3

DEVELOPMENT AND APPLICATIONS OF DECISION TREES

HUSSEIN ALMUALLIM

Information and Computer Science Department, King Fahd University of Petroleum & Minerals, Dhahran 31261, Saudi Arabia

SHIGEO KANEDA

Graduate School of Policy and Management, Doshisha University, Imadegawa-Karasuma-Higashi-iru, Kamigyou-ku, Kyoto 602-8580, Japan

YASUHIRO AKIBA

NTT Communication Science Laboratories, 2-4 Hikari-dai, Seika-cho, Souraku-gun, Kyoto 619-0237, Japan

I. INTRODUCTION
II. CONSTRUCTING DECISION TREES FROM EXAMPLES
  A. A Basic Tree Construction Procedure
  B. Which Test to Select
III. EVALUATION OF A LEARNED DECISION TREE
IV. OVERFITTING AVOIDANCE
V. EXTENSIONS TO THE BASIC PROCEDURE
  A. Handling Various Attribute Types
  B. Incorporating More Complex Tests
  C. Attributes with Missing Values
VI. VOTING OVER MULTIPLE DECISION TREES
  A. Bagging
  B. Boosting
VII. INCREMENTAL TREE CONSTRUCTION
VIII. EXISTING IMPLEMENTATIONS
IX. PRACTICAL APPLICATIONS
  A. Predicting Library Book Use
  B. Exploring the Relationship Between the Research Octane Number and Molecular Substructures
  C. Characterization of Leiomyomatous Tumors
  D. Star/Cosmic-Ray Classification in Hubble Space Telescope Images
X. FURTHER READINGS
ACKNOWLEDGMENTS
REFERENCES

Expert Systems, Vol. 1

I. INTRODUCTION

A critical issue in artificial intelligence research is to overcome the so-called "knowledge-acquisition bottleneck" in the construction of knowledge-based systems. Experience in typical real-world domains has shown that the conventional approach of extracting knowledge directly from human experts is associated with many problems and shortcomings. Interviews with experts are usually slow, inefficient, and frustrating for experts and knowledge engineers alike [1]. This is particularly true in those application domains in which decisions made by experts are "intuitive" ones, guided by imprecise and imperfect knowledge. In such domains, different experts may make substantially different judgments, and even the same expert may not give the same solution when confronted with the same problem twice over a period of time. These problems become more acute when dealing with large knowledge-based systems in which upgrading the knowledge base, fixing previous erroneous knowledge, and maintaining integrity are extremely challenging tasks.

A promising approach to ease the knowledge-acquisition bottleneck is to employ some learning mechanism to extract the desired knowledge automatically or semiautomatically from actual cases or *examples* that have been previously handled by the domain experts. This *machine learning* approach enjoys several advantages: (i) In problems for which knowledge is expert-dependent, one can simply learn from examples handled by different experts, with the hope that this will *average* the differences among different experts. (ii) Being able to construct knowledge automatically makes the upgrading task easier because one can rerun the learning system as more examples accumulate. Some learning methods are indeed "incremental" in nature. (iii) Machine learning can be applied to problems for which no experts exist. This is the case in data mining and knowledge discovery in databases, for which machine learning techniques are employed to automatically discover *new* knowledge.

Considerable attention has been devoted by the machine learning research community to the task of acquiring "classification knowledge" for which, among a predeclared set of available classes, the objective is to choose the most appropriate class for a given case. The goal in such research is to develop methods that induce the desired classification knowledge from a given set of preclassified examples. Significant progress has been made in the last decade toward this goal, and various methods for automatically inducing *classifiers* from data are now available. In particular, constructing classifiers in the form of decision trees has been quite popular, and a number of successful real-world applications that employ decision tree construction methods has been reported.

For knowledge-based systems, decision trees have the advantage of being comprehensible by human experts and of being directly convertible into production rules. Moreover, when used to handle a given case, a decision tree not only provides the solution for that case, but also states the reasons behind its choice. These features are very important in typical application domains in which human experts seek tools to aid in conducting their job while remaining "in the driver's seat." Another advantage of using decision trees is the ease and efficiency of their construction compared to that of other classifiers such as neural networks.

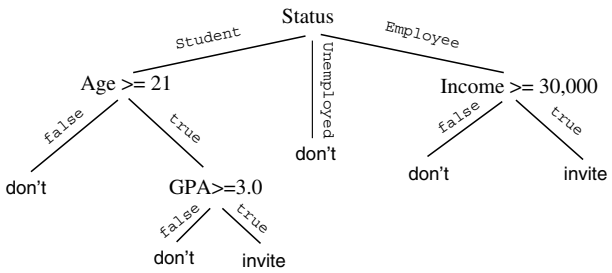

FIGURE 1 A decision tree that determines whether or not to offer a credit card invitation.

In this chapter, we first present a basic method for automatically constructing decision trees from examples and review various extensions of this basic procedure. We then give a sample of real-world applications for which the decision tree learning approach has been reported to be successful.

II. CONSTRUCTING DECISION TREES FROM EXAMPLES

A decision tree is used as a classifier for determining an appropriate action (among a predetermined set of actions) for a given case. Consider, for example, the task of targeting good candidates to be sent an invitation to apply for a credit card: given certain information about an individual, we need to determine whether or not he or she can be a candidate. In this example, information about an individual is given as a vector of *attributes* that may include sex (male or female), age, status (student, employee, or unemployed), college grade point average (GPA), annual income, social security number, etc. The allowed actions are viewed as *classes,* which are in this case to offer or not to offer an invitation. A decision tree that performs this task is sketched in Fig. 1. As the figure shows, each internal node in the tree is labeled with a "test" defined in terms of the attributes and has a branch for each possible outcome for that test, and each leaf in the tree is labeled with a class.

Attributes used for describing cases can be nominal (taking one of a prespecified set of values) or continuous. In the above example, *Sex* and *Status* are nominal attributes, whereas *Age* and *GPA* are continuous ones. Typically, a test defined on a nominal attribute has one outcome for each value of the attribute, whereas a test defined on a continuous attribute is based on a fixed threshold and has two outcomes, one for each interval as imposed by this threshold.[1] The decision tree in Fig. 1 illustrates these tests.

To find the appropriate class for a given case (individual), we start with the test at the root of the tree and keep following the branches as determined by the values of the attributes of the case at hand, until a leaf is reached. For example, suppose the attribute values for a given case are as follows:

$$\textit{Name} = \text{Andrew};\ \textit{Social Security No.} = 199199;\ \textit{Age} = 22;\ \textit{Sex} = \text{Male};$$

$$\textit{Status} = \text{Student};\ \textit{Annual Income} = 2,000;\ \textit{College GPA} = 3.39.$$

[1] Other kinds of attributes and tests also exist as will be explained later.

To classify this case, we start at the root of the tree of Fig. 1, which is labeled *Status*, and follow the branch labeled *Student* from there. Then at the test node $Age \geq 21$, we follow the *true* branch, and at the test node $GPA \geq 3.0$, we again follow the "true" branch. This leads finally to a leaf labeled "invite", indicating that this person is to be invited according to this decision tree.

Decision tree *learning* is the task of constructing a decision tree classifier, such as the one in Fig. 1, from a collection of *historical cases*. These are individuals who are already marked by experts as being good candidates or not. Each historical case is called a *training example*, or simply an *example*, and the collection of such examples from which a decision tree is to be constructed is called a *training sample*. A training example is assumed to be represented as a pair $\langle X, c \rangle$, where X is a vector of attribute values describing some case, and c is the appropriate class for that case. A collection of examples for the credit card task is shown in Fig. 2. The following subsections describe how a decision tree can be constructed from such a collection of training examples.

A. A Basic Tree Construction Procedure

Let $S = \{\langle X_1, c_1 \rangle, \langle X_2, c_2 \rangle, \ldots, \langle X_k, c_k \rangle\}$ be a training sample. Constructing a decision tree form S can be done in a divide-and-conquer fashion as follows:

Step 1: If all the examples in S are labeled with the same class, return a leaf labeled with that class.

Step 2: Choose some test t (according to some criterion) that has two or more mutually exclusive outcomes $\{o_1, o_2, \ldots, o_r\}$.

