An Incremental Method for Finding Multivariate Splits for Decision Trees[1]

PAUL E. UTGOFF (UTGOFF@CS.UMASS.EDU)
CARLA E. BRODLEY (BRODLEY@CS.UMASS.EDU)

Department of Computer and Information Science, University of Massachusetts, Amherst, MA 01003 U.S.A.

Abstract

Decision trees that are limited to testing a single variable at a node are potentially much larger than trees that allow testing multiple variables at a node. This limitation reduces the ability to express concepts succinctly, which renders many classes of concepts difficult or impossible to express. This paper presents the PT2 algorithm, which searches for a multivariate split at each node. Because a univariate test is a special case of a multivariate test, the expressive power of such decision trees is strictly increased. The algorithm is incremental, handles ordered and unordered variables, and estimates missing values.

1 Introduction

For inductive learning, decision-tree methods are attractive for three principal reasons. First, the methods find trees that generalize well to the unobserved instances, assuming that the instances are described in terms of features that are correlated with the target concept. Second, the methods are efficient, generally requiring a total amount of computation that is proportional to the number of observed training instances. Finally, the resulting decision tree provides a representation of the concept that appeals to humans because it renders the classification process self-evident.

This paper presents a new decision-tree algorithm, named PT2, that is designed to provide a richer space of possible tests at a node and to provide a uniform treatment of ordered and unordered variables. Section 2 lays out the issues that motivated the design of PT2, which is presented in Section 3. Following this, Section 4 illustrates the algorithm on three standard learning tasks. Finally, Section 5 draws conclusions about the algorithm and identifies new problems that require further research.

2 Issues for Decision-Tree Induction

This section discusses the principal issues that motivated the design of the PT2 algorithm. These issues arose from attempting to extend the perceptron tree algorithm (Utgoff, 1988) to handle ordered variables, but are central to research on decision-tree induction in general.

2.1 Splitting Criterion

A significant shortcoming of decision-tree algorithms such as ID3 (Quinlan, 1986) is that the space of legal splits at a node is impoverished. A *split* (Moret, 1982) is a partition of the instance space that results from placing a test at a decision node. Each subset of the partition corresponds uniquely to one outcome of applying the test to the instance. ID3 and its descendants only allow testing a single variable (attribute) and branching on the outcome of that test.

[1]The correct citation for this article is: Utgoff, P. E., & Brodley, C. E. (1990). An incremental method for finding multivariate splits for decision trees. *Proceedings of the Seventh International Conference on Machine Learning* (pp. 58-65). Austin, TX: Morgan Kaufmann.

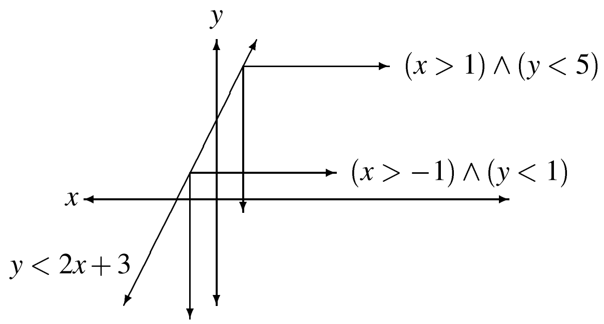


Figure 1. Limitations of Univariate Splits

In order to facilitate generalization, one would like to avoid a large number of branches at a node. This means that a variable should not have a large number of possible values. This is typically the case for nonnumeric variables. However, numeric variables, both continuous and integer-valued, are problematic due to the unlimited range of possible values. A more fundamental distinction among variables is whether a variable's values are ordered or not. Accordingly, an *ordered variable* is one whose possible values are totally ordered. This class includes continuous and integer-valued variables. An *unordered variable* is one whose values are not totally ordered. One standard technique for handling an ordered variable with a large number of possible values is to map its values onto a small set of intervals. Another technique is to partition the values of a numeric variable into two open-ended intervals: those values that are greater than a dynamically determined constant and those that are not (Breiman, Friedman, Olshen & Stone, 1984; Quinlan, 1987).

