Learning easily understandable decision rules from examples is one of the classic problems in machine learning. Most learning algorithms for this problem employ some variation of a greedy separate-and-conquer algorithm. In this paper, we describe a system called LERILS that learns highly accurate and comprehensible rules from examples using a randomized iterative local search inspired by algorithms like WalkSat and simulated annealing. We compare its performance to C4.5, RIPPER, and CN2 on 11 data sets from the UCI machine learning repository. We show that LERILS can outperform C4.5 most of the time and sometimes it can even best RIPPER. While its accuracy is comparable to CN2, its rules are shorter and fewer, and hence are more human-comprehensible.
Learning Classification Rules by Randomized Iterative Local Search

by

Michael Chisholm

A Thesis Submitted
to
Oregon State University

In Partial Fulfillment of the requirements for the degree of

Master of Science

Presented November 22, 1999
Commencement June 2000
Master of Science thesis of Michael Chisholm presented on November 22, 1999

Approved:

__________________________________________
Major Professor, representing Computer Science

__________________________________________
Head of Department of Computer Science

__________________________________________
Dean of Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

__________________________________________
Michael Chisholm, Author
Acknowledgement

I wish to acknowledge the assistance of Dr. Prasad Tadepalli, with whom the algorithm was co-developed, and who helped extensively with the wording, layout, and content of this thesis. Thanks are also due to my committee members: Dr. Thomas Dietterich, Dr. Bruce D’Ambrosio, and Dr. Tevian Dray for their helpful suggestions.
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2 Other Rule Learning Algorithms</td>
<td>4</td>
</tr>
<tr>
<td>3 Rule Learning Algorithm</td>
<td>6</td>
</tr>
<tr>
<td>3.1 Overview</td>
<td>6</td>
</tr>
<tr>
<td>3.2 Rule Generation</td>
<td>7</td>
</tr>
<tr>
<td>3.3 Ruleset Generation</td>
<td>10</td>
</tr>
<tr>
<td>3.4 Continuous Attributes</td>
<td>11</td>
</tr>
<tr>
<td>3.5 Complexity Analysis</td>
<td>13</td>
</tr>
<tr>
<td>4 Results</td>
<td>16</td>
</tr>
<tr>
<td>5 Conclusions and Further Work</td>
<td>20</td>
</tr>
<tr>
<td>Bibliography</td>
<td>21</td>
</tr>
</tbody>
</table>
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Pseudocode</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>Relabeled Examples</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>Thresholds</td>
<td>12</td>
</tr>
</tbody>
</table>
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>An Example of an Example</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>Descriptions of Datasets</td>
<td>17</td>
</tr>
<tr>
<td>3</td>
<td>Accuracies of the Algorithms</td>
<td>17</td>
</tr>
<tr>
<td>4</td>
<td>Hypothesis Size</td>
<td>18</td>
</tr>
</tbody>
</table>
Learning Classification Rules by Randomized Iterative Local Search

1. Introduction

Abstractly, supervised learning is a process whereby an agent learns by observing examples of correct behavior provided by a “teacher”. In particular, the teacher repeatedly draws examples from a space of all correctly labeled examples (the concept) and shows it to the agent. The agent’s task is to formulate its own approximation to the target concept based on the sequence of examples it is shown. This approximation is called a hypothesis. Our algorithm requires that the labels be discrete-valued. It is more common for these discrete-valued labels to be called classes, and the task of learning discrete-valued labels for examples to be called a concept learning task.

In the real world, even the teacher can make mistakes. This can be due to many things, including faulty sensors and human error. This is manifested in a classification task as incorrectly labeled training and/or test examples, otherwise known as noise. In the interests of robustness, it is therefore desirable that an agent be able to learn even when noise is present, which may mean that the final hypothesis classifies some of the training examples (presumably the incorrectly labeled ones) differently than the teacher did. A hypothesis that labels some training examples differently than the teacher did is called an inconsistent hypothesis.

If the hypothesis labels a noisy training example the way the teacher did, then of course it will get that example wrong when tested (assuming the example is labeled correctly in the test data), and so accuracy will suffer. This phenomenon, where the learner learns the training data so well that it learns the noise, resulting in decreased test set performance, is known as overfitting.

Noise in training data tends to be somewhat erratic and random, and therefore any hypothesis that classifies every training example the same way the teacher did tends to be overly
large and complex. It is this fact that most techniques rely upon to combat overfitting. In particular, the hypothesis may be simplified at the expense of slightly reduced training set accuracy. In order to do this, it is convenient to define a quantity that reflects both the size of the hypothesis and the number of errors it makes, and then try to minimize it. This quantity is known as the description length of the hypothesis, and that we should try to minimize it is known as the Minimum Description Length principle. We use description length heuristics of various flavors throughout our algorithm.

