Boosting Lazy Decision Trees

Xiaoli Zhang Fern XZ@ECN.PURDUE.EDU
Carla E. Brodley BRODLEY@ECN.PURDUE.EDU
School of Electrical and Computer Engineering, Purdue University, West Lafayette, IN 47907

Abstract

This paper explores the problem of how to construct lazy decision tree ensembles. We present and empirically evaluate a relevance-based boosting-style algorithm that builds a lazy decision tree ensemble customized for each test instance. From the experimental results, we conclude that our boosting-style algorithm significantly improves the performance of the base learner. An empirical comparison to boosted regular decision trees shows that ensembles of lazy decision trees achieve comparable accuracy and better comprehensibility. We also introduce a novel distance-based pruning strategy for the lazy decision tree algorithm to address the problem of over-fitting. Our experiments show that the pruning strategy improves the accuracy and comprehensibility of both single lazy decision trees and boosted ensembles.

1. Introduction

Boosting (Freund & Schapire, 1997) has been shown to be extremely effective at increasing the performance of various learning algorithms, among which decision tree algorithms have been extensively investigated (Quinlan, 1996; Freund & Schapire, 1996; Bauer & Kohavi, 1999; Dietterich, 2000). Interestingly, the lazy decision tree (LazyDT) algorithm (Friedman et al., 1996), a lazy learning variation of the C4.5 decision tree algorithm (Quinlan, 1993), has not been explored in the context of boosting research. Previous work (Margineantu & Dietterich, 2002) has used the lazy option tree algorithm, a variant of LazyDT, in a bagging approach (Breiman, 1996) to perform probability estimation. However, to our knowledge, no boosting-style algorithm has been proposed. A possible reason is that, as explained in Section 2, an effective and straightforward method of applying existing boosting algorithms to LazyDT is not readily apparent.

We chose to explore the topic of boosting lazy decision trees because LazyDT has some strengths in comparison with regular decision tree algorithms (Friedman et al., 1996). First, the decision paths built by LazyDT are often shorter and therefore more comprehensible than paths of regular decision trees. Second, it is well known that given limited training data, regular decision tree algorithms can suffer from the data fragmentation problem (Pagallo & Haussler, 1990). Regular decision tree algorithms select a test for the root of each sub-tree based on the *average improvement* of the test selection criterion (such as entropy). Because the choice is based on average improvement, a particular child branch of the test may see a decrease in the value of the criterion or remain the same. For instances taking such a branch, the test may be detrimental since it fragments the data unnecessarily. This may cause the resulting path to be less accurate because the remaining tests are selected based on fewer training instances. In contrast, LazyDT constructs a customized "tree" for each test instance, which consists of only a single path from the root to a labeled leaf node. Given a test instance, LazyDT selects a test by focusing on the branch that will be taken by the test instance. By doing so it avoids unnecessary data fragmentation and may produce a more accurate classifier for the specific instance. Given the above strengths of LazyDT, we are interested in further improving LazyDT by applying boosting-style ensemble construction techniques.

In this paper, we propose a relevance-based boosting-style algorithm[1] to build a customized LazyDT ensemble for each test instance. We show that our method significantly improves the performance over the base learner LazyDT and produces significantly (on average 50%) less complex ensembles than AdaBoost while maintaining comparable accuracy. To ameliorate the

[1]Technically a boosting algorithm is one that transforms a weak learning algorithm into a strong learning algorithm. Our algorithm does not have this property but it uses a boosting-style weight changing process. Thus, we refer to our algorithm as a boosting-style algorithm and use the term *boosted LazyDT* for the resulting ensemble to distinguish it from other ensembles such as *bagged LazyDT*.

Proceedings of the Twentieth International Conference on Machine Learning (ICML-2003), Washington DC, 2003.

Table 1. A generic lazy decision tree algorithm

Inputs: S is the training set
y is the test instance to be classified
Output: class label for the test instance y

1. If all instances in S are from a single class l, return l.
2. Otherwise, select a test T and let t be the value of the test on instance y. Let S' be the set of training instances satisfying $T = t$ and apply the algorithm to S' and y.

over-fitting problem for LazyDT, we also propose a new distance-based pruning technique to generate simpler and more accurate lazy decision trees. Currently this technique is implemented and tested only on data sets with numerical features. In our experiments, the proposed pruning method improves the accuracy and comprehensibility for both single lazy decision trees and LazyDT ensembles.