In general, one would like to allow a richer space of possible splits than those afforded by testing just one variable at a time (Pagallo & Haussler, 1988; Pagallo, 1989). For example, as shown in Figure 1, if the concept to be learned is the set of points in the half-plane $\{(x,y)|y < 2x+3\}$, then a decision tree based on univariate tests must approximate it with a disjunction of quarter-planes, e.g. $\{(x,y)|(x > -1 \wedge y < 1) \vee (x > 1 \wedge y < 5)\}$. This example illustrates the well known problem that a univariate test can only split a space with a boundary that is orthogonal to that variable's axis. This limits the space of regions in the instance space that can be represented succinctly, which can result in a large tree and poor generalization to the unobserved instances. The bias imposed by allowing only univariate tests may be inappropriate for a target concept.

2.2 Incremental Tree Revision

For learning tasks in which training instances are provided serially, one would like to be able to revise the existing decision tree, if necessary, instead of rebuilding it from scratch. One does not want a new training instance to render previous learning obsolete. In particular, one would like to be able to conduct the search for a multivariate test incrementally. Several groups, including Qing-Yun and Fu (1983), Breiman et al. (1984), Pagallo and Haussler (1989), Clark and Niblett (1989), and Chan (1989) have devised methods for searching for multivariate tests, but these methods are not incremental.

2.3 Sequential Testing and Understandability

There are two important advantages of decision-tree classifiers that one does not want to lose by permitting multivariate splits. First, the tests in a decision tree are performed sequentially by following the branches of the tree. Thus, only those variables that are required to reach a decision are evaluated. On the assumption that there is some cost in obtaining the value of a variable, it is desirable to test only those variables that are needed. Second, a decision tree provides a clear statement of a sequential decision procedure for determining the classification of an instance. A small tree with simple tests is most appealing because a human can understand it. There is a tradeoff to consider in allowing multivariate tests; simple tests may result in large trees that are difficult to understand, yet multivariate tests may result in small trees with tests that are difficult to understand.

2.4 Mixing Variable Types

One of the well-known problems with decision-tree methods is how to handle ordered variables. As discussed above, such variables are problematic because they may have many possible values, e.g. a continuous variable. For a decision tree, one requires variables with a small number of possible values, so that the number of possible branches at a node is kept small. Techniques exist for mapping an ordered variable to an unordered variable, thereby reducing the number of possible values for the ordered variable from many to as few as two. These techniques are special cases of the more general approach of partitioning an n-dimensional Euclidean space into regions, defined by inequalities. For example, a region in x-y space might be a disc defined by $\{(x,y)|x^2+y^2<3\}$. In general, one can map an n-dimensional point to a region and then treat containment within the region as a two-valued unordered variable.

How can one mix ordered and unordered variables freely? There are two standard approaches. First, as already discussed, one can map each ordered variable to an unordered variable, and then find a Boolean combination that represents the concept. The principal problem with this approach is that Boolean combinations of intervals can only define regions via boundaries that are each orthogonal to one of the coordinate axes. This renders the class of concepts that require other boundary orientations difficult to represent. Alternatively, one can map each variable, ordered or unordered, to a numeric variable and then find a numerical combination that represents the concept. This latter approach allows an accurate and succinct represention for a large class of concepts. For decision-tree induction, one would like to be able to consider boundaries in any orientation and then form regions by splitting the space in terms of these boundaries.

In order to map an unordered variable to a numeric variable, one needs to be careful not to impose an order on the values of the unordered variable. For a two-valued variable, one can simply assign 1 to one value and -1 to the other. If the variable has a range of more than two values, then each variable-value pair can be mapped to a propositional variable, which is TRUE if and only if the variable has the paritucular value in the instance (Hampson & Volper, 1986). This avoids imposing any order on the unordered values of the variable. With this mapping, one can represent concepts over unordered variables, ordered variables, or a mix of such variables. Mapping many-valued variables to two-valued variables has been observed to result in trees with higher classification accuracy (Cheng, Fayyad, Irani & Qian, 1988; Mooney, Shavlik, Towell & Gove, 1989). There are two reasons. First, the two-valued variables provide a finer-grained representation, which makes it possible to find a better decision tree. Cheng et al. point out that this approach also eliminates the irrelevant values of a variable. Second, a binary split of the instance space leaves just two subspaces, placing a larger proportion of the available training instances in each subspace (Pagallo & Haussler, 1988). Because learning in each subspace is based on a larger number of instances, the subtree found will be better determined.