Step 3: Partition S into disjoint subsets $S_1, S_2, \ldots, S_r$, such that S_i consists of those examples having outcome o_i for the test t, for $i = 1, 2, \ldots, r$.

Step 4: Call this tree-construction procedure recursively on each of the subsets $S_1, S_2, \ldots, S_r$, and let the decision trees returned by these recursive calls be $T_1, T_2, \ldots, T_r$.

Step 5: Return a decision tree T with a node labeled t as the root and the trees $T_1, T_2, \ldots, T_r$ as subtrees below that node.

For illustration, let us apply the above procedure on the set of examples of Fig. 2. We will use the Case IDs 1–15 (listed in the first column) to refer to each of these examples.

- $S = \{1, 2, 3, \ldots, 15\}$ has a mixture of classes, so we proceed to Step 2.
- Suppose we use the attribute *Status* for our test. This test has three outcomes, "*Student*", "Unemployed", and "Employee". It partitions S into the subsets $S_1 = \{1, 4, 6, 7, 9, 10, 11\}$, $S_2 = \{5, 8, 12, 13\}$, and $S_3 = \{2, 3, 14, 15\}$, respectively, for these outcomes.
- Note that S_1 has a mixture of classes. Suppose we choose the test $Age \geq 21$?. This test partitions S_1 into $S_{11} = \{6, 10\}$ for the *false* outcomes and $S_{12} = \{1, 4, 7, 9, 11\}$ for the *true* outcome.
- $S_{11} = \{6, 10\}$ has just one class "don't", so a leaf labeled with this class is returned for the call on S_{11}.
- For the set S_{12}, which has a mixture of classes, if we choose $GPA \geq 3.0$?, then the set will be partitioned into $S_{121} = \{7, 9\}$ and $S_{122} = \{1, 4, 11\}$.

Case ID	*Name*	*Social Security No.*	*Age*	*Sex*	*Status*	*Income*	*GPA*	Class
1	John	123321	22	Male	Student	3,000	3.22	Invite
2	Mary	343422	38	Female	Employee	32,000	2.00	Invite
3	Ali	876345	46	Male	Employee	69,000	2.90	Invite
4	Lee	673245	23	Male	Student	3,500	3.1	Invite
5	Ted	451087	45	Male	Unemployed	5,000	3.1	Don't
6	Nick	239847	19	Male	Student	1,300	3.8	Don't
7	Liz	229951	23	Female	Student	12,000	2.8	Don't
8	Debby	234819	33	Female	Unemployed	5,000	0.00	Don't
9	Pat	258199	32	Male	Student	1,000	2.1	Don't
10	Peter	813672	20	Male	Student	32,000	3.9	Don't
11	Dona	501184	23	Female	Student	6,600	3.3	Invite
12	Jim	619458	40	Male	Unemployed	35,000	3.3	Don't
13	Kim	654397	31	Female	Unemployed	14,000	3.0	Don't
14	Pan	350932	59	Male	Employee	29,000	2.8	Don't
15	Mike	357922	33	Male	Employee	19,000	2.6	Don't

FIGURE 2 Examples for the credit card task.

- The calls on the sets S_{121} and S_{122} will return leaves labeled "don't" and "invite", respectively, and thus, the call on the set S_{12} will return the subtree of Fig. 3a.
- Now that we are done with the recursive calls on S_{11} and S_{12}, the call on the set S_1 will return the subtree of Fig. 3b.
- The call on the set S_2 will return a leaf labeled "don't".
- For S_3, which contains a mixture of classes, suppose we choose the test $Income \geq 30,000$?. This will partition S_3 into $S_{31} = \{14, 15\}$ for the *false* outcome and $S_{32} = \{2, 3\}$ for the *true* outcome.
- The recursive calls on S_{31} and S_{32} will return leaves labeled "don't" and "invite", respectively, and thus, the call on S_3 will return the subtree of Fig. 3c.
- Finally, the call on the entire training sample S will return the tree of Fig. 1.

Obviously, the quality of the tree produced by the above top-down construction procedure of decision trees depends mainly on how tests are chosen in Step 2.

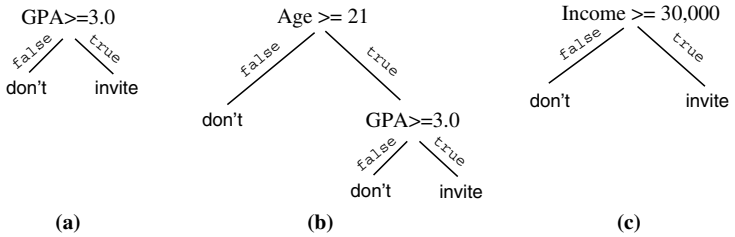

FIGURE 3 Subtrees returned by recursive calls on subsets of the training examples for credit card invitations.

Moreover, the stopping criterion of Step 1 (which requires that the passed set of training examples have a single class) may not be the strategy to quit recursion and stop growing the tree. We will elaborate on these points in the following subsection and in Section IV.

B. Which Test to Select?

Regardless of the test selection criterion adopted in Step 2 of the above tree-construction procedure, the procedure would eventually lead to a decision tree that is *consistent* with the training examples. That is, for any training example $\langle X, c \rangle \in S$, the learned tree gives c as the class for X. Nevertheless, the tree-building process is not intended to merely do well for the training examples themselves. Rather, the goal is to build (among many possible consistent trees) a tree that reveals the underlying structure of the domain, so that it can be used to "predict" the class of new examples not included in the training sample and can also be used by human experts to gain useful insight about the application domain. Therefore, some careful criterion should be employed for test selection in Step 2 so that important tests (such as *Income* and *Status* in our credit card example) are preferred and irrelevant ones (such as *Name* and *Sex*) are ignored. Ideally, one would like the final tree to be as compact as possible, because this is an indication that attention has been focused on the most relevant tests.

Unfortunately, finding the most compact decision tree is an intractable problem (as shown in [2]), so one has to resort to heuristics that help in finding a "reasonably small" one. The basic idea is to measure the importance of a test by estimating how much influence it has on the classification of the examples. In this way, correct classification is obtained using a small number of tests, meaning that all paths in the tree will be short and the tree as a whole will be small.

Note that, at any stage, the absolutely best test would be a test that partitions the passed training sample S into subsets $S_1, S_2, \ldots, S_r$ such that each subset S_i contains examples that are all of the same class (such subsets are called "pure"). Choosing such a test would immediately lead us to stop further recursive partitioning. The goodness of a test can thus be estimated on the basis of how close it is to this "perfect" behavior. In other words, the higher the purity of the subsets $S_1, S_2, \ldots, S_r$ resulting from a test t, the better that test is.

A popular practice in applying this idea is to measure the expected amount of information provided by the test based on information theory. Given a sample S, the average amount of information needed to find the class of a case in S is estimated by the function

$$\text{info}(S) = -\sum_{i=1}^{k} \frac{|S^i|}{|S|} \times \log_2 \frac{|S^i|}{|S|} \text{ bits},$$

where $S^i \subseteq S$ is the set of examples S of class i and k is the number of classes. For example, when $k = 2$ and when S has equal numbers of examples of each class, the above quantity evaluates to 1, indicating that knowing the class of a case in S is "worth" one bit. On the other hand, if all the examples is S are of the same class, then the quantity evaluates to 0, because knowing the class of a case in S provides no information. In general, the above non-negative quantity (known as the *entropy*

of the set S) is maximized when all the classes are of the same frequency and is equal to 0 when S is pure, that is, when all the examples is S are of the same class.

Suppoe that t is a test that partitions S into $S_1, S_2, \ldots, S_r$; then the *weighted average entropy* over these subsets is computed by

$$\sum_{i=1}^{r} \frac{|S_i|}{|S|} \times \text{info}(S_i).$$

Evaluation of the test t can then be based on the quantity

$$\text{gain}(t) = \text{info}(S) - \sum_{i=1}^{r} \frac{|S_i|}{|S|} \times \text{info}(S_i)$$

which measures the reduction in entropy obtained if test t is applied. This quantity, called the *information gain* of t, is widely used as the basis for test selection during the construction of a decision tree.

