Decision Tree Induction
Based on Efficient Tree Restructuring

Paul E. Utgoff

Technical Report 95-18
March 17, 1995

Department of Computer Science
University of Massachusetts
Amherst, MA 01003

Telephone: (413) 545-4843
Net: utgoff@cs.umass.edu

Contents

1 Introduction **1**

2 Tree Revision **2**

2.1 Representation 2
2.2 Information Maintained at Each Decision Node 3
2.3 Incorporating a Training Instance 4
2.4 Recursive Tree Transposition 5
2.4.1 The Base Cases 5
2.4.2 The Recursive Case 6
2.5 When to Apply Recursive Transposition 6

3 Incremental Tree Induction **7**

3.1 Algorithm ITI 8
3.2 Inconsistent Training Instances 9
3.3 Virtual Pruning 9
3.4 Incremental Update Cost 10
3.5 Training Modes 14
3.5.1 Normal Mode 14
3.5.2 Error-Correction Mode 15
3.5.3 Lazy Mode 16

4 Direct Metric Tree Induction **16**

4.1 Algorithm DMTI 16
4.2 Direct Metrics 17
4.2.1 Expected Number of Tests 17
4.2.2 Number of Leaves 17
4.2.3 Minimum Description Length 18

4.2.4 Expected Classification Expense 18
4.2.5 Expected Misclassification Cost 18
4.3 Discussion 18

5 Leave-One-Out Cross Validation **19**

6 Software **20**

7 Summary **20**

Abstract

The ability to restructure a decision tree efficiently enables a variety of approaches to decision tree induction that would otherwise be prohibitively expensive. This report describes two such approaches, one being incremental tree induction, and the other being non-incremental tree induction using a measure of tree quality instead of test quality. The algorithm ITI for incremental tree induction includes several significant advances from its predecessor ID5R, and the algorithm DMTI for employing a direct metric of tree quality is entirely new.

1 Introduction

Decision tree induction offers a highly practical method for generalizing from instances whose class membership is known. The most common approach to inducing a decision tree is to partition the labelled instances recursively until a stopping criterion is met. The partition is defined by way of selecting a test that has a manageable set of outcomes, creating a branch for each possible outcome, passing each instance down the corresponding branch, and treating each block of the partition as a subproblem, for which a subtree is built recursively. A common stopping criterion for a block of instances is that they all be of the same class.

This non-incremental approach to inducing a decision tree is quite efficient because exactly one tree is generated, without the need to generate and evaluate explicit alternatives. Consider the process in terms of searching the space of all possible decision trees. When one determines that a particular node shall be a decision node with a specified test, one implicitly rejects all other trees that would differ in this regard. Similarly, when one determines that a particular node shall be a leaf with a specified class label, one implicitly rejects all other trees that would differ in this way. Only one tree is explicitly constructed, and the entire tree construction process implements an efficient general-to-specific search. This greedy tree construction process implements a function that maps a particular set of instances to a particular tree.

There are alternative strategies for searching tree-space, two of which are presented here. First, for incremental decision tree induction, one maps an existing tree and a new training instance to a new tree. Second, for decision tree induction using a measure of tree quality, hereafter called *direct metric tree induction*, one simply maps one tree to another. As explained below, each of these methods requires the ability to restructure an existing decision

tree efficiently. The next section presents the tree revision mechanism, and the following two sections present the two tree induction algorithms based on it.

2 Tree Revision

Both of the decision tree induction algorithms presented in this paper depend on the ability to transform one decision tree into another. For simplicity, the discussion is limited to inducing a tree for a consistent set of instances, but this limitation will be relaxed further below. For any particular set of consistent instances, there exists a multitude of decision trees that are each consistent with those instances. For example, if we were to place a test t_1 at the root, and then build the tree recursively as before, we would obtain one specific tree. However, if instead we were to place a test t_2 at the root, and then build the tree recursively as before, we would obtain a different specific tree. Each tree would be consistent with the training instances, but we may prefer one to the other because of an *a priori* bias about the relative desirability of two consistent trees. We return to the issue of bias in tree selection in Section 4. For now, we simply assume that sometimes we will have need to change the test at a decision node, and that we would like to be able to effect such a change by revising the existing tree, instead of building a new tree from the original training instances.

2.1 Representation

We assume that every possible test at a decision node has exactly two possible outcomes, which means that the decision tree is always binary. There is no loss of generality in this choice because for every non-binary tree, there are one or more binary trees that produce the identical partition of the instance space. This means that symbolic variables with more than two possible values are mapped automatically to an equivalent set of propositional variables. For example, the non-binary test *color* $\in$ { *red, green, blue* } would be converted to the three binary tests *(color = red)* $\in$ { *true, false* }, *(color = green)* $\in$ { *true, false* }, and *(color = blue)* $\in$ { *true, false* }. For numeric variables, the conversion to a binary test is done in the same manner as C4.5 (Quinlan, 1993), by finding a cutpoint and incorporating it into a threshold test, e.g. *(x < cutpoint)* $\in$ { *true, false* }.