Human comprehensibility has recently gained importance as a desirable property of hypotheses [1]. This is because humans can learn from the hypotheses output by a computer, provided the hypothesis is in a human comprehensible form. Human experts can also participate in the learning process by critiquing or improving the learned hypothesis if it is understandable to them. Decision trees and rules are two widely used symbolic hypothesis representations, and it appears that of the two, rules are easier for people to understand, especially if they are independent. Independent rules are those which can be evaluated and interpreted on their own individually, without regard to their relative order in a list of rules. Neural networks for example, though highly accurate, are also highly incomprehensible to humans.

Most rule induction algorithms, including ID3 [3] and RIPPER [4], use some form of greedy search. While greedy search is fast, it might sacrifice accuracy and/or human comprehensibility. Our motivation is to sacrifice efficiency in favor of accuracy and comprehensibility, by doing a more exhaustive search. Our algorithm is based on a randomized iterative local search inspired by GSAT and Walksat [8, 10].

GSAT is a greedy algorithm for solving boolean satisfiability problems. It requires that the given boolean expression be in conjunctive normal form. All boolean variables in the expression are initially assigned random values. GSAT then keeps track of all literals that appear in an unsatisfied clause and repeatedly flips the value of the one that would result in the greatest reduction in the number of unsatisfied clauses. When the number of flips has exceeded a
predetermined flip count limit, the values of all variables are re-randomized and the flipping process begins again. If the number of restarts exceeds a predetermined restart limit, then the algorithm halts, and is deemed to have failed to have found a satisfying assignment. It has been shown that GSAT's performance improves by adding a random component [10]. Randomized versions of GSAT have also been successfully applied to planning by encoding planning problems as satisfiability problems [11]. It is from this randomized local search idea that we take our cue in creating a rule inducer.

Our algorithm has two parts. The first part finds rules and adds them to a large rule pool. The second part combines the rules in the rule pool to form a final ruleset. Both parts use GSAT-like search with a minimum description length [7] heuristic to trade off rule/ruleset accuracy with simplicity. The algorithm can handle missing attribute values and real-valued attributes.

We used 9 datasets from the UCI machine learning databases repository and 2 synthetic databases of our own creation to compare the performance of our algorithm with RIPPER, c4.5 [2], and CN2 [5]. Our algorithm was significantly better than RIPPER on two of the datasets, and was never significantly worse. It was significantly better than CN2 on 3 datasets, and significantly worse on 2 (though CN2’s hypotheses were excessively large on these two). It was significantly better than C4.5 on 4 datasets and significantly worse on only one. All significance is at the 5% level. LERILS’s hypotheses are similar in size to RIPPER’s, without the rule dependence. CN2 was found to have excessively large hypotheses. From this we conclude that LERILS’s hypotheses are the most comprehensible of the 4 algorithms tested.
2. Other Rule Learning Algorithms

There have been several rule learning algorithms in the literature, most of which work by some form of greedy search. For example, Swap-1 [12] builds its rules greedily by removing as well as adding literals during the rule generation phase. To avoid overfitting, it prunes the rules by creating a series of decreasing-complexity rulesets. These rulesets are created by repeatedly pruning the original ruleset at its “weakest link”. The weakest link is defined as the prune that would result in the fewest new errors per component pruned. Prunable components include single rules or literals. In order to approximate the error rate of a ruleset, cross validation is used. This is done by inducing an auxiliary ruleset of similar complexity to the original for each fold of the cross validation, and averaging the errors on the test sets.

Inducing auxiliary rulesets for each candidate prune in order to find the weakest link is computation-intensive. RIPPER avoids this expensive computation by making pruning incremental. First, rules are induced greedily; literals may only be added to the end of a rule, not deleted or swapped out. A rule is greedily pruned immediately after it is created. Examples are removed as rules are created. The classes are ordered by frequency in the training data, and rules are not induced for the most frequently occurring class – it becomes the default. After all rules are induced and pruned, the ruleset is further optimized by pruning the ruleset in a way that more closely resembles conventional reduced error pruning. In particular, two variants of each rule are created. A replacement rule is regrown from scratch and pruned so as to minimize the error of the entire ruleset on the pruning data. A revision is grown by adding literals to the original rule. If either the replacement or revision is deemed better than the original, the original is replaced.

SLIPPER improves upon the accuracy of RIPPER by using a form of AdaBoost [13], where the weak hypotheses are rules grown and pruned similarly to RIPPER’s rules, but using different heuristics. Each rule is assigned a confidence rating. Classification of an unseen
example is done by adding the signed confidence ratings of all rules that cover the example. A positive sum indicates prediction of the positive class, while a negative sum indicates prediction of the negative class. The confidence rating of all rules is positive, with the exception of the default rule (SLIPPER so far only works on binary classification tasks). SLIPPER’s hypotheses are accurate, but the rules in an ensemble are not independent of one another and so are less comprehensible than the hypotheses of non-ensemble rule based learners.

