & | & | & | & | & | & | & | \\
\text{(normalized)} & | & | & | & | & | & | & | \\
\end{array}
\]

\[
\begin{array}{cccccccc}
\hline
\text{1} & \text{2} & \text{3} & \text{4} & \text{5} & \text{6} & \text{7} & \text{8} \\
\hline
\end{array}
\]

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

\[
H(X) = - \sum_{i=1}^{8} \frac{1}{8} \log_2 \frac{1}{8}
\]

\[
= - \sum_{i=1}^{8} \frac{1}{8} \log_2 2^{-3}
\]

\[
= 3 \text{ bits}
\]
• Now consider the following example in which the uniformly distributed random variable $X$ takes one of 64 possible values:

\[
\begin{array}{cccccccc}
1/64 & | & | & | & | & | & | & | \\
\text{histogram} & | & | & | & | & | & | & | \\
\text{(normalized)} & | & | & | & | & | & | & | \\
\end{array}
\]

\[
1 \quad 2 \quad 3 \quad 4 \quad \ldots \ldots \quad 63 \quad 64
\]

In this case,

\[
H(X) = - \sum_{i=1}^{64} \frac{1}{64} \log_2 \frac{1}{64}
\]

\[
= - \sum_{i=1}^{8} \frac{1}{8} \log_2 2^{-6}
\]

\[
= 6 \quad \text{bits}
\]

• 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:

![Histogram](histogram.png)

In this case, we obviously have

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

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

$$H(X) = - \sum_{i=1}^{N} p(x_i) \log_2 p(x_i) = - [p_1 \log_2 p_1 + ... p_k \log_2 p_k + ... + p_N \log_2 p_N]$$
\[ = - 1 \times \log_2 1 \quad \text{bits} \]

\[ = 0 \quad \text{bits} \]

where we use the fact that as \( p \to 0^+ \),
\( p \log p \to 0 \) 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

- 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,

$$H(Y|X) = H(X,Y) - H(X)$$

Given two random variables $X$ and $Y$, the entropy contained in both when taken together is $H(X,Y)$. The above definition says that, if $X$ and $Y$ are inter-dependent, and if we know $X$, we can reduce our measure of chaos in $Y$ by the chaos that is attributable to $X$. 
• But what do we mean by knowing \( X \) in the context of \( X \) and \( Y \) being inter-dependent? Before we answer this question, let’s first look at the formula for the joint entropy \( H(X, Y) \), which is given by

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

• 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:

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

• The formula shown for \( H(Y|X) \) on the previous page is the average of \( H(Y|X=a) \) over all possible instantiations \( a \) for \( X \).
4. Average Entropies

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

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

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

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

where $H(Y|X=a)$ is given by the formula at the bottom of the previous slide.
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 \( \{f_1, f_2, \ldots, f_K\} \).

- 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:

$$H(C|f) = \sum_a H(C|v(f) = a) \times p(v(f) = a)$$

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

\[ H(C | v(f) = a) = - \sum_m p(C_m | v(f) = a) \times \log_2 p(C_m | v(f) = a) \]

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 \( p(C_m | v(f) = a) \) that is needed on the right hand side?

• We will next present two different ways for calculating \( p(C_m | v(f) = a) \). 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 \( p(C_m|v(f) = a) \) 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: \( f_1, f_2, \ldots, f_K \). 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 \( p(C_m|v(f) = a) \) 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 \( p(C_m|v(f) = a) \) uses the Bayes’ Theorem:

\[
p(C_m|v(f) = a) = \frac{p(v(f) = a|C_m) \times p(C_m)}{p(v(f) = a)}
\]

This formula also allows us to carry out separate measurement experiments for objects belonging to different classes.
• Another advantage of the formula shown at the bottom of the previous slide is that it is no longer a problem if only a small number of objects are available for some of the classes — such non-uniformities in object populations are taken care of by the $p(C_m)$ term.

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

$$\sum_{m} p(C_m|v(f)=a) = 1$$

• What’s interesting is that if we do obtain $p(v(f)=a)$ 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 19. Otherwise, $p(v(f)=a)$ 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 19.

• 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 multi-dimensional 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 $H(C|v(f_j) = a_j, f)$.

[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:

\[
H\left(C \mid f, v(f_j) = a_j \right) = \\
\sum_b H\left(C \mid v(f) = b, v(f_j) = a_j \right) \times p\left(v(f) = b, v(f_j) = a_j \right)
\]

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*:

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

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

$$H(C \mid v(f) = b, v(f_j) = a_j) = -\sum_{C_m} p(C_m \mid v(f) = b, v(f_j) = a_j) \times \log_2 p(C_m \mid v(f) = b, v(f_j) = a_j)$$

for any given feature $f \neq f_j$.