The adoption of only binary tests brings two principal benefits. The first is that there can be no bias among tests that is due to the tests having a different number of possible outcomes. This is important because many common methods for selecting a test are biased

in this manner (White & Liu, 1994). Second, choosing a binary split at a decision node is a conservative approach to partitioning, because a block of instances is divided into at most two smaller blocks. This is beneficial because each block can be further subdivided, if necessary, by slection of a test that is best for that block.

2.2 Information Maintained at Each Decision Node

To be able to change the test that is used at a decision node, hereafter called the *installed test*, one needs to maintain at that node the information that provides the basis for evaluating the quality of each possible test for that node. For each test that is based on a specific value of a symbolic variable, e.g. $(color = blue) \in \{ true, false \}$, the frequency counts for each outcome-class combination are kept and updated as necessary. For each test that is based on a specific cutpoint of a numeric variable, it is too costly to keep a separate set of frequency counts for each one. Instead, the list of values observed in the instances at that node is maintained in sorted order by value, with each value tagged by the class of the instance in which it was observed. For each pair of adjacent values of different classes (Fayyad & Irani, 1992), the midpoint of the two values defines a possible cutpoint. The legal cutpoints and the merit of each one can be computed efficiently during a single pass over the sorted list of tagged values.

For efficiency, every set of information items kept at a decision node is maintained as an almost-balanced binary search tree (AVL). This organization provides $O(\log n)$ insert, delete and lookup. Specifically, the set of variables is maintained at a decision node as an attached AVL-tree of variables. For each variable (node) in this attached AVL-tree, the set of observed values for that variable is maintained as its own attached AVL-tree. Similarly, for each value (node) in that attached AVL-tree, the set of observed classes with frequency counts, is kept as an attached AVL-tree. This tree of trees of trees is independent of the semantics of the decision tree itself, and serves merely as an efficient scheme for tracking the information that must be maintained at the decision node. Due to this organization, neither a large number of variables, nor a large number of values, nor a large number of classes is debilitating computationally. This is all the information that is needed to evaluate all of the possible binary tests that are permitted at a decision node.

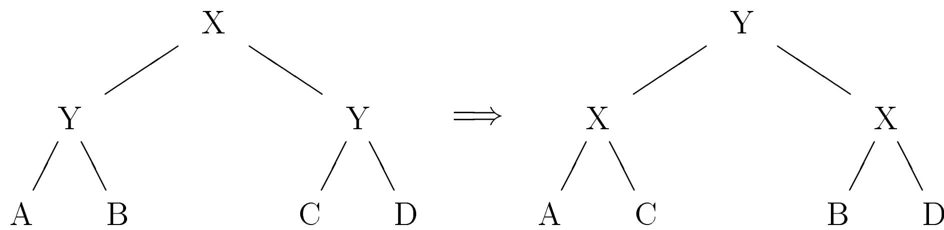

Figure 1. Tree Transposition Operator (branch left on TRUE, right on FALSE)

2.3 Incorporating a Training Instance

To be able to restructure a tree, we need the ability to incorporate a training instance at a node. As will be seen below, although this can occur during incremental training, it can also occur when a subtree is reduced, which is accomplished by removing a superfluous decision node and collapsing its two leaves into a single leaf. The need for reduction can arise during the restructuring process, and is described below.

When an instance is to be incorporated into an empty tree, the tree is replaced by a leaf node that indicates the class of the leaf, and the instance is attached to (saved at) the leaf node. Whenever an instance is to be incorporated, the branches of the tree are followed as far as possible according to the values in the instance, typically until a leaf is reached. If the instance has the same class as the leaf, the instance is simply added to the set of instances saved at the node. If the instance has a different class label from the leaf, the algorithm attempts to turn the leaf into a decision node, picking the best attribute according to the attribute-selection metric. The instances saved at the node (just converted from a leaf node to a decision node) are then incorporated by sending each one down its proper branch according to the new test. Whenever a value needed for a test is missing from an instance, the instance is simply saved at the decision node, without passing it down either branch. Thus, we assume that an instance can be 'added' to a node, and that it will work its way as far down the tree as it can, possibly sprouting branches at leaves as it moves downward through the tree.

2.4 Recursive Tree Transposition

We assume that the impetus to change the installed test at a decision node comes from the algorithm that makes use of the tree revision mechanism that we are describing. This need to change the installed test at a decision node initiates a tree revision process that is a composition of tree transpositions. Locally, one revises the tree as necessary to install the desired test at a particular node. After that has been accomplished, one needs to revisit the subtrees to ensure that each decision node has its own best test installed. This visiting of the subtrees accomplishes a more global objective of producing the most desirable tree.

Changing the installed test at a decision node, and visiting the subtrees thereafter are achieved by two recursive procedures. To cause a specific test to be installed at a particular decision node, one invokes the recursive tree transposition procedure, which is described here. Section 2.5 describes the second recursive procedure for deciding when to select and install the best test at each node in the subtrees.

