A Simple, Fast, and Effective Rule Learner

William W. Cohen **Yoram Singer**
AT&T Labs–Research 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 (for Simple Learner with Iterative Pruning to Produce Error Reduction). 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

Copyright © 1999, American Association for Artificial Intelligence. All rights reserved.

"abstain" (vote with confidence zero) on examples not covered by the rule, and vote with an appropriate non-zero confidence on covered examples.

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 $\langle(x_1, y_1), \ldots, (x_m, y_m)\rangle$ 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 $h : \mathcal{X} \to \mathbb{R}$, where

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 $h_t : \mathcal{X} \to \mathbb{R}$.
- Choose $\alpha_t \in \mathbb{R}$.
- Update: $D_{t+1}(i) = D_t(i) \exp(-\alpha_t y_i h_t(x_i))/Z_t$

Output final hypothesis: $H(x) = \text{sign}\left(\sum_{t=1}^{T} \alpha_t h_t(x)\right)$

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

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 $h(x) = 0$. Pseudo-code describing the generalized boosting algorithm is given in Figure 1; here Z_t is a normalization constant that ensures the distribution D_{t+1} sums to 1, and α_t depends on the weak-learner.

The weak-hypotheses that we use here are *rules*. In SLIPPER, rules are conjunctions of primitive conditions. As used by the boosting algorithm, however, a rule R can be any hypothesis that partitions the set of instances $\mathcal{X}$ into two subsets: the set of instances which satisfy (are covered by) the rule, and those which do not satisfied the rule. If x satisfies R, we will write $x \in R$.

In order to make the strong-hypothesis similar to a conventional ruleset, we will force the weak-hypothesis based on a rule R to abstain on all instances unsatisfied by R, by setting the prediction $h(x)$ for $x \notin R$ to 0. We will also force the rules to to predict with the same confidence C_R on every $x \in R$; in other words, for the t-th rule R_t generated by the weak learner, we will require that $\forall x \in R_t, \alpha_t h_t(x) = C_{R_t}$. Thus, to classify an instance x with the strong-hypothesis, one simply adds up the confidence C_{R_t} for each rule R_t that is satisfied by x, and predicts according to the sign of this sum. As a final constraint, we will require each rule R to be in one of two forms: either R is a "default rule" (i.e., $x \in \mathcal{X} \Rightarrow x \in R$) or else R is such that C_R is positive. Thus each non-default rule R is associated with a single real-valued confidence C_R, and can be interpreted as follows: if R is satisfied then predict class "positive" with confidence C_R, and otherwise abstain.

In Figure 1, Z_t is a real value used to normalize the distribution: $Z_t = \sum_i D_t(i) \exp(-\alpha_t y_i h_t(x_i))$. Thus Z_t depends on both h_t and α_t. Schapire and Singer (1998) showed that to minimize training error, the weak-learning algorithm should pick, on each round of boosting, the weak hypothesis h_t and weight α_t which lead to the smallest value of Z_t. Assume that a rule R has been generated by the weak learner. We will now show how the confidence value C_R for rule R can be set to minimize Z_t. Omitting the dependency on t, Z can rewritten in our case as

$$Z = \sum_{x_i \notin R} D(i) + \sum_{x_i \in R} D(i) \exp(-y_i C_R), \qquad (1)$$

where $C_R = \alpha h(x)$. Let $W_0 = \sum_{x_i \notin R} D(i)$, $W_+ = \sum_{x_i \in R: y_i = +1} D(i)$, and $W_- = \sum_{x_i \in R: y_i = -1} D(i)$. We can now further simplify Equ. (1) and rewrite Z as

$$Z = W_0 + W_+ \exp(-C_R) + W_- \exp(+C_R) \ . \qquad (2)$$

Following Schapire and Singer (1998), to find C_R we need to solve the equation $\frac{dZ}{dC_R} = 0$, which implies that Z is minimized by setting

$$C_R = \tfrac{1}{2} \ln \left(\frac{W_+}{W_-} \right) \ . \qquad (3)$$

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_-:

$$\hat{C}_R = \tfrac{1}{2} \ln \left(\frac{W_+ + 1/(2n)}{W_- + 1/(2n)} \right) \qquad (4)$$

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 Equ. (2) we get that

$$\begin{aligned} Z &= W_0 + 2\sqrt{W_+ W_-} \\ &= 1 - \left(W_+ - 2\sqrt{W_+ W_-} + W_- \right) \\ &= 1 - \left(\sqrt{W_+} - \sqrt{W_-} \right)^2 \ . \qquad (5) \end{aligned}$$

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

$$\tilde{Z} = \sqrt{W_+} - \sqrt{W_-} \ . \qquad (6)$$

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 $\tilde{Z}$ 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 $\tilde{Z}$ as given by Equ. (6).