The rest of the paper is organized as follows. In Section 2 we briefly introduce the LazyDT algorithm and explain the difficulties of applying existing boosting algorithms to LazyDT. Section 3 describes our boosting-style algorithm for LazyDT. In Section 4, we empirically evaluate our boosted LazyDT ensembles on ten data sets and compare their performance with the base learner LazyDT, bagged LazyDT ensembles and boosted regular decision tree ensembles. Section 5 introduces the distance-based pruning technique for LazyDT and illustrates its effectiveness with preliminary experimental results. Section 6 concludes with a discussion of related issues and the future work.

2. The Difficulties of Applying Existing Boosting Algorithms to LazyDT

In Table 1 we briefly describe the general steps of the LazyDT algorithm. The core part of the algorithm is how to select a test – LazyDT chooses a test that "optimizes" the resulting branch taken by the given test instance. Once a test is selected, only instances that take the same branch as the test instance are kept to build the remaining part of the lazy decision tree. We omit the details of the algorithm and refer readers to the original paper (Friedman et al., 1996) for an exact description of the LazyDT algorithm.[2]

Commonly used boosting algorithms, such as AdaBoost (Freund & Schapire, 1997), iteratively apply a base learner to different distributions of the training data. Typically it is assumed that the produced base classifiers can be applied to classify the entire instance space. In each iteration, AdaBoost adjusts the weights of training instances according to the classification decisions made by the previously learned classifiers. Misclassified instances will be assigned larger weights in the successive iteration to make the base learner focus on the "hard" instances.

When LazyDT is used as the base learner, it differs from regular decision algorithms in two ways. First, LazyDT generates a single decision path for a given test instance. This path can only be used to give predictions to instances that satisfy all of its tests. For those instances that fail to satisfy the tests, no classification information is available. Without complete classification information, the weight changing process in the above framework can not be directly applied. Second, the LazyDT algorithm has a rather special goal – building a decision path that correctly classifies a given test instance. If a training instance contributes little information to correctly classifying the given test instance, even it is a "hard" instance for the current classifier, not much leverage can be gained by increasing its weight in subsequent iterations.

A tentative solution to the first of these two problems is to use the confidence-rated boosting algorithm (Schapire & Singer, 1999), which is capable of handling incomplete classification information. However, it does not address the second problem. Moreover, despite the fact that it has been used in SLIPPER (Cohen & Singer, 1999) to boost decision rules, the confidence-rated boosting algorithm is not an ideal choice for our task. In SLIPPER, a rule makes predictions for the training instances that satisfy all of its tests with a positive confidence and abstains on the others by making a prediction with a zero confidence. The boosting process changes the weight of a training instance only if it is given a classification with non-zero confidence. Our investigation of this method did not give promising results, which we conjecture was due to the lack of diversity in the resulting ensembles. LazyDT grows a decision rule (path) until all the instances that are covered by the rule are from the same class. Typically, only a small part of the training set will be predicted with non-zero confidence and be assigned different weights. Given only small changes of the distribution, LazyDT tends to produce very similar if not identical decision paths for each iteration, resulting in an ensemble that lacks diversity.

In summary, we believe that in order to successfully boost the performance of LazyDT, we need our algo-

[2]Our implementation slightly differs from the original algorithm in its handling of numeric features. We use C4.5's approach rather than discretizing the features before the tree construction.

rithm to 1) work with incomplete classification information; 2) produce diverse lazy decision paths; and 3) take into account the special goal of classifying the given test instance. To our knowledge none of the existing boosting algorithms successfully satisfies all three requirements.

