

# I<sup>2</sup>REP++, a Faster Rule Learning Algorithm\*

Oliver Dain

Robert K. Cunningham

Stephen Boyer

22nd December 2003

## Abstract

We present I<sup>2</sup>REP++, a rule learning algorithm similar to RIPPER and IREP. Like these other algorithms I<sup>2</sup>REP++ produces accurate, human readable rules from noisy data sets. However I<sup>2</sup>REP++ is able to produce such rule sets more quickly and can often express the target concept with fewer rules and fewer literals per rule resulting in a concept description that is easier for humans to understand. The new algorithm is fast enough for interactive training with very large data sets.

## 1 Introduction

Classifiers that produce if-then rules have become popular as they produce human readable rule sets and often have a high classification accuracy [11]. The most common ways to produce such rule sets from data are to first learn a decision tree and then extract the rules from the tree [13] or to learn the rules directly from the data [5, 7, 4]. RIPPER is an algorithm belonging to the latter class that has proved effective and fast even with large, noisy data sets [5, 6].

While RIPPER is a very fast algorithm, the training time was too long for an information assurance application of interest to the authors. Our application required an algorithm that could be trained on data sets with over one million training patterns and more than thirty features fast enough to be used in an interactive environment where training times of more than a few minutes would be unacceptable. We therefore used RIPPER as a starting point and attempted to develop an algorithm that achieved comparable accuracy but ran faster. The result of these efforts is I<sup>2</sup>REP++. The algorithm has proven to have equivalent accuracy while being significantly faster at developing new rule sets. The speed improvements were achieved by making several changes to the RIPPER algorithm including a better pruning metric, some novel data structures, and a more efficient stopping criteria.

I<sup>2</sup>REP++ handles data with categorical features more efficiently than earlier systems [7, 5] and is thus often able

to produce fewer rules while achieving similar accuracy on such data sets. We show that the FOIL gain metric [12] used for growing rules has a property we call “sortable”. Exploiting this property allows us to produce individual rules that cover more data than comparable RIPPER rules while requiring little more execution time per learned rule.

## 2 Algorithm Overview

I<sup>2</sup>REP++ is based on RIPPER [5] which in turn is based on Fürnkranz and Widmer’s IREP algorithm [7]. These algorithms all share the common structure described here. It should be noted that RIPPER is able to handle data sets with targets that take on more than two unique values. While this is certainly useful in a machine learning algorithm, I<sup>2</sup>REP++ and IREP are designed for boolean valued targets only and this application shall be the focus of the following discussion. If desired, I<sup>2</sup>REP++ could be similarly extended.

All three algorithms begin with a default rule (target = false) and, using a training data set, attempt to learn rules that predict exceptions to the default. Each rule learned is a conjunction of literals. Each literal corresponds to a split of the data based on the value of a single feature. For numerical features, each literal is of the form  $f_i > c$ , or  $f_i < c$  for some feature  $f_i$  and some constant  $c$ . For categorical features IREP and RIPPER produce literals of the form  $f_i = c$  where  $c$  is a value that may be attained by feature  $f_i$ . RIPPER also has an option that allows it to produce literals of the form  $f_i \neq c$ . I<sup>2</sup>REP++ produces literals of the form  $f_i \in \{c_1, c_2, \dots, c_m\}$  where each  $c_i$  is a value that may be attained by feature  $f_i$ . If all the literals are true for a given data point, then that data point is said to be “covered” by the rule and we predict a target value of true for that pattern. If a data point is not covered by any of the rules we predict its target value is false using the default rule.

The basic algorithm structure is depicted in Algorithm 1. On each iteration of the main loop (steps 2 through 9) a new rule is learned. First the training data is randomly split into a grow set and a prune set preserving the prior probabilities of the target (2/3 of the data in the grow set, 1/3 in the prune set). Next, a new rule is learned using the grow set. This new rule is immediately pruned using the prune set to compensate for over-fitting. It must now be decided if the newly formed rule is a good rule that should be added to the rule set or if it

\*This work is sponsored by the Department of Defense under Air Force Contract F-19628-00-C-0002. Opinions, interpretations, conclusions and recommendations are those of the author and are not necessarily endorsed by the United States Government.

**Algorithm 1:** Algorithm Overview**Input:** A training data set**Output:** A rule set

LEARN(TrainingData)

- (1) RuleSet  $\leftarrow$  NULL
- (2) **repeat**
- (3) (GrowSet, PruneSet) = SPLIT(TrainingData)
- (4) NewRule  $\leftarrow$  GROWRULE(GrowSet)
- (5) NewRule  $\leftarrow$  PRUNERULE(NewRule, PruneSet)
- (6) **if** KEEP(NewRule)
- (7) RuleSet  $\leftarrow$  RuleSet + NewRule
- (8) TrainingData  $\leftarrow$  NOTCOVERED(RuleSet, TrainingData)
- (9) **until** stopping criteria is met
- (10) **return** RuleSet

is a bad rule that should be discarded. All three algorithms make this determination differently. Finally the grow set and prune set are combined and any patterns covered by the new rule are removed from this combined set to form a new training set. This new set is used on the next iteration of the loop. At some point we must stop learning new rules and return a final rule set. All three algorithms have different stopping criteria. It should be noted that IREP++, in contrast to RIPPER, does not remove patterns that are covered by the new rule in step 8 unless the newly learned rule is deemed good enough to keep (the if statement in step 6 evaluates to true). If the rule is not worth keeping then we do not remove the patterns it covers. Since the next iteration will randomly split the data into a new grow and prune set, which should be different than the grow and prune sets of the previous iteration, we are unlikely to learn the same bad rule on the next iteration.

