An Improved Algorithm for Incremental Induction of Decision Trees

Paul E. Utgoff

Technical Report 94-07
February 7, 1994 (updated April 25, 1994)

Department of Computer Science
University of Massachusetts
Amherst, MA 01003
utgoff@cs.umass.edu

This paper will appear in *Proceedings of the Eleventh International Conference on Machine Learning.*

Abstract

This paper presents an algorithm for incremental induction of decision trees that is able to handle both numeric and symbolic variables. In order to handle numeric variables, a new tree revision operator called 'slewing' is introduced. Finally, a non-incremental method is given for finding a decision tree based on a direct metric of a candidate tree.

Contents

1 Introduction

Incremental induction is desirable for a number of reasons. Most importantly, 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 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). Finally, the ability to revise knowledge in an efficient manner opens new possibilities for algorithms that otherwise would remain prohibitively expensive. For example, a non-incremental algorithm is presented that searches the space of decision trees more deliberately than the typical greedy approach. Moving through the space of trees would be highly impractical without the ability to revise an existing tree inexpensively.

Considerable attention has been devoted to incremental induction, including Samuel's (1959) checkers player, Winston's (1975) algorithm, Mitchell's (1978) Candidate Elimination Algorithm, Fisher's (1987) COBWEB, Laird et al's (1986) SOAR, and the explosion of recent work in reinforcement learning, e.g. Sutton's (1988) temporal-difference learning. Non-incremental induction has also received much attention, including Michalski's (1980) AQ, Quinlan's (1993) C4.5, Clark & Niblett's (1989) CN2, and Ragavan & Rendell's (1993) LFC. Many non-incremental algorithms possess desirable properties, such as efficiency, high classification accuracy, or an intelligible classifier, 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 the desirable properties of the non-incremental algorithms but at the low incremental cost of the incremental algorithms.

2 Design Goals

The algorithm that is described in the next section was motivated by a variety of design goals, which are discussed in greater detail in the next section:

1. The incremental cost of updating the tree should be much lower than the 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.

2. The update cost should be independent, or nearly so, 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 accept instances described by any mix of symbolic and numeric variables (attributes).

5. The algorithm should handle multiple classes, not just two.

6. The algorithm should not falter when given inconsistent training instances.
7. The algorithm should not favor a variable because it has a larger value set.
8. The algorithm should execute efficiently in time and space.
9. The algorithm should handle instances with missing values.
10. The algorithm should avoid overfitting noise in the instances.
11. The algorithm should have the ability to find compound tests, similar to Pagallo's FRINGE.
12. The algorithm should be capable of grouping the values of a variable.

The earlier ID5R algorithm (Utgoff, 1989b) meets only the first three of these goals. Regarding the fourth goal, ID5R accepts only symbolic variables. ID5R does not meet the remaining eight of these goals.

3 An Improved Algorithm

This section presents the algorithm ITI, which aims to meet the design goals described above. It meets the first nine, and plans for addressing the last three are described in Section 9. The basic algorithm follows from ID5R, but adds the ability to handle numeric variables, instances with missing values, and inconsistent training instances. Furthermore, in order to partition as conservatively as possible at each decision node, a test at a decision node is always binary. Each multi-valued symbolic variable is dynamically encoded as a set of boolean variables. For example, the symbolic variable 'color' with value set {'red', 'blue', 'green' } would be automatically encoded as the three boolean variables 'color=red', 'color=blue', and 'color=green'. Each numeric variable is dynamically encoded as a threshold test in a manner similar to Quinlan's C4.5. For example, the value 'x' might be automatically encoded as the boolean variable 'x<21.3'.

The basic task of ITI is to accept a new training instance, and to update the tree in response. The update process revises the tree to the extent necessary that it becomes the tree that corresponds to the set of training instances seen so far. Updating the tree involves two steps. The first, as described more fully below, is to incorporate the instance into the tree by passing it down the proper branches until it reaches the proper leaf. The second step, also described below, is to traverse the tree from root to leaves, restructuring it as necessary so that each decision node employs the best available test by which to partition the training instances.