In summary, Swap-1 and RIPPER remove examples from the pool as rules are created, increasing rule dependence and decreasing comprehensibility. In addition, Swap-1 uses a rather computation intensive pruning algorithm. SLIPPER’s hypotheses are accurate, but the rules in its ensembles are perhaps even less independent than Swap-1 or RIPPER’s rules. These considerations led us to design a learning algorithm that achieves high accuracy while keeping the rules independent, by iteratively learning accurate rules from the same training set. This algorithm is described in the next section.
3. Rule Learning Algorithm

This section describes our learning algorithm in detail and analyzes its asymptotic complexity. Our algorithm is motivated by the need to learn accurate and comprehensible rules. This implies that the rules should be simple and few in number and should be independently accurate. This rules out learning multiple classes by separate-and-conquer, and boosting that relies on majority voting of several low accuracy rules.

3.1 Overview

LERILS iteratively works on each class, and operates in two stages:

1. Create a large pool of rules.
2. Try to find an optimal subset of these rules.

In the first stage, positive examples are picked at random, and rules are created from them. These rules are then refined by doing a random walk, using a description length heuristic to guide the search. The goal is to get a variety of good rules to choose from during the second stage. In the second stage, rulesets are created from random subsets of the rules in the rule pool. These rulesets are then refined by doing a greedy walk using another description length heuristic. These two stages are repeated for each class, and then the final rulesets are greedily ordered to produce the final list of rulesets.

Parameters were tuned by observing runs on a few of the datasets – initially on the synthetic datasets, and then when the algorithm worked on more than just binary attributes and classes, on a couple of the other datasets, such as iris and ecoli. minGain needs to be small enough that good rules can be made, but not so small that excessively many thresholds are found, as this would slow the algorithm down. M needs to be large enough that there are good variety of
rules to choose from when constructing the rulesets, but again, the larger $M$ is, the slower the algorithm runs. $k_1$ needs to be chosen so that there is sufficient time for the search to find a good rule, but the larger $k_1$ is, the longer it takes to create a rule. The ideal value for $p_1$ is sufficiently high that local minima can be escaped from, but not so high that local minima are never found. Restarts needs to be high enough that good rulesets are found. But the restarts take time, so the value should be kept as low as possible while maintaining ruleset accuracy.

### 3.2 Rule Generation

Rule generation begins by picking a random positive seed example. A rule is essentially represented as an example with extra relevance bits to indicate which literals of the example are a part of the rule. Each literal in the seed example is chosen to be relevant with 50% probability in the initial rule created from it. The seed example always remains covered to ensure a good variety of rules. As an illustration of this, consider the 4-attribute example

<table>
<thead>
<tr>
<th>Sky</th>
<th>Temp</th>
<th>Time</th>
<th>Wind</th>
<th>EnjoyTennis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sunny</td>
<td>Cool</td>
<td>?</td>
<td>NotWindy</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 1: An Example of an Example

where EnjoyTennis is the class attribute and ? represents a missing attribute, and consider the bit pattern 1001. The rule represented by these would be $\text{EnjoyTennis} = \text{Yes} :\text{- Sky=}\text{Sunny}, \text{Wind=}\text{NotWindy}$. Since the seed example has its third attribute missing, the third bit in the relevance bitset must always be 0. During the random walk, with probability $p_1$, a random relevance bit is flipped. With probability $1-p_1$, that bit is flipped which minimizes the
/* Find thresholds */
for each continuous attribute a
    sort all training examples according to attribute a
    for each class c
        collect all threshold values that result in an information gain
greater than minGain and add them to the threshold list for
class c, attribute a.
    end for
end for

/* induce rules */
for each class c

/* make rule pool */
for r = 1 to M do
    pick random seed example of class c and create rule rule from it
    for t = 1 to k1*N do
        with probability p1, flip a random relevance bit of rule
        (or randomize the thresholds, if the chosen attribute is continuous)
        with probability 1-p1, flip the bit or adjust the threshold
        boundary that minimizes the description length of the
        resulting rule
        if the description length is better than the description length
        of the best rule found so far, bestRule = rule
    end for
    add bestRule to the rule pool
end for

/* make ruleset */
remove duplicate rules from rule pool
for t = 1 to restarts do
    create ruleset ruleSet from the rules in the rule pool
    do
        add or remove the rule from ruleSet that minimizes the ruleset
        description length of the resulting ruleset
        if the ruleset description length of ruleSet is better than the
        best description length found so far, bestRuleSet = ruleSet
        while the ruleset description length continues to decrease
    end for
    add bestRuleSet to the final set of rulesets
end for

/* order the rulesets */
EX = set of all training examples
for t = 1 to number-of-classes
    Add to the decision list the ruleset r with the minimum number of
    exceptions (false positives + false negatives) with respect to EX
    Remove from EX all examples covered by r as well as all false negatives
end for

return the decision list of rulesets

Parameter values:

\[ \text{minGain} = 0.1 \]
\[ M = 100 \]
\[ k_1 = 3 \]
\[ p_1 = 0.35 \]
\[ \text{restarts} = 5 \]

Figure 1: Pseudocode
description length of the rule. At the end of the walk, the best rule found is added to the rule pool. In the experiments of this paper, we used $p_i = .35$ and added 100 rules to the rule pool.