3. A Relevance-Based Boosting-Style Algorithm for LazyDT

Based on the observations made in Section 2, we propose a relevance-based boosting-style algorithm to generate a customized lazy decision tree ensemble for each test instance. Given a test instance, a training set and an ensemble size T, our algorithm iteratively constructs T decision paths. In each iteration, a decision path customized for the given test instance is produced by applying LazyDT to the training set with a given distribution represented by instance weights. In the initial distribution all the training instances are equally weighted. The instance weights are then adjusted according to the learned decision path to form a new distribution for the next iteration. This process repeats for T iterations and the majority vote of the resulting T decision paths is used to give the final prediction for the given test instance. Our algorithm adjusts the weight of a training instance according to how *relevant* this instance is to classifying the given test instance and whether its class label is predicted by the current decision path. The remainder of this section first defines what we mean by *relevant* and then presents the details of the weight modification procedure.

To understand how *relevant* a training instance is to classifying a test instance, we shall look at how training instances are used by LazyDT in the *path building process*. Given a training set and a test instance, LazyDT starts by selecting a test for the root node using all the training instances. Once the test is selected the training set will be partitioned according to results of the selected test. Because LazyDT only extends the branch that is taken by the test instance, those instances that take other branches will be "discarded" and not used any further in the learning process for the given test instance. The remaining instances will then be used to select the next test for the path. This process repeats until all remaining instances belong to the same class, at which point a leaf node is generated and labeled with that class. The later an instance is "discarded" in this process the more *relevant* we consider it is to classifying the given test instance. This is because it was used to select more tests for the decision path and hence contributed more information to the classification decision.

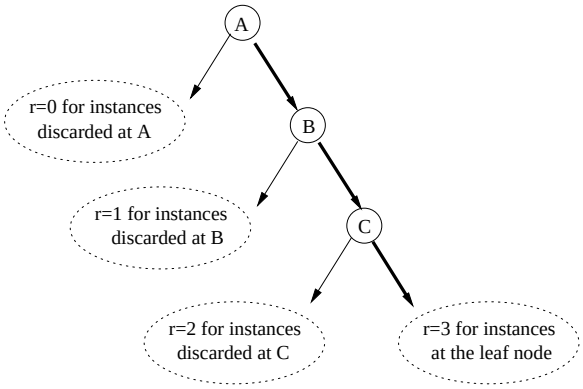


Figure 1. A decision path with three test nodes. The training instances are divided into four groups, each of which has a different relevance level.

Given a test instance y and a decision path generated for y, We define the *relevance level*, r, of a training instance x to be the depth of the node at which the training instance is "discarded" when building the given decision path. Figure 1 shows an example of a decision path with three test nodes. This path divides the training instances into four groups each with a different relevance level. Note that the relevance level of a training instance is defined with respect to a specific test instance and decision path. In each iteration of our algorithm, after a decision path is built the relevance level of each training instance is used to regulate by what magnitude to change its weight for the next iteration. A higher relevance level results in a more dramatic weight change for an instance.

In addition to the magnitude, we also need to decide the direction of the weight change. Recall that AdaBoost increases the weight of incorrectly classified training instances and decreases the weight of correctly classified instances. For LazyDT only a small subset of the training instances are classified by the decision path and in most cases all of these are classified correctly (because the path is grown until purity). To apply the above idea, we assume the decision path "classifies" all the training instances as the class of the leaf node. Therefore, if a training instance has the same class label as the leaf node, its weight will be decreased and vice versa. Obviously our assumption could be erroneous for some training instances. However, for a particular instance, the higher the relevance level, the more "reasonable" our assumption is to that particular instance because it shows more consistency with the decision path. Recall that we control the magnitude of the weight change according to the instance's relevance level.

Table 2. Comparing LazyDT, boosted LazyDT(BO-LazyDT) and bagged LazyDT(BA-LazyDT)