IREP, RIPPER, and IREP++ also share the same basic structure in the GrowRule and PruneRule procedures which we now address. The GrowRule algorithm is depicted in Algorithm 2. On each iteration of its main loop (steps 2 through 10) GrowRule adds another literal to current rule.

To construct the new literal, GrowRule considers each feature in the GrowSet in turn and attempts to find the best literal using that feature. For continuous-valued features this involves sorting the data set using the feature under consideration and then considering all possible split points. Each split point corresponds to two possible literals:  $f_i < c$  and  $f_i > c$  where  $c$  is a number mid way between the values on either side of the split point. The value of the split is determined using Quinlan's information gain criteria from his FOIL algorithm [12]. FOIL gain is an information theoretic criteria and can be written

$$(2.1) \quad p \left( \log_2 \frac{p}{p+n} - \log_2 \frac{p_0}{p_0+n_0} \right).$$

**Algorithm 2:** GrowRule Overview**Input:** A grow data set**Output:** A rule

GROWRULE(GrowSet)

- (1) NewRule  $\leftarrow$  NULL
- (2) **repeat**
- (3) GlobalBestSplit  $\leftarrow$  NULL
- (4) **foreach** feature in GrowSet
- (5) NewSplit  $\leftarrow$  FINDSPLIT(feature, GrowSet)
- (6) **if** NewSplit is better than GlobalBestSplit
- (7) GlobalBestSplit  $\leftarrow$  NewSplit
- (8) NewRule  $\leftarrow$  NewRule + GlobalBestSplit
- (9) GrowSet  $\leftarrow$  NOTCOVERED(NewRule, GrowSet)
- (10) **until** no errors on GrowSet
- (11) **return** NewRule

If  $R$  is the original rule and  $R'$  is the rule after adding our candidate literal, then  $p(n)$  is the number of positive (negative) patterns covered by  $R'$ , and  $p_0(n_0)$  is the number of positive (negative) patterns covered by  $R$ . IREP and RIPPER handle categorical features by having each call to FindSplit return a literal of the form  $f_i = c$  where  $c$  is a value attained by the feature  $f_i$ . As with numerical features, literal quality is assessed using the FOIL gain metric. A discussion of the way categorical features are handled by IREP++ can be found in Section 3.2.

The split having the highest FOIL gain metric on all the features is retained and the corresponding literal is added to the rule. All patterns in the GrowSet covered by this new literal are removed and, as long as there are still some negative patterns remaining in the GrowSet, we go back through the loop and add another literal. The loop terminates when the new rule makes no errors on the GrowSet. At this point the rule has over-fit the data and must be pruned.

Pruning involves removing literals from the new rule until performance on the PruneSet decreases. The metric used to assess performance is different in all three algorithms as is the sequence of literals considered for removal. The following sections will explore such differences in the algorithms.

**3 Differences Between IREP++ and RIPPER**

RIPPER was designed to improve the accuracy of IREP. Cohen demonstrated that RIPPER is significantly more accurate than IREP, but at the cost of being slightly slower [5]. Our goal was to preserve the accuracy of RIPPER while improving on its speed. As a result this discussion will focus primarily on RIPPER.

**3.1 Handling of Numerical Features** Finding the best split on a numerical feature requires us to sort the data set on that feature and then search for the split point that maximizes the FOIL gain. Each pass through the main loop of GrowRule requires us to access the sorted values

of each feature. RIPPER simply re-sorts the data on each pass. Using data structures similar to those used by the SLIQ system [10] we have been able to keep the feature values in sorted order so sorting is only necessary on the first iteration.

When the GrowSet is formed, a separate “feature matrix”, is used to store each feature. The first column of this matrix stores the values of the feature in sorted order. The second column stores the number of the row in the GrowSet from which the value came. Thus each feature value is represented by an ordered pair  $(v_i, r_i)$  where  $v_i$  is the  $i^{\text{th}}$  value of the feature and  $r_i$  is the row in the unsorted GrowSet where this value occurred. Another array, the “keep table”, is used to indicate which rows of the original GrowSet are still uncovered by the current rule. Each row in the keep table corresponds to one row in the initial, unsorted GrowSet. Initially each row of the keep table is set to true. These data structures are depicted in Figure 1.

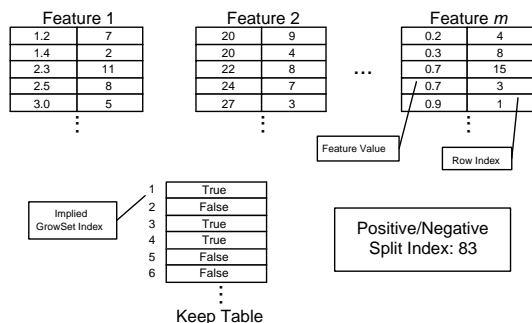


Figure 1: Data Structures Used by GrowRule

Once we have determined the best split on all features (step 8 of Algorithm 2) we can use this split to update the keep table, setting to false all rows of the keep table not covered by the new literal. This can be accomplished by iterating through the feature matrix starting at the split point for the feature used in the split. The row indices in the feature matrix indicate which rows of the keep table should be updated. The remaining feature matrices can then be compressed so they only contain values whose rows indices are marked true in the keep table. This can be accomplished at a cost that is proportional to the number of patterns remaining in the GrowSet.