2.5 Missing Values

For some instances, it may be that not all variable values are available. In such a case, one would like to estimate the missing values. This is true for both training and classification. Quinlan (1989) describes a variety of approaches for handling missing values of unordered variables. These include ignoring any instance with a missing value, filling in the most likely value, and combining the results of classification using each possible value according to the probablility of that value.

Another approach for handling a missing value is to estimate it using the sample mean, which is an unbiased estimator of the expected value. For a numeric (ordered) variable, one need only maintain a running total of the values seen and a running count of the number of values seen. This approach can also be applied to missing values of unordered variables. If, as suggested above in Section 2.4, one has mapped every nonnumeric variable to a numeric variable, then this single mechanism also estimates a missing value for a nonnumeric variable. There are two types of mappings to consider. First, if the original variable is two-valued, then it is mapped to a single numeric variable, with one value corresponding to 1 and the other to -1. For a missing value, one simply uses the sample mean. Second, if the original variable is many-valued, then it is mapped to a set of propositional (two-valued) variables, with each one treated as above. For a missing value, one uses the sample mean for each of the propositional variables. One would like to use the sample mean so that the location of the instance in the encoded Euclidian n-space is estimated as accurately as possible. Note that the sample mean of a variable at one node can differ from the sample mean of the same variable at a different node due to the different sets of observed instances on which each is based.

3 The PT2 Decision Tree Algorithm

This section presents the PT2 algorithm for inducing a decision tree from a stream of training instances. The design of the algorithm was motivated by the issues discussed above. The algorithm maintains the decision tree as a global data structure, and revises it incrementally, as necessary, in response to each received training instance. A legal decision tree is either NIL, for the empty tree, or an answer node containing a class name, or a decision node containing one or more Boolean variables, each of which defines a binary split of the instance space. One of these variables is designated as *current*, and has a branch to a decision tree for each of its two possible values. The initial tree is empty.

Table 1 specifies the PT2 algorithm, which is designed to learn single concepts over two classes. At each decision node, it searches for a binary split of the instance space. This search proceeds on two fronts. First, a possible binary split is represented as a linear threshold unit (LTU) over the original input variables. As training instances are observed at an LTU, its weights are adjusted as necessary in order to move its hyperplane in an attempt to separate the positive instances from the negatives. Second, the algorithm attempts to find an LTU at a node that is based on a reduced set of the original input variables. To this end, a set of LTUs is trained at each node in an attempt to identify those variables that can be removed without sacrificing classification accuracy at the node. The rest of this section describes this search in greater detail and discusses the basis on which one split is judged better than another.

Table 1. The PT2 Decision-Tree Update Algorithm.

1. If TREE is empty, then set TREE to new answer node containing class name of training instance, return.
2. If TREE is an answer node, then
   (a) If class name of instance is same as TREE, then return.
   (b) Set TREE to new decision node, initialize current LTU to one based on all n original variables, set $\mathbf{W} \leftarrow 0$, set list of alternate LTUs to empty, initialize other bookkeeping variables.
3. Update decision node at root of TREE, i.e.
   (a) Update the active weight vector $\mathbf{W}$ of every LTU at the node via the absolute error correction procedure. Also update the pocket vector $\mathbf{P}$, and other LTU variables as necessary. If $\mathbf{P}$ of current LTU has changed, then discard its subtrees if they exist. Use the sample mean for any missing value. Whenever $\mathbf{P}$ changes, also save the sample means that correspond to $\mathbf{P}$.
   (b) If the current LTU is not yet determined (see text), then return.
   (c) If some alternate LTU is determined and is at least as good (see Table 2) as the current LTU, then
      i. Replace the current LTU with a best such alternate LTU.
      ii. Remove from the current LTU all original variables for which all the associated weights of the encoded variables are 0.
      iii. Reset the list of alternate LTUs to those based on all $n-1$ variable combinations of those in the new current LTU. For each alternate LTU, initialize its $\mathbf{W}$ by setting each w_i to the corresponding p_i of the new current LTU, update the LTU via the absolute error correction procedure. Use sample mean for any missing value.
   (d) Descend recursively along branch below $\mathbf{P}$ of current LTU as per instance, return.

