A Simple, Fast, and Effective Rule Learner

William W. Cohen       Yoram Singer
AT&T Shannon Laboratory
180 Park Avenue
Florham Park, NJ 07932-0971 USA
{wcohen,singer}@research.att.com

Abstract
We describe SLIPPER, a new rule learner that generates rulesets by repeatedly boosting a simple, greedy, rule-builder. Like the rulesets built by other rule learners, the ensemble of rules created by SLIPPER is compact and comprehensible. This is made possible by imposing appropriate constraints on the rule-builder, and by use of a recently-proposed generalization of AdaBoost called confidence-rated boosting. In spite of its relative simplicity, SLIPPER is highly scalable, and an effective learner. Experimentally, SLIPPER scales no worse than \( O(n \log n) \), where \( n \) is the number of examples, and on a set of 32 benchmark problems, SLIPPER achieves lower error rates than RIPPER 20 times, and lower error rates than C4.5rules 22 times.

Introduction
Boosting (Schapire 1990; Freund 1995; Freund & Schapire 1997) is usually used to create ensemble classifiers. It is popular because it is simple, easy to implement, well-understood formally, and effective at improving accuracy. One disadvantage of boosting is that improvements in accuracy are often obtained at the expense of comprehensibility. If comprehensibility is important, it is more appropriate to use some learner that produces a compact, understandable hypothesis—for instance, a rule learning system like CN2 (Clark & Niblett 1989), RIPPER (Cohen 1995), or C4.5rules (Quinlan 1994). However, the rule learning systems that perform best experimentally have the disadvantage of being complex, hard to implement, and not well-understood formally.

Here, we describe a new rule learning algorithm called SLIPPER.\(^1\) SLIPPER generates rulesets by repeatedly boosting a simple, greedy, rule-builder. SLIPPER's rule-builder is much like the inner loops of RIPPER (Cohen 1995) and IREP (Fürnkranz & Widmer 1994). However, SLIPPER does not employ the "set-covering" process used by conventional rule learners—rather than removing examples covered by a new rule, SLIPPER uses boosting to reduce the weight of these examples.

Like the rulesets constructed by RIPPER and other rule learners, SLIPPER's rulesets have the desirable property that the label assigned to an instance depends only on the rules that "fire" for that instance. This property is not shared by earlier applications of boosting to rule learning (see for instance (Freund & Schapire 1996)), in which the behavior of the entire ensemble of rules can affect an instance's classification. This property makes classifications made by the rulesets easier to understand, and is made possible by imposing appropriate constraints on the base learner, and use of a recently-proposed generalization of AdaBoost (Schapire & Singer 1998).

SLIPPER is simpler and better-understood formally than other state-of-the-art rule learners. In spite of this, SLIPPER scales well on large datasets, and is an extremely effective learner. Experimentally, SLIPPER's run-time on large real-world datasets scales no worse than \( O(n \log n) \), where \( n \) is the number of examples. On a set of 32 benchmark problems, SLIPPER achieves lower error rates than RIPPER 20 times, and lower error rates than C4.5rules 22 times. The rulesets produced by SLIPPER are also comparable in size to those produced by C4.5rules.

The SLIPPER Algorithm
SLIPPER uses boosting to create an ensemble of rules. The weak learner that is boosted finds a single rule, using essentially the same process as used in the inner loops of IREP (Fürnkranz & Widmer 1994) and RIPPER (Cohen 1995). Specifically, the weak learner splits the training data, grows a single rule using one subset of the data, and then prunes the rule using the other subset. In SLIPPER, the ad hoc metrics used to guide the growing and pruning of rules are replaced with metrics based on the formal analysis of boosting algorithms. The specific boosting algorithm used is a generalization of Freund and Schapire's AdaBoost (Freund & Schapire 1997) that employs confidence-rated predictions (Schapire & Singer 1998). This generalization allows the rules generated by the weak learner to "abstain" (vote with confidence zero) on examples not covered by the rule, and vote with an appropriate non-zero confidence on covered examples.

\(^1\)For Simple Learner with Iterative Pruning to Produce Error Reduction.
The current implementation of SLIPPER only handles two-class classification problems. The output of SLIPPER is a weighted ruleset, in which each rule $R$ is associated with a confidence $C_R$. To classify an instance $x$, one computes the sum of the confidences of all rules that cover $x$, then predicts according to the sign of this sum: if the sum is greater than zero, one predicts the positive class. In order to make the ruleset more comprehensible, we further constrain SLIPPER to generate only rules that are associated with a positive confidence rating—that is, all rules predict membership in the positive class. The only rule with a negative confidence rating (i.e., that predicts membership in the negative class) is a single default rule. This representation is a generalization of propositional DNF, and is similar to that used by many other rule learners: for most rule learners the classifier is a set of rules, often with some associated numerical confidence measure, and often with some sort of voting scheme for resolving possible conflicts in the predictions.