*Some folks 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:

\[
p(C_m | v(f) = b, v(f_j) = a_j) =
\]

\[
= \frac{p(v(f) = b, v(f_j) = a_j | C_m) \times p(C_m)}{p(v(f) = b, v(f_j) = a_j)}
\]

\[
= \frac{p(v(f) = b | C_m) \times p(v(f_j) = a_j | C_m) \times p(C_m)}{p(v(f) = b) \times p(v(f_j) = a_j)}
\]

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 $H(C\mid f_j)$ 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 $H(C\mid f_k, v(f_j) = a_j)$ is less than the entropy $H(C\mid f_j)$ inherited by the child from the root.
• If the condition \( H(C \mid f, v(f_j) = v_j) < H(C \mid f_j) \) 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:

\[
p\left(C_m \ rivals \left| \ v(f_j) = a_j, \ v(f_k) = b_k, \ldots \right. \right)
\]

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

• The above probability may be estimated with Bayes Theorem:

\[
p\left(C_m \ rivals \left| \ v(f_j) = a_j, \ v(f_k) = b_k, \ldots \right. \right) = \frac{p\left( v(f_j) = a_j, \ v(f_k) = b_k, \ldots \mid C_m \right) \times p\left( C_m \right)}{p\left( v(f_j) = a_j, \ v(f_k) = b_k, \ldots \right)}
\]
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:

\[
p\left( v(f_j) = a_j, \ v(f_k) = b_k, \ldots \right) = \prod_{f \text{ on branch}} p\left( v(f) = \text{value} \right)
\]

\[
p\left( v(f_j) = a_j, \ v(f_k) = b_k, \ldots \mid C_m \right) = \prod_{f \text{ on branch}} p\left( v(f) = \text{value} \mid C_m \right)
\]
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:

\[
H(C \mid v_{th}(f) = \theta) = H(C \mid v(f) \leq \theta) \times p(v(f) \leq \theta) + H(C \mid v(f) > \theta) \times p(v(f) > \theta)
\]

where \( v_{th}(f) = \theta \) means that we have set the decision threshold for the values of the feature \( f \) at \( \theta \) 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 $\theta$ and the other for which all values of $f$ are greater than $\theta$. 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

$$H\left( C \mid v(f) \leq \theta \right) = -\sum_{m} p(C_m \mid v(f) \leq \theta) \times \log_2 p(C_m \mid v(f) \leq \theta)$$

and

$$H\left( C \mid v(f) > \theta \right) = -\sum_{m} p(C_m \mid v(f) > \theta) \times \log_2 p(C_m \mid v(f) > \theta)$$
• We can estimate \( p(C_m|v(f) \leq \theta) \) and \( p(C_m|v(f) > \theta) \) by using the Bayes' Theorem:

\[
p(C_m|v(f) \leq \theta) = \frac{p(v(f) \leq \theta | C_m) \times p(C_m)}{p(v(f) \leq \theta)}
\]

and

\[
p(C_m|v(f) > \theta) = \frac{p(v(f) > \theta | C_m) \times p(C_m)}{p(v(f) > \theta)}
\]

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:

\[
\sum_{m} p(C_m \mid v(f) \leq \theta) = 1
\]

and

\[
\sum_{m} p(C_m \mid v(f) > \theta) = 1
\]

• 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:

![Feature Tested at Root](image)

• 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 $\theta$, 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 $\theta_j$, 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 $\theta_k$. 
The choice \((f_k, \theta_k)\) 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:

\[
H\left( C \mid v(f_j) \leq \theta_j, \ v(f_k) \leq \theta_k \right) \times p\left( v(f_j) \leq \theta_j, \ v(f_k) \leq \theta_k \right) \\
+ \\
H\left( C \mid v(f_j) \leq \theta_j, \ v(f_k) > \theta_k \right) \times p\left( v(f_j) \leq \theta_j, \ v(f_k) > \theta_k \right)
\]

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-2.30 for Constructing a Decision Tree and for Classification

**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-2.30/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 2.30 when you get to the CPAN page for the module.]
• There are two ways to use this module: the new way that allows you to incorporate numeric features in constructing a decision tree, and the old way that works for just the symbolic features. From a usage standpoint, an important difference between the two is with regard to how you supply your training data to the module. If your training data includes numeric features, you must use a CSV file for feeding the data into the module. On the other hand, if your data is just symbolic, you can continue to use the ‘.dat’ files.

• To use Version 2.30 when you have numeric features (or a mixture of numeric and symbolic features), you first need to construct an instance of the Algorithm::DecisionTree class as shown on the next slide.
• A constructor call when your training data is in a CSV file looks like:

```perl
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,
);
```

• 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 feature. 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.