3.1 Training an LTU

A linear threshold unit can be used as a Boolean variable, specifically as a test at a decision node, because it is a predicate over its inputs. The LTU maps its variables to either the positive or negative side of its hyperplane. All original variables are encoded as necessary, as described in Section 2.4, to achieve a set of numeric variables, called the encoded variables. These are normalized dynamically so that the maximum observed value a of a variable maps to 0.5 and the minimum observed value maps to -0.5. The variables are normalized so that each has the same influence during the error computation for the absolute error correction rule. The mapping $M(a)$ is a function of the historical minimum and maximum observed for the variable. Let

$$\begin{aligned} a &= \text{value of the encoded variable} \\ hmin &= \text{historical minimum for variable} \\ hmax &= \text{historical maximum for variable} \end{aligned}$$

Table 2. Procedure to Determine Whether LTU1 is Better than LTU2.

1. If either of LTU1 or LTU2 is undetermined then return FALSE.
2. If the pocket count of LTU1 is higher than that of LTU2, then return TRUE.
3. If the pocket count of LTU1 is identical to that of LTU2, and LTU1 is based on fewer original variables, then return TRUE;
4. Return FALSE.

Then

$$M(a) = \begin{cases} -0.5 & \text{if } hmax = hmin \\ \frac{a-hmin}{hmax-hmin} - 0.5 & \text{otherwise} \end{cases}$$

As training instances are observed, each LTU is adjusted as necessary via the absolute error correction procedure (Nilsson, 1965). Combining input features to form multivariate splits is a form of constructive induction; new terms are created based on linear combinations of subsets of the original variables.

3.2 Eliminating Variables

If, at a node, the designated current LTU based on n original variables is no better than one of the alternate LTUs, each based on $n-1$ of these variables, then the set of LTUs being considered at the node is changed. A best such alternate LTU is designated as current, the others are discarded, and a new set of alternate LTUs is created, based on leaving out one variable from those of the new current LTU. This is done in order to find a test on fewer variables if possible. The idea of removing one original variable at a time is taken from CART (Breiman, Friedman, Olshen & Stone, 1984), and is based on the assumption that it is better to search for a useful projection onto fewer dimensions from a relatively well informed state than it is to search for a projection onto more dimensions from a relatively uninformed state. Note that when one original variable is removed, one or more encoded variables are removed.

3.3 Comparing Splits

How does one decide whether one split, as manifested by an LTU, is better than another, so that a best LTU can be designated current? This is accomplished with a procedure that depends on Gallant's (1986) Pocket Algorithm. For a linear threshold unit, the algorithm saves in **P** the best weight vector **W** that occurs during normal perceptron training, as measured by the longest run of consecutive correct classifications, called the *pocket count*, assuming that the observed instances are chosen in a random order. Gallant shows that the probability of an LTU based on the pocket vector **P** being optimal approaches 1 as training proceeds. The pocket vector is optimal in the sense that no other weight vector visited so far is likely to be a more accurate classifier. The Pocket Algorithm fulfills a critical role when searching for a separating hyperplane because the classification accuracy of the LTU based on **W** is unpredictable when the instances are not linearly separable (Duda & Hart, 1973).

One might simply assume that the LTU with the highest pocket count provides the best split, but there are two problems with such an assumption. The procedure shown in Table 2 was devised to decide whether one LTU provides a better split than another, and it depends on the definition of *determined* given below. To understand the need for this procedure, consider the problems that would arise from using the pocket count alone.

First, one cannot select an LTU if its pocket vector fails to discriminate among the instances. Consider a learning problem in which the training instances belong predominantly to one class. An LTU can classify a long sequence of instances correctly if it always classifies each one as the more frequently occurring class. Indeed, such a split may result in higher classification accuracy than any split that actually discriminates instances in one class from the other. However, a split that does not discriminate is no split at all. If the space of instances at a node is not split, then the space of instances at one subtree will be identical to the space at the parent. Thus, the same null split would be found at the subtree, the process would repeat, and the tree would become infinitely deep. To avoid this, an LTU with a pocket vector that is not based on having observed instances from more than one class can never be considered better than another LTU.

