Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP), Philadelphia, July 2002, pp. 17-24. Association for Computational Linguistics.

An Incremental Decision List Learner

Joshua Goodman
Microsoft Research
One Microsoft Way
Redmond, WA 98052
joshuago@microsoft.com

Abstract

We demonstrate a problem with the standard technique for learning probabilistic decision lists. We describe a simple, incremental algorithm that avoids this problem, and show how to implement it efficiently. We also show a variation that adds thresholding to the standard sorting algorithm for decision lists, leading to similar improvements. Experimental results show that the new algorithm produces substantially lower error rates and entropy, while simultaneously learning lists that are over an order of magnitude smaller than those produced by the standard algorithm.

1 Introduction

Decision lists (Rivest, 1987) have been used for a variety of natural language tasks, including accent restoration (Yarowsky, 1994), word sense disambiguation (Yarowsky, 2000), finding the past tense of English verbs (Mooney and Califf, 1995), and several other problems. We show a problem with the standard algorithm for learning probabilistic decision lists, and we introduce an incremental algorithm that consistently works better. While the obvious implementation for this algorithm would be very slow, we also show how to efficiently implement it. The new algorithm produces smaller lists, while simultaneously substantially reducing entropy (by about 40%), and error rates (by about 25% relative.)

Decision lists are a very simple, easy to understand formalism. Consider a word sense disambiguation task, such as distinguishing the financial sense of the word "bank" from the river sense. We might want the decision list to be probabilistic (Kearns and Schapire, 1994) so that, for instance, the probabilities can be propagated to an understanding algorithm. The decision list for this task might be:

IF "water" occurs nearby, output "river: .95", "financial: .05"
ELSE IF "money" occurs nearby, output "river: .1", "financial: .9"
ELSE IF word before is "left", output "river: .8", "financial: .2"
ELSE IF "Charles" occcurs nearby, output "river: .6", "financial: .4"
ELSE output "river: .5", "financial: .5"

The conditions of the list are checked in order, and as soon as a matching rule is found, the algorithm outputs the appropriate probability and terminates. If no other rule is used, the last rule always triggers, ensuring that some probability is always returned.

The standard algorithm for learning decision lists (Yarowsky, 1994) is very simple. The goal is to minimize the entropy of the decision list, where entropy represents how uncertain we are about a particular decision. For each rule, we find the expected entropy using that rule, then sort all rules by their entropy, and output the rules in order, lowest entropy first.

Decision lists are fairly widely used for many reasons. Most importantly, the rule outputs they produce are easily understood by humans. This can make decision lists useful as a data analysis tool: the decision list can be examined to determine which factors are most important. It can also make them useful when the rules must be used by humans, such as when producing guidelines to help doctors determine whether a particular drug should be administered. Decision lists also tend to be relatively small and fast and easy

to apply in practice.

Unfortunately, as we will describe, the standard algorithm for learning decision lists has an important flaw: it often chooses a rule order that is suboptimal in important ways. In particular, sometimes the algorithm will use a rule that *appears* good – has lower average entropy – in place of one that *is* good – lowers the expected entropy given its location in the list. We will describe a simple incremental algorithm that consistently works better than the basic sorting algorithm. Essentially, the algorithm builds the list in reverse order, and, before adding a rule to the list, computes how much the rule will reduce entropy *at that position*. This computation is potentially very expensive, but we show how to compute it efficiently so that the algorithm can still run quickly.

2 The Algorithms

In this section, we describe the traditional algorithm for decision list learning in more detail, and then motivate our new algorithm, and finally, describe our new algorithm and variations on it in detail. For simplicity only, we will state all algorithms for the binary output case; it should be clear how to extend all of the algorithms to the general case.