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 Equ. (6).
   (c) `PruneRule`: starting with the output of `GrowRule`, delete some final sequence of conditions to minimize Equ. (7), where $\hat{C}_{R'}$ is computed using Equ. (4) and `GrowSet`.
   (d) Return as R_t either the output of `PruneRule`, or the default rule, whichever minimizes Equ. (5).
2. *Construct $h_t : \mathcal{X} \to \mathbb{R}$:*
   Let $\hat{C}_{R_t}$ be given by Equ. (4) (evaluated on the entire dataset). Then
   $$h_t(x) = \begin{cases} \hat{C}_{R_t} & \text{if } x \in R_t \\ 0 & \text{otherwise} \end{cases}$$
3. *Update:*
   (a) For each $x_i \in R_t$, set $D(i) \leftarrow D(i) / \exp(y_i \cdot \hat{C}_{R_t})$
   (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: $H(x) = \text{sign}\left(\sum_{R_t : x \in R_t} \hat{C}_{R_t} \right)$

Figure 2: The SLIPPER algorithm

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 $A_c \leq \theta$ or $A_c \geq \theta$, where A_c is a continuous variable and θ is some value for A_c that occurs in the training data. `GrowRule` then adds the condition that attains the maximal value for $\tilde{Z}_t$ on `GrowSet`. This process is repeated until the rule covers no negative examples from `GrowSet`, or no further refinement improves $\tilde{Z}_t$.

This rule is often too 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 Equ. (4) on the W_+, W_- associated with `GrowSet`. `PruneRule` minimizes the formula

$$(1 - V_+ - V_-) + V_+ \exp\left(-\hat{C}_{R'}\right) + V_- \exp\left(+\hat{C}_{R'}\right) . \quad (7)$$

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 Equ. (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 Equ. (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 Equ. (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 $R_1, \ldots, R_k$, 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.[1]

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 sets (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 artificially made two-class problems—in each case, the goal is to separate most frequent class from the remaining classes; for `adult`, 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 (hence-

[1]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.

Problem Name	Source	#Train	#Test	#Feat	RIPPER −opt	RIPPER +opt	C4.5 Trees	C4.5 Rules	C5.0 Rules	SLIPPER
Prospective:										
adult	uci	5000	16281	14	17.2	16.0	16.0	15.0	15.1	14.7
blackjack	att	5000	10000	4	29.1	29.1	27.9	28.0	27.8	27.9
market2	att	5000	6000	68	43.1	41.3	45.5	43.1	41.4	42.6
market3	att	5000	15000	4	9.1	8.6	9.5	9.3	8.6	8.9
splice-junction	uci	2190	1000	60	6.5	5.9	4.3	4.8	4.5	5.9
hypothyroid	uci	2514	1258	29	1.0	0.9	0.4	0.4	0.4	0.7
breast-wisc	uci	699	10CV	9	3.7	4.6	6.6	5.2	5.0	4.2
bands	uci	540	10CV	39	28.3	27.0	30.0	30.0	30.2	22.8
crx	uci	690	10CV	15	15.5	15.2	14.2	15.5	14.0	15.7
echocardiogram	uci	74	10CV	12	2.9	5.5	5.5	6.8	4.3	4.3
german	uci	1000	10CV	20	28.6	28.7	27.5	27.0	28.3	27.2
hepatitis	uci	155	10CV	19	20.7	23.2	18.8	18.8	20.1	17.4
heart-hungarian	uci	294	10CV	13	19.7	20.1	20.8	20.0	21.8	19.4
ionosphere	uci	351	10CV	34	10.3	10.0	10.3	10.3	12.3	7.7
liver	uci	345	10CV	6	32.7	31.3	37.7	37.5	31.9	32.2
horse-colic	uci	300	10CV	23	17.0	16.3	16.3	16.0	15.3	15.0
Development:										
mushroom	uci	3988	4136	22	0.2	0.0	0.2	0.2	0.7	0.2
vote	uci	300	135	16	3.7	3.0	3.0	5.2	3.0	3.0
move	att	1483	1546	10	35.1	29.3	27.8	25.3	26.8	23.9
network1	att	2500	1077	30	25.0	25.7	28.4	26.6	26.3	25.1
network2	att	2600	1226	35	21.7	21.3	23.3	22.3	22.9	26.6
market1	att	1565	1616	10	23.4	22.3	21.8	20.5	21.2	20.1
weather	att	1000	4597	35	28.5	28.9	31.3	28.7	29.2	28.7
coding	uci	5000	15000	15	34.3	32.8	34.1	32.6	32.4	30.2
ocr	att	1318	1370	576	3.0	3.4	3.6	3.6	4.7	2.0
labor	uci	57	10CV	16	18.0	18.0	14.7	14.7	18.4	12.3
bridges	uci	102	10CV	7	13.7	13.7	15.7	17.5	15.7	13.7
promoters	uci	106	10CV	57	18.1	19.0	22.7	18.1	22.7	18.9
sonar	uci	208	10CV	60	29.8	24.2	30.3	29.8	28.3	25.5
ticket1	att	556	10CV	78	1.8	1.6	1.6	1.6	1.6	2.7
ticket2	att	556	10CV	53	6.3	4.7	4.2	4.9	4.2	4.5
ticket3	att	556	10CV	61	4.5	3.2	2.9	3.4	2.9	4.3
Average: Prospective Set					17.83	17.75	18.20	17.97	17.55	16.65
Average: Development Set					16.70	15.70	16.60	15.93	16.31	15.11
Average: All Problems					17.26	16.72	17.40	16.95	16.93	15.88
Average Rank: All Problems					4.05	3.36	4.06	3.59	3.41	2.53
#Lowest Error Rates: All Problems					6	9	6	3	8	13