2.4.1 The Base Cases

Consider one of the base cases, as illustrated in Figure 1. If each subtree of the root, whose test is to be changed, is a decision node whose installed test is already the one that we want to install at the root, then one can transpose the tree as indicated. Notice that the subtrees A, B, C, and D are simply reattached, and that they are neither inspected nor visited for any purpose. The sets of instances on which each of A, B, C, and D are based have not changed. Hence, the test information maintained at any of these decision nodes has not changed. Similarly, the set of instances corresponding to the root has also not changed (though the designation of the installed test has), which means that the test information maintained at the root also remains unchanged. Only the sets of instances at the two children of the root have changed. For example, the left subtree formerly corresponded to the instances used to build subtrees A and B, but now the left subtree corresponds to the instances used to build subtrees A and C. This raises the problem of how to maintain the test information at each child of the root.

Fortunately, the problem has an inexpensive solution that does not require rebuilding the information from the training instances. One simply recreates the test information by merging the test information of the two grandchildren. For example, for the left subtree, one would define the test information as the 'sum' of the information at nodes A and C.

For a symbolic variable, one adds the corresponding frequency counts, and for a numeric variable one merges the two sorted tagged lists of values. For each instance that may have been attached to either of the two children of the root, the instance is simply removed and reincorporated at the root node, as described above, so that it finds its way down to the proper node.

The other base cases are all special cases of the above. If one of the subtrees of the root is a leaf, instead of a decision node, transposition is accomplished somewhat differently. For example, consider the case in which the right subtree is a leaf. Then transposition is accomplished by discarding the root node, reattaching the left subtree in its place, discarding the right subtree (the leaf), and reincorporating its instances at the root. For the base case in which both subtree are leaves, one discards both subtrees, installs the desired test at the root, and then reincorporates the instances from both leaves at the root.

All of these base cases revise the tree in such a way that the tree is kept in reduced form. There are a few other base cases that arise during the transposition process when a subtree may not exist because no instances had that outcome for the installed test, but these represent temporary states that are handled in a straightforward manner.

2.4.2 The Recursive Case

The base cases presuppose that each subtree is a leaf or is a decision node whose test is the one that is to be installed at the root. The only case that this excludes is one in which at least one of the children is a decision node whose currently installed test differs from the one that is to be installed at the root. In this case, each such subtree is first transposed recursively, which will always produce one of the base cases, which is then handled as described above.

2.5 When to Apply Recursive Transposition

As described above, the recursive tree transposition operator provides the ability to restructure a given tree into another that has the designated test installed at the root. This is quite useful, but it is not enough by itself for producing the best tree because it has only caused one decision node to have the desired test installed. During the recursive transposition, it may be that the subtrees have been transposed as a by-product of bringing the desired test to the root. The installed test of each decision node in the subtrees may be there as the result of transposition rather than as the result of a deliberate choice. Again

consider the transposed tree in Figure 1. Each of the two subtrees has test X installed, but that is only because a transposition was performed from above, not because X was identified as the best choice and installed intentionally.

To ensure that every decision node has the desired test installed, according to the attribute selection metric, one needs to visit the subtrees recursively. At each decision node that requires that a different test be installed, the algorithm transposes the tree to install the best test at the node. It could become costly to check every decision node of the subtrees after a transposition. Often, a subtree is not touched during a transposition. To this end, the algorithm includes a marker in each decision node for whether the choice of the installed test is stale.

The algorithm marks as *stale* any decision node whose test information has changed. This is because changing the test information invalidates the basis on which the installed test was selected. Whenever a desired test has been identified and installed (if different) one removes its stale mark. To ensure that every decision node has the desired test installed, one proceeds recursively in the following manner: at the root, identify the desired test and install it via recursive transposition; for each subtree, if it is marked stale, then recursively identify its desired test and install it.

3 Incremental Tree Induction

We have seen in Section 2 that one can transform one tree into another instead of building a new tree from scratch. This section presents an incremental tree induction algorithm ITI (incremental tree inducer) that makes extensive use of the tree transformation mechanism.

Incremental induction is desirable for a number of reasons. Revision of existing knowledge presumably underlies many human learning processes, such as assimilation and generalization. Secondly, knowledge revision is typically much less expensive than knowledge creation. For example, upon receiving a new training instance, it is generally much less expensive to revise a decision tree than it is to build a new tree from scratch, based on the now-augmented set of accumulated training instances (Utgoff, 1989b).

Many non-incremental algorithms possess desirable properties, such as efficiency, high classification accuracy, or intelligibility, making them highly useful tools for data analysis. For someone in need of an inductive algorithm to embed in an agent or knowledge maintainer,

a non-incremental algorithm is impractical because one cannot afford to run it repeatedly. One wishes instead for both the desirable inductive properties of a proven non-incremental algorithm and the low incremental cost of an incremental algorithm.

3.1 Algorithm ITI

The algorithm described here was motivated by several design goals:

1. The average incremental cost of updating the tree should be much lower than the average cost of building a new decision tree from scratch. It is not necessary however that the sum of the incremental costs be less because we care only about the cost of being brought up to date at a particular point in time.

2. The update cost should be independent, to the extent possible, of the number of training instances on which the tree is based.

3. The tree that is produced by the incremental algorithm should depend only on the set of instances that has been incorporated into the tree, without regard to the sequence in which those instances were presented.