Below, we describe the SLIPPER algorithm in detail.

### Boosting Confidence-rated Rules

The first boosting algorithms (Schapire 1990; Freund 1995) were developed for theoretical reasons—to answer certain fundamental questions about PAC-learnability (Kearns & Valiant 1994). While mathematically beautiful, these two algorithms were rather impractical. Later, Freund and Schapire (1997) developed the AdaBoost algorithm, which proved to be a practically useful meta-learning algorithm. AdaBoost works by making repeated calls to a weak learner. On each call the weak learner generates a single weak hypothesis, after which the examples are re-weighted. The weak hypotheses are combined into an ensemble called a strong hypothesis.

Recently, Schapire and Singer (1998) studied a generalization of AdaBoost, in which a weak-hypothesis can assign a real-valued confidence to each prediction. The weak-hypothesis can assign different confidences to different instances, and in particular, it can “abstain” on some instances by making a prediction with zero confidence. The ability to abstain is important for our purposes. We now give a brief overview of this extended boosting framework and describe how it is used for constructing weighted rulesets. Since we have thus far implemented only a two-class version of SLIPPER, we will focus on the two-class case; however, the theory extends nicely to multiple classes.

Assume that we are given a set of examples $\{(x_1, y_1), \ldots, (x_m, y_m)\}$ where each instance $x_i$ belongs to a domain $\mathcal{X}$ and each label $y_i$ is in $\{-1, +1\}$. Assume also that we have access to a weak learning algorithm, which accepts as input the training examples along with a distribution over the instances (initially uniform). In the generalized boosting setting, the weak learner computes a weak hypothesis $h$ of the form $h : \mathcal{X} \rightarrow \mathbb{R}$, where the sign of $h(x)$ is interpreted as the predicted label and the magnitude $|h(x)|$ as the confidence in the prediction: large numbers for $|h(x)|$ indicate high confidence in the prediction, and numbers close to zero indicate low confidence. The weak hypothesis can abstain from predicting the label of an instance $x$ by setting $h(x) = 0$.

Figure 1: A generalized version of AdaBoost with real-valued predictions (Schapire & Singer 1998).

Given: $$(x_1, y_1), \ldots, (x_m, y_m) ; \; x_i \in \mathcal{X}, y_i \in \{-1, +1\}$$

Initialize $D_1(i) = 1/m$.

For $t = 1, \ldots, T$:

- Train weak learner using distribution $D_t$.
- Get weak hypothesis $h_t : \mathcal{X} \rightarrow \mathbb{R}$.
- Choose $\alpha_t \in \mathbb{R}$.
- Update: $D_{t+1}(i) = D_t(i) \exp(-\alpha_t h_t(x_i))/Z_t$.

Output final hypothesis: $H(x) = \text{sign} \left( \sum_{t=1}^{T} \alpha_t h_t(x) \right)$.
rewritten in our case as
\[ Z = \sum_{x \in R} D(i) + \sum_{x \in R} D(i) \exp(-y_i C_R), \tag{1} \]
where \( C_R = \alpha h(x) \). Let \( W_0 = \sum_{x \in R} D(i) \), \( W_+ = \sum_{x \in R, y_i = +1} D(i) \), and \( W_- = \sum_{x \in R, y_i = -1} D(i) \). We can now further simplify Eqn. (1) and rewrite \( Z \) as
\[ Z = W_0 + W_+ \exp(-C_R) + W_- \exp(+C_R). \tag{2} \]
Following Schapire and Singer (1998), to find \( C_R \) need to solve the equation \( \frac{\partial Z}{\partial C_R} = 0 \), which implies that \( Z \) is minimized by setting
\[ C_R = \frac{1}{2} \ln \left( \frac{W_+}{W_-} \right). \tag{3} \]
Since a rule may cover only a few examples, \( W_- \) can be equal to 0, leading to extreme confidence values: to prevent this, in practice, we “smooth” the confidence by adding \( \frac{1}{2n} \) to both \( W_+ \) and \( W_- \):
\[ C_R = \frac{1}{2} \ln \left( \frac{W_+ + 1/(2n)}{W_- + 1/(2n)} \right). \tag{4} \]
The smoothed confidence value of any rule \( R \) is therefore bounded from above by \( \frac{1}{2} \ln(2n) \).