The number of steps taken during the random walk is dependent on the number of attributes, $N$, in each example. More attributes require a longer walk. For example, if there are 20 attributes and most are irrelevant, it would take around 20 steps to flip the relevance bits (possibly more if some attributes are continuous), whereas if there were only 10 attributes, it would take closer to 10 steps. We have been using $3N$ steps in these experiments.

The relative description length of a rule is computed as $(s+n)/p$, where $s$ is the number of relevant literals (the size of the rule), $n$ is the number of false positives (negative examples covered), and $p$ is the number of positive examples covered by the rule. Missing attributes in the seed example are kept permanently irrelevant in the rule. Note that unlike RIPPER and CN2, after a rule is created and added to the rule pool, the examples it covers are not removed from the training example set. So every rule is built using all the training examples. This helps ensure that every rule created is as accurate as possible. Continuous attributes require special handling, as described later.

To understand the relative description length formula, first consider the simpler formula $s+n$ and the rules it favors. The ideal rule under this heuristic would be one with, say 1 literal, and covers no negative examples. Now there is no incentive to prefer rules that cover any positive examples either, so the rule may not cover any at all except the seed example. A rule like this, though it may be general, is probably quite useless when it comes to classifying unseen examples. There needs to be an incentive to cause rules to cover as many positive examples as possible while still excluding negative examples and remaining short. Hence the division with $p$. 
3.3 Ruleset Generation

Duplicate rules are first removed from the rule pool. A ruleset is simply a collection of rules for the same class, and initially it contains a random subset of the rule pool. Every rule in the rule pool has a 50% chance of being included in the initial ruleset. So on average, the initial ruleset contains half the rules in the rule pool. The choice of which rule to add or remove from the ruleset is purely greedy: that rule is removed or added to the ruleset that minimizes the resulting description length of the ruleset. A pure greedy method was found to be faster than doing randomized walking for a fixed number of steps, and the quality of the rulesets generated didn’t decrease significantly. The ruleset description length is computed as \( e + l \), where \( e \) is the total number of exceptions, both positive and negative, to the ruleset, and \( l \) is the total number of literals in all the rules in the ruleset. Adding a count of literals to the description length causes the learner to prefer fewer and more general (i.e. those with fewer literals) rules to more specific rules. The walk continues as long as the ruleset description length continues to decrease. There are a fixed number of restarts, where a new ruleset is randomly chosen from the rule pool, and the greedy walk is begun again. This compensates for the lack of randomness in the walks. We used 5 restarts for each ruleset. The best ruleset found during all steps of all greedy walks becomes the final ruleset for the class.

To resolve classification conflicts, the rulesets are ordered in a decision list structure. A decision list is simply a way of resolving classification conflicts, where each rule is tried on an example in a certain fixed order. The class of the first rule to cover the example becomes the example’s predicted class. In this case, there are multiple rules per class, and the order of rules within a ruleset doesn’t matter, so it’s just the rulesets themselves that are ordered. Ordering of rulesets is done in a greedy way. The exception count of the decision list is simply the number of exceptions to the rulesets in the list. At any point, that ruleset is added to the decision list which
minimizes the exception count of the resulting decision list. For efficiency’s sake, an incremental algorithm is used that adds the ruleset which minimizes its contribution to the exception count. All covered examples and false negatives from the example list are removed after each ruleset is added, and succeeding rulesets are only checked on the remaining examples. This way, the rulesets added never need to be checked again, since they will always cover the same examples and have the same exceptions.

3.4 Continuous Attributes

There are no relevance bits maintained for continuous attributes. Instead, a pair of threshold boundary values is stored for each continuous attribute, which represents the allowable range of the attribute. Thresholds are computed prior to learning, using the information gain criterion developed by Quinlan for his ID3 [3] algorithm. The information gain criterion, as used in ID3, measures the difference in the information content of a correct labeling before and after a particular attribute is known, and selects the attribute with the highest gain. We use the criterion to select splitting thresholds rather than attributes. In particular, all the training examples are sorted according to a continuous attribute. All split points that result in an information gain greater than some minimum threshold amount are then collected. There is one set of thresholds for each continuous attribute, for each class. Each set has $-\infty$ as its first threshold, and $+\infty$ as its last. An interval such as $(-\infty, 5)$ for an attribute $a$ is shown as $a < 5$, $[6, 10)$ is shown as $6 \leq a < 10$, and $[14, \infty)$ is shown as $a \geq 14$. A continuous attribute is deemed irrelevant if its threshold boundaries are $\pm \infty$. When a new rule is created from a seed example and a continuous attribute has been decided to be relevant (which happens with 50% probability as mentioned previously), the threshold boundary values are initialized to the narrowest interval containing the seed example’s attribute value.
The specific values chosen as threshold values are those that begin a run of examples of the class that rules are being induced for, or a run of all other classes. For example, if we label examples of class 3, say, as ‘+’ and all others as ‘-‘, then the sorted list of examples can be split into alternating segments of examples labeled ‘+’ and examples labeled ‘-‘.