4. The algorithm should not be biased toward selection of a test because it has a larger set of possible outcomes than that of another test.

Additional well-accepted design goals are that the algorithm should also: accept instances described by any mix of symbolic and numeric variables (attributes), handle multiple classes, handle inconsistent training instances, handle instances with missing values, and avoid fitting noise in the instances.

The basic ITI incremental decision tree induction algorithm is based on the tree revision mechanism described above in Section 2, and thus can be stated simply. When given a training instance that is to be incorporated into the tree, pass it down the proper branches as far as possible. This includes updating the test information kept at each node through which it passes (including marking each such node stale). It also includes the process of incorporating an instance at a leaf, which may cause additional growth of the tree below that leaf. After the instance has been incorporated, visit each stale node recursively, as described above, ensuring that the desired test is installed at that node.

As usual, a test is considered best if it has the most favorable value of the attribute-selection metric. ITI uses the same gain-ratio metric that is used in C4.5. For the order of the training instances to remain immaterial, a tie for the best test must be broken deterministically.[1] With each new training instance, various frequency counts at each node traversed by the instance will change. Because the attribute-selection metric is a function of particular probabilities that are based on these frequency counts, the value of the attribute-selection metric for each test will also change, which may in turn change the assessment of which test is considered to be best at the node.

3.2 Inconsistent Training Instances

Two instances are inconsistent if they are described by the same variable values but have different class labels. When inconsistent instances occur, they will be directed to the same leaf. If one were to split every impure leaf, this would cause an infinite recursion. However, since converting the leaf to a decision node would provide no information, and this is easily detected by the gain-ratio metric, ITI keeps the node as a leaf and simply adds the instance to the set of instances retained at the leaf, making an impure leaf. This causes no trouble for classification because the vote for the class name is a function of the class distribution at the leaves. Typically, just one leaf is reached for classification, and the most frequently observed class is assigned, but when classifying an instance that is missing a value that is needed for a test, the weighted class distributions of the subtrees are combined in the same manner as is done for C4.5.

3.3 Virtual Pruning

An important component of decision tree induction is to avoid overfitting the training data, especially when the data are known to contain attribute or classification error (noise). A variety of methods have come into existence, and the question is which of them is best suited to the incremental induction problem. All of the approaches that maintain a separate pruning set are oxymoronic for incremental induction. For ITI, the most suitable is one based on the minimum description length principle (Rissanen, 1978; Quinlan & Rivest, 1989).

[1]For ITI, a tie is broken by selecting the test based on the input variable whose symbolic name has a lower hash table address. This comes about naturally from the in-order traversal of the AVL-tree of variable names.

The basic approach is to consider whether each subtree could be represented more compactly by a leaf with a default class and a list of exceptions, where each exception is an index into the list of instances and an indication of its non-default class label. For any subtree that we would want to be pruned (replaced with a leaf), we mark its root decision node as being pruned, but we do not discard anything. For incremental induction, we preserve all information so that it is possible to change whether a subtree should or should not be virtually pruned. To unprune it, we simply remove the mark that it is considered to be pruned. For all practical purposes, such as classifying instances with the tree, or inspecting the tree by printing it, a virtually pruned tree behaves and appears as though it had truly been pruned.

The virtual pruning process is accomplished by a post-order traversal of the decision tree that sets a marker in each decision node to indicate whether that decision node is to be considered pruned to a leaf. The previous status of whether the node was marked as pruned is immaterial. The procedure winds its way down to the leaves via the post-order traversal, and sets each subtree as pruned (or not) based on the minumum description length (MDL). For each leaf, the number of bits needed to encode the leaf is $1+\log(c)+x(\log(i)+\log(c-1))$, where c is the number of classes observed at the leaf, x is the number of instances at the leaf that are not of the default class, and i is the total number of instances at the leaf. This number of bits is stored in the leaf node. For each decision node, the number of bits needed to encode the subtree is $1 + \log(t) + l + r$, where t is the number of possible tests at the node, l is the MDL of the left subtree (already set), and r is the MDL of the right subtree (already set). To decide whether to mark a decision node as pruned, the MDL for the node is computed as though it were a leaf. If the virtual leaf would require fewer bits to encode, then the node is marked as pruned, and the MDL of the virtual leaf is saved at the node. Otherwise, the node is marked as not pruned, and the MDL of the subtree is saved instead.

3.4 Incremental Update Cost

Consider the cost of moving from one tree to the next, as a result of incorporating a single training instance. Hereafter, this cost of making such a single step is called the incremental update cost.