Recall that finding the best split on a feature requires that we know the target value associated with each value of a feature. As we have removed the features from the data set to reduce the sorting time we will need another way to access this information. Our solution is to group all the positive training patterns at the top of the unsorted GrowSet and all the negative patterns at the bottom. We can then determine the target value by simply looking at the row index. If the row index is less than the number of positive patterns in the

GrowSet the target value must be positive; otherwise it must be negative.

**3.2 Handling of Categorical Features** RIPPER and IREP both handle categorical features by forming literals of the form  $f_i = c$  for some categorical feature  $f_i$  and some value  $c$ . We will refer to such literals as “single valued splits”. Given a categorical feature that takes on many unique values (“levels”) it is likely that many values of the feature predict a positive target value. As RIPPER can not learn rules of the form  $f_i \in \{c_1, c_2, \dots, c_k\}$  (“set-valued splits”) it must learn a separate rule for each value of the feature that predicts a positive target value. This not only requires more rules and hence more processing time, but may also lead to less accurate rules. A literal of the form  $f_i = c$  will cover fewer data patterns than a literal of the form  $f_i \in \{c_1, c_2, \dots, c_k\}$ . Thus, once the uncovered patterns have been removed, subsequent splits must be made with less evidence and hence will tend to be less accurate. Larger rule sets also take more time to evaluate when we wish to classify new data patterns. RIPPER has an option to learn negations for categorical features. With this option enabled RIPPER is able to learn literals of the form  $f_i \neq c$  or  $f_i = c$ . This increases the expressive power of RIPPER and helps mitigate some of the problems mentioned above.

There are, however, potential problems with allowing set-valued splits. It is known that mixing categorical features which attain many unique values and continuous valued features presents a problem when information theoretic metrics are used as the splitting criteria [11, pg. 73-75]. The problem arises because a feature which attains many unique values is capable of splitting the training data into many small subsets. If we can pick any subset of values as a split there will be so many possible subsets that we will tend to get some high valued splits by chance. Hence there will a bias for splits on these categorical features over splits on potentially more predictive continuous features. Our experiments indicate that this potential drawback is no larger than the accuracy drawback mentioned for single valued splits. Since the two approaches produce rules with similar accuracy but the set valued features produce smaller rule sets we have opted to use set valued features.

If we are to use set valued splits we must find a way to determine the optimal subset of values to use for the split in a computationally realistic manner. Calculating the FOIL gain of every possible subset would require time that is exponential in the number of levels. This is not necessary because FOIL gain is “sortable”. Specifically, suppose we have a feature  $f$  which can attain  $k$  possible values. Each value of the feature,  $v_j, 0 \leq j \leq k$  has  $p_j$  positive and  $n_j$  negative training patterns associated with it. If we order the levels such that  $\frac{p_1}{n_1} \geq \frac{p_2}{n_2} \geq \dots \geq \frac{p_k}{n_k}$  then the subset of values that maximizes the FOIL gain will always be of the

form  $\{v_1, v_2, \dots, v_l\}$  for some  $l \leq k$ . Thus finding the optimal subset requires time that is linear in the number of unique values attained by the feature.

A class of metrics was known to have this property [3], but FOIL gain is not a member of that class<sup>1</sup>. As the Gini metric is a member of the class [3] we tried using it as the metric in GrowRule. Gini assesses the impurity of a set of data as

$$I(S) = \left( \frac{p_S}{p_S + n_S} \right) \left( \frac{n_S}{p_S + n_S} \right)$$

where  $p_S$  ( $n_S$ ) is the number of positive (negative) data patterns in the set  $S$ . In GrowRule a split divides the initial data set,  $S$ , into two new sets  $S_1$ , and  $S_2$ . If the weighted Gini impurity of these two sets is less than the impurity of  $S$  then the proposed split is a good one. The split that causes the greatest reduction in Gini impurity is the best split. Here reduction in Gini impurity is calculated as

$$\Delta I = I(S) - \left( \frac{p_{S_1} + n_{S_1}}{p_S + n_S} \right) I(S_1) - \left( \frac{p_{S_2} + n_{S_2}}{p_S + n_S} \right) I(S_2).$$

This gave results that had roughly the same accuracy as those produced with FOIL gain, but it produced many more rules. This can be attributed to the fact that Gini has no built in bias for literals with large coverage while FOIL gain, with its leading factor of  $p$ , does. Since Gini did not perform the way we wanted we began to examine FOIL gain to see if it had the same properties. Monte Carlo simulation indicated that FOIL gain has the desired property, but we were unable to prove it. We therefore enlisted the help of Professor Madhu Sudan who produced the elegant proof that appears in Appendix A.