(Columns RIPPER through SLIPPER: Percent Error on Test Data)

Table 1: Summary of the datasets used, and error rates for SLIPPER, four alternative rule learners (RIPPER with and without optimization, C4rules, and C5rules), and the C4.5 decision tree learner.

forth, C4rules); and the C5.0 rule learner[2] (henceforth, C5rules), a proprietary, unpublished descendent of C4rules. RIPPER without optimization is included as a relatively simple separate-and-conquer variant; this algorithm has been evaluated elsewhere under the names IREP∗ (Cohen 1995) and IRIP (Fürnkranz 1998).

The results are shown in detail in Table 1. SLIPPER obtains the average lowest error rate for both sets of benchmarks; also, among the rule learners SLIPPER, RIPPER, C4rules, and C5rules, SLIPPER obtains the lowest error rate 17 times, C5rules 10 times, RIPPER 9 times, and C4rules 5 times. Also among these rule learners, the average rank of SLIPPER is 2.0, compared to 2.6 for RIPPER and C5rules, and 2.8 for C4rules.[3]

Summaries of the experimental results are given in Figure 3 and Table 2. In the scatterplot of Figure 3,

[2]That is, C5.0 run with the `-r` option.

[3]The corresponding figures across all learning algorithms compared are given in the table.

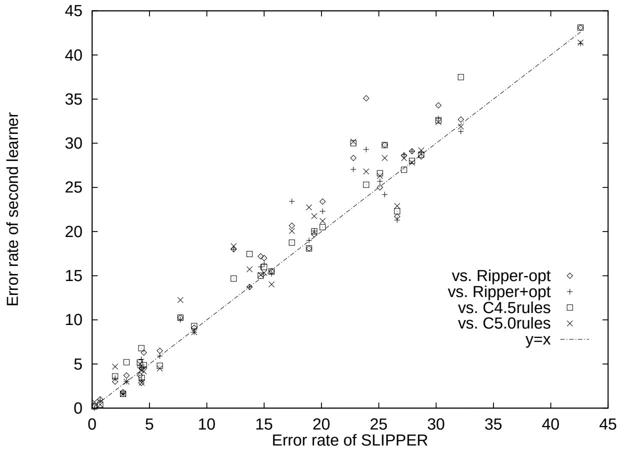

Figure 3: Summary of experimental results. Points above the lines $y = x$ correspond to datasets for which SLIPPER performs better than some second learner.

	RIPPER	C4rules	C5rules	SLIPPER
RIPPER		1.023	0.993	0.961
C4rules	14-17-1		1.066	0.971
C5rules	14-15-3	16-14-2		0.977
SLIPPER	<u>20-9-3</u>	**22-8-2**	19-11-2	
SLIPPER	11-4-1	<u>12-4-0</u>	8-7-1	*(Prosp.)*
	9-5-2	10-4-2	11-4-1	*(Devel.)*

Table 2: Summary of experimental results. If L_R and L_C are the learners corresponding to a row and column, respectively, the upper triangle entries are the average of error(L_C)/error(L_R). 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.

each point compares SLIPPER to some second learning system L on a single dataset: the x-axis position of the point is the error rate of SLIPPER, and the y-axis position is the error rate of L. Thus, points above the lines $y = x$ correspond to datasets for which SLIPPER 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.[4]

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.[5] 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 records 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.[6] 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.[7] 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.[8] 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. SLIPPERalso 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.

---

[4]The sole exception to this is `network2`, on which SLIPPER performs noticeably worse than the other methods.

[5]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.

[6]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.

[7]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).

[8]Timing results are given in CPU seconds on a MIPS Irix 6.3 with 200 MHz R10000 processors.