Data set	Insts	Attrs	Classes	LazyDT	BO-LazyDT	BA-LazyDT
Cleveland	303	13	2	71.95±2.19	76.73±1.06 ⊕	74.59±1.15 ⊕
Crx	541	15	2	86.32±1.18	85.84±0.69	85.84±0.47
German	1000	20	2	72.05±0.89	73.88±0.92 ⊕	74.40±0.37 ⊕
Hepatitis	155	19	2	85.87±1.10	85.42±0.72	84.65±0.99 ⊖
Lympho	148	18	4	81.35±1.82	80.61±1.01	79.12±2.04 ⊖
Monk2	601	6	2	86.99±1.29	91.40±0.93 ⊕	76.74±1.11 ⊖
Pima	768	8	2	72.04±0.70	73.13±0.74 ⊕	74.61±0.74 ⊕
Road	2056	7	9	78.52±0.26	79.86±0.45 ⊕	80.25±0.27 ⊕
Tahoe	2237	6	5	72.31±0.41	74.31±0.37 ⊕	75.07±0.37 ⊕
Yeast	1484	8	10	49.44±0.47	51.72±0.78 ⊕	53.20±0.78 ⊕

Specifically, the weight changing process works as follows. Given a decision path R, we first calculate for each training instance x its relevance level r as the depth of the node where x is "discarded" when building R. Then we compare x's class label with R's leaf node class. If x has the same class label, its weight is multiplied by a factor α^r, otherwise it is multiplied by a factor β^r, where $\alpha < 1$ and $\beta > 1$. After all individual updates, the instance weights are normalized to form a valid distribution for the next iteration. Consider the example shown in Figure 1, the weight update factor for instances that are *located at* the leaf node will be α^3 or β^3 based on whether the instance has the same class label as the leaf node.[3] Note that the weight update factor will be one for those instances that have zero relevance level (i.e., those discarded at node A), but the normalization step will adjust the weights of those instances.

In implementing the algorithm, we need to specify the values for α and β. In our experiments we arbitrarily selected a basic setting $\alpha = 0.98$ and $\beta = 1.15$. How to optimize the choice of α and β is an open problem. Fortunately within a reasonable range the performance of the algorithm appeared to be rather robust – in the tests of other settings we did not observe significant performance changes when the parameters are within range $0.85 < \alpha < 1$ and $1 < \beta < 1.15$. When values outside of this range were used, the performance started to decrease. This is possibly because the weight change magnitude increases exponentially as the relevance level increases. When extreme parameter values are used, the weight change magnitude can be so big that a few training instances may have most of the weight and the resulting distribution may fail to retain sufficient information to build an accurate decision path for the given test instance.

[3]LazyDT stops growing a decision path when only one training instance remains. Our implementation of LazyDT with weighted instances stops if the total weight of all remaining instances is less than one, making it possible to produce leaf nodes with instances from more than one class.

4. Experiments

In this section we compare boosted LazyDT with its base learner LazyDT, bagged LazyDT and boosted regular decision trees. Ten well-known benchmark data sets from the UCI data collection (Blake & Merz, 1998) are used in the comparison. The characteristics of these data sets are summarized in columns 1-4 of Table 2. For each data set, ten runs of a stratified ten-fold cross-validation are conducted and the reported results are averaged over ten runs. An ensemble size of ten is used for all ensemble methods. Experiments with an ensemble size of twenty show similar trends as those of size ten and are thus omitted from the result tables.

Comparison of boosted LazyDT, LazyDT, and bagged LazyDT: The base learner LazyDT was implemented based on MLC++ library (Kohavi et al., 1994). Bagging was implemented as follows. Given a test instance and the training set, bagging generates T bootstrap samples of the original training set, where T is the ensemble size. Each sample is generated by uniformly sampling m instances from the training set with replacement, where m is set to be the size of the original training set. On average, each sample contains about 63.2% distinct instances from the original training set. Each bootstrap sample is then used as the training set to build a lazy decision tree to classify the given test instance. A majority vote among the resulting T lazy decision trees is used as the final output of the bagged LazyDT ensemble.