For illustration, let us compute the gain of the test on the nominal attribute *Status* in the set of training examples of Fig. 2. In this set of examples, there are 5 and 10 examples of the classes "invite" and "don't", respectively. Therefore, the entropy of the set is computed as

$$-\frac{5}{15}\log_2\frac{5}{15} - \frac{10}{15}\log_2\frac{10}{15} = 0.918 \text{ bits.}$$

The test on *Status* partitions the set into three subsets, $S_1 = \{1, 4, 6, 7, 9, 10, 11\}$, $S_2 = \{5, 8, 12, 13\}$, and $S_3 = \{2, 3, 14, 15\}$. In S_1 there are 3 and 4 examples of the classes "invite" and "don't", respectively. Therefore, the entropy of S_1 is computed as

$$-\frac{3}{7}\log_2\frac{3}{7} - \frac{4}{7}\log_2\frac{4}{7} = 0.985 \text{ bits.}$$

All the examples of S_2 are of one class "don't", and thus the entropy of this set is 0. Finally in S_3, there are 2 examples of each of the classes "invite" and "don't". Therefore, the entropy of S_3 is

$$-\frac{2}{4}\log_2\frac{2}{4} - \frac{2}{4}\log_2\frac{2}{4} = 1 \text{ bit.}$$

Thus, the weighted average entropy after applying the test "status" becomes

$$\frac{7}{15} \times 0.985 + \frac{4}{15} \times 0 + \frac{4}{15} \times 1 = 0.726 \text{ bits,}$$

and the gain of this test is

$$0.918 - 0.726 = 0.192 \text{ bits.}$$

To select the most informative test, the above computation is repeated for all the available tests and the test with the maximum information gain is then selected.

Although the information gain test selection criterion has been experimentally shown to lead to good decision trees in many cases, it was found to be biased in favor of tests that induce finer partitions [3]. As an extreme example, consider the (meaningless) tests defined on attributes *Name* and *Social Security Number* in our

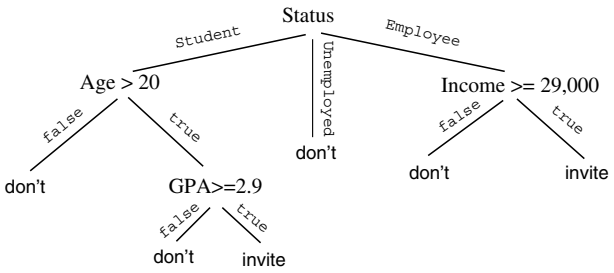

FIGURE 4 Decision tree learned from the credit card training examples using information gain as the test selection criterion.

credit card application. These tests would partition the training sample into a large number of subsets, each containing just one example. Because these subsets do not have a mixture of examples, their entropy is just 0, and so the information gain of using these trivial tests is maximal.

This bias in the gain criterion can be rectified by dividing the information gain of a test by the entropy of the test outcomes themselves, which measures the extent of splitting done by the test [3]

$$\text{split}(t) = -\sum_{i=1}^{r} \frac{|S_i|}{|S|} \log \frac{|S_i|}{|S|},$$

giving the *gain-ratio* measure

$$\text{gain-ratio}(t) = \frac{\text{gain}(t)}{\text{split}(t)}.$$

Note that $\text{split}(t)$ is higher for tests that partition the examples into a large number of small subsets. Therefore, although the tests on the *Name* and *Social Security Number* attributes have high gain, dividing by $\text{split}(t)$ in the above manner inflicts a high "penalty" on their scores, not allowing them to be selected.

Applying the basic tree-construction procedure to the set of examples of Fig. 2, using the gain-ratio as the criterion for test selection, leads to the decision tree given in Fig. 4.[2]

III. EVALUATION OF A LEARNED DECISION TREE

In evaluating a decision tree, it is necessary to distinguish between the *training error* of a decision tree, which is the percentage of *training* examples that are misclassified by the tree, and the *generalization error*, which is the probability that a randomly selected *new* case is misclassified by that tree. This latter quantity measures the tree's prediction power, and thus, it is a reasonable measure of how well the learning process was able to capture the underlying regularities of the application domain.

[2] Although the gain-ratio test selection criterion is the most widely used in decision tree learning, many other criteria are found in the literature. For experimental comparisons of various criteria see [4–6].

The generalization error is usually estimated by *cross-validation* where the available set of preclassified examples is randomly divided into *training* and *test* sets. Only the training set is used during tree construction. For each example in the test set, the class *predicted* by the learned tree is compared to the *actual* class as given in the test set. The percentage of misclassified examples is then used as an approximation of the generalization error.

Note that in the above procedure, different partitions of the examples into training and test sets may lead to different error estimates. A common practice to avoid this sensitivity to partitioning is to perform *m-fold* cross-validation. Here, the data are partitioned into m disjoint subsets of sizes that are as equal as possible. Then, one of these subsets is kept aside as a test set, and the remaining examples are used as a training set. This is repeated m times, each time using a different subset as a test set, so that each of the m subsets is used exactly once as a test set. The resulting errors are then averaged over the m runs to get the final approximation of the generalization error.

Usually, m is chosen to be about 10. A special case of m-fold cross-validation is when m is equal to the number of examples, in which case, the test set each time will have exactly one example. This is usually called the *leave-one-out* method.

IV. OVERFITTING AVOIDANCE

A major concern in decision tree construction is the risk of "overfitting" the training data. In most practical applications, the training cases are usually expected to have some level of *noise*, that is, some incorrect attribute measurements and/or class labels. In such situations, taking the training examples "too seriously" by attempting to completely eliminate the training error eventually leads to a decision tree that deviates from the actual underlying regularities of the application domain by the modeling the noise present in the training examples. Consequently, this overfitting of the training examples would hurt the generalization performance as well as the intelligibility of the learned tree.

By assuming no conflicts in the training examples (no two identical training cases are labeled with different classes), Step 1 of the tree construction procedure ensures that the tree classifies all the training cases correctly; that is, zero training error is guaranteed. However, splitting of the training examples eventually makes the number of examples available at lower nodes too small to evaluate the available test reasonably. If we insist on growing the tree all the way until pure leaves are reached, this will result in an overly complex decision tree that (although it fits the training examples well) is expected to have a high generalization error.

Experience in decision tree learning has shown that it is often the case that smaller trees that are less consistent with the training examples can outperform (in terms of generalization error) more complex trees that fit the training examples perfectly. Simplifying decision trees to avoid overfitting of the data is usually achieved by the process of *pruning*, which is the removal of those lower parts of the tree where tests are chosen based on an inadequately small number of examples. Pruning can take place during the construction of the tree or by modifying an already constructed complex tree. These approaches are sometimes called *prepruning* and *postpruning*, respectively.

In the prepruning approach, splitting of the sample is stopped as soon as we reach a conclusion that further growing of the tree is not useful. This is done by changing Step 1 of the tree-construction procedure to be as follows:

> **Step 1:** If one *stopping conditions* is satisfied, return a leaf labeled with the *most frequent class* in S.

For example, early stopping of the recursive splitting process may be forced in the following situations:

- When the information gain score for all the available tests falls below a certain threshold, and so, further error reduction is not expected using these tests.
- When all the available tests are statistically found to be irrelevant. Based on the χ^2 test, the procedure neglects any test whose irrelevance cannot be rejected with high confidence.

In practice, however, it is usually hard to design good stopping rules with perfect thresholds so that splitting is terminated at just the right time. This fact makes the other approach, postpruning, more popular. In this approach, the tree-construction procedure is allowed initially to keep growing the tree, leading eventually to an overly complex decision tree. This tree is then simplified by explicitly substituting some of its subtrees by leaves.

When plenty of training examples are available, one can divide these into two sets: one used as a training sample for the actual construction of the tree and the other used as a test sample for assessing the performance of the tree, that is, its generalization error on unseen cases. Replacement of subtrees by leaves can then be carried out such that this estimated generalization error is minimized. This can be done using the OPT-2 algorithm introduced in [7], which for any given error level finds the smallest tree whose error is within that level.

Constructing the decision tree based only on a subset of the available training examples, however, is not an attractive approach when the number of available examples is not large (which is usually the case in practice). There are two well-known approaches to get around this problem.