3.1 Incorporating a Training Instance

The initial tree is the empty tree NIL. When an instance is to be incorporated into a tree, if the tree is NIL then 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 according to the values in the instance 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 Quinlan's gain-ratio metric, which is implemented exactly as described for C4.5. Another metric could be used instead, such as de Mántaras's (1991) distance metric. The instances saved at the former leaf are then incorporated by sending each one down its proper branch according to the new test. Whenever an instance is missing the value that is needed for the test at a decision node, the instance is simply saved at the decision node, without passing it down either branch.

3.2 Ensuring a Best Test at Each Decision Node

Immediately after an instance has been incorporated into the tree, the tree is traversed from the root, recursively ensuring that the best test possible at a node is the one that is tested at a node. As usual, a test is considered best if it has the most favorable value of the attribute-selection metric. For the order of the training instances to remain immaterial, a tie for the best test should be broken deterministically, such as by lexicograhic order. 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 attribute-selection-metric value for each test will also change, which may in turn change the assessment of which test is considered to be best at the node.

An efficiency in the implementation is to mark each node as stale whenever its test information changes, which occurs either when an instance is passed through a decision node during incorporation, or when a subtree of a node is revised, as explained below. Then, during the traversal from the root to ensure the best test at each node, any subtree below a node that has not been marked as stale does not need to be checked or revised, and can be skipped entirely.

3.3 Information Kept at a Decision Node

At every decision node, the ITI algorithm maintains a list of possible tests that could be used as the test at the decision node. For each binary test based on a symbolic variable, a table of frequency counts for each class and value combination is maintained. For each numeric variable, a sorted list of the values seen, each tagged with its class, is maintained with the variable, along with the best cutpoint for the variable. The best cutpoint is calculated as it is for C4.5, by considering the midpoints between each adjacent pair of values as a possible cutpoint. Following Fayyad (1992), only values between adjacent values that are tagged with differing classes need be considered. By storing the best cutpoint for the numeric variable, one is effectively encoding just the best binary test based on the numeric variable.

In the current implementation, the cost of an insertion or deletion of a numeric value is linear in the number of such items. In the next version, this cost will be reduced to the log of the number of items by using an AVL tree to maintain the sorted list of values. This balanced binary tree representation will also be used for maintaining the value list of each symbolic variable.

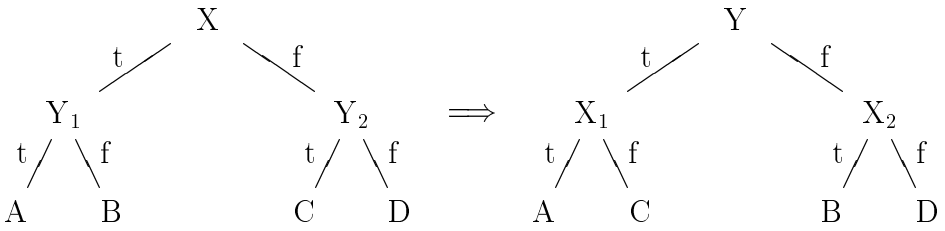

Figure 1. Tree Transposition Operator

3.4 Tree Transposition

One of the two tree revision operators employed by ITI is to transpose a tree. When a different test should replace the current one at a decision node, each non-leaf subtree is first recursively revised so that the new test occurs at the root of each subtree. Then the tree is transposed at the decision node, as illustrated in Figure 1. Whereas ID5R forces expansion of an immediate subtree that was a leaf, ITI does not. One would rather not force a decision node into existence just for the sake of complying with the preconditions for the tree transposition operator. Instead, the tree transposition operator has been enhanced to handle the cases in which either subtree is a leaf.

An important observation is that the subtrees of the children of a node (A, B, C, and D in the figure) do not need to be examined or revised during tree transposition. Each subtree corresponds to a set of instances, and because that set has not changed during the transposition, the subtree does not need to be changed in any way. Similarly, the set of instances corresponding to the root of the tree being transposed does not change, and the test information maintained at that decision node does not change. For these reasons, only the information kept at the nodes of the two children (X_1 and X_2 in the figure) needs to be revised. This is done inexpensively, without re-examining training instances, by simply combining the information kept in the each child's children nodes. For symbolic variables, one adds the frequency counts, and for numeric variables, one merges the two sorted value lists. Instances that may have been stored at each child node are simply removed and reincorporated at their parent node so that they find their way down to the proper node, typically a leaf.