Table 2 presents the accuracies of LazyDT, boosted LazyDT and bagged LazyDT in columns 5, 6 and 7 respectively. A paired t-test with 0.05-level is used to compare each ensemble method to their base method

Table 3. Comparing boosted LazyDT(BO-LazyDT) and AdaBoost applied to C4.5(ADAC45)

	Accuracy		Decision path length	
Data set	BO-LazyDT	ADAC45	BO-LazyDT	ADAC45
Cleveland	76.73±1.06	78.95±0.87	2.87±0.67	4.78±0.78
Crx	85.84±0.69	85.42±0.98	2.91±0.69	4.17±0.72
German	73.88±0.92	70.44±1.24	3.21±0.65	6.20±1.23
Hepatitis	85.42±0.72	82.45±1.83	3.01±1.33	3.35±0.76
Lympho	80.61±1.01	78.65±1.82	2.25±0.77	2.42±0.61
Monk2	91.40±0.93	64.11±1.45	5.12±0.98	1.28±0.46
Pima	73.13±0.74	72.45±1.02	2.89±0.63	5.64±1.37
Road	79.86±0.45	83.03±0.36	2.87±0.72	7.69±1.37
Tahoe	74.31±0.37	74.71±0.31	3.07±0.88	8.41±2.01
Yeast	51.72±0.78	56.48±0.87	3.39±0.92	10.31±2.54

LazyDT. In Table 2, "$\oplus$" is used to indicate a significantly better performance than LazyDT and "$\ominus$" is used to indicate a significantly worse performance than LazyDT.

Among the ten data sets, boosted lazy decision trees perform significantly better than the base learner for seven data sets and never significantly worse. Bagged lazy decision trees show similar performance improvement over LazyDT for most of the data sets but behave less consistently. For three data sets, Hepatitis, Lympho and Monk2, bagging significantly degrades the performance of the base learner. This is possibly caused by the sub-sampling procedure used by bagging to generate different distributions, which may lose important information in the training set. In contrast, our boosting-style algorithm generates an ensemble of classifiers through adjusting instance weights and avoids such detrimental information loss.

Comparison of boosted LazyDT and AdaBoost: To compare the performance of our algorithm to boosted regular decision trees, we implemented the AdaBoost algorithm based on MLC++ (Kohavi et al., 1994) and the C4.5 source code. Note that C4.5 was run in its default mode with pruning enabled.

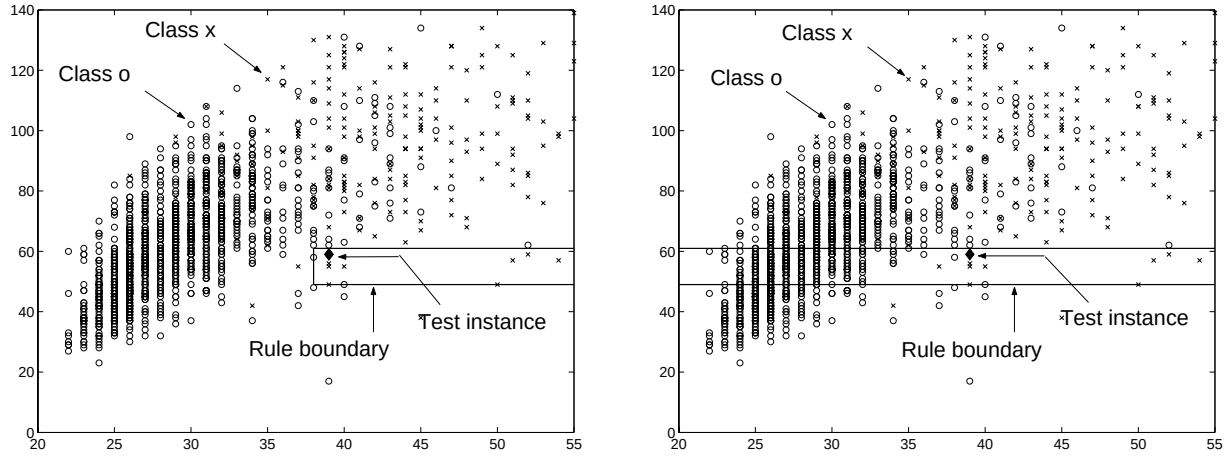
Table 3 presents the accuracy and the average decision path length of the resulting ensembles of the two algorithms. The average decision path length is calculated as follows. For each test instance, we calculate the decision path length averaged over all classifiers in the ensemble and then take the average across all test instances. For a regular decision tree, the path that is taken by the test instance is used as its decision path.