The analysis of Singer and Schapire also suggests an objective function to be used by the weak-learner which constructs rules. Plugging the value of \( C_R \) into Eqn. (2) we get that
\[ Z = W_0 + 2\sqrt{W_+ W_-} = 1 - \left( W_+ - 2\sqrt{W_+ W_-} + W_- \right) = 1 - \left( \sqrt{W_+} - \sqrt{W_-} \right)^2. \tag{5} \]
Thus, a rule \( R \) minimizes \( Z \) iff it maximizes \( |\sqrt{W_+} - \sqrt{W_-}| \). Note that a rule which minimizes \( Z \) by maximizing \( \sqrt{W_-} - \sqrt{W_+} \) may be negatively correlated with the positive class, and hence its confidence value \( C_R \) is negative. As described earlier, in SLIPPER we restrict ourselves to positively correlated rules, hence the objective function we attempt to maximize when searching for a good rule is
\[ Z = \sqrt{W_+} - \sqrt{W_-}. \tag{6} \]

In summary, this use of boosting corresponds roughly to the outer “set-covering” loop found in many rule learners (Pagallo & Haussler 1990; Quinlan 1990; Brunk & Pazzani 1991; Fürnkranz & Widmer 1994; Cohen 1995). The major difference is that examples covered by a rule are not immediately removed from the training set. Instead, covered examples are given lower weights; further, the degree to which an example's weight is reduced depends on the accuracy of the new rule. The formal analysis of boosting given by Schapire and Singer also suggests a new quality metric for rules: notice that \( Z \) encompassed a natural trade-off between accuracy (the proportion of the positive examples satisfied by a rule to the total number of examples that the rule satisfies) and coverage (the fraction of examples that satisfy the rule).

Below, we will discuss how to construct rules based on the objective function \( Z \) as given by Eqn. (6).

**Rule growing and pruning**

We will now describe the weak-learner which generates individual rules. This procedure is similar to the heuristic rule-building procedure used in RIPPER (Cohen 1995) and IREP (Fürnkranz & Widmer 1994).

The rule-builder begins by randomly splitting the dataset into two disjoint subsets, GrowSet and PruneSet. The split is constrained so that the total weight of examples in GrowSet is about 2/3.

The rule-builder then invokes the GrowRule routine. GrowRule begins with an empty conjunction of conditions, and considers adding to this conjunction any condition in one of the following forms: \( A_n = v \), where \( A_n \) is a nominal attribute and \( v \) is a legal value for \( A_n \); or \( A_n \leq \theta \) or \( A_n \geq \theta \), where \( A_n \) is a continuous variable and \( \theta \) is some value for \( A_n \) that occurs in the training data. GrowRule then adds the condition that attains the maximal value for \( Z_t \) on GrowSet. This process is repeated until the rule covers no negative examples from GrowSet, or no further refinement improves \( Z_t \).

---

**Given:** \((x_1, y_1), \ldots, (x_m, y_m)\); \( x_i \in \mathcal{X}, y_i \in \{-1, +1\} \)

**Initialize:** \( D(i) = 1/m \).

**For** \( t = 1, \ldots, T \):

1. **Train the weak-learner using current distribution** \( D \):
   a. Split data into GrowSet and PruneSet.
   b. **GrowRule**: starting with empty rule, greedily add conditions to maximize Eqn. (6).
   c. **PruneRule**: starting with the output of GrowRule, delete some final sequence of conditions to minimize Eqn. (7), where \( C_R \) is computed using Eqn. (4) and GrowSet.
   d. Return as \( R_t \) either the output of PruneRule, or the default rule, whichever minimizes Eqn. (5).

2. **Construct** \( h_t : \mathcal{X} \rightarrow \mathbb{R} \):
   a. For each \( x_i \in R_t \), set \( D(i) \leftarrow D(i) / \exp(y_i \cdot C_R) \).
   b. Let \( Z_t = \sum_{i=1}^m D(i) \).
   c. For each \( x_i \), set \( D(i) \leftarrow D(i) / Z_t \).