3.5 Slewing a Cutpoint

The remaining tree revision operator employed by ITI is to slew a cutpoint. For a numeric variable, it may be that the variable at the node whose test is to be altered is already the basis of the boolean test at the node, but used in comparison with a different cutpoint. If the same numeric variable, but with a different cutpoint, is required for the new test, one must change (slew) the cutpoint to a different value. It is not necessary to restructure the tree through transposition, but one cannot simply change the cutpoint. Training instances were previously passed down the left or right branch of the tree according to whether the

variable's value in the instance was less than the old cutpoint. Simply changing the cutpoint could wreak havoc because some instances that were passed down a branch with the old cutpoint would have been passed down the other branch with the new cutpoint.

When slewing the cutpoint, the instances everywhere below the current node need to be checked and possibly moved to the other subtree. This is done by traversing the two subtrees with the new cutpoint in hand. Each instance that is in the wrong subtree is backed out to the current node by removing its information from the tests along the way. Then the cutpoint is changed and the instances that have been backed out are reincorporated into the correct subtree by passing them down the proper branch, reaching the proper node somewhere below.

3.6 How to Ensure a Best Test Everywhere

We have seen how to bring a desired boolean test to the root of a tree via transposition and by slewing. The general procedure for bringing a desired test, symbolic or numeric, to the root of a given subtree can now be stated. If the desired test is based on the current numeric variable, but the wrong cutpoint, then slew the cutpoint. Otherwise, if the desired test is based on the wrong variable, recursively bring the desired test to the root of each immediate nonleaf subtree and then transpose the tree. Finally, recursively ensure the best test for each of the two subtrees. As discussed above, one need not make the recursive call for a subtree that is not marked stale.

4 Incremental Training Cost

The most important goal for an incremental method is that its incremental cost be less than the cost of starting from scratch. The incremental cost is the cost of incorporating a new instance and revising the tree to the extent necessary.

In general, the incremental cost is proportional to the number of nodes in the tree. 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 training cost continues to grow even after the size of the tree has more or less stabilized, and that is the context of the discussion here.

Although it is the case that when only symbolic variables are involved one can show analytically that the incremental cost of tree revision is entirely independent of the number of training instances seen (Utgoff, 1989b), this is not the case for numeric variables. For each numeric variable at each node, a sorted list of the values observed in the instances is maintained. More training instances means a greater cost to maintain each such list. If one uses an AVL tree, the cost of insertions and deletions is small, but is nevertheless dependent on the log of the number of training instances. 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 uns on a variety of problems suggests that cutpoints do not typically change very much, but the issue has not yet been studied carefully. For practical purposes, based on the variety of problems tried to date, the incremental training cost appears to be effectively independent of training effort.

To illustrate that incremental training cost is nearly independent of training effort, the