We observe from Table 3 that boosted LazyDT gives comparable performance to boosted regular decision trees. There is no strong evidence for superiority of either algorithm. However, we observe that the average decision path length of the boosted lazy decision trees is significantly (on average 41.77%) shorter than boosted regular decision trees. The Monk2 data set is an interesting outlier in the results. It contains an artificial concept that can not be represented by the C4.5 decision tree. In this case, the path length of the regular decision tree ensemble is extremely short but such simplicity comes with a large sacrifice of accuracy – it fails to capture the underlying concept and performs poorly as opposed to the LazyDT ensemble. Excluding the Monk2 data set, the average decision path length of boosted lazy decision trees is 50% shorter than that of boosted regular decision trees.

5. Further Improvement with Pruning

A drawback of LazyDT is that it is highly prone to over-fitting. This is because LazyDT grows a decision path until a pure node is reached. We believe an effective pruning strategy will reduce over-fitting and improve the performance of LazyDT—in fact, lack of pruning is considered "the weakest point" of the LazyDT algorithm in the original paper (Friedman et al., 1996). In this section we introduce our distance-based pruning strategy for LazyDT.

As the decision path generated by LazyDT has the form of a decision rule, LazyDT pruning is closely related to rule pruning. Traditional rule pruning techniques (Quinlan, 1993; Cohen, 1995; Cohen & Singer, 1999) prune an overly-specific rule by deleting tests from the rule and evaluating the resulting rules with a hold-out pruning set or the original training set. Typically the goal of such pruning is to produce a more general rule for the particular class that is covered by the rule. For LazyDT, a decision rule is generated to classify a given test instance. A proper pruning heuris-



(a) Original decision rule

(b) Pruned decision rule

Figure 2. An example of LazyDT pruning from Tahoe dataset

tic should evaluate a rule's *effectiveness in classifying the given test instance*. However, traditional rule pruning heuristics such as the class coverage over a pruning set or the training set can not fulfill this task. A more fundamental problem is that applying traditional rule pruning techniques to LazyDT will not correct any classification mistakes since these techniques always associate the rule with the same class label throughout the pruning process.

Visually a lazy decision rule defines a region containing the test instance in the instance space and labels the test instance based on the majority class within this region. Intuitively we would like the test instance to be at the center of the defined region, which ideally is homogeneous, compact and far away from the other classes. We designed our pruning heuristic based on this intuition.

Let $S = \{< x_i, y_i >: i = 1, 2, ..., n\}$ be the training set, where x_i is the feature vector of instance i and y_i is its class label that takes a value from the set $Y = \{1, 2, ..., m\}$. Let C_i represent the set of training instances belonging to class i. A rule R partitions S into two disjoint sets $\Re$ and $\bar{\Re}$, where $\Re$ contains all the instances that are covered by the rule and $\bar{\Re}$ contains the rest. To evaluate the goodness of R with respect to classifying test instance x, we calculate the $\Re$-*distance* and $\bar{\Re}$-*distance* of R with respect to x as follows.

Definition 1: $D_f(x, A)$, the distance between an instance x and an instance set A, is defined as the average Euclidean distance between x and its k-nearest neighbors in set A where the Euclidean distance is calculated on *feature set* f.

Assume rule R predicts class j and let f_r be the feature set that is used by R, we define:

Definition 2:

$$\Re\text{-}distance = D_{fr}(x, C_j \cap \Re)$$

i.e., the distance between x and the instance set $C_j \cap \Re$ calculated on the feature set f_r.

Definition 3:

$$\bar{\Re}\text{-}distance = \min_{y \in Y - \{j\}} D_{fr}(x, C_y)$$

The above distance measurements are both defined in a subspace formed with the set of features used by the rule R. $\Re$-*distance* measures the distance between the test instance and the majority class in the set $\Re$ and $\bar{\Re}$-*distance* measures the distance between the test instance and the closest cluster of instances from other classes. A rule with a small $\Re$-*distance* and a large $\bar{\Re}$-*distance* for the test instance is considered good since it has the properties we described above. In light of the fact that for different rules different numbers of features are used to calculate the distance, the ratio $\frac{\bar{\Re}\text{-}distance}{\Re\text{-}distance}$ is used to remove the influence of dimensionality. A larger heuristic value indicates a better rule. Currently our heuristic is limited to data sets with only numeric features and k in Definition 1 is arbitrarily set to be ten.