**3.3 Pruning** When pruning rules we must make two important choices: which literals should be considered for removal, and what metric should be used to assess the quality of a pruned rule. IREP iteratively considers the final literal for removal (where final refers to the last literal added) while RIPPER considers any final sequence of literals. After some performance comparisons we decided on RIPPER's approach.

In [7] the authors describe two possible pruning metrics:

$$(3.2) \quad M_1(p, n, P, N) = \frac{p + (N - n)}{P + N}$$

$$(3.3) \quad M_2(p, n) = \frac{p}{p + n}$$

where  $p$  ( $n$ ) is the number of positive (negative) patterns covered by the pruned rule and  $P$  ( $N$ ) is the number of

<sup>1</sup>The theorem in [3] applies to minimizing a function  $\Delta i = p_L \phi(p_L) + (1 - p_L) \phi(1 - p_L)$  where  $p_L$  is probability of being in one of the two subsets and  $\phi(p)$  is a concave function of  $p$ . While FOIL gain is a concave function of  $p$ , this parameter does not represent a probability nor are we defining optimality as required by the theorem.

positive (negative) patterns in the entire prune set. They note that metric  $M_1$  performs better than  $M_2$  and Cohen's experiments confirm this [5]. However, Cohen notes that the metric  $M_1$  can cause convergence problems [5]. In an attempt to find a better solution, RIPPER adopted the metric

$$M_3(p, n) = \frac{p - n}{p + n} = 2 \left( \frac{p}{p + n} \right) - 1$$

which is a linear function of  $M_2$  and so will have the same effect on pruning.

To summarize, the metric  $M_3$  is equivalent to  $M_2$  which is known to be inferior to  $M_1$ . As  $M_1$  suffers from convergence problems the authors investigated other pruning metrics and settled on a version of FOIL gain. We use the FOIL gain formula in Equation 2.1 but here  $p_0$  ( $n_0$ ) is the number of positive (negative) patterns in the prune set covered by the pruned rule and  $p$  ( $n$ ) is the number of positive (negative) patterns covered by the original, unpruned rule. If the value of this metric is negative then adding the literal under consideration to the pruned rule does not produce any information gain on the prune set and the literal should be removed. The sequence of literals that produces the lowest (most negative) FOIL gain with the fewest literals is the final pruned rule.

The execution time of PruneRule may be reduced by noting that a pattern covered by a rule will be covered by any pruned version of the same rule as removing literals always makes the rule more general. Thus a pattern covered by the current rule will be covered by the rule produced in the next iteration of PruneRule. We therefore only consider patterns which were not covered by the rule on the previous iteration.

**3.4 Stopping Criteria** Step 9 of Algorithm 1 refers to a stopping criteria. In order to maximize the speed of the rule learner we would like to stop learning as soon as possible. However, there is a danger that we might stop too soon. Thus the stopping criteria is a trade-off between computational efficiency and accuracy. The original IREP algorithm stops learning the first time a rule is learned whose accuracy on the prune set is worse than that accuracy of the empty rule (the rule which always predicts "false"). To improve the accuracy of IREP, RIPPER used a description length principal as its stopping criteria. After each rule is learned RIPPER calculates the description length of the rule set and the examples covered. Learning terminates when the description length becomes sixty four bits longer than the smallest description length obtained so far. It is then necessary to examine each rule in turn and delete those rules which decrease the description length.

IREP++ uses a simpler approach than RIPPER while avoiding the early stopping that caused accuracy problems with IREP. If a newly learned rule covers more negative than positive training patterns that rule is considered a bad rule

and is immediately removed from the rule set. In this case the patterns covered by the bad rule are not removed from the data set (e.g. step 8 of Algorithm 1 is executed only if the new rule is not a bad rule). When five bad rules have been learned in a row the algorithm terminates. The number of bad rules to learn before terminating was determined experimentally. Note that we avoid an infinite loop since the training data is randomly split before each rule is learned. Thus even though we don't remove the training data covered by the newly learned rule, a different split of the data is likely to result in a different rule being learned on the next iteration of the loop. Since bad rules are immediately removed from the rule set we don't need to consider which rules to remove after completing the steps of Algorithm 1 like RIPPER does.

**3.5 Optimization Loop** To improve classification accuracy RIPPER employs an optimization loop after completing all the steps of Algorithm 1. During the optimization phase alternatives to each rule are learned and a description length metric is used to choose which of the alternatives to keep. The optimization may be repeated any number of times. If  $k$  optimization loops are employed the algorithm is called RIPPER $k$ . Experiments with RIPPER indicate that while the optimization loop does improve performance some, the effect is not large (see Section 4.1). As the primary goal of IREP++ was to develop a faster algorithm, we didn't include any optimization loop.

## 4 Experimental Results

The accuracy and speed of IREP++ were assessed using a variety of realistic and synthetic data sets. The realistic data sets were gathered from the University of California, Irvine's data repository [2], the data from The Third International Knowledge Discovery and Data Mining Tools Competition [8], some of the data distributed with the LNKnet data mining package [9], the phoneme data set from the Esprit ROARS project [1], and some information assurance data sets produced by the authors. In some instances the data set did not initially represent a two class problem. Since IREP++ can only work on data with boolean targets, these data sets were modified to change them into a two-class problem. This was achieved by picking one value to represent a positive target and allowing all other values to represent a negative target. Data sets were randomly split, preserving the prior probability of the target, into a train set and a test set with two thirds of the data going to the train set and the remaining going to the test set. All error rates reported reflect performance on the test data.