2.1 Traditional Algorithm

Decision list learners attempt to find models that work well on test data. The test data consists of a series of inputs $x_1, ..., x_n$, and we are trying to predict the corresponding results $y_1, ..., y_n$. For instance, in a word sense disambiguation task, a given x_i could represent the set of words near the word, and y_i could represent the correct sense of the word. Given a model D which predicts probabilities $P_D(y|x)$, the standard way of defining how well D works is the *entropy* of the model on the test data, defined as $\sum_{i=1}^{n} -log_2 P_D(y_i|x_i)$. Lower entropy is better. There are many justifications for minimizing entropy. Among others, the "true" probability distribution has the lowest possible entropy. Also, minimizing training entropy corresponds to maximizing the probability of the training data.

Now, consider trying to learn a decision list. Assume we are given a list of possible questions, $q_1, ..., q_n$. In our word sense disambiguation example, the questions might include "Does the word 'water' occur nearby," or more complex ones, such as "does the word 'Charles' occur nearby and is the word before 'river."' Let us assume that we have some training data, and that the system has two outputs (values for y), 0 and 1. Let $C(q_i, 0)$ be the number of times that, when q_i was true in the training data, the output was 0, and similarly for $C(q_i, 1)$. Let $C(q_i)$ be the total number of times that q_i was true. Now, given a test instance, x, y for which $q_i(x)$ is true, what probability would we assign to $y = 1$? The simplest answer is to just use the probability in the training data, $C(q_i, 1)/C(q_i)$. Unfortunately, this tends to overfit the training data. For instance, if q_i was true only once in the training data, then, depending on the value for y that time, we would assign a probability of 1 or 0. The former is clearly an overestimate, and the latter is clearly an underestimate. Therefore, we smooth our estimates (Chen and Goodman, 1999). In particular, we used the interpolated absolute discounting method. Since both the traditional algorithm and the new algorithm use the same smoothing method, the exact smoothing technique will not typically affect the relative performance of the algorithms. Let $C(0)$ be the total number of ys that were zero in the training, and let $C(1)$ be the total number of ys that were one. Then, the "unigram" probability y is $P(y) = \frac{C(y)}{C(0)+C(1)}$. Let $N(q_i)$ be the number of non-zero ys for a given question. In particular, in the two class case, $N(q_i)$ will be 0 if there were no occurences of the question q_i, 1 if training samples for q_i always had the same value, and 2 if both 1 and 0 values occurred. Now, we pick some value d (using heldout data) and discount all counts by d. Then, our probability distribution is

$$P(y|q_i) = \begin{cases} \frac{(C(q_i,y)-d)}{C(q_i)} + \frac{dN(q_i)}{C(q_i)}P(y) & \text{if } C(q_i, y) > 0 \\ \frac{dN(q_i)}{C(q_i)}P(y) & \text{otherwise} \end{cases}$$

Now, the predicted entropy for a question q_i is just

$$\textit{entropy}(q_i) = -P(0|q_i)log_2 P(0|q_i) - P(1|q_i)log_2 P(1|q_i)$$

The typical training algorithm for decision lists is very simple. Given the training data, compute the predicted entropy for each question. Then, sort the questions by their predicted entropy, and output a decision list with the questions in order. One of the

questions should be the special question that is always TRUE, which returns the unigram probability. Any question with worse entropy than TRUE will show up later in the list than TRUE, and we will never get to it, so it can be pruned away.

2.2 New Algorithm

Consider two weathermen in Seattle in the winter. Assume the following (overly optimistic) model of Seattle weather. If today there is no wind, then tomorrow it rains. On one in 50 days, it is windy, and, the day after that, the clouds *might* have been swept away, leading to only a 50% chance of rain. So, overall, we get rain on 99 out of 100 days. The lazy weatherman simply predicts that 99 out of 100 days, it will rain, while the smart weatherman gives the true probabilities (i.e. 100% chance of rain tomorrow if no wind today, 50% chance of rain tomorrow if wind today.)

Consider the entropy of the two weathermen. The lazy weatherman always says "There is a 99% chance of rain tomorrow; my average entropy is $-.99 \times log_2.99 - .01 \times log_2.01 = .081$ bits." The smart weatherman, if there is no wind, says "100% chance of rain tomorrow; my entropy is 0 bits." If there is wind, however, the smart weatherman says, "50% chance of rain tomorrow; my entropy is 1 bit."