\[
\begin{array}{cccccc}
- & + & - & + & - \\
\end{array}
\]

*Figure 2: Relabeled Examples*

The examples under consideration as split points are the first examples in each segment, labeled $t_0$ through $t_4$ below.

\[
\begin{array}{cccccc}
t_0 & - & t_1 & + & t_2 & - & t_3 & + & t_4 & - \\
\end{array}
\]

*Figure 3: Thresholds*

Assuming $t_2$ is the split point under consideration, the contents of the two example subsets created by the split point are shaded. If we assume the examples are sorted in increasing order from left to right in figure 2 (and 1) then threshold value $t_2$ belongs to the larger group. Hence, if $t_4$ was also chosen as a threshold value, then the interval represented by those thresholds is the half open interval $[t_2, t_4)$.

When doing a random adjustment to a continuous attribute, both threshold bounds are moved to random thresholds, making sure the seed example remains covered. A greedy adjustment to a continuous attribute is an adjustment to exactly one of the threshold boundaries that minimizes the description length of the resulting rule, while still keeping the seed example
covered. We chose to only consider adjustments to one threshold bound to speed the algorithm up.

3.5 Complexity Analysis

Here we derive the time complexity of the algorithm. Recall from the pseudocode in figure 1 that the algorithm has five parameters; all except \textit{minGain} and \textit{p}_1 will be used in the following derivation. In addition, a set of training data may be characterized in terms of three parameters: the number of attributes per example, the number of examples, and the number of classes. In order to derive a worst-case bound on running time, two assumptions will be made: (1) all attributes are continuous, and (2) only greedy adjustments are made. The variables used in the following derivation are:

\begin{align*}
    a &= \text{number of attributes per example} \\
    e &= \text{number of training examples} \\
    c &= \text{number of classes} \\
    M &= \text{number of rules added to the rule pool during rule generation} \\
    k_1 &= \text{multiplier of } a \text{ that determines the length of the walk during rule generation} \\
    r &= \text{restarts from the pseudocode: the number of restarts during ruleset generation} \\
    t &= \text{the length of the longest threshold list}
\end{align*}

First of all, checking an example’s attribute value against a rule’s corresponding literal can be done in constant time. Therefore a rule/example coverage check may be done in \( O(a) \) time. Finding the relative description length of a rule involves finding \( s, n, \) and \( p \) from the relative description length formula. This may be done in a single pass through the training data, hence finding the description length of a rule may be done in \( O(ea) \) time. Doing a greedy adjustment to a rule involves trying all single-threshold adjustments to the threshold bounds for every attribute. There are \( \leq t \) thresholds per attribute, and \( a \) attributes to check, therefore doing a
single greedy adjustment to a rule takes \( O(te^2) = O(e^2a^2) \) time, since \( t \leq e \). There are \( k_ia \) adjustments to be made, so creation of a single rule takes \( O(k_ie^2a^3) \) time. Finally, \( M \) rules are created, so rule generation for a single class takes \( O(Mk_ie^2a^3) \) time.

When doing ruleset generation, finding the ruleset description length involves finding \( e \) and \( l \) from the ruleset description length formula. \( l \) can be easily updated in constant time as rules are added to and removed from the ruleset. Finding \( e \), the exception count takes \( O(Mea) \) time, since there are \( \leq M \) rules in the ruleset and we must check each example for coverage by one of the rules in the ruleset. Doing a greedy adjustment to a ruleset involves finding exception counts for all possible rule inclusions and exclusions, of which there are \( \leq M \). Hence a single greedy adjustment takes \( O(M^2ea) \) time. The number of greedy adjustments that will be performed is no greater than \( e + Ma \) because the maximum possible exception count is \( e + Ma \), and it can only decrease during the course of the greedy walk. Therefore a single restart takes \( O(M^2e^2a + M^3ea^2) \) time. Finally, \( r \) restarts are performed, resulting in a final time complexity of \( O(rM^2e^2a + rM^3ea^2) \) for ruleset generation for one class.

Each of the above stages must be done once for each of the \( c \) classes, hence the final time complexities for rule and ruleset generation are \( O(cMk_ie^2a^3) \) and \( O(crM^2e^2a + crM^3ea^2) \). The final stage in the process is the ordering of the rulesets. Finding the best ruleset to add to the end of the ruleset decision list is done incrementally, by removing all covered examples and all false negatives after each new ruleset is added. In this way, we are actually looking for the ruleset that will result in the smallest increase in the exception count; the actual exception count never needs to be found until the end. Finding the exception count increase merely involves counting all exceptions to a single ruleset, which as previously noted, takes \( O(Mea) \) time. At first, there are \( c \) candidates for the first ruleset, then \( c - 1 \) candidates for the second, etc… which
results in an overall time complexity of $O(c^2Mea)$. Adding all the time complexities results in an overall time complexity of $O(cMK_1e^2a^3 + cM^2e^2a + cM^3ea^2 + c^2Mea) = O(c^2rk_1M^3e^2a^3)$ for the algorithm.