The most important goal for an incremental method is that its average incremental cost be less than the average cost of building a new tree from scratch. The incremental cost is the

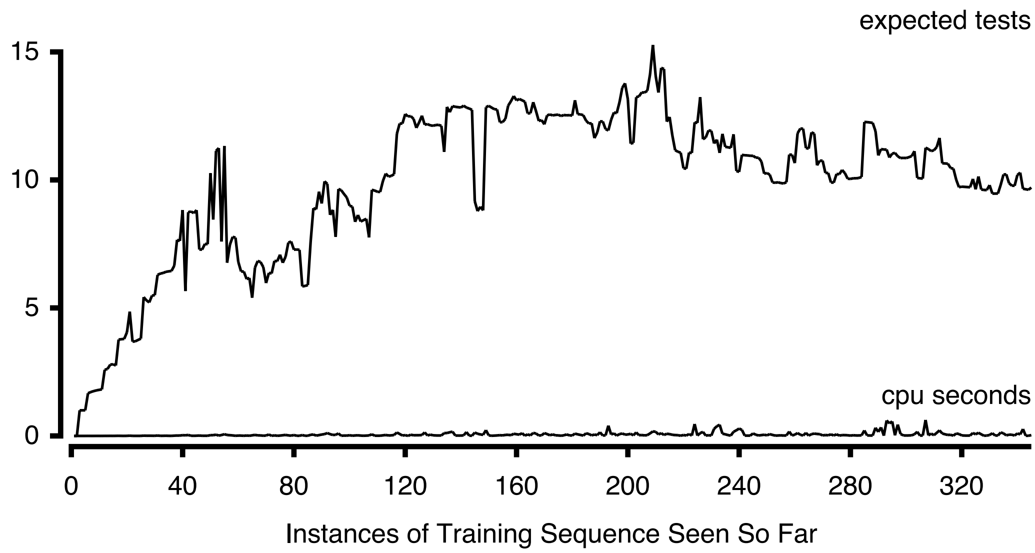

Figure 2. Liver Disorders - All Numeric Variables

cost of incorporating a new instance and revising the tree to the extent necessary so that it is the tree that corresponds to the set of training instances incorporated so far. In general, the incremental cost is proportional to the number of nodes in the tree, because the cost of revising the tree is largely the cost of all the transpositions. The size of a tree generally grows to its approximate final size early in the training, with the rest of the training serving to improve the selection of the test at each node. Of concern is whether the incremental update cost continues to grow even after the size of the tree has more or less stabilized.

When only symbolic variables are involved, one can show analytically that the incremental cost of tree revision is independent of the number of training instances seen (Utgoff, 1989b). This is because the information gleaned from an instance is kept as counting information. Seeing additional instances does not increase the amount of information that must be maintained. It only changes the actual counters that are maintained at the node. However, because ITI keeps the tree in reduced form, there is some expense that relates to the occasional redistribution of instances near the leaves of the tree.

When numeric variables are included,the incremental cost of tree revision is not independent of the number of training instances. For each numeric variable at each node, a sorted list (AVL-tree) of the values observed in the instances is maintained. More training instances means a greater cost to maintain each such list. The cost of insertions and deletions is proportional to the log of the number of distinct values observed for the variable in the training instances. The set of distinct values can be as large as the number of instances if all observed values are unique. The question then is how often and how drastically cutpoints change in practice, since this is where the greatest potential lies for requiring a large number of insertions and deletions. Numerous runs on a variety of problems suggest that cutpoints do not typically change very much. A worst-case analysis is hopelessly pessimistic, and empirical measurements are inevitably limited to a small set of tasks. One can only reason that as one sees more training instances, the various probabilities will tend to stabilize, which means that the need to revise the tree will diminish, driving down the average incremental update cost.

To illustrate that incremental update cost is nearly independent of training effort, the results of running ITI on a problem with only numeric variables is shown in Figure 2. The graph shows no noticeable evidence of growth in the incremental cost after the tree size has stabilized. This behavior is typical of all the runs on numerical data to date. The measure

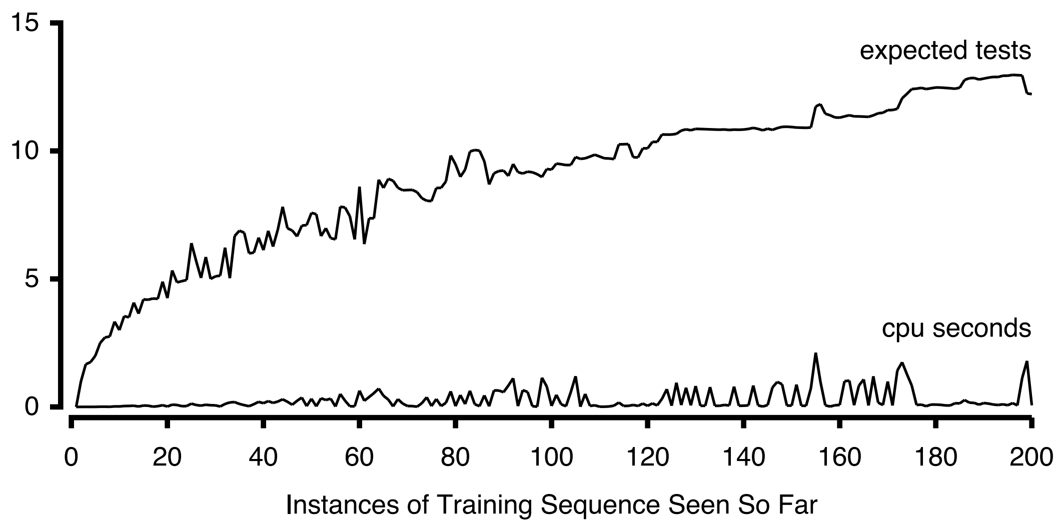

Figure 3. Audiology - All Symbolic Variables

of expected number of tests that is shown in the graph corresponds closely to tree size. The expected number of tests is the sum of the number of tests evaluated when classifying all the training instances, divided by the number of training instances. This measure can be computed inexpensively in a single traversal of the tree.