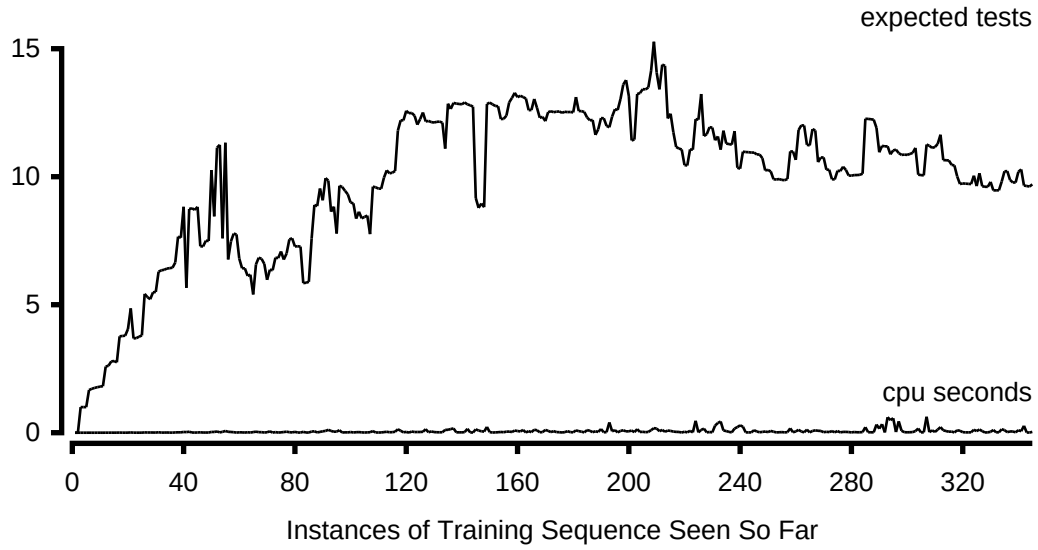

Figure 2. Liver Disorders - All Numeric Variables

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 of expected number of tests that is shown in the graph corresponds closely to tree size. The expected number of tests is 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 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.

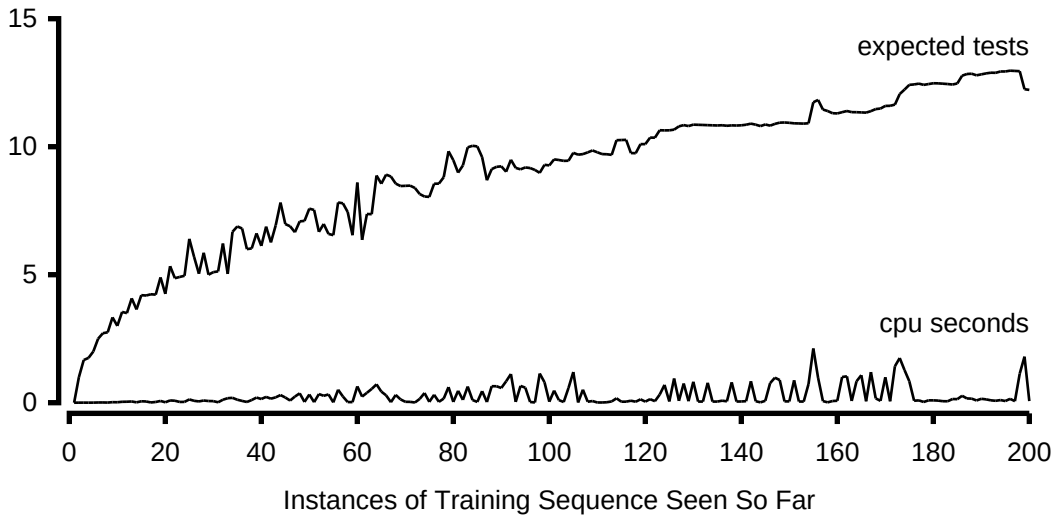

Figure 3. Audiology - All Symbolic Variables

5 Error-Correction Mode

Although the ITI algorithm saves instances in the tree, these instances are not typically re-examined to reconstruct counting information. Rather, instances are retained so that when a leaf is converted to a decision node, the counting information can be initialized immediately without loss of information. Because an instance is not counted or otherwise inspected at the leaf, but just attached, there is no extra expense in counting it later when the node is converted to a decision node. However, this is not strictly the case because, as we can see from the discussion above, slewing can cause an instance to be backed out of one subtree and added to another.

A known alternative to incorporating every training instance in 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 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 can cycle through the pool repeatedly, removing an incorrectly classified instance from the pool and incorporating it in 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 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. For example, compare the results on the Wisconsin breast cancer task using a ten-fold cross-validation with a 90/10 split for training and test data. In normal mode, ITI incorporated at average of 629 instances in building a tree that has an average accuracy of 91.7% on the test data. When run in error-correction mode on the same splits, ITI incorporated only an average of 153 instances for an average accuracy of 94.3%. This kind of behavior is typical of many tasks that have been tried with ITI.

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 stream of instances that contain noise, the error-correction mode will save each instance on which it was incorrect at the time the instance was received. This could lead to unbounded saving of instances in the tree, though one could institute a scheme for discarding older instances.

For a pool of instances, using the pool regimen, error-correction mode still works properly, in the sense that it leads ITI to build a well-formed tree and that ITI halts. ITI also handles inconsistent instances because it 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. So, even though the tree may not become a perfect classifier on all instances that it has incorporated, it does continue to train until there are no misclassified instances remaining in the pool.