Our pruning algorithm works iteratively. In each iteration, the algorithm performs a greedy search to select a deletion of test from the rule conditions. Each possible deletion produces a candidate rule, whose goodness is evaluated using the heuristic defined above. The deletion that results in the largest heuristic improvement will be performed—resulting in a new rule. This

Table 4. Accuracy of LazyDT and boosted LazyDT with and without pruning

DATASET	CLEVELAND	PIMA	ROAD	TAHOE
LAZYDT	71.95±2.19	72.04±0.70	78.52±0.26	72.31±0.41
PRUNED LAZYDT	73.01±2.15	74.27±1.14	79.84±0.21	74.95±0.52
BO-LAZYDT	76.73±1.06	73.13±0.74	79.86±0.45	74.31±0.37
PRUNED BO-LAZYDT	78.25±1.09	76.00±0.61	80.58±0.28	75.66±0.20

process repeats until no further improvement can be achieved. Note that a pruned rule will give a different prediction if the new region it defines has a different majority class. This is fundamentally different from regular rule pruning techniques – making it possible for the proposed pruning strategy to correct a classification mistake made by an original rule.

Figure 2 shows a pruning example from the Tahoe data set. This data set has five classes, among which two are shown in the figure as "o" and "x". The others are omitted because they are either not present near the region defined by the rule or are too rare to be noticed. The original rule, shown in Figure 2(a), consists of three tests on two features. It defines a region containing only class "x"instances and classifies the test instance as class "x". However, the test instance is close to the region boundary and is visually closer to class "o" than to the predicted class "x". The pruned rule, shown in Figure 2(b), defines a new region with a different majority class and classifies the test instance as class "o"—the real class of the test instance in this case. Our pruning strategy successfully corrected the mistake made by the original rule. Note that the proposed pruning strategy is built on an intuitive heuristic without a certain guarantee, therefore it may also reverse a correct classification in some cases. However, we hypothesize that on average this strategy will improve the classification accuracy and our preliminary experiments support this hypothesis.

We applied the proposed pruning strategy to LazyDT and boosted LazyDT with an ensemble size of ten. In Table 4 we report the accuracy of LazyDT and boosted LazyDT with and without pruning on four data sets. Only four data sets are used here because our current implementation of the pruning method can not handle non-numeric features or missing values. On average, pruning improves the accuracy of LazyDT by 1.81%. For boosted LazyDT, pruning also improves the accuracy by an average of 1.61%. In addition, pruning significantly reduces the complexity of the resulting rules – on average we observe a 22% size reduction for regular LazyDT and 25% reduction for boosted LazyDT. The average length of pruned LazyDT and LazyDT ensemble is 2.13 and 2.19 respectively. This indicates that most resulting rules use only two or three features, forming a low dimensional feature space in which we can easily visualize the rules. We believe this can greatly enhance the comprehensibility of the resulting decision rules.

Because the experiments are somewhat limited, we can not conclusively comment on the effectiveness of our pruning algorithm in general. However, the preliminary results do indicate that the pruning algorithm is potentially useful for handling the over-fitting problem for the LazyDT algorithm and it merits more thorough examination. The ensemble performance gain induced by pruning is rather interesting because previous work (Bauer & Kohavi, 1999; Dietterich, 2000) has indicated that pruning often has no significant impact on the performance of boosted decision tree ensembles. Here we provide two possible explanations. First, our experiments set the ensemble size to be ten, the effect of pruning may become negligible if larger ensembles are used. Second, this is also possibly due to the fact that our "boosting" algorithm is only a "boosting-style" algorithm – pruning may have a different impact in this case.

A drawback of our pruning heuristic is that it is biased toward the majority class, thus minority class performance may degrade. For example, the Tahoe data set has two minority classes - class 2 and 4, together covering only 4.74% of the whole data set. Pruning decreased the accuracy of LazyDT on these two classes from 14.34% to 9.06%. We conjecture that this is because on average a test instance is more likely to be closer to majority class instances because there are many more of them in the instance space. Future work will address how to incorporate a cost function into our pruning process to alleviate this problem for data sets where the misclassification cost of a minority class is greater than that of a majority class.