The graph also indicates a low incremental update cost at every step. The worst incremental cost is apparently much lower than the cost of rebuilding the tree from scratch at that point. This behavior of no discernible growth in the incremental cost has been observed in all runs to date, except one, which had an unusual cause that was quickly remedied. The audiology data from the UC Irvine Repository contains a variable that is the unique identifier of each instance. As training proceeded, the value set for the 'identifier' variable was growing with each instance, and to a large number of values, progressively slowing the algorithm. Removal of this unusual variable eliminated the growth, as shown in Figure 3.

It is somewhat disturbing to see how the size of the tree, as measured by the expected number of tests to classify an instance, varies so much during training. This indicates considerable sensitivity of the attribute-selection metric to the underlying probabilities computed from the various frequency counts. One observes this same phenomenon for the number of nodes in the tree.

3.5 Training Modes

A variety of training modes are possible, three of which are described here. Upon presentation of a training instance, one can decide whether or not to incorporate that training instance into the tree. Any policy for making this decision constitutes one element of a training mode. When one does elect to incorporate a new training instance into the tree, one can then decide whether or not to ensure immediately afterward that the best test is installed at each decision node. Because the process of adding an instance to a tree can be accomplished independently from revising the tree, it is possible to accept more than one instance between each occurrence of revising the tree.

3.5.1 Normal Mode

We denote as 'normal' the mode in which each instance that is presented is incorporated into the tree, and in which the tree is then immediately restructured as necessary so that every decision node has its most desired test installed. This mode always produces the same

tree that one would obtain with a non-incremental version.

3.5.2 Error-Correction Mode

A known alternative to incorporating every training instance into the tree is instead to incorporate an instance only if the existing tree would misclassify it. This mode of training is akin to the error correction procedures of statistical pattern recognition, but it was also suggested in the context of decision tree induction by Schlimmer and Fisher (1986). For a stream of training instances, one effectively discards instances for which the tree is currently correct. However, for a fixed pool of instances, ITI cycles through the pool repeatedly, removing an incorrectly classified instance from the pool and incorporating it into the decision tree, until the tree does not misclassify any instance still remaining in the pool. Although instances are examined one at a time, this training regimen departs somewhat from the notion of a linear stream of training instances.

Training in error-correction mode with the pool regimen often results in a tree that is based on fewer instances and that is built at lower cost. Such a tree is often smaller and more accurate than a tree based on all the instances presented (Utgoff, 1989a), though the reason for this phenomenon is still unknown.

When the instances are noise-free, training in error-correction mode leads to saving a set of training instances that is sufficient to cause a consistent tree to be found. Other instances are discarded upon receipt.

For a pool of instances, ITI will always build a tree and halt. This is true even when the instances are noisy, because ITI continues through the pool until every instance remaining in the pool is classified correctly. When an instance in the pool is misclassified, it is removed from the pool and added to the tree. Thus, even though the tree may not become a perfect classifier on all the training instances that it has incorporated (when there is noise or inconsistency or pruning is turned on), it does continue to select and remove currently misclassified training instances from the pool until no misclassified instances remain in the pool. This occurs either when the current tree classifies all training instances still in the pool correctly or when the pool becomes empty (all instances incorporated in the tree).

3.5.3 Lazy Mode

Most of the effort in ITI goes to ensuring, after each training instance, that the tree has the best test installed at each decision node. However, when a batch of instances is received at one time, it would be wasteful to restructure the tree after each one. Instead, one can add each instance to the tree without revising the tree (beyond the simple operations that occur when incorporating an instance). Then, after all the instances from the batch have been incorporated, a single call to the procedure for ensuring that the best test is installed at each node brings the tree to its proper form. Thus, ITI can run purely incrementally, purely nonincrementally, or anywhere in between. Since one does not generally know when the tree will be needed, one strategy for improving efficiency would be to revise the tree only if it is needed for some purpose, such as classification, and the root node has been marked as stale.

4 Direct Metric Tree Induction

We now turn to the non-incremental DMTI (direct metric tree inducer) algorithm, which also makes use of the tree revision mechanism described above in Section 2. The ability to revise a tree inexpensively makes it practical to install a particular test at a node, and then measure directly the quality of the resulting tree. This provides an attribute-selection metric that is a function of a tree (direct metric), instead of being a function of various frequency counts tallied locally at the node (indirect metric). Without the ability to revise an existing tree, it would usually be prohibitively expensive to take such an approach.