So in practice, which stage dominates the time taken for rule induction? The answer is: it depends. If you look only at those factors that vary with the data and not for a fixed variant of the algorithm, it would appear the time is dominated by the first term, which is the rule generation, because of the cubic $a$ factor. However, the number of rules in the rule pool is also dependent on the data: with simple datasets, there will be many duplicate rules added to the rule pool, so the size after duplicate removal will be small. But with complex datasets there may be few duplicates, and the actual number of rules in the rule pool may be close to $M$. We used $M=100$ in these experiments, and if there are few duplicates, this results in a constant factor of 1,000,000 in the second term. The size of the ruleset decreases with each step, so making the greedy steps gets faster the farther along in the walk you are. Nevertheless, making those first steps can be very time consuming. In practice, the time for rule generation can often be outweighed by the time for ruleset generation. In any case, ruleset ordering is very fast, and the actual time is usually negligible in comparison to the other two stages.
4. Results

We used 9 datasets from the UCI Machine Learning Databases Repository [9] and 2 synthetic datasets of our own creation (CNF and DNF) to compare our algorithm to RIPPER, c4.5, and CN2. Table 3 contains the results of 10 fold cross validation for each algorithm and dataset.

c4.5 greedily builds a decision tree using information gain to select an attribute to divide the training set. RIPPER was described in section 1.2. CN2 conducts a general-to-specific search through a space of rules by adding selectors (conditions to be checked, e.g. Sky=Sunny or Height<6) to rules and using a statistical significance test and an entropy measure to keep track of the best rule found so far. Having found this best rule, it adds the rule to its list of rules and removes all examples from the training set that are covered by the rule.

RIPPER’s default behavior is to first order the classes by frequency of occurrence in the training data. Assuming \( n \) classes, it finds rules to separate the first class from all others, then class 2 from class 3 through \( n \), etc. No rules are induced for the last (most frequently occurring) class; it becomes the default. To resolve ordering conflicts RIPPER tries the rules in the order they were induced. When using the unordered rules option, rules are created to separate class \( k \) from all other classes, for all classes \( k \). To resolve conflicts, the rule that was most accurate on the training data is preferred. If no rules cover the example in question, the most commonly occurring class in the training data is predicted.

CN2 has the same capability to create unordered and ordered rules [6]. When CN2 creates unordered rules, the algorithm changes a little bit. Instead of removing all covered examples after a new rule is found, only those examples of the new rule’s class are removed. This promotes the accuracy of succeeding rules, as the negative examples still remain in the training data to help shape the new rules. The conflict resolution strategy for classification is
Table 2: Descriptions of Datasets

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Attributes</th>
<th>Examples</th>
<th>Attribute Type</th>
<th># Classes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CNF</td>
<td>26</td>
<td>1000</td>
<td>Binary</td>
<td>2</td>
<td>Syn. 3-clause CNF concept</td>
</tr>
<tr>
<td>DNF</td>
<td>13</td>
<td>500</td>
<td>Binary</td>
<td>2</td>
<td>Syn. 5-term DNF concept</td>
</tr>
<tr>
<td>Xd6</td>
<td>10</td>
<td>200</td>
<td>Binary</td>
<td>2</td>
<td>DNF concept, 10% noise</td>
</tr>
<tr>
<td>lost2p-high</td>
<td>24</td>
<td>928</td>
<td>Binary</td>
<td>2</td>
<td>Chess domain</td>
</tr>
<tr>
<td>balance-scale</td>
<td>4</td>
<td>625</td>
<td>Nominal</td>
<td>3</td>
<td>Do objects balance?</td>
</tr>
<tr>
<td>Balloons</td>
<td>4</td>
<td>20</td>
<td>Binary</td>
<td>2</td>
<td>Conditions of experiment</td>
</tr>
<tr>
<td>Iris</td>
<td>4</td>
<td>150</td>
<td>Continuous</td>
<td>3</td>
<td>Identify species of Iris</td>
</tr>
<tr>
<td>Ecoli</td>
<td>7</td>
<td>336</td>
<td>Continuous</td>
<td>8</td>
<td>Localization sites of proteins</td>
</tr>
<tr>
<td>Hepatitis</td>
<td>19</td>
<td>155</td>
<td>13 bin, 6 cont</td>
<td>2</td>
<td>Patient lives or dies?</td>
</tr>
<tr>
<td>cred-screening</td>
<td>15</td>
<td>690</td>
<td>4bin,5nom,6cont</td>
<td>2</td>
<td>Credit approval</td>
</tr>
<tr>
<td>Glass</td>
<td>9</td>
<td>214</td>
<td>Continuous</td>
<td>7</td>
<td>Predict type of glass</td>
</tr>
</tbody>
</table>