- **Cost-complexity pruning with cross-validation** [8]: Under this approach, a score for a given decision tree is computed as a weighted sum of its complexity (number of leaves) and its training error. Note that for any fixed weighting, lower complexity means higher training error and vice versa. The goal of this approach is to strike a good balance between the tree size and its training error, that is, to minimize the weighted sum of these quantities for some appropriate weighting. Such weighting is determined through cross-validation over the set of the training examples: the training examples are randomly divided into, say, m equally sized subsets (usually $m = 10$). The examples of $m - 1$ subsets of these are used to construct a decision tree, and the mth subset is used to estimate the generalization error of various pruned trees generated from the learned decision tree. This is repeated m times, using each of the m subsets exactly once for the evaluation of the pruned trees. The appropriate weighting is then taken as the weighting that minimizes the average error over the m runs. Then,

once this appropriate weighting is determined, a decision tree is learned from the entire training sample (all the m subsets), and by using that appropriate weighting, the pruned tree that minimizes the weighted sum of the complexity and training error is returned.

- **Reduced-error pruning** [3]: Quinlan introduced the idea of computing at each leaf an amount U such that the probability that the generalization error rate at the leaf exceeds U is very small (with respect to some preset confidence factor). The amount U is then used as an estimate of the generalization error at that leaf, and pruning is based on this estimate. That is, a subtree is replaced by a leaf if the estimated error is reduced by such an action.

V. EXTENSIONS TO THE BASIC PROCEDURE

A. Handling Various Attribute Types

In Section II.B we explained how scores are computed for the available tests in Step 2 of the tree-construction procedure to choose the most significant test for the current node in the decision tree. The actual computation of a test score depends on the type of the attribute used in that test. The computation is straightforward for a test defined on a nominal attribute having a reasonably small number of values. All that is needed is to partition the training sample according to this attribute, to compute the class frequency in each of the resulting subsets, and then to apply the gain-ratio formula directly, Score computation for other attribute types, however, may be more involved as discussed below.

1. Continuous Attributes

At a first glance, it may seem that continuous attributes are difficult to handle because arbitrary thresholds lead to an infinite number of tests to be considered. This is not true, however. For a given attribute x, suppose that we sort the training examples according to the values of x in each example. For two consecutive examples e_1 and e_2, using any threshold θ that lies between the values of x in e_1 and e_2 would obviously lead to the same partitioning of the training examples. All these thresholds are, thus, equivalent as far as the training examples are concerned because they lead to exactly the same tree. Therefore, one has to consider only one "representative" threshold from each interval (usually, the midpoint) between each two consecutive examples, and so the number of thresholds to be considered is not more than the number of training examples themselves.

In fact, the process can be made even more efficient based on results of Fayyad and Irani [9], in which they show that the threshold that gives the best gain-ratio score must be in some interval lying between two consecutive training examples labeled with different classes. This means that one can safely ignore any interval between two examples of the same class.

Another approach for handling continuous attributes is to *discretize* them. Based on the training examples, a sequence of "break points" is determined, and the continuous attribute is treated as a nominal one with each interval between the break points considered as one value for this attribute. Techniques for discretization of continuous attributes can be found in [9, 10].

2. Set-Valued Attributes

Unlike a nominal attribute, a *set-valued* attribute is one whose value is a *set* of strings (rather than a single string). For example:

The *Hobby* attribute for a person who likes soccer, volleyball, and skiing would have the value {*Soccer, Volleyball, Skiing*}.
The value of the *Color* attribute for a "white and black" dog is {*White, Black*}.

The sets of elements (hobbies and colors in the above examples) are not assumed to be taken from some small, predetermined set, because otherwise one could simply use a boolean attribute for each possible element (for example, an attribute *Soccer*, which indicates whether or not a person likes soccer) to replace the original set-valued attribute.

Tests for this kind of attribute take the form $s \in x$? where x is a set-valued attribute and s is a string. The outcome of this test is *true* for objects in which the string s appears in their set-value of x and *false* otherwise. For instance, in the above hobby example, a possible test is $Soccer \in Hobby$?, which is true if and only if *Soccer* is included in the set of hobbies of a person.

A procedure for finding the best test defined on a set-valued attribute x is given in [11]. The procedure computes the class frequency in the subsets that results from partitioning the training sample using a test of the form $s \in x$ for every string s that appears in the training examples within the set-values for the attribute x. These class frequencies are then used to evaluate the possible tests and return the best one.

3. Tree-Structured Attributes

In some domains, discrete attributes may be associated with *hierarchical* structures such as the *Shape* and *Color* hierarchies shown in Fig. 5. Such attributes are called *tree-structured* attributes. Only the values at the leaves of the hierarchy are observed in the training examples. That is, only values such as *Triangle, Square*, and *Star*, would appear in the training examples as values for *Shape*. These low-level values, however, may be too specific to concisely describe the underlying regularities in the domain. Consider, for example, a situation in which "colored polygons" constitute one class, and all other objects constitute another class. Suppose our goal is to construct from a given training sample a decision tree that discriminates between these two classes. If we are to use only those low-level values of the hierarchies shown in Fig. 5, that is, the values observable in the training examples, then the resulting tree would be overly complex and not at all comprehensible. Allowing tests in the tree that are defined using higher level categories from the given hierarchies (such as *Chromatic* and *Polygon* in this case) would greatly simplify the tree.

In general, tests defined using categories from hierarchies could by binary tests or have multiple outcomes. A binary test checks whether an object is an instance of a specific category. For example, the test $Shape = Polygon?$ is a binary test that gives *true* for objects of shape *Triangle*, *Hexagon*, or *Square* and *false* otherwise. An algorithm for finding the best binary test for a given tree-structured attribute can be found in [12].

A multiple-split test corresponds to a *cut* or a *partition* in the hierarchy. For example, for the attribute *Shape* we may have a test with three outcomes {*Polygon*,

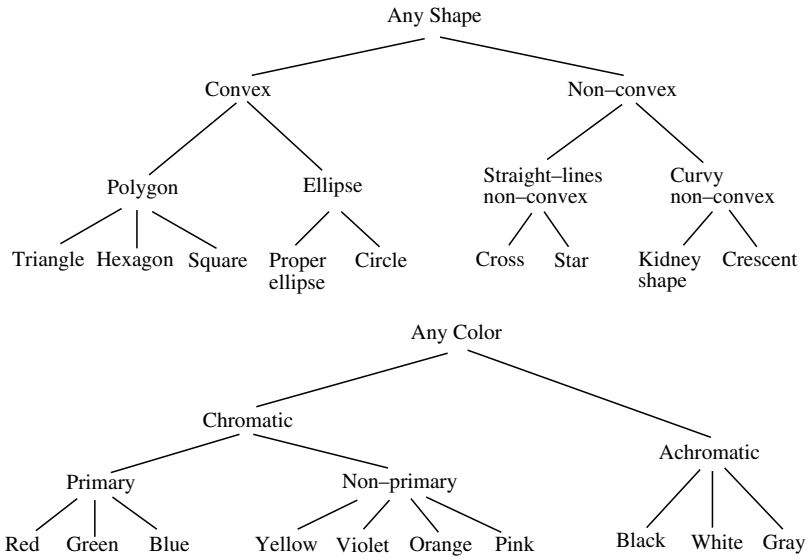

FIGURE 5 Hierarchies for *Shape* and *Color*.

Ellipse, *Nonconvex*}. In this case, objects that are *Triangle*, *Hexagon*, or *Square* give the first outcome, those that are *Proper Ellipse* or *Circle* give the second outcome, and all other objects give the third one. Note that the set of categories constituting a cut should be chosen such that any object would give exactly one outcome for the resulting test.

Finding the cut that gives the test with the best gain-ratio score for a given tree-structured attribute is a rather complicated task because the number of all possible cuts grows exponentially in the number of leaves of the associated hierarchy. An algorithm that solves this optimization problem is given in [13].

B. Incorporating More Complex Tests

In our discussion so far, we have mentioned only simple tests that are defined on a single attribute. Although the restriction to such simple tests may be justified for computational efficiency reasons, more complex tests are sometimes needed to construct decision trees with improved performance. Examples of such more complex tests are discussed here.