4.1 Algorithm DMTI

The DMTI algorithm is a version of the classical top-down approach, because one finds the best test to install at the root, installs it, and then solves the subproblems recursively. There are three important differences. First, one starts the process with a decision tree instead of a set of instances. Second, a test is evaluated directly by installing it, including automatic revision of the subtrees using the (indirect) gain-ratio metric, and then by evaluating the direct metric on the resulting tree. Thus, for n permissible tests at a node, DMTI evaluates n different trees. Third, it would typically be quite expensive to consider all possible tests at a node, so the set of permissible tests is limited to the best test for each input variable according to the indirect metric. For a symbolic variable, the best test is the best derived

propositional variable, and for a numeric variable, the best test is the derived propositional variable based on the best cutpoint. So, for DMTI, the set of permissible tests at a node is limited in size to the number of input variables.

4.2 Direct Metrics

We are now able to decide much more directly what bias we desire in preferring one tree to another. A direct metric can be viewed as an objective function, and the process of visiting the n trees at a node can be viewed as a kind of local optimization. On what basis shall we prefer one tree to another? With no additional information, there can be no universally correct answer to this question, but in practice we have additional information, and it is usually possible to make a good choice. For example, if one has a strong prior belief that the target concept can be modelled best by a decision tree that is a simple function of the input variables, then one would prefer a smaller consistent tree to a larger consistent tree. Similarly, it is often the case that a designer of a set of training instances will try to choose input variables that are as predictive of the class label as possible; so, a metric that is biased toward simpler functions is often appropriate. Other prior knowledge can be used to determine a corresponding bias that will be appropriate for a given tree induction task.

4.2.1 Expected Number of Tests

The direct metric *expected-number-of-tests* returns the number of tests that one would expect to evaluate in order to classify an instance, assuming that testing instances are drawn according to the same probability distribution as training instances. As mentioned in Section 2, the expected number of tests can be computed by counting the total number of tests evaluated while classifying all the training instances, and dividing by the total number of training instances. It is possible to calculate the value of this metric during a single traversal of the tree, without actually classifying any instances. All the instances that have been incorporated into the tree are attached to some node in the tree, and all node heights are known during the traversal.

4.2.2 Number of Leaves

The direct metric *leaf-count* returns the number of leaf nodes in the tree. This count can be computed during a single traversal of the tree. Though the number of leaves is related to the number of tests, it is not directly related to the expected number of tests. For example,

it is possible for a tree t_1 to have a higher leaf count and a lower expected number of tests than another tree t_2.

4.2.3 Minimum Description Length

The direct metric *minimum-description-length* returns the number of bits needed to encode the tree, using the encoding scheme described in Section 3.3. DMTI can be run with virtual pruning turned on or off, independently of the direct metric that is being used. If pruning has been enabled, then immediately after a tree is revised, the pruning procedure is invoked to mark each decision node as virtually pruned or not. For example, one can try to minimize the expected number of tests with pruning turned on. As another example, one could attempt to minimize the MDL of the tree with pruning turned off.

4.2.4 Expected Classification Expense

The direct metric *expected-classification-cost* is identical to expected-number-of-tests except that each test has a specified evaluation cost, instead of the implied uniform evaluation cost. In some applications, such as diagnosis, some tests are much more expensive than others, and the cost of producing the answer is an important factor (Tan & Schlimmer, 1990).

4.2.5 Expected Misclassification Cost

The direct metric *expected-misclassification-cost* measures the penalty that one would pay when misclassifying an instance, assuming that testing instances are drawn according to the same probability distribution as training instances. Often, tree induction algorithms embody the assumption that all classification errors incur the same cost (Pazzani, Merz, Murphy, Ali, Hume & Brunk, 1994). To be more comprehensive, one could include an explicit cost matrix that specifies the cost of labelling an instance with class X when it should have been class Y. It is possible to calculate the value of this metric in a single traversal of the tree.

4.3 Discussion

DMTI often produces dramatic improvement over ITI. Consider three examples in which DMTI used the expected-number-of-tests metric. For the classic 6-bit multiplexor, DMTI finds the optimal tree whereas ITI finds a much poorer tree because it is fooled by the indirect metric. More specifically, on a ten-fold cross validation, ITI produced a tree with an average

of 4.04 expected tests (37.6 nodes, accuracy 88.57%), and DMTI produced a tree with an average of 3.0 expected tests (15.0 nodes, accuracy 100.0%). For the monks-2 problem, which has fixed training and testing sets, ITI produced a tree of 6.25 expected tests (135 nodes, accuracy 75.46%), whereas DMTI produced a tree of 5.15 expected tests (71 nodes, 95.14% accuracy). For the hepatitis problem, using a ten-fold cross validation, ITI produced a tree with an average of 5.24 expected tests (38.0 nodes, accuracy 75.63%), and DMTI produced a tree with an average of 1.61 expected tests (16.0 nodes, accuracy 83.75%).

It is important to remember that DMTI is finding a tree according to a direct metric. Although these examples indicate improved classification accuracy, there is no known cause-and-effect relationship, and one cannot expect that this improvement will always occur. However, with respect to the direct metric, one can count on DMTI finding a tree that is no worse than that found by ITI. This is because the tree found by ITI is one of trees generated by DMTI, so another tree will be picked only if it is not worse. One can reasonably expect that DMTI will find a better tree, according to the direct metric, than ITI.