The second problem is that if the current LTU is not a perfect classifier, then subtrees will be needed to split the space further. It can be wasteful to try to grow subtrees before there is any strong indication that a given LTU is the best that can be found at the node. This is because the algorithms calls for discarding both subtrees of a node whenever the pocket vector **P** of the current LTU is redefined. Such activity will eventually cease because the pocket count of the current LTU increases monotonically. However, to reduce lost effort, no subtrees are allowed to grow below an LTU until enough evidence has accumulated to indicate that an improved pocket vector is not apt to be found anytime soon. The status of an LTU is considered to be *determined* if and only if the following conditions are true:

1. At least kn, $k \geq 2$, instances must have been presented to the LTU, where n is the number of original variables. This is a minimal test based on the capacity of a hyperplane (Duda & Hart, 1973); if fewer than $2n$ instances have been observed, then the LTU is known to be underdetermined.

2. As discussed above, the pocket vector for the LTU must separate at least one instance from each class.

3. Either; the pocket vector **P** is identical to the active weight vector **W**, or each and every weight in **W** has been varying within its historical minimum and maximum for a number of weight adjustments greater than the log of the number of weights in **W** (Utgoff, 1988; Utgoff, 1989).

Note that the status of the LTU can change back and forth between determined and undetermined if a new historical minimum or maximum is established infrequently. Also note that one can raise k to cause the algorithm to be more conservative about determining the status of an LTU. In the current implementation of PT2, the default value of k is 5.

4 Illustrations

This section illustrates the PT2 algorithm on three standard learning tasks. The first is the DNF concept used to illustrate the FRINGE algorithm (Pagallo, 1989). This concept is expressed succinctly by allowing multivariate tests at a node. Second is the multiplexor, both for the six-bit

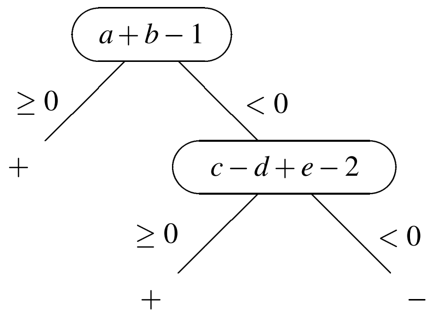


Figure 2. Tree for the DNF Task

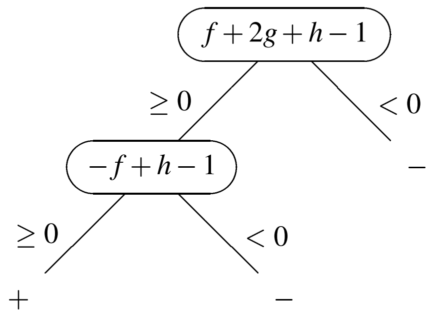


Figure 3. Tree for the Three-Bit Multiplexor Task

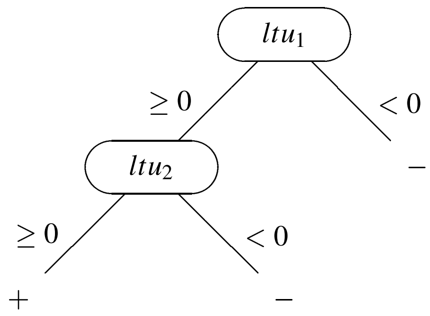
and three-bit cases. Decision-tree algorithms normally do very poorly on this problem because the best variable to test at the root is not correlated with the classification. The final illustration is Quinlan's (1987) hyperthyroid concept, in which instances are described by a mix of unordered and ordered (numeric) attributes, some with missing values.

4.1 The DNF Task

The DNF task was used previously to illustrate the ability of FRINGE to find multivariate tests at a node, though by a much different mechanism from that of PT2. The concept to be learned is the Boolean function $ab \vee c\bar{d}e$. Although there are only 32 possible instances, the algorithm always obtains its next training instance by selecting randomly from the full space of 32 instances. Figure 2 shows the tree found by PT2 after training on 775 instances. For clarity, the weights have been scaled by a constant factor. The tree consists of two tests and three leaves, and is logically equivalent to that found by FRINGE. Recall that PT2 automatically encodes TRUE as 0.5 and FALSE as -0.5. The tree found by ID3 for this task consists of eight tests and nine leaves.