Our speed assessments are based on the above mentioned data sets and some synthetic data sets. The synthetic data sets will be discussed in Section 4.2.

**4.1 Accuracy** We compared IREP++ to RIPPER0 and RIPPER2 (RIPPER with no optimization loop and with 2 optimization loops) on several realistic data sets. Both RIPPER0 and RIPPER2 were tried with and without the negation feature, which enables it to learn literals of the form  $f_i \neq c$ . A summary of these results appears in Table 1. RIPPER2 does only marginally better than RIPPER0 on these data sets. The won-lost-tied record of RIPPER2 compared to RIPPER0 is 10-6-3 with only one of the wins being statistically significant. Here, and in the following, statistical significance is calculated using the method described in [11]. We assume the difference between the error rates of the two classifiers on the test data set is approximately normally distributed. If the difference between the error rates is significant at the 95% level we deem the difference significant.

The won-lost-tied record of IREP++ compared to RIPPER2 is 6-7-6 with two of those wins and no losses being statistically significant. Against RIPPER0, IREP++ has a 10-6-3 won-lost-tied record with four statistically significant wins and one significant loss. The negation option doesn't affect the results very much. Comparing IREP++ to RIPPER0 with negation changes the won-lost-tied record to 11-6-2 with three wins and two losses statistically significant. RIPPER2 with negation has the same won-lost-tied record as RIPPER2 without negation. Two of the wins and one loss are statistically significant. Given these results we conclude that there is very little difference in the accuracy of RIPPER2 and IREP++ while IREP++ and RIPPER2 are both a bit more accurate than RIPPER0<sup>2</sup>. However, our experimental results indicate that IREP++ is significantly faster than either variant of RIPPER.

**4.2 Speed** Several factors affect the running time of these algorithms. The number of features, type of features, number of levels of categorical features, complexity of the target concept, amount of noise present in the data, and number of training patterns all affect the learning time required. Speed assessments of these algorithms required a combination of realistic and synthetic data. The data sets discussed in Section 4.1 were used to assess training times on realistic data. Two synthetic data sets were used to measure how well the algorithms scale as data set sizes increase. In all cases execution time was determined by running the algorithms on a Pentium III Linux machine with 512MB of RAM and using the Unix "time" command to measure the user and system times.

The training times on the smaller data sets in Table 1 were so small as to be unmeasurable since the time command has a resolution of only 0.01 seconds. We therefore only

<sup>2</sup>We are performing nineteen statistical significance tests here and so would expect roughly one Type I Error at the 95% significance level. Thus the fact that IREP++ has two statistically significant wins when compared to RIPPER2 should not be given too much weight.

Data Set	Number of Train Patterns	Number of Test Patterns	Number of Features	Categorical Features	RIPPER0 Errors	RIPPER0 Negation Errors	RIPPER2 Errors	RIPPER2 Negation Errors	IREP++ Errors
Credit*	551	139	15	9	26	26	18	22	25
Defcon_categorical	751,076	250,360	17	10	22	9	16	7	21
Defcon_intrusion	751,075	250,361	33	0	9	9	7	7	11
Hypothyroid*	2,529	634	25	18	9	9	8	8	8
Ionosphere*	280	71	34	0	4	4	5	5	8
KDD_cup†	395,216	98,805	41	7	44	50	37	46	55
Kr-vs-kp*	2,555	640	36	36	6	8	7	7	6
Letter**	16,998	3,002	16	0	6	6	9	9	18
Monks-1*	432	124	6	6	32	32	0	0	0
Monks-2*	432	169	6	6	64	64	64	64	53
Monks-3*	432	122	6	6	6	6	6	6	6
Mushroom*	6,498	1,626	22	22	0	0	0	0	0
Pbvowel**	1,194	300	5	1	8	8	6	6	6
Phoneme‡	4,322	1,082	5	0	200	200	170	170	134
Pima-diabetes*	614	154	8	0	38	38	39	39	48
Promoters*	84	22	57	57	10	7	7	7	2
Sonar*	165	43	60	0	10	10	12	12	8
Votes*	347	88	16	16	5	8	6	6	6
Wdbc*	454	115	30	0	8	8	5	5	0

Table 1: Performance of IREP++, RIPPER0 and RIPPER2 on several realistic data sets. The columns “RIPPER0 Negation Errors” and “RIPPER2 Negation Errors” indicate accuracy rates for RIPPER with the negation option enabled. Data sets marked with a \* come from the UC Irvine Repository. † sets come from the 1999 KDD Cup competition, \*\* sets are from the LNKnet data mining package, the ‡set is from the ROARS project, and unmarked sets were created by the authors.