1. Linear Combination Tests

In some domains, the underlying regularities of the domain are best described using some linear combination of numerical attributes. A linear combination test is a test of the form

$$w_1x_1 + w_2x_2 + \cdots + w_nx_n > \theta?,$$

where x_1 through x_n are numerical attributes and $w_1, w_2, \ldots, w_n\theta$ are real-valued constants. This is a binary test which may be viewed as a hyperplane that partitions the space of objects into two halves. Under this view, a test on a single numerical

attribute is an "axis-parallel" hyperplane. In domains that involve oblique (non-axis-parallel) hyperplanes, the standard tree-construction procedure would generate large trees that perform poorly, because the procedure attempts to approximate the needed oblique hyperplanes using axis-parallel ones.

Abandoning the restriction to axis-parallel hyperplanes makes the task of finding the best test considerably harder. In [14], it is shown that this task is nondeterministic polynomial-hard; that is, it probably has no polynomial time algorithm. A heuristic is introduced in [8] in which attributes are considered one at a time. For each attribute x_i, the current hyperplane is perturbed to find w_i and θ (with all other coefficients fixed) that give the best result. This is a hill-climbing heuristic, and as such, it may get trapped at local maxima. Randomized approaches are proposed in [14] and [15] to avoid this problem.

2. Boolean Combination Tests

Boolean combination tests are binary tests defined by applying logic operators, such as *and*, *or*, and *not*, on simpler binary tests. Boolean combination tests are important in domains in which one has to check several attributes simultaneously to proceed to the final decision. For example, in medical diagnosis, a useful test may look like "Is either symptom *A* or *B* present and is the result of test *C* negative?"

In such domains, the basic tree-construction procedure can still generate a tree using only single-attribute tests that simulate tests of the above form. However, the cost will be too many splits that eventually lead to an overly complex tree in which the lower level subtrees may be based on inadequately small numbers of examples.

Given the fact that the number of all Boolean combinations is extremely large, Breiman *et al.* [8] restrict their attention to only those "conjunctive" combinations; that is, tests that look like

$$\text{test}_1 \text{ and } \text{test}_2 \text{ and } \cdots \text{ test}_d$$

where each test_i is a binary test on a single attribute (for example, $Status = Student$? or $Age > 23$?). Note that *and* is the only operator used.

Even with such a restriction to conjunctive combinations, finding the best possible test remains computationally hard. An iterative heuristic is given by Breiman *et al.* [8], in which the combined test is constructed by adding one single-attribute test to the current conjunct each time. Starting with the single-attribute test that gives the best possible score, they add each time the single-attribute test that leads to the best improvement in the score, and so on, stopping when the improvement in the score, and so on, stopping when the improvement in the score falls below a certain threshold.

A different approach is followed in the FRINGE family of algorithms [16, 17]. In this approach, a decision tree is initially constructed as usual using only single-attribute tests. Then, by examining this tree, new attributes are defined by combining tests that appear at the *fringe* of the tree. These combined attributes are then added to the description of the training examples.

For example, let us consider again the decision tree of Fig. 4 which was constructed from the examples of Fig. 2. Among the new combined attributes that would be defined from this tree by the FRINGE family of algorithms is the attribute "$Age > 20$ and $GPA > 2.9$". The set of training examples of Fig. 2 is then

modified by adding a new column for each newly defined attribute. The column of the attribute "$Age > 20$ and $GPA > 2.9$" would have the value *true* for the examples $\{1, 4, 5, 6, 10, 11, 12, 13\}$ and *false* for the rest of the examples.

A new decision tree is then constructed from the training examples with the new combined attributes included. This new tree may include tests defined on these combined attributes if such attributes score well during the tree-construction process. A new set of attributes is then defined again from this new tree, and a new tree is constructed after adding these attributes to the training examples. This process is iterated several times until the tree becomes stable or until a maximum number of iterations is reached.

Note that because new attributes may be defined in terms of previously introduced combined attributes and so on, arbitrary Boolean combinations of attributes can be generated in this approach. Unlike the work of Breiman *et al.* [8] which restricts attention to pure conjuctive tests, the FRINGE approach allows the introduction of candidate arbitrarily combined attributes, which are then filtered by the test selection criterion based on the training examples.

3. Grouping Attribute Values

In this discussion so far, we have assumed that a test defined on a nominal attribute has one branch for each value of that attribute. In some applications, however, it may be advantageous to group some of the values together in one branch. For example, consider an attribute *Day* with the 7 days of the week as its values. Imagine that, for the task at hand, the only concern is whether the day is a weekend day or not. Thus, the same conclusion is reached whether $Day = Saturday$ or $Day = Sunday$, and similarly, the conclusion for all working days is the same. In this case, having seven branches for the test on the attribute *Day* is not desirable since it imposes unnecessary fragmentation of the training examples over these many branches. The generalization performance and the intelligibility of the decision tree would improve if the tree construction procedure introduces only two branches for the attribute *Day*, one for *Saturday* and *Sunday*, and the other for the rest of the days.

Of course, if we know in advance that such grouping of the values is more suitable for the application at hand, preprocessing can be done so that the attribute *Day* is turned into a binary attribute. The discussion here, however, is meant for situations in which "background knowledge" about which values should be grouped together is not available, and the goal is to let the tree-construction procedure "discover" the most appropriate grouping that improves the final tree.

Considering *all* possible groupings of values is computationally infeasible. In [3], a hill-climbing heuristic is introduced that iteratively mergers values together in the best way that improves the gain-ratio measure. The initial value groups are just the individual values of the attribute under consideration and, at each cycle, the procedure evaluates the consequences of merging every pair of groups. This is repeated in a hill-climbing manner until no improvement in the gain-ratio score is observed. Alternatively, if so desired, this may be forced to continue until only two groups remain, leading eventually to a binary test (just like the *Day* example above).

In the GID3$*$ algorithm [18], an alternative approach is introduced which allows one branch per value for certain values, while grouping the rest of the values in one "default" branch. For example, for our *Day* attribute, it may be that *Monday* and *Friday* are of special interest (say, being the first and last working days of the

week), whereas the rest of the days are all indistinguishable for the application at hand. In this case, the test on *Day* is to have three branches, one for *Monday*, another for *Friday*, and more for the remaining 5 days. For an attribute A with values $\{a_1, a_2, \ldots, a_r\}$, a decision has to be made for each value a_i whether to have a separate branch or to let the value be part of the default branch. To handle this task, Fayyad [18] introduced a measure called $\text{Tear}(a_i)$. This quantity measures the degree to which the partition induced by the test $A = a_i?$ avoids separating training examples of the same class (see [18] for details). Because it is preferred to have examples of the same class go to the same branch as much as possible, large values for $\text{Tear}(a_i)$ make the attribute value a_i more "qualified" to have its own branch.

In GID3$*$, the gain-ratio score is first computed for each binary test of the form $A = a_i?$, for $1 \leq i \leq r$, and the value, say a_j, that scores best will have its own branch. Then, $\text{Tear}(a_i)$ for each value a_i other than a_j is compared to $\text{Tear}(a_j)$, and any value a_i with $\text{Tear}(a_i) \geq \text{Tear}(a_j)$ will also have its own branch. The rest of the values constitute the default branch.

C. Attributes with Missing Values

In real-world applications, it is usually unrealistic to expect that all the attribute values are specified for each case seen. Quite often cases with incomplete descriptions are encountered in which one or more of the values are missing. This may occur, for example, because these values could not be measured or because they were believed to be irrelevant during the data collection stage. For example, in our previous credit card application example, the *GPA* attribute may be missing in some of the cases either because such information is not available or because *GPA* becomes not relevant once we know that the person is employed.

The problem of missing values has to be dealt with both when processing the training examples during the decision tree construction and when we wish to classify a new case using a learned tree. We describe here one solution to this problem that was introduced in [3].