Now, if today is windy, who should we trust? The smart weatherman, whose expected entropy is 1 bit, or the lazy weatherman, whose expected entropy is .08 bits, which is obviously much better.

The decision list equivalent of this is as follows. Using the classic learner, we learn as follows. We have three questions: if TRUE then predict rain with probability .99 (expected entropy = .081). If NO WIND then predict rain with probability 1 (expected entropy = 0). If WIND then predict rain with probability 1/2 (expected entropy = 1). When we sort these by expected entropy, we get:
IF NO WIND, output "rain: 100%" (entropy 0)
ELSE IF TRUE, output "rain: 99%" (entropy .081)
ELSE IF WIND, output "rain: 50%" (entropy 1)
Of course, we never reach the third rule, and on windy days, we predict rain with probabiliy .99!

The two weathermen show what goes wrong with a naive algorithm; we can easily do much better. For the new algorithm, we start with a baseline question, the question which is always TRUE and predicts the unigram probabilities. Then, we find the question which if asked *before* all other questions would decrease entropy the most. This is repeated until some minimum improvement, ϵ, is reached.[1] Figure 1 shows the new algorithm; the notation *entropy*(*list*) denotes the training entropy of a potential decision list, and *entropy*(*prepend*(q_i, *list*)) indicates the training entropy of *list* with the question "If q_i then output $p(y|q_i)$" prepended.

list = { TRUE }
do
  for each question q_i
    entropyReduce(i) =
      entropy(*list*) $-$ *entropy*(*prepend*$(q_i, list)$)
  $l = i$ such that *entropyReduce*(i) is largest
  if *entropyReduce*$(l) < \epsilon$ then
    return *list*
  else
    list $= prepend(q_l, list)$

Figure 1: New Algorithm, Simple Version

Consider the Parable of the Two Weathermen. The new learning algorithm starts with the baseline: If TRUE then predict rain with probability 99% (entropy .081). Then it prepends the rule that reduces the entropy the most. The entropy reduction from the question "NO WIND" is $.081 \times .99 = .08$, while the entropy for the question "WIND" is 1 bit for the new question, versus $.5 \times 1 + .5 \times -log_2.01 = .5 + .5 \times 6.64 = 3.82$, for the old, for a reduction of 2.82 bits, so we prepend the "WIND" question. Finally, we learn (at the top of the list), that if "NO WIND", then rain 100%, yielding the following decision list:
IF NO WIND, output "rain: 100%" (entropy 0)
ELSE IF WIND, output "rain: 50%" (entropy 1)
ELSE IF TRUE, output "rain: 99%" (entropy .081)
Of course, we never reach the third rule.

Clearly, this decision list is better. Why did our entropy sorter fail us? Because sometimes a smart learner knows when it doesn't know, while a dumb rule, like our lazy weatherman who ignores the wind, doesn't know enough to know that in the current sit-

[1] This means we are building the tree bottom up; it would be interesting to explore building the tree top-down, similar to a decision tree, which would probably also work well.

list = {TRUE}
for each training instance x_j, y_j
    instanceEnt$(j) = -log_2\ p(y_j)$
for each question q_i
    // Now we compute $entropyReduce(i) =$
    // *entropy*(*TRUE*) − *entropy*$(q_i$, *TRUE*$)$
    entropyReduce$(i) = 0$
    for each x_j, y_j such that $q_i(x_j)$
        entropyReduce(i) += $log_2\ p(y_j) - log_2\ p(y_j|q_i)$
do
    $l = \arg\max_i$ *entropyReduce*(i)
    if *entropyReduce*$(l) < \epsilon$ then
        return *list*
    else
        list $= prepend(q_l$, *list*$)$
    for each x_j, y_j such that $q_l(x_j)$
        for each k such that $q_k(x_j)$
            entropyReduce(k) += *instanceEnt*(j)
        instanceEnt$(j) = -log_2\ p(y_j|q_l)$
        for each k such that $q_k(x_j)$
            entropyReduce(k) −= *instanceEnt*(j)

Figure 2: New Algorithm, Efficient Version

uation, the problem is harder than usual.