**Output final hypothesis:** \( H(x) = \text{sign} \left( \sum_{R_t : x \in R_t} \hat{C}_R \right) \)

---

**Figure 2: The SLIPPER algorithm**

---

accuracy (the proportion of the positive examples satisfied by a rule to the total number of examples that the rule satisfies) and coverage (the fraction of examples that satisfy the rule).
This rule is often specific, and "overfits" the training data; thus the resulting rule is immediately pruned using the PruneRule routine. PruneRule considers deleting any final sequence of conditions from the rule. Each sequence of deletions defines a new rule whose goodness is evaluated on PruneSet. As before, each candidate rule $R'$ partitions the PruneSet into two subsets, depending on whether or not $R'$ is satisfied. Similar to the definition of $W_+$ and $W_-$, let $V_+$ (respectively $V_-$) be the total weight of the examples in PruneSet that are covered by $R'$ and labeled +1 (respectively −1). Denote by $\hat{C}_R$ the (smoothed) prediction confidence obtained by evaluating Eq. (4) on the $W_+, W_-$ associated with GrowSet. PruneRule minimizes the formula

$$(1 - V_+ - V_-) + V_+ \exp(-\hat{C}_R) + V_- \exp(+\hat{C}_R), \quad (7)$$

This can be interpreted as the loss (as defined by Singer and Schapire) of the rule $R'$, with associated confidence $\hat{C}_R$, as estimated on the examples in PruneSet.

Subject to the limitations of this greedy, incomplete search procedure, this rule will have a low Z score. It is also guaranteed to be positively correlated with the positive class. We also allow a default rule (a rule that is satisfied for all examples) to be used in a hypothesis—indeed, without such a rule, it would be impossible for the strong-hypothesis to classify any instances as negative. The rule-builder will thus return to the booster either the output of PruneRule, or the default rule—whichever rule has the lowest Z value, as determined by Eq. (5). (This behavior is different from other rule-learners, which typically add a single default rule after all other rules have been learned.)

Note that the value of Eq. (7) and the confidence value $\hat{C}_R$ which was calculated on GrowSet is used only in the weak-learner search for a good rule—the booster will assign a confidence using Eq. (4) on the entire dataset.

Pseudo-code for SLIPPER is given in Figure 2.

Other details

It is possible for the weak-learner to generate the same rule several times—for instance, the default rule is often generated many times during boosting. Therefore, after the last round of boosting, the final strong-hypothesis is "compressed" by removing duplicate rules. Specifically, if the strong-hypothesis contains a set of identical rules $R_1$, $R_2$, ..., $R_k$, these are replaced by a single rule $R'$ with confidence $C_R' = \sum_{i=1}^{k} C_{R_i}$. This step reduces the size of the strong-hypothesis, thus reducing classification time and improving comprehensibility.

As described above, SLIPPER has one free parameter—the number of rounds of boosting $T$. Although there are theoretical analyses of the number of rounds needed for boosting (Freund & Schapire 1997; Schapire et al. 1997), these tend not to give practically useful bounds. Therefore, we use internal five-fold cross-validation (on the training set) to fix $T$. Five training/holdout divisions of the data are created in the usual way, and the algorithm of Figure 2 is run five times for $T_{max}$ rounds on each training set (where $T_{max}$ is an upper bound set by the user). The number of rounds $T^*$ which produces the lowest average error on the holdout data is then determined, breaking ties in favor of smaller values of $T^*$, and the algorithm is finally run again for $T^*$ rounds on the entire dataset. In the experiments below, we always used a value of $T_{max} = 100$.

Experiments

To evaluate SLIPPER, we used two sets of benchmark problems, each containing 16 two-class classification problems. The first set, the development set, was used in debugging SLIPPER and evaluating certain variations of it. The second set, the prospective set, was used as a secondary evaluation of the SLIPPER algorithm, after development was complete. This two-stage procedure was intended as a guard against the possibility of "overfitting" the benchmark problems themselves; however, since the experimental results are qualitatively similar on both the development and prospective sets, we will focus on results across all 32 benchmark problems in the discussion below. These results are summarized in Table 2 and Figure 3, and presented in more detail in Table 1.

The benchmark problems are summarized in Table 1. The problems from the development set are discussed elsewhere (Cohen 1995). The problems in the prospective set are taken without modification from the UC/Irvine repository (Blake, Keogh, & Merz 1989), with these exceptions: the hypothyroid and splice-junction problems were artificial made two-class problems—in each case, the goal is to separate most frequent class from the remaining classess; for adults, we used a 5000-element subsample of the designated training set; and market1 and market2 are real-world customer modeling problems provided by AT&T.

To measure generalization error, we used a designated test set, when available; a single random partition of the training set, for the larger problems; and stratified 10-fold cross-validation otherwise, as indicated.