1. Handling Missing Values During Tree Construction

To handle missing values in the training examples, the basic tree-construction procedure is modified as follows:

- A real-valued *weight* is assigned to each training example. This weight is initially set to 1.0 for every example and may decrease later during the construction of the tree. Counting the *number* of examples in a given set S (i.e., quantities of the form $|S|$ that appear in the gain-ratio computation) is then replaced by summing the weight of all the examples in S, that is, by $\Sigma_{e \in S}\, \text{weight}(e)$.
- To evaluate the gain-ratio for a given test t, we first exclude those training examples for which the outcome of t cannot be computed due to missing attribute values. We next initially compute the gain of t using the rest of the training examples and call this the *initial-gain*. The actual gain is then computed as

$$\text{gain}(t) = F \times \text{initial} - \text{gain}(t),$$

where F is the fraction of the excluded examples (those with missing values). For computing the *split* of t, we consider t as having one more outcome that covers those examples with missing values. So, if t has n outcomes, its split information will be computed as if it divides the cases into $n+1$ subsets.

- The remaining issue now is how to partition the training sample into subsets in Step 3 of the tree-construction procedure, after the best test t has been chosen. That is, suppose t has the outcomes $\{o_1, o_2, \ldots, o_r\}$ that will partition the training examples S into the corresponding subsets $S_1, S_2, \ldots, S_r$. The question is to which subset should we send a training example for which the outcome of t cannot by specified due to missing features?

  Let S' be the subset of S with known outcomes on the test t. We partition S' into the disjoint subsets $S_1, S_2, \ldots, S_r$, where, as usual, each S_i contains the examples in S' with outcome o_i. For each outcome o_i, we then estimate the probability of that outcome as

$$p_i = \frac{\text{The sum of the weights of the examples in } S_i}{\text{The sum of the weights of the examples in } S'}.$$

  Then, for each training example $e \in S - S'$, we create r copies $e_1, e_2, \ldots, e_r$ and set the weight for each copy to be

$$\text{weight}(e_i) = \text{weight}(e) \times p_i.$$

  Each copy e_i is the included in the subset S_i with the above weight.

  Note that under this approach, the sets $S_1, S_2, \ldots, S_r$ are no longer disjoint. However, for any example with weight w in S, if we sum up the weights of all the copies of that example in all these subsets, the result would obviously be w.
- Step 1 of the tree-construction procedure is modified so that each leaf stores the "class probability" information, which is estimated by counting the number of training examples of each class that reach that leaf. This information is stored for later use when a case with missing attribute value(s) is classified as explained below.

2. Classifying a New Case with Missing Values

As usual, a new case is classified by starting at the root of the decision tree and following the branches as determined by the attribute values for the case. However, when a test t is encountered for which the outcome cannot be determined because of missing values, all the outcomes of t are considered. The classification results for all these outcomes are then combined by considering the probability of each outcome. More precisely, to classify a case e using a decision tree T, we run the following recursive procedure ClassProb(e, T) which eventually returns a vector of class probabilities:

- If T is a leaf return the class probability vector associated with the leaf.
- Let t be the test at the root, where t has the outcomes $o_1, o_2, \ldots, 0_r$, leading to subtrees $T_1, T_2, \ldots, T_r$. If the outcome of t for e is o_i, then

return ClassProt(e, T_i). Otherwise, if the outcome of t cannot be determined due to missing value(s), then return

$$\sum_{i=1}^{r} \times \text{ClassProb}(e, T_j),$$

where p_i is the estimated probability of outcome o_i as explained above.

Finally, the class probability vector returned by ClassProb(e, T) is scanned and the class with the highest probability is returned as the classification result for e.

VI. VOTING OVER MULTIPLE DECISION TREES

Significant reduction in the generalization error can be obtained by learning *multiple* trees from the training examples and then letting these trees *vote* when classifying a new case. Bagging (short for *b*ootstrap *agg*regat*ing*) [19] and boosting [20, 21] are two techniques that follow this approach, which has recently been shown to provide excellent improvement [22] in the final generalization performance. Note that this improvement is, however, bought by a significant increase in computation as well as by degradation of the intelligibility of the final classifier for human experts.

In the following, we will denote by T^k the decision tree generated in the kth iteration and by T the final composite classifier obtained by voting. For a case e, $T^k(e)$ and $T(e)$ are the classes returned by T^k and T, respectively.

A. Bagging

In this approach, in each iteration $k = 1, 2, \ldots, K$ (where K is a prespecified constant), a training set S^k is sampled (with replacement) from the original training examples S, such that $|S^k| = |S|$. From each S^k, a decision tree T^k is learned, and a final classifier T is formed by aggregating the trees $T^1, T^2, \ldots, T^K$. To classify a case e, a *vote* is given for the class $T^k(e)$, for $k = 1, 2, \ldots, K$, and $T(e)$ is then the class with the maximum number of votes.

B. Boosting

Boosting was experimentally found to be superior to bagging in terms of the generalization performance [22]. In boosting, each training example is assigned a real-valued *weight*, quantifying its influence during tree construction. A decision tree is learned from these weighted examples, and at each iteration, the weight of those examples that are misclassified in the previous iteration is increased. This means that such examples will have more influence when the next tree is constructed. Finally, after generation of several decision trees, *weighted* voting is conducted, for which the weight of each tree in the voting process is a function of its training error.

More precisely, let w_e^k denote the weight of case e at iteration k, where for every e, $w_e^1 = 1/|S|$. The following is repeated for $k = 1, 2, 3, \ldots$:

1. A tree T^k is constructed, taking into account the weights of the training example. That is, during the computation of gain-ratio, the size $|S|$ of a set S is replaced by the total weight of the examples in S; that is, $\sum_{e \in S} w_e^k$.

2. The training error ϵ^k of T^k is then measured as the sum of the weights of the training example that are misclassified by T^k.
3. Let $\beta^k = \epsilon^k/(1-\epsilon^k)$. Note that $\beta^k < 1$ because $\epsilon^k < 0.5$.
4. For each example e that is correctly classified by T^K, the weight of e is decreased to become $w_e^{k+1} = w_e^{k+1} \times \beta^k$.
5. Renormalize the weights of all the examples so that $\sum_{e \in S} w_e^{k+1} = 1$.

The preceding loop is terminated whenever one of the following cases holds:

- $k = K$.
- $\epsilon^k = 0$ (that is, T^k commits no errors on S). In this case K becomes the current value of k.
- $\epsilon^k \geq 0.5$ (that is, T^k is of a very bad quality). In this case T^k is discarded and K becomes $k-1$.

When classifying a new case e, we first find $T^k(e)$ for all the generated trees T^k. Voting is then conducted such that the vote of T^k is worth $\log(1/\beta^k)$ units, where β^k is as defined previously. Thus, unlike the bagging approach, the contribution of each tree in the voting processing is dependent on its training error.

Note that the above vote weight of $\log(1/\beta^k)$ is fixed for each tree regardless of the case being classified. An alternative is proposed by Quinlan [22] in which the vote weight depends on the leaf of the tree that is used to classify the new case: Let ℓ_e be the leaf used by T^k to classify a new case e as belonging to class $j = T^k(e)$, and let $S(\ell_e)$ denote the set of training examples at that leaf. The subset of these examples that belongs to class $j = T^k(e)$ is denoted $S(\ell_e)^j$. In the work of Quinlan [22], the vote weight of tree T^k is then set to be

$$\frac{|S| \times \Sigma_{e \in S(\ell_e)^j} w_e^k + 1}{|S| \times \Sigma_{e \in S(\ell_e)} w_e^k + 2},$$

where S is the entire set of training examples. Using this quantity was experimentally reported to yield better generalization performance compared to the fixed vote weight of $\log(1/\beta^k)$ [22].

VII. INCREMENTAL TREE CONSTRUCTION

In our discussion so far, we have assumed that all the training examples are available prior to tree construction. This kind of "batch" learning, however, is not suitable in real-time applications in which new training examples keep arriving over time. Upon the availability of new data, one may consider discarding the current tree and learning a completely new decision tree by including the newly available data in the training sample. To reduce the computational costs, however, it is more attractive to attempt to modify the current decision tree in the light of the new examples, rather than to start from scratch each time an example arrives. The objective here is to produce a tree by incorporating the new data that is the same as the tree that would have been generated by batch learning using all the available examples. This "incremental" approach has been studied by Utgoff in his ID5R algorithm [23] and then more recently in his ITI algorithm [24].