4.2 The Multiplexor Task

PT2 was run on both the six-bit and the three-bit multiplexor tasks (Barto, 1985; Wilson, 1987; Quinlan, 1988). The three-bit multiplexor is of interest for the sake of illustration rather than the complexity of the task. The three-bit multiplexor is a Boolean function over three Boolean variables. One variable, here called f, is called the address bit, and it serves as a selector of one



$$\begin{aligned} ltu_1 \;=\;& 61.6ftival + 20.9t3val - 20.5onthy + 15.4tt4val + 8.5tt4 - \\ & 7.3t4uval + 6.5tbg + 6.5hpit - 6.5psyc + 5.5ref.other - 5.5thsurg + \\ & 5.0ref.svi + 4.5preg + 4.5ref.svhc - 4.5qhypo - 4.5qhyper - 3.9thsval + \\ & 3.5ref.stmw + 3.5lith - 3.5onanti - 3.5sick + 0.5goit + 0.5ths - \\ & 0.5qonthy - 0.5i131 + 6.5 \end{aligned}$$

$$\begin{aligned} ltu_2 \;=\;& -38.7thsval + 20.9t4uval + 13.1ftival - 11.4tt4val + 4.5sick + 4.5preg + \\ & 4.5i131 + 4.5qhypo + 4.5t4u + 4.5fti - 3.5ths + 3.0ref.svhc - \\ & 1.5qonthy - 1.5ref.other + 0.6age + 0.5tbg + 0.5t3val + 0.5tt4 + \\ & 0.5thsurg + 0.5onthy + 0.5onanti + 0.5lith + 0.5goit + 0.5hpit + \\ & 0.5psyc - 0.5ref.svi - 0.5tum + 0.5 \end{aligned}$$

Figure 4. Tree for the Hyperthyroid Task

of the two variables g, and h, called the data bits. The value of the function is the value of the selected data bit. The simplest tree for this concept corresponds to the expression $fg \vee \bar{f}h$. The problem is interesting because the address bit is the best test at the root, yet it is not correlated with the classification, although the data bits are. Thus, for algorithms that choose a split on such a basis, any data bit looks like a better choice than any address bit. There exists a split on all three variables that is correct for 7 of the 8 instances, and PT2 finds it, as the pocket count is highest for this split. PT2 found a correct tree after training on 480 instances, as shown in Figure 3. This PT2 tree contains two tests and three leaves, whereas the tree found by ID3 contains five tests and six leaves respectively. Note that the test at the root evaluates all three variables, which is also suboptimal. One would prefer a tree in which only necessary tests are performed.

For the six-bit multiplexor task PT2 found a correct tree of ten tests and eleven leaves after training on 3,968 instances from the full space of 64 possible instances. Total CPU time during training was 143 seconds. A characteristic of the PT2 algorithm is that the first correct tree that it finds may not be the smallest that it would find if training were to continue. It is possible that the current LTU at a node will be replaced by one that is better. Such improvements can result either from an improved pocket vector in the current LTU, or by replacing the current LTU with one that is based on fewer of the original input variables. For the six-bit multiplexor task, PT2 was left to continue training until it had seen 74,800 training instances. Although the final tree was the same size as the first correct tree, in terms of tests and leaves, some of the LTUs were replaced by LTUs based on fewer variables. The total number of variables in all ten LTUs of the first correct tree was 38, but this total in the final tree was 25. The tree that ID3 produces for this task contains 25 tests and 26 leaves.

4.3 The Hyperthyroid Task

The hyperthyroid task (Quinlan, 1987) is of interest because its instances are described in terms of both numeric and nonnumeric variables, and because there are training instances with missing values for both types of variables. The training set contains 2,800 instances, each falling into one of four classes. Because PT2 currently handles only two-class problems, the task was cast as learning the concept of "hyperthyroid". Instances labelled as "hyperthyroid" were considered to be positive, and all other instances were considered to be negative. Each instance is described by 28 variables (one variable whose value is missing for all the instances is not included). Across all 28 variables for the 2800 training instances, there are 1,756 missing values, for an average of 0.63 missing values per instance. The independent test set contains 972 instances described in terms of the same variables, with a total of 536 missing values, for an average of 0.55 missing values per instance.