show the timing results for data sets containing over one thousand training patterns. The results are summarized in Table 2. IREP++ is faster than RIPPER2 in all cases and faster than RIPPER0 in all but two cases. The two data sets on which RIPPER0 was faster are small and total training time for RIPPER0 and IREP++ was less than one second on these sets. However, on the larger data sets where training times were longer, IREP++ was always faster than either variant of RIPPER. In some instances this difference was quite dramatic. For example, on the KDD\_cup data set IREP++ trained in about 1 minute 30 seconds, RIPPER0 required about 8 minutes, and RIPPER2 required over a half an hour. Since the KDD\_cup data set is representative of the data sets we wish to work with interactively (in terms of size and number of features) the difference in training times represents the difference between interactive and batch processing. In almost all cases use of RIPPER’s negation option decreased the required training time. In some cases the difference was dramatic while in others it made little difference. For example, RIPPER2 required less than 17 minutes to train on the Defcon\_categorical data set with negation and almost 27 minutes to train without it. On the other hand, the KDD\_cup data set which also contains many categorical features showed very little difference in training times. Mushroom, which contains only categorical features, saw an increase in training times when the negation option was employed. This variance can be traced to the reduction in rules caused by the negation option. As indicated in Table 3, RIPPER2 with negation learned half as many rules on the Defcon\_categorical data as it did without negation while the

reduction in learned rules was less for the KDD\_cup and Mushroom sets.

To assess how well the algorithms scale we constructed two synthetic data sets: one containing only categorical features and one containing only numerical features. In both cases data was generated by first selecting random feature values. For the numerical data set all features values were floating point values between 0 and 1. For the categorical data set all feature values were randomly chosen from a set of eleven possible values. A rule set of the form learned by RIPPER and IREP++ was then used to decide if the feature vector should be given a true or false target. Noise was added to the data set by changing the target value of fifteen percent of the data patterns. All data sets contained ten features. The rules for generating the continuous valued data were

```
[1]<0.2 and [3]>0.8 and [7]<0.5;
[2]>0.7 and [3]<0.6 and [9]>0.25;
```

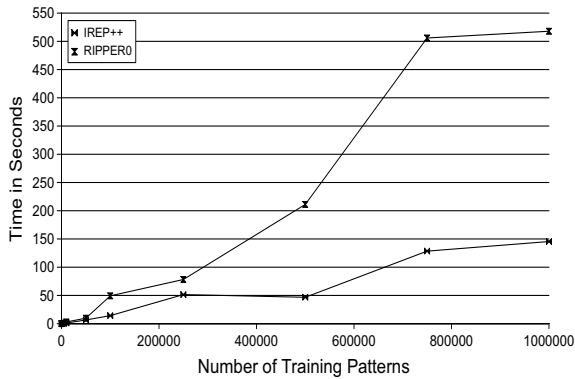
where  $[i] > c$  indicates that feature  $i$  must be greater than the value  $c$ . Each semi-colon terminate sequence of literals is a single rule. If the randomly generated data was covered by either rule it was assigned a target value of true. For the categorical features the rule set was

```
[1] in {a} and [3] in {a, b, c,
d, e, f} and [7] in {g, h, i,
j, k, a};
[2] in {k, a, b, c, d, e, f} and
[3] in {g, h, i} and
[9] in {b, c, d, e, f};
```

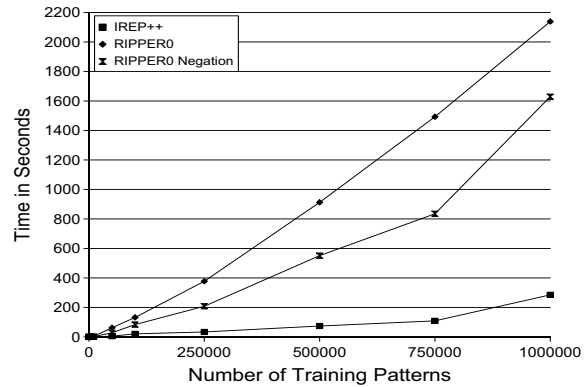
Data sets of various sizes were generated this way and RIPPER0 and IREP++ were run on the resulting sets. The

Data Set	Training Patterns	Number Features	Percent Categorical	IREP++ Time	RIPPER0 Time	RIPPER0 Negation Time	RIPPER2 Time	RIPPER2 Negation Time
Defcon_intrusion	751,075	33	0%	08:41.48	10:15.15	10:14.23	37:25.40	36:52.08
Defcon_categorical	751,076	17	59%	02:25.59	06:30.09	04:43.36	26:57.50	16:47.74
KDD_cup	395,216	41	17%	01:31.19	08:26.94	07:54.23	33:18.69	30:39.23
Letter	16,998	16	0%	00:04.61	00:06.21	00:06.11	00:20.92	00:20.65
Phoneme	4,322	5	0%	00:01.72	00:01.35	00:01.32	00:04.83	00:04.82
Kr-vs-kp	2,555	36	100%	00:00.55	00:00.41	00:00.39	00:01.41	00:01.46
Mushroom	6,498	22	100%	00:00.35	00:00.57	00:00.68	00:01.54	00:01.59
Hypothyroid	2,529	25	72%	00:00.34	00:00.23	00:00.24	00:00.42	00:00.42
Pbvwel	1,194	5	20%	00:00.02	00:00.06	00:00.05	00:00.19	00:00.18

Table 2: IREP++ and RIPPER training times. The percent categorical column indicates the fraction of features which are categorical.