In the ITI algorithm, to permit later modification, counts used for test evaluation are stored at the nodes of the tree, and the training examples are stored at the leaves. When a new example arrives, the branches of the tree are followed according to the values in the example until a leaf is reached. If the example has the same class as the leaf, the example is simply added to the set of examples saved at that leaf. Otherwise, if the example has a different class from the leaf, the algorithm attempts to replace the leaf by a subtree generated by running the tree-construction procedure on the examples of the leaf.

After the example has been incorporated into the tree, the tree is traversed from the root recursively, ensuring that the best test possible each node (according to the test evaluation criterion) is the test actually conducted at that node. Otherwise, when a different test should replace the current one, tree revision operators (see [24] for details) are used to restructure the tree as necessary.

VIII. EXISTING IMPLEMENTATIONS

A number of packages are now available that can be used for decision tree generation. The most well-known and widely used package is Quinlan's C4.5 [3]. This package is capable of handling nominal as well as numerical attributes and is equipped with several features such as pruning, handling missing values, and grouping attribute values. It also includes routines for running cross-validation experiments and for converting learned decision trees to production rules that are more suitable for expert system development. The source code of the package (in C) is distributed with Quinlan's book, which makes it possible to experiment with variants of the original tree-construction procedures.

An alternative package is *MLC++* reported in [25]. This package implements in C++ a variety of learning algorithms including decision tree learning algorithms. It is meant to aid in the development and comparison of learning algorithms, but can also be utilized by end users. This package can be retrieved using the URL http://robotics.stanford.edu://users/ronnyk/mlc.html.

IX. PRACTICAL APPLICATIONS

In this section, we present some recent successes in applying decision tree learning to solve real-world problems. This sample is not comprehensive by any means but is only meant to demonstrate the usefulness of the decision tree learning approach in practical applications.

A. Predicting Library Book Use

In [26], decision trees are developed that predict the future use of books in a library. Forecasting book usage helps librarians to select low-usage titles and move them to relatively distant and less expensive off-site locations that use efficient compact storage techniques. For this task, it is important to adopt a book choice strategy that minimizes the expected frequency of requesting removed titles. For any choice policy, this frequency depends, of course, on the percentage of titles that have to

be removed for off-site storage (as dictated by the capacity of the main library); the higher this percentage is, the higher this frequency is expected to be. Taking the random choice policy as the benchmark, the quality of a given choice policy is evaluated using a measure called the "expected advantage over random" (EAR). This evaluation measure for choice policy is calculated given no assumptions about how many titles to store off-site.

For the Harvard College Library, where the work of Silverstein and Shieber [26] was conducted, an ideal *clairvoyant* policy can achieve an EAR of 90.51% at best, whereas a choice policy that is based on books' checkout history alone achieves an EAR of 58.86%.

In simulation experiments, book usage records for 80,000 titles during the period July 1975 to June 1984 were used to prepare examples, where each example is described using six attributes (checkout history, last use, publication date, language, country, and alphabetic prefix of the Library of Congress classification). These examples are then labeled by classes, indicating how often they are checked out in the "future," where the future information is simulated using records of the period July 1984 to June 1993. It is shown that a choice policy based on a decision tree learned from these examples gives an EAR as high as 73.12%. This is a considerable achievement given that, in the same setting, a choice that is based on books' checkout history alone gives an EAR of 58.86% and a choice policy that is based on last use alone (which is what is recommended to be best by experts) gives an EAR of 60.02%.

Furthermore, Silverstein and Shieber [26] show that the improvement is particularly striking if the percentage of titles to be moved off-site is between 20 and 40%. For example, if the Harvard College Library had implemented the last-use choice policy to choose which 20% of its collection to move to the depository in 1985, they would have had to retrieve volumes from the depository about 34,000 times per year. If they had, instead, used the decision tree constructed from examples, there would have been less than one-fifth as many retrievals—only 6,200.

B. Exploring the Relationship Between the Research Octane Number and Molecular Substructures

Figuring out what molecule information one needs to predict the research octane number (RON) is a nontrivial problem of particular interest to chemists. In the work of Blurock [27], substructure presence–absence information is used for RON prediction, not only because this is believed to give good prediction results, but also because asking directly about the presence or absence of substructures in molecules is easily interpretable by chemists, and hence, valuable intuitive information can be gained by studying the substructure–RON relationship.

The 230 hydrocarbons for which RON is known in the literature were used in the study of Blurock [27]. The attributes used are predicates about whether a particular substructure (e.g., hexane) is a member of the molecule being analyzed. Given these attributes, the goal is to predict whether RON is less than a given threshold α. Thus, there are two classes in this task: RON $< \alpha$ and RON $\geq \alpha$. Different decision trees are learned for different values of α, and these are then used to predict a range in which RON falls, that is, α_1 and α_2 such that $\alpha_1 \leq$ RON $< \alpha_2$, in which case, the predicted RON is taken as $(\alpha_1 + \alpha_2)/2$.

Of the 230 hydrocarbons, 48 were used as a test set and the rest were used for generating decision trees. The overall RON range was between 40 and 105. For this wide range, the predicted RON was within 10, 5, and 3 RON units for 65, 58, and 44% of the test cases, respectively. The greatest accuracy was achieved for a RON range of 90 to 105 units, which is also the range with the greatest concentration of cases. For this range, the predicted RON was within 10, 5, and 3 RON units for 85, 93, and 98% of the test cases, respectively. These results show that decision tree learning achieves reasonable results in this application domain given enough data points of good quality.

In addition to demonstrating the predictive power of the learned decision trees, analyzing these trees was useful in providing insight about the significance of different substructures for RON prediction. It was reported that among a total of 230 substructures, only 31 were actually needed to make the prediction. Moreover, within this subset, conclusions were drawn based on the decision trees on which substructures are important to which RON range. These findings are viewed as a contribution to better understanding of the underlying principles that determine RON of molecules.

C. Characterization of Leiomyomatous Tumors

This application was reported in [28], in which the goal was to generate hypotheses about tumor diagnosis/prognosis problems when confronted with a large number of features. For a given tumor, it is desired to know to *which* group this tumor belongs and *why*.

Traditionally, tumor characterization is made on the basis of features (such as tumor differentiation, cellularity, mitotic count, age, location, and cell types) that are difficult for a pathologist to evaluate. The task is, thus, carried out subjectively, and the quality of the results is determined by the pathologist's experience with the group of tumors concerned. To achieve a higher level of objectivity, many more quantitative measurements (related to DNA content, morphonuclear characteristics, and immunohistochemical specificities) need to be considered. Furthermore, useful information can result from interactions between several of these features that cannot be detected using traditional univariate statistical analysis.

In the work of Decaestecker *et al.* [28], decision tree learning was applied to the difficult problem of leiomyomatous (or soft muscle) tumor diagnosis. In this application, there are 31 features and 2 classes (benign leiomyomas and malignant leiomyosarcomas). The C4.5 package was run on a collection of 23 cases, each of which was preclassified independently by three pathologists. Because of the limited number of available cases, the leave-one-method was used to estimate the generalization error.

In the reported experiments, the goal of the authors was first to determine the subset of features that are most relevant to the task. Using the leave-one-out method, inclusion and exclusion of features were conducted while the authors watched the estimated generalization error, and the subset of features that gave the best results was isolated. Only those features eventually selected were given to C4.5 as the actual attributes. Decaestecker *et al*. [28] reported that the decision tree learning approach was superior (in terms of the generalization accuracy) to other

classifiers (logistic regression and neural networks) for the studied task. Furthermore, the authors note that the decision tree approach is more suitable for this task because it led to explicit logical rules that can be interpreted by human experts, which meet the exploratory nature of their job.

D. Star/Cosmic-Ray Classification in Hubble Space Telescope Images

Salzberg *et al.* [29] applied decision tree learning to the task of distinguishing between stars and cosmic rays in images collected by the Hubble Space Telescope. In addition to high accuracy, a classifier for this task must be fast due to the large number of classifications and to the need for online classification. In their experiments, a set of 2211 preclassified images was used as a training sample for decision tree construction, and a separate set of 2282 preclassified images was used to measure the generalization performance of the learned decision tree. Each of these images was described using 20 numerical features and labeled as either a star or a cosmic ray. The reported experiments show that quite compact decision trees (no more than 9 nodes) achieve generalization accuracy of over 95%. Moreover, the experiments suggest that this accuracy will get even higher when methods for eliminating background noise are employed.