Due to the lopsided distribution of the training instances, with 97.79% being negative, a variation of random sampling was used for selecting the next instance during training. Instead of sampling randomly from the entire population of training instances, the instances from the two classes were kept as two separate populations. An instance was selected randomly within a population but the choice of population alternated each time. This training strategy supplies an even distribution of instances at the root, but the effect diminishes at the subtrees because the tree is attempting to separate the positives from the negatives. PT2 does not depend on this training strategy. The rationale is simply that with a lopsided distribution, most instances will come from one class and therefore be uninformative.

Figure 4 shows the tree found by PT2, having trained on a total of 372,400 instances for approximately one clock week. The tree classifies 99.46% of the training data correctly and 99.07% of the test data correctly. In the figure, the notation $a.v$ indicates a propositional variable corresponding to value v of the original input variable a. Thus, ltu_1 contains 25 weights based on 22 original variables, and ltu_2 contains 27 weights based on 25 original variables.

It is unclear whether 99.07% correct classification on the test data is good, especially given that simply always guessing negative yields 98.25% correctly classified. Precise classification accuracies of previous solutions related to this task (Quinlan, 1987; Chan, 1989) were not published. It is worth noting that both C4 and PT2 chose $ftival$ as the most important variable at the root. Also note that for PT2, one can rank the relative importance of the variables by comparing the magnitudes of their respective weights. It is meaningful to compare weights because PT2 normalizes all the variable values.

5 Discussion

This section identifies strengths and weaknesses of the PT2 algorithm, and indicates the directions in which the research is proceeding.

5.1 Strengths

There are four strengths of the algorithm. First, it is incremental, which means that additional training instances received at a later time will not obviate previous learning by requiring that a new tree be built from scratch. This is highly desirable for learning algorithms that are imbedded in systems that learn from experience.

Second, the algorithm finds a multivariate test at a node by training a linear threshold unit. This allows defining regions in the instance space that have boundaries in any orientation. This richer space of splits, compared to allowing only univariate splits, enables the program to find more compact trees. The algorithm attempts to find a linear split that is optimal in terms of classification, but that is based on a small subset of the original variables.

Third, the algorithm treats all variable types uniformly by encoding their ranges numerically. The encoded values are normalized by mapping each range onto the interval [-0.5,0.5]. Encoding and mapping are determined dynamically. This encoding allows one to learn concepts over instances that are described by a mix of ordered and unordered variables, rather than one or the other.

Finally, the algorithm estimates missing values via sample mean, which is an unbiased estimator of the expected value. This is well defined for all variable types because they are all encoded numerically.

5.2 Weaknesses

There are four clear weaknesses in the algorithm, which are being addressed as this work continues. First, the mechanism for dropping variables from a LTU is too costly. The algorithm may train $O(n^2)$ LTUs at a node while searching for a good split. This is worse than the predecessor perceptron tree algorithm (Utgoff, 1988), which trained just one LTU at each node.

Second, there is no guarantee that PT2 will find a tree that satisfies the sequential testing and understandability goals outlined in Section 2.3. Indeed, the tree found for the hyperthyroid task is poor in this regard.

Third, the algorithm does not take advantage of pruning techniques that have been designed to prevent overfitting the training data (Breiman, Friedman, Olshen & Stone, 1984; Mingers, 1989). Furthermore, the algorithm is also subject to underfitting error. If too few unique instances are delivered to a node, with respect to the dimensionality of the instance descriptions, then an LTU will be underdetermined (Duda & Hart, 1973).

Finally, the algorithm is limited to discriminating just two classes.

5.3 Near-Term Extensions

The algorithm is being extended in three ways. First, a better method for discarding variables at a node is being investigated. Given that the encoded variables are normalized, one can identify the most important variables by the magnitudes of the weights. One could base the LTU for the split on just those variables that have sufficiently large magnitudes, compared to the largest magnitude among the weights.

Second, the ability to discriminate more than two classes is being added. Several known methods are being considered, but the most promising is to separate the instances into two superclasses at each node (Breiman, Friedman, Olshen & Stone, 1984). Such a scheme finds "strategic" splits, in the sense that splits near the top of the tree group together those classes that are similiar, while splits near the leaves isolate single classes.

Third, a pruning mechanism is being added that will attempt to prevent overfitting and underfitting problems.