These results indicate that there is plenty of room for improvement in the standard indirect attribute selection techniques. For example, if one expects that the indirect gain-ratio metric will lead to small trees, then it is easy to see that still smaller trees are typically overlooked.

DMTI works poorly on problems in which there are many missing values. This is because an instance is left attached to a decision node when the installed test at the node cannot be evaluated for the instance. When minimizing the number of leaves or the expected number of tests, picking a test that causes a large number of instances to 'stop' at the node has the effect that fewer instances are passed down, and less tree structure is needed to separate the instances.

5 Leave-One-Out Cross Validation

For some learning methods, it is possible to perform a leave-one-out cross validation inexpensively because it is easy to modify the classifier incrementally. For example, one can change an instace-based classifier simply by adding or subtracting an instance from its instance base. This makes leave-one-out cross validation inexpensive because for each instance in the base, one removes it from the base, classifies it, and then puts it back into the base. The cross-validated accuracy is the percentage of classifications that were correct.

With efficient tree revision, leave-one-out cross validation is practical for decision tree classification (Kohavi, 1995). One first builds a tree from all the instances. Then for each instance, one subtracts it from the tree, classifies it, and adds it to the tree. This requires adding a primitive that subtracts an instances from the tree, which is straightforward; it is the inverse of adding an instance. When an instance is subtracted or added, the algorithm uses the indirect metric to identify the best test at each decision node. The cost of subtracting an instance and adding it back is dramatically less than the cost of building the tree from scratch.

6 Software

An implementation of the ITI and DMTI algorithms is available via anonymous ftp to ftp.cs.umass.edu on directory /pub/iti. The distribution includes the C source code for the two algorithms and for several additional small programs for running experiments, plotting performance graphs, and plotting decision trees. This latter program PST generates postscript that draws a tree on as many pages as necessary for the specified font and pointsize.

Everything discussed in this report has been implemented except for the ITI lazy mode, the DMTI expected-classification-expense metric, and the DMTI expected-misclassification-cost metric. Several useful operators have been implemented that have not been discussed here, such as save-tree and restore-tree.

7 Summary

This report has presented a set of fundamental tree revision operators, and shown how two decision tree induction algorithms can be built from them. The ITI algorithm performs incremental decision tree induction on symbolic or numeric variables, and handles noise and missing values. The algorithm also includes a virtual pruning mechanism that can operate in conjunction with a tree induction algorithm.

The non-incremental DMTI algorithm uses an attribute selection metric that is a function of a tree instead of a function of counting information kept at a node. This makes it possible to choose from among a set of trees based on an explicit bias. It also lends itself to studies of how well the indirect metrics do at identifying tests that lead to induction of the most preferred trees.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. IRI-9222766, by a grant from the Digital Equipment Corporation, and by a grant to Ross Quinlan from the Australian Research Council. I am indebted to Ross Quinlan for his many excellent suggestions during my stay at the University of Sydney. I thank Jeff Clouse, Neil Berkman, David Skalak, and Gunnar Blix for helpful comments. Neil Berkman implemented the leave-one-out cross validation, save-tree and restore-tree procedures. The idea of using the tree revision mechanism to implement an efficient leave-one-out cross validation was suggested independently by each of Ron Kohavi, Neil Berkman, and Mike Pazzani.

References

Fayyad, U. M., & Irani, K. B. (1992). On the handling of continuous-valued attributes in decision tree generation. *Machine Learning, 8*, 87-102.

Kohavi, R. (1995). The power of decision tables. *Proceedings of the European Conference on Machine Learning.*

Pazzani, M., Merz, C., Murphy, P., Ali, K., Hume, T., & Brunk, C. (1994). Reducing misclassification costs. *Machine Learning: Proceedings of the Eleventh International Conference* (pp. 217-225). New Brunswick, NJ: Morgan Kaufmann.

Quinlan, J. R., & Rivest, R. L. (1989). Inferring decision trees using the minimum description length principle. *Information and Computation, 80*, 227-248.

Quinlan, J. R. (1993). *C4.5: Programs for machine learning.* Morgan Kaufmann.

Rissanen, J. (1978). Modeling by shortest data description. *Automatica, 14*, 465-471.

Schlimmer, J. C., & Fisher, D. (1986). A case study of incremental concept induction. *Proceedings of the Fifth National Conference on Artificial Intelligence* (pp. 496-501). Philadelpha, PA: Morgan Kaufmann.

Tan, M., & Schlimmer, J. C. (1990). Two case studies in cost-sensitive concept acquisition. *Proceedings of the Eighth National Conference on Artificial Intelligence* (pp. 854-860). Boston, MA: Morgan Kaufmann.

Utgoff, P. E. (1989a). Improved training via incremental learning. *Proceedings of the Sixth International Workshop on Machine Learning*. Ithaca, NY: Morgan Kaufmann.

Utgoff, P. E. (1989b). Incremental induction of decision trees. *Machine Learning, 4*, 161-186.

White, A. P., & Liu, W. Z. (1994). Bias in information-based measures in decision tree induction. *Machine Learning, 15*, 321-329.