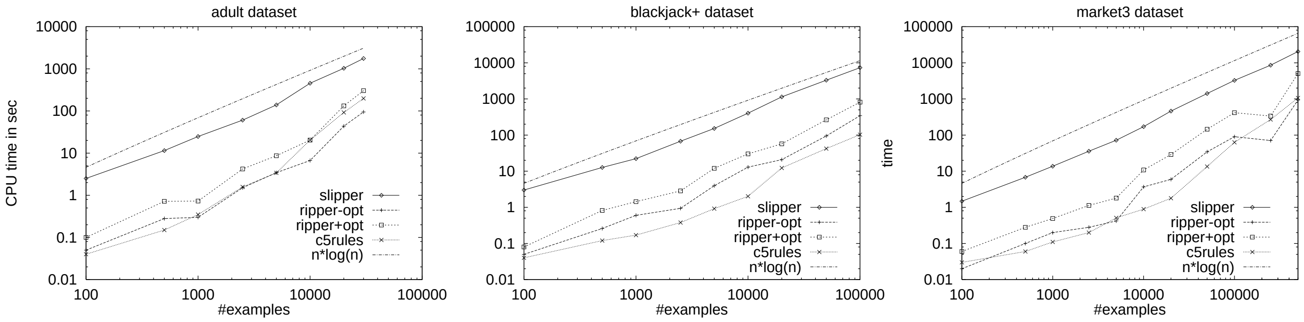

Figure 4: Run-time performance of SLIPPER, RIPPER, and C5rules on large datasets.

Concluding remarks

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

References

Blake, C.; Keogh, E.; and Merz, C. J. 1989. UCI repository of machine learning databases [www.ics.uci.edu/~mlearn/MLRepository.html]. Irvine, CA: University of California, Department of Information and Computer Science.

Brunk, C., and Pazzani, M. 1991. Noise-tolerant relational concept learning algorithms. In *Proceedings of the Eighth International Workshop on Machine Learning*. Ithaca, New York: Morgan Kaufmann.

Clark, P., and Niblett, T. 1989. The CN2 induction algorithm. *Machine Learning* 3(1).

Cohen, W. W. 1995. Fast effective rule induction. In *Machine Learning: Proceedings of the Twelfth International Conference*. Lake Tahoe, California: Morgan Kaufmann.

Domingos, P. 1996. Unifying instance-based and rule-based induction. *Machine Learning* 24(2):141–168.

Frank, E., and Witten, I. 1998. Generating accurate rule sets without global optimization. In *Machine Learning: Proceedings of the Fifteenth International Conference*.

Freund, Y., and Schapire, R. E. 1996. Experiments with a new boosting algorithm. In *Machine Learning: Proceedings of the Thirteenth International Conference*, 148–156.

Freund, Y., and Schapire, R. E. 1997. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences* 55(1):119–139.

Freund, Y. 1995. Boosting a weak learning algorithm by majority. *Information and Computation* 121(2):256–285.

Fürnkranz, J., and Widmer, G. 1994. Incremental reduced error pruning. In *Machine Learning: Proceedings of the Eleventh Annual Conference*. New Brunswick, New Jersey: Morgan Kaufmann.

Fürnkranz, J. 1998. Integrative windowing. *Journal of Artificial Intelligence Research* 8:129–164.

Hsu, D.; Etzioni, O.; and Soderland, S. 1998. A redundant covering algorithm applied to text classification. In *AAAI Workshop on Learning for Text Categorization*.

Kearns, M., and Valiant, L. G. 1994. Cryptographic limitations on learning Boolean formulae and finite automata. *Journal of the Association for Computing Machinery* 41(1):67–95.

Margineantu, D. D., and Dietterich, T. G. 1997. Pruning adaptive boosting. In *Machine Learning: Proceedings of the Fourteenth International Conference*, 211–218.

Pagallo, G., and Haussler, D. 1990. Boolean feature discovery in empirical learning. *Machine Learning* 5(1).

Quinlan, J. R. 1990. Learning logical definitions from relations. *Machine Learning* 5(3).

Quinlan, J. R. 1994. *C4.5: programs for machine learning*. Morgan Kaufmann.

Schapire, R. E., and Singer, Y. 1998. Improved boosting algorithms using confidence-rated predictions. In *Proceedings of the Eleventh Annual Conference on Computational Learning Theory*, 80–91.

Schapire, R. E.; Freund, Y.; Bartlett, P.; and Lee, W. S. 1997. Boosting the margin: A new explanation for the effectiveness of voting methods. In *Machine Learning: Proceedings of the Fourteenth International Conference*.

Schapire, R. E. 1990. The strength of weak learnability. *Machine Learning* 5(2):197–227.

This paper has been published by the American Association for Artificial Intelligence (http://www.aaai.org). This document has been copyrighted by the American Association for Artificial Intelligence and is being made available solely for the use and benefit of authors preparing papers for AAAI Press publications. No other use of this document may be made without permission in writing from the copyright holder.