Table 3: Accuracies of the Algorithms

<table>
<thead>
<tr>
<th>Dataset</th>
<th>RIPPER (unordered)</th>
<th>RIPPER (ordered)</th>
<th>C4.5</th>
<th>CN2 (unordered)</th>
<th>CN2 (ordered)</th>
<th>LERILS</th>
</tr>
</thead>
<tbody>
<tr>
<td>CNF</td>
<td>99.6 ± 0.27</td>
<td>100 ± 0</td>
<td>97.3 ± 0.40†</td>
<td>98.9 ± 0.35†</td>
<td>100 ± 0</td>
<td>100 ± 0</td>
</tr>
<tr>
<td>DNF</td>
<td>99.8 ± 0.20</td>
<td>94.4 ± 0.93</td>
<td>96.8 ± 0.90†</td>
<td>97.8 ± 0.63†</td>
<td>98.2 ± 0.92</td>
<td>99.4 ± 0.43</td>
</tr>
<tr>
<td>Xd6</td>
<td>79.5 ± 2.93†</td>
<td>84.5 ± 3.20</td>
<td>74.5 ± 3.37†</td>
<td>77.5 ± 2.71†</td>
<td>77.0 ± 1.86</td>
<td>87.5 ± 2.50</td>
</tr>
<tr>
<td>Lost2p-high</td>
<td>97.1 ± 0.64</td>
<td>96.7 ± 0.48</td>
<td>96.9 ± 0.65</td>
<td>98.7 ± 0.27*</td>
<td>98.5 ± 0.37</td>
<td>97.4 ± 0.56</td>
</tr>
<tr>
<td>Balance-scale</td>
<td>71.5 ± 1.49</td>
<td>72.1 ± 2.07</td>
<td>60.6 ± 1.97†</td>
<td>79.4 ± 1.53*</td>
<td>87.7 ± 1.12</td>
<td>73.1 ± 2.11</td>
</tr>
<tr>
<td>Balloons</td>
<td>100 ± 0</td>
<td>100 ± 0</td>
<td>100 ± 0</td>
<td>100 ± 0</td>
<td>100 ± 0</td>
<td></td>
</tr>
<tr>
<td>Iris</td>
<td>94.0 ± 2.32</td>
<td>93.3 ± 2.22</td>
<td>94.5 ± 1.72</td>
<td>95.3 ± 1.42</td>
<td>93.3 ± 1.72</td>
<td>96.7 ± 1.49</td>
</tr>
<tr>
<td>Ecoli</td>
<td>81.8 ± 1.50</td>
<td>81.2 ± 2.06</td>
<td>82.8 ± 1.26*</td>
<td>79.7 ± 2.22</td>
<td>80.6 ± 2.37</td>
<td>78.0 ± 1.77</td>
</tr>
<tr>
<td>Hepatitis</td>
<td>77.5 ± 3.16†</td>
<td>80.0 ± 3.85</td>
<td>81.2 ± 3.09</td>
<td>79.9 ± 4.41</td>
<td>81.8 ± 3.81</td>
<td>85.3 ± 3.15</td>
</tr>
<tr>
<td>Cred-screening</td>
<td>87.5 ± 1.50</td>
<td>86.5 ± 1.79</td>
<td>85.4 ± 1.58</td>
<td>82.6 ± 1.54</td>
<td>80.9 ± 0.80</td>
<td>84.5 ± 1.61</td>
</tr>
<tr>
<td>Glass</td>
<td>58.4 ± 2.11</td>
<td>69.0 ± 2.15</td>
<td>68.9 ± 3.33</td>
<td>64.5 ± 2.81</td>
<td>71.2 ± 3.08</td>
<td>60.8 ± 3.31</td>
</tr>
</tbody>
</table>

* indicates a performance increase relative to LERILS significant at the 0.05 level. † indicates a performance decrease relative to LERILS significant at the 0.05 level.

changed as well. Instead of using the decision list structure, the preferred class for an unseen example is chosen probabilistically from the classes of all rules that cover the example.

The last column contains LERILS’s scores when making the last class in the ruleset list a default class. Note that both RIPPER and CN2, whether creating ordered or unordered rules,
Table 4: Hypothesis Size (rules/literals)