6. Conclusions

In this paper, we presented and empirically evaluated a relevance-based boosting-style algorithm to build lazy

decision tree ensembles. Given a test instance, the algorithm builds an ensemble of lazy decision trees by adjusting training instance weights and then uses majority vote to make the final classification. The training instance weights are adjusted according to their relevance to classifying the given test instance. Experimental results show that our boosted lazy decision trees significantly outperform the base learner LazyDT. Compared to bagged LazyDT, our algorithm shows more consistent performance improvement over LazyDT. In comparison with boosted regular decision trees, boosted LazyDT achieves better comprehensibility while maintaining comparable accuracy. A problem of the proposed algorithm is that it is time-inefficient. We see its best use in situations where the number of test instances is limited and the classifier's accuracy and comprehensibility are both important for individual test instances.

A second contribution of this paper is a novel distance-based pruning technique for LazyDT. Different from traditional rule pruning methods, our method is capable of correcting classification mistakes for the given test instance. Demonstrated by preliminary experimental results, our pruning method produces simpler lazy decision trees with improved classification performance. As explained in Section 5, our pruning method is biased against minority classes, thus should not be used when minority class accuracy is more important than overall accuracy. Indeed an open question is how to incorporate misclassification costs into LazyDT and our pruning technique.

In boosting algorithms such as AdaBoost, when a base learner fails to generate a classifier with an accuracy over 50%, the boosting process will stop or be reset and the "bad" classifier will be discarded. Such a stopping criterion is missing in our relevance-based LazyDT ensemble algorithm. In our future work we will investigate using the distance-based heuristic to weight the rules to avoid strong detrimental influence of "bad" rules.

Acknowledgment

The authors were supported by NASA under Award number NCC2-1245. Many thanks to the anonymous reviewers for their valuable comments that helped improve this paper.

References

Bauer, E., & Kohavi, R. (1999). An empirical comparison of voting classification algorithms: Bagging, boosting, and variants. *Machine Learning*, *36*, 105–139.

Blake, C. L., & Merz, C. J. (1998). UCI repository of machine learning databases.

Breiman, L. (1996). Bagging predictors. *Machine Learning*, *24*, 123–140.

Cohen, W. W. (1995). Fast effective rule induction. *Proceedings of the Twelfth International Conference on Machine Learning* (pp. 115–123).

Cohen, W. W., & Singer, Y. (1999). A simple, fast, and effective rule learner. *Proceedings of the Sixteenth National Conference on Artificial Intelligence* (pp. 335–342).

Dietterich, T. G. (2000). An experimental comparison of three methods for constructing ensembles of decision trees: Bagging, boosting, and randomization. *Machine Learning*, *40*, 139–157.

Freund, Y., & Schapire, R. E. (1996). Experiments with a new boosting algorithm. *Proceedings of the Thirteenth International Conference on Machine Learning* (pp. 148–156). Morgan Kaufmann.

Freund, Y., & Schapire, R. E. (1997). A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Science*, *55*, 119–139.

Friedman, J. H., Kohavi, R., & Yun, Y. (1996). Lazy decision trees. *Proceedings of the Thirteenth National Conference on Artificial Intelligence* (pp. 717–724). AAAI Press.

Kohavi, R., John, G., Long, R., Manley, D., & Pfleger, K. (1994). MLC++: A machine learning library in C++.

Margineantu, D. D., & Dietterich, T. G. (2002). Improved class probability estimates from decision tree models. *Nonlinear Estimation and Classification; Lecture Notes in Statistics* (pp. 169–184).

Pagallo, G., & Haussler, D. (1990). Boolean feature discovery in empirical learning. *Machine Learning*, *5*, 71–99.

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

Quinlan, R. J. (1996). Bagging, boosting, and C4.5. *Proceedings of the Thirteenth National Conference on Artificial Intelligence* (pp. 725–730).

Schapire, R. E., & Singer, Y. (1999). Improved boosting algorithms using confidence-rated predictions. *Machine Learning*, *37*, 297–336.