X. FURTHER READINGS

Although we have attempted to summarize in this chapter the most central issues in learning decision trees from examples, many important details and extensions have not been discussed. For the interested reader, we give in this last section information on where some of the related literature can be found.

Extended explanations of the basic tree-construction procedure and many of the extensions presented in this chapter can be found in books by Breiman *et al.* [8] and Quinlan [3] devoted to decision tree learning.

Studies that compare the information gain and the gain-ratio criteria to other test selection criteria can be found in Mingers [4], Buntine and Niblett [5], and Liu and White [6]. An interesting experimental work on improving the gain-ratio criterion is presented by Dietterich *et al.* [30], which is based on the theoretical work of Kearns and Mansour [31].

Construction of *fuzzy* decision trees in domains with fuzzy attributes is studied in Umano *et al.* [32], Yuan and Shaw [33], and Ichihashi *et al.* [34].

A larger variety of applications of decision tree learning in real-world applications can be found in the literature, for example, for electrical power systems [35], air defense [36], and monitoring of manufacturing processes [37].

ACKNOWLEDGMENTS

The first author acknowledges partial support from NTT Communications Science Laboratory, Japan, through a research grant. He also thanks King Fahd University of Petroleum & Minerals for their support.

REFERENCES

1. Buchanan, B. G. and Wilkins, D. C. (Eds.). *Readings in Knowledge Acquisition and Learning.* Morgan Kauffman, San Mateo, CA, 1993.
2. Hyafil, L. and Rivest, R. L. Constructing optimal binary decision trees is NP-complete. *Inform. Process. Lett.* 5(1):15–17, 1976.
3. Quinlan, J. R. *C4.5: Programs for Machine Learning.* Morgan Kauffmann, San Mateo, CA, 1993.
4. Mingers, J. An empirical comparison of selection measures for decision tree induction. *Mach. Learn.* 3:319–342, 1989.
5. Buntine, W. and Niblett, T. A further comparison of splitting rules for decision-tree induction. *Mach. Learn.* 8:75–86, 1992.
6. Liu, W. Z. and White, A. P. The importance of attribute selection measures in decision tree induction. *Mach. Learn.* 15:25–41, 1994.
7. Almuallim, H. An efficient algorithm for optimal pruning of decision trees. *Artif. Intell.* 83:347–362, 1996.
8. Breiman, L., Friedman, J. H., Olshen, R. A., and Stone, C. J. *Classification and Regression Trees*. Wadsworth, Belmont, CA, 1984.
9. Fayyad, U. M. and Irani, K. B. Multi-interval discretization of continuous valued attributes for classification learning. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, 1993, pp. 1022–1027.
10. Kohavi, R. and Sahami, M. Error-based and entropy-based discretization of continuous features. In *Proceedings of the 2nd International Conference on Knowledge Discovery and Data Mining,* 1996, pp. 114–119.
11. Cohen, W. Learning trees and rules with set-valued features. In *Proceedings of the 13th National Conference on Artificial Intelligence (AAAI96)*, 1996, pp. 709–716.
12. Almuallim, H., Akiba, Y., and Kaneda, S. On handling tree-structured attributes in decision tree learning. In *Proceedings of the 12th International Conference on Machine Learning,* 1995, pp. 12–20.
13. Almuallim, H., Akiba, Y., and Kaneda, S. An efficient algorithm for finding optimal gain-ratio multiple-split tests on hierarchical attributes in decision tree learning. In *Proceedings of the 13th National Conference on Artificial Intelligence (AAAI96)*, 1996, pp. 703–708.
14. Heath, D., Kasif, S., and Salzberg, S. Induction of oblique decision trees. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence,* 1993, pp. 1002–1007.
15. Murthy, S., Kasif, S., Salzberg, S., and Beigel, R. OC1: Randomized induction of oblique decision trees. In *Proceedings of the 11th National Conference on Artificial Intelligence,* 1993, pp. 322–327.
16. Pagallo, G. and Haussler, D. Boolean feature discovery in empirical learning. *Mach. Learn.* 5:71–99, 1990.
17. Yang, D., Rendell, L., and Blix, G. A scheme for feature construction and a comparison of empirical methods. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence,* 1991, pp. 699–704.
18. Fayyad, U. M. Branching on attribute values in decision tree generation. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI94),* 1994, pp. 601–606.
19. Breiman, L. Bagging predictors. *Mach. Learn.* 24:123–140, 1996.
20. Fruend, Y. and Schapire, R. E. A decision theoretic generalization of online learning and an application to boosting. *Journal of Computers and System Sciences,* 55(1):119–139, 1997.
21. Fruend, Y. and Schapire, R. E. Experiments with a new boosting algorithm. In *Proceedings of the 13th International Conference on Machine Learning,* 1996, pp. 148–156.
22. Quinlan, J. R. Bagging, boosting and C4.5. In *Proceedings of the 13th National Conference on Artificial Intelligence (AAAI96),* 1996, pp. 725–730.
23. Utgoff, P. E. Incremental induction of decision trees. *Mach. Learn.* 4:161–186, 1989.
24. Utgoff, P. E. An improved algorithm for incremental induction of decision trees. In *Proceedings of the 11th International Conference on Machine Learning,* 1994, pp. 318–325.
25. Kohavi, R., Sommerfield, D., and Dougherty J. *MLC++*: A machine learning library in C++. In *Proceedings of the 8th International Conference on Tools with Artificial Intelligence,* 1996, pp. 234–245.
26. Silverstein, C. and Shieber, S. M. Predicting individual book use for off-site storage using decision trees. *Lib. Q.* 66(3):266–293, 1996.

27. Blurock, E. S. Automatic learning of chemical concepts: Research octane number and molecular substructures. *Comput. Chem.* 19(2):91–99, 1995.
28. Decaestecker, C., Remmelink, M., Salmon, I., Camby, I., Goldschmidt, D., Petein, M., Van Ham, P., and Pasteels, J. Methodological aspects of using decision trees to characterize leiomyomatous tumors. *Cytometry* 24:83–92, 1995.
29. Salzberg, S., Chandar, R., Ford, H., Murthy, S. K., and White, R. Decision trees for automated identification of cosmic-ray hits in Hubble Space Telescope images. *Publ. Astron. Soc. Pacific* 107:279–288, 1995.
30. Dietterich, T. G., Kearns, M., and Mansour, Y. Applying the weak learning framework to understand and improve C4.5. In *Proceedings of the 13th International Conference on Machine Learning*, 1996, pp. 96–104.
31. Kearns, M. and Mansour, Y. On the boosting ability of top-down decision tree learning algorithms. In *Proceedings of the 28th ACM Symposium on the Theory of Computing,* 1996.
32. Umano, M., Okamoto, H., Hatono, I., Tamura, H., Kawachi, F., Umedzu, S., and Kinoshita, J. Fuzzy decision trees by Fuzzy ID3 algorithm and its application to diagnosis systems. In *Proceedings of the IEEE International Conference on Fuzzy Systems,* 1994, pp. 2113–2118.
33. Yuan, Y. and Shaw, M. J. Induction of fuzzy decision trees. *Fuzzy Sets Syst.* 69:125–139, 1995.
34. Ichihashi, H., Shirai, T., Nagasaka, K., and Miyoshi, T. Neuro-fuzzy ID3: A method of inducing fuzzy decision trees with linear programming for maximizing entropy and an algebraic method for incremental learning. *Fuzzy Sets Syst.* 81:157–167, 1996.
35. Hatziargyriou, N. D., Papathanassiou, S. A., and Papadopoulos, M. P. Decision trees for fast security assessment of autonomous power systems with a large penetration from renewables. *IEEE Trans. Energy Conversion* 10(2):315–325, 1995.
36. Lee, Y. and Lo, C. Optimizing an air defense evaluation model using inductive learning. *Appl. Artif. Intell.* 8:645–661, 1994.
37. Du, R., Elbestawi, M. A., and Wu, S. M. Automated monitoring of manufacturing processes. Part 1. Monitoring methods. *J. Eng. Ind.* 117:121–132, 1995.