<table>
<thead>
<tr>
<th>Dataset</th>
<th>RIPPER (unordered)</th>
<th>RIPPER (ordered)</th>
<th>C4.5</th>
<th>CN2 (unordered)</th>
<th>CN2 (ordered)</th>
<th>LERILS</th>
</tr>
</thead>
<tbody>
<tr>
<td>CNF</td>
<td>23/71</td>
<td>4/11</td>
<td>45</td>
<td>33/120</td>
<td>18/44</td>
<td>4/11</td>
</tr>
<tr>
<td>DNF</td>
<td>16/52</td>
<td>11/38</td>
<td>41</td>
<td>22/81</td>
<td>12/24</td>
<td>6/14</td>
</tr>
<tr>
<td>lost2p-high</td>
<td>13/35</td>
<td>4/19</td>
<td>23</td>
<td>25/72</td>
<td>14/32</td>
<td>4/20</td>
</tr>
<tr>
<td>balance-scale</td>
<td>27/53</td>
<td>15/31</td>
<td>26</td>
<td>148/474</td>
<td>56/102</td>
<td>18/36</td>
</tr>
<tr>
<td>Balloons</td>
<td>4/4</td>
<td>2/2</td>
<td>5</td>
<td>4/4</td>
<td>3/2</td>
<td>2/2</td>
</tr>
<tr>
<td>Iris</td>
<td>6/7</td>
<td>3/3</td>
<td>7</td>
<td>10/16</td>
<td>8/11</td>
<td>3/3</td>
</tr>
<tr>
<td>Ecoli</td>
<td>15/36</td>
<td>9/16</td>
<td>33</td>
<td>24/61</td>
<td>19/48</td>
<td>7/11</td>
</tr>
<tr>
<td>Hepatitis</td>
<td>6/8</td>
<td>4/6</td>
<td>13</td>
<td>18/47</td>
<td>12/36</td>
<td>4/7</td>
</tr>
<tr>
<td>credit-screening</td>
<td>8/22</td>
<td>6/16</td>
<td>29</td>
<td>31/99</td>
<td>37/104</td>
<td>6/12</td>
</tr>
<tr>
<td>Glass</td>
<td>11/19</td>
<td>8/14</td>
<td>39</td>
<td>26/70</td>
<td>18/43</td>
<td>5/9</td>
</tr>
</tbody>
</table>

always have a default class. If they had no default class, their accuracies would drop. The appropriate comparisons to LERILS to make are with unordered variants of the algorithms. Scores for the ordered variants appear in the table as a reference, because the ordered variants are more commonly used. LERILS outperforms unordered RIPPER and CN2 on many of the datasets. Ordered systems are generally more accurate than unordered systems, but they lose comprehensibility.

LERILS significantly outperformed C4.5 on four datasets: CNF, DNF, xd6, and balance-scale. It was outperformed only on ecoli. LERILS outperformed unordered CN2 on CNF, DNF, and xd6, and was outperformed on lost2p-high and balance-scale. CN2’s hypothesis for these datasets contained 6 and 8 times (respectively) the number of rules that LERILS did. Clearly, this extra accuracy came at a price. LERILS outperformed unordered RIPPER on only hepatitis and xd6, but was not significantly outperformed by RIPPER on any of the datasets.

Table 4 contains hypothesis sizes for a single run of the algorithms. These are indicative of comprehensibility, although comprehensibility involves more than the rule size and there is no widely accepted measure. For example, it appears that rules are more comprehensible than trees,
and rules that are independent of class order are even better than order-dependent rules. For the
rule learners there are two numbers per cell: the number of rules, followed by the total number of
literals in all rules. The default rule counted as one rule with no literals. For C4.5, the values
represent the total number of nodes in the decision tree.

LERILS always induced hypotheses that were smaller than unordered RIPPER’s. The
number of literals in LERILS’s rules were always smaller than the number of nodes in C4.5’s
trees. In almost all cases, CN2’s hypotheses were significantly larger than the rest.

A final remark should be made about running time. The above results may look good,
but our algorithm is much slower than CN2 and RIPPER. Each of the ten trials with the iris
dataset took less than 2 minutes to complete. In contrast, each trial of the credit-screening dataset
took 2.5 to 3.5 hours to complete. The average time per trial RIPPER reported for iris was 0.16
seconds, and for credit-screening, 2.28 seconds. But our goal with this algorithm is to sacrifice
speed for accuracy and comprehensibility. We took the view that it is acceptable to take longer to
learn, as long as good, independent, comprehensible rules are produced. Perhaps a more serious
limitation is the asymptotic complexity of the algorithm. Ways to get around this might include
preprocessing training data by taking a subset of features and/or training data.
5. Conclusions and Further Work

We have demonstrated an algorithm that induces human-comprehensible rules with accuracies comparable to the best rule inducers. It can handle datasets with any number of classes, and both missing and continuous-valued attributes. The last five datasets in the table have one or more continuous attributes. Unordered RIPPER outperformed LERILS on three of them, so there still seems to be room for improvement in the handling of continuous attributes. Any improvements made to the speed of the algorithm would also greatly increase its attractiveness. Our algorithm is certainly not better than the rest in terms of time. A greater range of tests should be made to more accurately tell how much better (or worse) we are than the other algorithms in terms of accuracy. Nevertheless, our accuracies seem to be comparable to the others. We have made a greater commitment to comprehensibility than the others, and that’s where our algorithm shines.

Due to the modularity of the algorithm, it is possible to add a “filtering module” in between the rule generator and the ruleset generator, which could look at the rules generated by the rule generator and throw out some rules before passing them on to the ruleset generator. This filtering could be done in different ways. Humans may have an idea of what rules make sense, and what rules don’t. The rules in the rule pool could be shown to a human expert who selects rules to throw away. In this way, a human could become involved in the rule learning. Or, a knowledge based solution might be used to replace the human. This would allow prior domain-specific knowledge to assist in rule induction, which has historically been difficult to do.
Bibliography