6 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.

7 Direct Metrics for Attribute Selection

The ability to restructure a tree inexpensively opens the possibility of searching more deliberately for a good test to use at a decision node. One can afford to change the test at a node to see whether there is a positive effect on the tree. Such an experiment would be prohibitively expensive if one had to rebuild the subtrees each time from scratch. This idea of trying different tests at a node, and measuring the quality of a resulting tree has led to a non-incremental approach to tree induction that searches the space of trees more deliberately than typical greedy approaches. One can move from state to state by tree revision, and evaluate each tree by a direct metric of the tree, instead of the more traditional indirect metric of probabilities estimated from the training distribution at a node.

After the tree has been built (no new instances expected anytime soon), one tries each possible test at the root. To try a test means to make it the test that would be used at the node and to restructure the subtrees as before using the indirect attribute-selection metric.

For each test tried, evaluate the tree according to the direct metric. For this mode, ITI currently measures the expected number of tests of the tree, but any direct metric of choice could be applied. When all tests have been tried at the root, select the test that gave the best tree, and then find the subtrees recursively. Although each boolean test based on a symbolic variable is tried, only the boolean test based on the best cutpoint of a numeric variable is considered. This algorithm is still greedy, but the attribute selection metric is direct because it is a function of a tree instead of a set of probabilities. This approach is more expensive than the traditional indirect approach because it involves tree restructuring to determine its value.

The direct-metric mode of ITI often produces dramatic improvement over the tree that would be found using only the indirect attribute-selection metric. For example, on the classic 6-bit multiplexor, ITI finds the optimal tree (fewest expected number of tests and fewest nodes) in 0.19 cpu seconds on a DEC 3000.

For the hepatitis problem, using a ten-fold cross validation as described above, ITI found a tree using the direct metric in an average of 4.49 seconds. The tree found using the indirect metric had an average of 5.19 expected tests (37.44 nodes) with an average accuracy on the remaining data of 75.00%, whereas the tree found using the direct metric for the same data had an average of 2.82 expected tests (19.44 nodes) with an average accuracy of 83.33%. Although this illustration is for a single problem, it stands to reason that a method that enumerates and evaluates trees directly is likely to find a tree of higher quality than a method that evaluates tests and builds just one tree.

For the monks-2 problem, finding the a tree with the direct metric required 2.85 seconds. The tree built with the indirect metric on the training data had 6.0 expected tests (109 nodes) with an accuracy on the test data of 78.24%, whereas the tree built with the direct metric had 5.2 expected tests (71 nodes) with an accuracy of 95.14%.

These results indicate that there is plenty of room for improvement in attribute selection techniques. In the case for ITI, the ability to revise a tree inexpensively makes it possible to consider a metric that is a function of an entire tree, not just probablistic information tallied at a node. It would be unworkable to build these trees on the fly with a nonincremental method.

8 Lazy Restructuring

It is worth pointing out that most of the effort in ITI goes to ensuring, after each training instance, that the tree has the best test 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 the best test 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.

9 Research Agenda

Reviewing the design goals of Section 2, ITI meets the first nine, providing a large improvement over ID5R, which meets only the first three. Thus, one should consider ID5R to be obsolete. These goals were discussed above, but it is worth adding here that ITI is implemented in C, with procedures that are easy to imbed in larger systems. ITI does not yet meet the last three goals, but these are being pursued actively and are discussed here.

ITI does not yet handle noisy instances, in the sense that it does not prune subtrees that overfit the data. To retain the characteristic that the same tree will be found for the same set of instances, independent of the order in which they are presented, one does not actually want to discard anything. Instead, one wants to mark a node as virtually pruned, thereby retaining the ability to unmark it at a later time without loss of information. Thus subtrees can be marked in and out of existence without the expense of destroying or reconstructing anything. This has an effect on classification of instances, not tree construction per se. When one arrives at a virtually pruned decision node, one treats it as a leaf, returning the corresponding class label. One method for deciding whether a subtree is virtually pruned is to apply the minimum-description-length principle. This would lend itself to an incremental approach.