We compared SLIPPER's performance to RIPPER (Cohen 1995), with and without its "optimization" step; the C4.5 decision-tree learner (Quinlan 1994), with pruning, and the C4.5rules rule learner (henceforth, C4rules); and the C5.0 rule learner$^3$ (henceforth, C5rules), a proprietary, unpublished descendent of C4rules. RIPPER without optimization is identical, but the output of PruneRule, or the default rule—whichever rule has the lowest Z value, as determined by Eq. (5). (This behavior is different from other rule-learners, which typically add a single default rule after all other rules have been learned.)

Note that this step does not alter the actual predictions of the learned ruleset. Other approaches that perform "lossy" compaction of the strong hypothesis by, for instance, deleting rules associated with low confidence values, might lead to better generalization error (see for instance (Margineantu & Dietterich 1997)) but are beyond the scope of this this paper.

$^3$That is, C5.0 run with the -r option.
The average rank of SLIPPER is 2.0, compared to 2.6 for RIPPER and C5rules, and 2.8 for C4rules. Summaries of the experimental results are given in Figure 3 and Table 2. In the scatterplot of Figure 3, each point compares SLIPPER to some second learner. The x-axis position is the error rate of L. Thus, points above the lines y = x correspond to datasets for which SLIPPER is better than the second learner, and vice versa.
PER performs better than some second learner. Visual inspection confirms that SLIPPER often substantially outperforms each of the other rule learners, and that its performance is almost always close to the best of the other rule learners.\(^5\)

In Table 2, let \(L_R\) be the learner corresponding to a row of the table, and let \(L_C\) correspond to a column. The upper triangle entries are the average, across all benchmarks, of the quantity error(\(L_C\))/error(\(L_R\)); for instance, the entries of the fourth column indicate that SLIPPER’s error rate is, on average, about 2% to 4% lower than the other rule learners. The lower triangle entries are the won-loss-tied record of learner \(L_R\) versus \(L_C\), a “win” indicating \(L_R\) achieved a lower error rate. A record is underlined if it is statistically significant at the 90% level, and bold-faced if it is statistically significant at the 95% level.\(^6\) For instance, the first entry of the fourth row indicates that SLIPPER achieves a lower error rate than RIPPER 20 times, a higher error rate 9 times, and the same error rate 3 times. SLIPPER’s rules versus C4rules and C5rules are similar. The last two lines of the table give SLIPPER’s won-loss-tied records for the development set and prospective set only, indicating that these results are generally comparable across both test sets. (An exception is SLIPPER’s performance versus C5rules: it appears to be superior on the development set, but only comparable on the prospective set.)

We also measured the size of the rulesets produced by the different algorithms.\(^7\) The most compact rulesets are produced by RIPPER: the average size of RIPPER’s rulesets is 6.0 rules (or 8.1 without optimization), and RIPPER virtually always produces the smallest ruleset.\(^8\) The remaining three learners produce similar sized rulesets, with SLIPPER tending to produce somewhat smaller rulesets than the other two. The average size rulesets for C4rules, C5rules, and SLIPPER are 22.1 rules, 30.7 rules, and 17.8 rules, respectively, and the respective average ranks among these three are 1.8, 2.3, and 1.9. The largest ruleset produced by SLIPPER is 49 rules (for coding).

Finally, we evaluated the scalability of the rule learners on several large datasets. We used adult, blackjack, with the addition of 20 irrelevant noise variables; and market3, for which many examples were available. C4rules was not run, since it is known to have scalability problems (Cohen 1995). The results are shown in the log-log plots of Figure 4.\(^9\) The fastest rule learner for these datasets is usually C5rules, followed by the RIPPER variants. SLIPPER (at least in the current implementation) is much slower than either C5rules or RIPPER; however, it scales very well with increasing amounts of data. In absolute terms, SLIPPER’s performance is still quite reasonable: SLIPPER needs 1-2 hours to process 100,000 examples of the blackjack+ and market3 datasets, and 30 minutes to process the 30,000 training examples from the adult dataset.

To summarize, SLIPPER obtains the lowest error rates on average. SLIPPER also scales well to large datasets, although it is somewhat less efficient than C5rules and RIPPER. SLIPPER’s rulesets are comparable in size to those of C4rules and C5rules, although somewhat larger than RIPPER’s.