(a)



(b)

Figure 2: Training times of RIPPER0 and IREP++ on synthetic data with continuous features (a) and categorical features (b). The line titled “RIPPER0 Negation” indicates training times for RIPPER0 with the negation option enabled.

results are summarized in Figures 2a and 2b. Since RIPPER0 is always faster than RIPPER2 we only present the results of RIPPER0 and IREP++ here. On the categorical data results for RIPPER0 with and without the negation option are presented. It is clear from these figures that IREP++ scales more efficiently than RIPPER and that the negation option improves the training times of RIPPER. From the figures it appears that both scale nearly linearly with IREP++ having a smaller slope. This is consistent with the analysis and results in [5].

**4.3 Number of Rules** Because of the differences in how RIPPER and IREP++ handle categorical features we expect IREP++ to be able to express the same target concept with fewer rules. Table 3 shows the number of rules learned by IREP++, RIPPER0, and RIPPER2 on all of the realistic data sets which contain categorical features except votes and hypothyroid. These two sets were omitted as votes contains only three values per categorical feature (yes, no, and undecided), and all the categorical features in the hypothyroid

Data Set	Number of Features	Number Categorical	RIPPER2 Rule Count	RIPPER2 Negation Rule Count	RIPPER0 Rule Count	RIPPER0 Negation Rule Count	IREP++ Rule Count
Credit	15	9	4	3	10	10	4
Defcon_categorical	17	10	23	12	20	12	8
KDD_cup	41	7	31	28	24	22	23
Kr-vs-kp	36	36	17	18	15	14	10
Monks-1	6	6	5	5	2	2	4
Monks-2	6	6	0	0	0	0	2
Monks-3	6	6	5	3	5	3	2
Mushroom	22	22	8	6	8	7	4
Pbvwel	5	1	6	6	2	2	5
Promoters	57	57	4	3	4	3	2

Table 3: Number of rules learned by IREP++, RIPPER0, and RIPPER2 on various data sets.

data set are boolean. We would not expect see the effects discussed in Section 3.2 on such data sets.

No variant of RIPPER learned *any* rules on the Monks-2 data set. If we exclude this data set from our analysis, then IREP++ learned an average of 4.56 fewer rules per problem than RIPPER2 without negation, and an average of 3.11 fewer rules per problem than RIPPER0 without negation. When negation is enabled RIPPER generally learns fewer rules. This is expected as the negation option allows a greater

expressive range for RIPPER rules and may thus allow the target concept to be expressed more efficiently. However, set-valued splits are more expressive than negated single-valued splits and so we would expect IREP++ to require fewer rules than RIPPER. The data confirms this: IREP++ requires an average of 2.44 fewer rules than RIPPER2 and an average of 1.44 fewer rules than RIPPER0 when negated single-valued predicates are permitted.

## 5 Summary

IREP++ is a new learning algorithm that builds on IREP and RIPPER. IREP++ contains several improvements that make it faster without sacrificing accuracy. These improvements include a new pruning metric, a simpler stopping criterion, and novel data structures that reduce the number of required sorts. We have also demonstrated a more efficient way to handle categorical features that allows for more compact rule sets and shorter training times. The resulting algorithm is as accurate as RIPPER2, but faster than RIPPER0. The speed improvements make it possible to use the algorithm on very large data sets in an interactive environment where fast results are required.

### A Proof that FOIL Gain is Sortable

Given a categorical feature,  $f$ , having  $m$  levels, we denote the levels of the feature as  $v_i$ . The  $v_i$  are ordered such that that  $p_1/n_1 \geq p_2/n_2 \geq \dots \geq p_m/n_m$  where  $p_i$  is the number of training patterns with value  $v_i$  where the target value is true, and  $n_i$  is the number of training patterns with value  $v_i$  where the target value is false. We wish to show that the subset of values that maximizes the FOIL gain metric always has the form  $S^* = \{v_1, v_2, \dots, v_k\}$  where  $k \leq m$ .

Each subset of levels,  $S$ , may be represented as a vector  $x_S$  whose  $i^{\text{th}}$  coordinate is 1 if  $v_i \in S$  and 0 otherwise. To prove the conjecture we temporarily revert to a continuous setting in which the values of  $x_S$  may instead take on any value in the interval  $[0, 1]$ . Let  $p(x) = \sum_{i=1}^m p_i x_i$  and let  $q(x) = \sum_{i=1}^m q_i x_i$  where  $q_i = p_i + n_i$ . Then we can write the FOIL gain as  $f(x) = p(x) (\log_2(p(x)/q(x)) - C)$  where  $C = \log_2(\sum_{i=0}^m p_i / \sum_{i=0}^m q_i)$ .