Pagallo (1990) showed that one can find good candidate compound tests by examining the fringe of an existing decision tree. By considering such a compound test and building a new tree, a better fit of the data is often found. Unfortunately, the FRINGE algorithm calls for building a new tree from scratch in each iteration. ITI should be able to accomplish the same result much less expensively by revising the tree. Upon identifying a compound test to add, only one traversal of the tree would be needed to add the compound variable at each decision node. In the recursion, one would inspect the instances at the leaves in order to tally the joint probability information for the compound variable. On the return up the tree, one would add the variable at each decision node and combine the counting information and value lists from the subtrees in the same manner as described above for tree transposition. Finally, a call to the routine to ensure the best test at each node would bring about any needed restructuring inexpensively.

Finally, to improve the fit of the tree to the training data, one should group those values of a variable that have similar distributions of instances. This avoids excessive partitioning, generally reducing size and improving accuracy (Quinlan, 1986). Fayyad (1991) has devised a greedy algorithm for grouping the values of a symbolic variable. Neil Berkman is currently investigating several approaches to grouping values for ITI.

10 Summary

The paper has presented the algorithm ITI for incremental induction of decision trees. The algorithm already meets most of the design goals set for it, and plans for achieving the remaining goals are in place. The ITI algorithm borrows the mechanism for tree transposition from ID5R, and adds several new capabilities, most notably the ability to handle numeric variables during incremental induction. The principal advantage of incremental learning is that it is much less expensive for serial learning than repeated running of a non-incremental algorithm. In addition, running in error-correction mode often boosts time and

space efficiency and classification accuracy. Finally, the notion of being able to revise a tree inexpensively makes searching the space of decision trees much more tractable because one can step from tree to tree more cheaply than one can generate each tree from scratch. A non-incremental method was presented that uses the inexpensive-revision capability to select a test for a node by directly measuring the quality of each resulting tree, instead of by computing a heuristic function of probabilities tallied for each test. Such an approach would be infeasible without the ability to revise trees inexpensively.

Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. IRI-9222766, by a grant from the Digitial 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 Mike Cameron-Jones, Carla Brodley, Jeff Clouse, Neil Berkman, Jamie Callan, Stephen Soderland, and Margie Connell for helpful comments.

References

Clark, P., & Niblett, T. (1989). The CN2 induction algorithm. *Machine Learning, 3*, 261-283.

de Mántaras, R. L. (1991). A distance-based attribute selection measure for decision tree induction. *Machine Learning, 6*, 81-92.

Fayyad, U. M. (1991). *On the induction of decision trees for multiple concept learning.* Doctoral dissertation, Computer Science and Engineering, University of Michigan.

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

Fisher, D. H. (1987). Knowledge acquisition via incremental conceptual clustering. *Machine Learning, 2*, 139-172.

Laird, J. E., Rosenbloom, P. S., & Newell, A. (1986). Chunking in SOAR: The anatomy of a general learning mechanism. *Machine Learning, 1*, 11-46.

Michalski, R. S., & Chilausky, R. L. (1980). Learning by being told and learning from examples: An experimental comparison of the two methods of knowledge acquisition in the context of developing an expert system for soybean disease diagnosis. *Policy Analysis and Information Systems, 4*, 125-160.

Mitchell, T. M. (1978). *Version spaces: An approach to concept learning.* Doctoral dissertation, Department of Electrical Engineering, Stanford University, Palo Alto, CA.

Pagallo, G. M. (1990). *Adaptive decision tree algorithms for learning from examples.* Doctoral dissertation, University of California at Santa Cruz.

Quinlan, J. R. (1986). Induction of decision trees. *Machine Learning, 1*, 81-106.

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

Ragavan, H., & Rendell, L. (1993). Lookahead feature construction for learning hard concepts. *Machine Learning: Proceedings of the Tenth International Conference* (pp. 252-259). Amherst, MA: Morgan Kaufmann.

Samuel, A. (1959). Some studies in machine learning using the game of Checkers. *IBM Journal of Research and Development, 3*, 211-229.

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.

Sutton, R. S. (1988). Learning to predict by the method of temporal differences. *Machine Learning, 3*, 9-44.

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.

Winston, P. H. (1975). Learning structural descriptions from examples. In Winston (Ed.), *The psychology of computer vision*. New York: McGraw-Hill.