## Concluding remarks

We have described SLIPPER, a new rule learning algorithm which uses confidence-rated boosting to learn

---

\(^5\)The sole exception to this is network2, on which SLIPPER performs noticeably worse than the other methods.

\(^6\)That is, if one can reject the null hypothesis that the probability of a win is 0.50, given there is no tie, with a two-tailed binomial test.

\(^7\)In the 10-CV experiments, we looked at the size of the ruleset generated by running on all the data, not the average of the cross-validation runs.

\(^8\)However, it has been argued that RIPPER over-prunes on the sort of the smaller problems that predominate in the UC/Irvine repository (Frank & Witten 1998).

\(^9\)Timing results are given in CPU seconds on a MIPS Irix 6.3 with 200 MHz R10000 processors.
an ensemble of rules. Although the SLIPPER algorithm is relatively simple, SLIPPER performs well on a set of 32 benchmark problems: relative to RIPPER, SLIPPER achieves lower error rates 20 times, and the same error rate 3 times; relative to C4.5rules, SLIPPER achieves lower error rates 22 times, and the same rate 2 times; and relative to C5.0rules, SLIPPER achieves lower error rates 19 times, and the same rate 2 times. Using a two-tailed sign test, these differences between RIPPER, C4.5rules, and C5.0rules are significant at 94%, 98%, and 80% levels respectively. SLIPPER also performs best among these three systems according to several measures of aggregate performance, such as average rank. SLIPPER's rulesets are of moderate size—comparable to those produced by C4.5rules and C5.0rules—and the algorithm also scales well on large datasets.

As noted above, SLIPPER is based on two lines of research. The first line of research is on scalable, noise-tolerant separate-and-conquer rule learning algorithms (Pagallo & Haussler 1990; Quinlan 1990), such as reduced error pruning (REP) for rules (Brunk & Pazzani 1991),IREP (Fürnkranz & Widmer 1994), and RIPPER (Cohen 1995). The second line of research is on boosting (Schapire 1990; Freund 1995), in particular the AdaBoost algorithm (Freund & Schapire 1997), and its recent successor developed by Schapire and Singer (1998).

SLIPPER is similar to an earlier application of boosting to rule learning (Freund & Schapire 1996), in which AdaBoost was used to boost a rule-builder called FindDecRule. In contrast to SLIPPER, Freund and Schapire used a heuristic based on an information gain criterion that has no formal guarantees. SLIPPER also places a greater emphasis on generating comprehensible rulesets; in particular, SLIPPER generates relatively compact rulesets, and SLIPPER's use of confidence-rated boosting allows it to construct rules that "abstain" on instances that are not covered by a rule; thus the label assigned to an instance depends only on the rules that "fire" for that instance. In Freund and Schapire's rule boosting algorithm, in contrast, the label for an instance always depends on all the rules in the ensemble. The algorithm also always generates a ruleset of fixed size (in their experiments, 100 rules).

SLIPPER's use of boosting is a departure from the separate-and-conquer approach used by many earlier rule learners. Another alternative is the RISE algorithm (Domingos 1996), which combines rule learning and nearest-neighbour classification using a bottom-up "conquering without separating" control structure. However, the ruleset constructed by RISE is somewhat more difficult to interpret, since the label assigned to an instance depends not on the rules that cover it, but on the rule that is "nearest".

More recently, Hsu, Etzioni, and Soderland (1998) described an experimental rule learner called DAIRY which extends the set-covering approach of traditional rule learners by "recycling" examples—that is, by reducing the weight of examples that have been "covered" by previous rules, rather than removing these examples. DAIRY's recycling method was shown experimentally to improve performance on a number of text classification problems. SLIPPER's combination of boosting and rule-building is similar to recycling, and could be viewed as a formally justified variant of it.

We note that there are important practical advantages to using learning methods that are formally well understood. For instance, existing formal analysis (Schapire & Singer 1998) generalizes the boosting method used here to multi-class learning problems, and also to a setting in which misclassification costs are unequal. In further work, we plan to implement a multi-class version of SLIPPER, and an extension of SLIPPER for minimizing an arbitrary cost matrix, which maps each pair of (predicted label, correct label) to an associated cost. We also plan to evaluate SLIPPER on text classification benchmarks: the current implementation of SLIPPER, which is based on code used in RIPPER, inherits from RIPPER the ability to handle text efficiently.

Acknowledgments
We would like to thank Rob Schapire for helpful discussions, and Haym Hirsh for comments on a draft of this paper.

Figure 4: Run-time performance of SLIPPER, RIPPER, and C5rules on large datasets.
References