**DEFINITION A.1.** *An  $m$  dimensional vector  $x$  is said to be left shifted if its first  $j$  coordinates are 1, its last  $m - j - 1$  coordinates are 0 and its  $(j + 1)^{\text{st}}$  coordinate is between 0 and 1.*

**THEOREM A.1.** *For every non-negative real number  $p$ , the function  $f(x)$ , subject to the condition  $p(x) = p$ , is maximized when the vector  $x$  is left shifted.*

*Proof.* Since  $p(x) = p$ ,  $f(x) = p (\log_2(p/q(x)) + C)$ . Thus  $f(x)$  is maximized by minimizing  $q(x)$ . Assume  $x$  is not left shifted. Then there exists  $i < j$  such that  $x_i < 1$  and  $x_j > 0$ . We can then form  $x'$  by subtracting a small amount,

$\delta$ , from  $x_j$  and adding  $\frac{p_j}{p_i} \delta$  to  $x_i$ .  $x'$  maintains the condition  $p(x) = p$ . Furthermore, we know that

$$\frac{q(x') - q(x)}{p_j} = \frac{n_i}{p_i} \delta - \frac{n_j}{p_j} \delta \leq 0$$

since  $p_j/n_j \leq p_i/n_i$ . Thus we can be assured that  $q(x') \leq q(x)$ . Thus  $x'$  has FOIL gain greater than or equal to the FOIL gain of  $x$ . If  $x'$  is not left shifted we may repeat the procedure described above until we arrive at a left shifted vector.

In order to complete the proof of our conjecture we must show that all coordinates of the vector  $x$  that maximizes the FOIL gain must be either 0 or 1. Theorem A.1 shows that  $x$  must be left shifted thus we know that at most one coordinate of  $x$  is not 0 or 1. Theorem A.2 finishes our proof.

**THEOREM A.2.** *For  $y \in [0, 1]$ , let  $x_y = \langle 1, \dots, 1, y, 0, \dots, 0 \rangle$ , where the  $i^{\text{th}}$  coordinate of  $x_y$  is  $y$ , and let  $g(y) = f(x_y)$ . Then  $g(y) \leq \max\{g(0), g(1)\}$ .*

*Proof.* We show that the second derivative of  $g$  with respect to  $y$  is non-negative and thus  $g(y)$  has no local maxima. It follows that  $g(0)$  or  $g(1)$  is a maximum for  $y \in [0, 1]$ .

Let  $\alpha = (\sum_{j=1}^{i-1} p_j) / p_i$  and let  $\beta = (\sum_{j=1}^{i-1} q_j) / q_i$ . Since  $p_1/q_1 \geq p_2/q_2 \geq \dots \geq p_i/q_i$ , it follows that  $\alpha \geq \beta$ . Now we may write  $g(y) = p_i(\alpha + y) (\log_2 p_i(\alpha + y) - \log_2 q_i(\beta + y) + C)$ . Since the linear term involving  $C$  has no contribution to the second derivative, and since positive constant multipliers don't affect the sign, we see that the second derivative of  $g$  has the same sign as the second derivative of  $h(y) = (\alpha + y)(\ln(\alpha + y) - \ln(\beta + y))$ . Differentiating we get  $h'(y) = \ln(\alpha + y) + 1 - \ln(\beta + y) - (\alpha + y)/(\beta + y)$ . Then the second derivative is

$$\begin{aligned} h''(y) &= \frac{1}{\alpha + y} - \frac{1}{\beta + y} - \frac{1}{\beta + y} + \frac{\alpha + y}{(\beta + y)^2} \\ &= \frac{1}{(\alpha + y)(\beta + y)^2} \\ &\quad ((\beta + y)^2 - 2(\alpha + y)(\beta + y) + (\alpha + y)^2) \\ &= \frac{(\alpha - \beta)^2}{(\alpha + y)(\beta + y)^2} \geq 0 \end{aligned}$$

### Acknowledgments

The authors wish to extend their sincere thanks to Professor Madhu Sudan for providing the proof that appears in Appendix A and to Rich Lippmann for comments and suggestions that improved this paper.

## References

- [1] P. Alinat. Periodic progress report. Technical Report TS. AMS 93/S/EGS/NC/079, ROARS Project Esprit II, February 1993.
- [2] C.L. Blake and C.J. Merz. UCI repository of machine learning databases, 1998. <http://www.ics.uci.edu/~mllearn/MLRepository.html>.
- [3] Leo Breiman, Jerome H. Friedman, Richard A. Olshen, and Charles J. Stone. *Classification and Regression Trees*. Chapman and Hall/CRC, Boca Raton, FL, 1984.
- [4] Peter Clark and Tim Niblett. The CN2 induction algorithm. *Machine Learning*, 3:261–283, 1989.
- [5] William W. Cohen. Fast effective rule induction. In *Proceedings of the Twelfth International Conference on Machine Learning*, Lake Tahoe, California, 1995.
- [6] Thomas G. Dietterich. Machine learning research: Four current directions. *AI Magazine*, 18:97–136, 1997.
- [7] Johannes Fürnkranz and Gerhard Widmer. Incremental reduced error pruning. In *Machine Learning: Proceedings of the Eleventh Annual Conference*, New Brunswick, New Jersey, 1994. Morgan Kaufmann.
- [8] S. Hettich and S. D. Bay. The UCI KDD archive, 1999. <http://kdd.ics.uci.edu>.
- [9] Richard P. Lippman, Linda Kukulich, et al. LNKnet: Neural network, machine learning, and statistical software for pattern classification. *Lincoln Laboratory Journal*, 6(2):249–268, 1993.
- [10] Manish Mehta, Rakesh Agrawal, and Jorma Rissanen. "SLIQ: A fast scalable classifier for data mining. In *Proc. Of the Fifth Int'l Conference on Extending Database Technology*, pages 18–32, March 1996.
- [11] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.
- [12] J. R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5:239–266, 1990.
- [13] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers, 1993.