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

```php
$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 classify a data record. 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.

• Note that if your features are purely symbolic, you can continue to use the same constructor syntax that was used in the older versions of this module. However, your old ‘.dat’ training files will not work with the new version. The good news is that with just a small fix, you can continue to use them. The fix and why it was needed is described in the file README_for_dat_files in the examples directory of the module.
• The module also allows you to generate your own training and testing datasets for experimenting with decision trees classifiers. For that, the module file contains the following classes: (1) TrainingDataGeneratorNumeric, (2) TrainingDataGeneratorSymbolic, and (3) TestDataGeneratorSymbolic.

• 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 places its output in a ‘.dat’ 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. The class `TestDataGeneratorSymbolic` is just a convenience class for creating test data — that is, data records without class labels. The labels are placed in a separate file.

• **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 or a ‘.dat’ file in one go.
9. The Python Module
DecisionTree-2.3.1 for Constructing a Decision Tree and for Classification

**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 2.3.1. To download Version 2.3.1:

  [http://pypi.python.org/pypi/DecisionTree/2.3.1](http://pypi.python.org/pypi/DecisionTree/2.3.1)

  The API of this software package is at the HTML link at the bottom of the web page at the above URL.
• Like its immediate predecessors, Version 2.3.1 accepts training data from two different kinds of files: `.csv` files and the old style `.dat` files. **Training data that includes numeric features can only be accepted through `.csv` files.** If you are not already familiar with `.csv` files, the acronym CSV stands for Comma Separated Values. The previous versions of the module used white-space separated values for the training data records in `.dat` files.

• When you are supplying training data through a `.csv` file, you need to tell the module which column contains the class labels for the training records and which columns are to be used as features. **Column indexing is zero based.** That is, the index of the first column is considered to be zero. This information is supplied to the module through constructor options as shown on the next slide.
• 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:

```python
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,
)
```

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.

• Shown below is an example of the constructor call for the case of purely symbolic features read from a `.dat` file.

```python
training_datafile = 'training.dat'

dt = DecisionTree.DecisionTree(
    training_datafile = training_datafile,
    entropy_threshold = 0.01,
    max_depth_desired = 3,
)
```

• 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.

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

```python
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:

```python
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:

```python
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.

• Note: The synthetic training datafiles generated by the older versions of this module will NOT work with versions 2.0 and higher.
• For large test datasets, see the next section for the demonstration scripts in Version 2.3.1 of the Python module that show you how you can classify all your data records in a CSV or a `.dat` file in one go.
10. Bulk Classification of All Test Data Records in a CSV or an ‘.dat’ File

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

- The Examples directories of Version 2.30 of the Perl module and Version 2.3.‘ of the Python 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.

- If your file-based test data is purely symbolic, you can also use the old-style ‘.dat’ files for storing the test data records.
For the case of Perl, see the following scripts in the examples directory of Version 2.30 of the module for bulk classification of data records in a CSV file for the first script and ‘.dat’ file for the second:

classify_test_data_in_a_file_numeric.pl
classify_test_data_in_a_file_symbolic.pl

And for the case of Python, check out the following scripts, the first for the case of a CSV file and the second for the case of a ‘.dat’ file:

classify_test_data_in_a_file_numeric.py
classify_test_data_in_a_file_symbolic.py

All scripts mentioned on this slide 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.
• **IMPORTANT:** The test datafiles for both the numeric/symbolic and the purely symbolic cases must include a column for the class label even when all the entries in this column are just the empty string ""

• If you are going to be using the old-style `.dat` files for the test datafiles, they must mention the order in which the features values are presented. For an example, see the file `testdata.dat` in the examples directory.
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. 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 the feature space is an important contributor to the **generalization abilities** of decision-tree-based classification.
• 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 two scripts in the Examples directory of the Python module for how to carry out DT introspection:

    introspection_in_a_loop.py
    introspection_at_one_node.py

and the following two scripts in the examples directory of the Perl module

    introspection_in_a_loop.pl
    introspection_at_one_node.pl

In both cases, the first script puts you in an interactive session in which you will be asked for the node numbers you are interested in. At each node, you will be asked for whether or not you are interested in specific questions that the introspector can provide answers for. The second script does the same but only at a specific node.
13. **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:

![Decision Tree Diagram]

• 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.
14. 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 three 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; and (3) The number of the training samples for the different classes not being in proportion to the real-world prior probabilities of the classes.
• 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 2.3.1 of the Python Decision Tree module and version 2.30 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 three 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 the following in mind 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 three 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 your training 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:

```python
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,
)
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:

```perl
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,
);
$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.
15. For Digging Deeper

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

• The learning phase in Lynne’s demonstrations consisted of merely showing 3D objects to the sensor suite. For each object shown, the human would tell the computer what its identity and pose was.
• From the human supplied class labels and pose information, the computer constructed a decision tree in the manner described in the previous 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 proof of the pudding lay in the robot successfully manipulating the objects.

• The details of this system are published in Lynne Grewe and Avinash Kak, "Interactive Learning of a Multi-Attribute Hash Table Classifier for Fast Object Recognition," Computer Vision and Image Understanding, pp. 387-416, Vol. 61, No. 3, 1995.
16. 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 to the work of Ross Quinlan in the mid 1990’s in the machine learning community.

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.