5.4 Longer-Term Extension

A longer term problem that may be addressed is how to detect when the piece-wise linear classification strategy is performing poorly and alter it automatically. For example, if the members of the target concept define a hyperregion with one or more curved boundaries, then a potentially infinite number of hyperplanes will be needed to approximate the hyperregion. This would lead to a very large tree, yet if the space were to be split with hyperspheres or hyperellipses then the tree would be smaller, which would facilitate learning.

Acknowledgments

This material is based upon work supported by the National Aeronautics and Space Administration under Grant No. NCC 2-658, and by the Office of Naval Research through a University Research Initiative Program, under contract number N00014-86-K-0764. Sharad Saxena, Jamie Callan, Tom Fawcett, David Haines, David Lewis, Jeff Clouse, Margie Connell, and Pat Langley provided helpful comments.

References

Barto, A. G. (1985). Learning by statistical cooperation of self-interested neuron-like computing elements. *Human Neurobiology, 4*, 229-256.

Breiman, L., Friedman, J. H., Olshen, R. A., & Stone, C. J. (1984). *Classification and regression trees*. Belmont, CA: Wadsworth International Group.

Chan, P. K. (1989). Inductive learning with BCT. *Proceedings of the Sixth International Workshop on Machine Learning* (pp. 104-108). Ithaca, NY: Morgan Kaufmann.

Cheng, J., Fayyad, U. M., Irani, K. B., & Qian, Z. (1988). Improved decision trees: A generalized version of ID3. *Proceedings of the Fifth International Conference on Machine Learning* (pp. 100-106). Ann Arbor, MI: Morgan Kaufman.

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

Duda, R. O., & Hart, P. E. (1973). *Pattern classification and scene analysis*. New York: Wiley & Sons.

Gallant, S. I. (1986). Optimal linear discriminants. *Proceedings of the International Conference on Pattern Recognition* (pp. 849-852). Paris, France: IEEE Computer Society Press.

Hampson, S. E., & Volper, D. J. (1986). Linear function neurons: Structure and training. *Biological Cybernetics, 53*, 203-217.

Mingers, J. (1989). An empirical comparison of pruning methods for decision tree induction. *Machine Learning, 4*, 227-243.

Mooney, R., Shavlik, J., Towell, G., & Gove, A. (1989). An experimental comparison of symbolic and connectionist learning algorithms. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence* (pp. 775-780). Detroit, Michigan: Morgan Kaufmann.

Moret, B. M. E. (1982). Decision trees and diagrams. *Computing Surveys, 14*, 593-623.

Nilsson, N. J. (1965). *Learning machines*. New York: McGraw-Hill.

Pagallo, G., & Haussler, D. (1988). *Feature discovery in empirical learning*, (unpublished), Santa Cruz, CA: University of California, Department of Computer and Information Science.

Pagallo, G. (1989). Learning DNF by decision trees. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence* (pp. 639-644). Detroit, Michigan: Morgan Kaufmann.

Qing-Yun, S., & Fu, K. S. (1983). A method for the design of binary tree classifiers. *Pattern Recognition, 16*, 593-603.

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

Quinlan, J. R. (1987). Decision trees as probabilistic classifiers. *Proceedings of the Fourth International Workshop on Machine Learning* (pp. 31-37). Irvine, CA: Morgan Kaufmann.

Quinlan, J. R. (1988). An empirical comparison of genetic and decision-tree classifiers. *Proceedings of the Fifth International Conference on Machine Learning* (pp. 135-141). Ann Arbor, MI: Morgan Kaufman.

Quinlan, J. R. (1989). Unknown attribute values in induction. *Proceedings of the Sixth International Workshop on Machine Learning* (pp. 164-168). Ithaca, NY: Morgan Kaufmann.

Utgoff, P. E. (1988). Perceptron trees: A case study in hybrid concept representations. *Proceedings of the Seventh National Conference on Artificial Intelligence* (pp. 601-606). Saint Paul, MN: Morgan Kaufmann.

Utgoff, P. E. (1989). Perceptron trees: A case study in hybrid concept representations. *Connection Science, 1*, 377-391.

Wilson, S. W. (1987). Classifier systems and the animat problem. *Machine Learning, 2*, 199-228.