2.2.1 Efficiency

Unfortunately, the algorithm of Figure 1, if implemented in a straight-forward way, will be extremely inefficient. The problem is the inner loop, which requires computing *entropy*(*prepend*$(q_i$, *list*$)$). The naive way of doing this is to run all of the training data through each possible decision list. In practice, the actual questions tend to be pairs or triples of simple questions. For instance, an actual question might be "Is word before 'left' and word after 'of'?" Thus, the total number of questions can be very large, and running all the data through the possible new decision lists for each question would be extremely slow.

Fortunately, we can precompute *entropyReduce*(i) and incrementally update it. In order to do so, we also need to compute, for each training instance x_j, y_j the entropy with the current value of *list*. Furthermore, we store for each question q_i the list of instances x_j, y_j such that $q_i(x_j)$ is true. With these changes, the algorithm runs very quickly. Figure 2 gives the efficient version of the new algorithm.

for each question q_i compute *entropy*(i)
list = questions sorted by *entropy*(i)
remove questions worse than TRUE
for each training instance x_j, y_j
    instanceEnt$(j) = -log_2\ p(y_j)$
for each question q_i in list in reverse order
    entropyReduce $= 0$
    for each x_j, y_j such that $q_i(x_j)$
        entropyReduce +=
            instanceEnt$(j) - log_2\ p(y_j|q_i)$
    if *entropyReduce* $< \epsilon$
        remove q_i from list
    else
        for each x_j, y_j such that $q_i(x_j)$
            instanceEnt$(j) = log_2\ p(y_j|q_i)$

Figure 3: Compromise: Delete Bad Questions

Note that this efficient version of the algorithm may consume a large amount of space, because of the need to store, for each question q_i, the list of training instances for which the question is true. There are a number of speed-space tradeoffs one can make. For instance, one could change the update loop from

for each x_j, y_j such that $q_i(x_j)$

to

for each x_j, y_j
    if $q_i(x_j)$ then ...

There are other possible tradeoffs. For instance, typically, each question q_i is actually written as a conjunction of simple questions, which we will denote Q_{i_j}. Assume that we store the list of instances that are true for each simple question Q_{i_j}, and that q_i is of the form $Q_{i_1}\&Q_{i_2}\&...\&Q_{i_I}$. Then we can write an update loop in which we first find the simple question with the smallest number of true instances, and loop over only these instances when finding the instances for which q_i is true:

$k = \arg\min_j$ number instances such that Q_{i_j}
for each x_j, y_j such that $Q_{i_k}(x_j)$
    if $q_i(x_j)$ then ...

2.3 Compromise Algorithm

Notice the original algorithm can actually allow rules which make things worse. For instance, in our lazy weatherman example, we built this decision list:

IF NO WIND, output "rain: 100%" (entropy 0)
ELSE IF TRUE, output "rain: 99%" (entropy .081)
ELSE IF WIND, output "rain: 50%" (entropy 1)
Now, the second rule could simply be deleted, and the decision list would actually be much better (although in practice we never want to delete the "TRUE" question to ensure that we always output some probability.) Since the main reason to use decision lists is because of their understandability and small size, this optimization will be worth doing even if the full implementation of the new algorithm is too complex. The compromise algorithm is displayed in Figure 3. When the value of ϵ is 0, only those rules that improve entropy on the training data are included. When the value of ϵ is $-\infty$, all rules are included (the standard algorithm). Even when a benefit is predicted, this may be due to overfitting; we can get further improvements by setting the threshold to a higher value, such as 3, which means that only rules that save at least three bits – and thus are unlikely to lead to overfitting – are added.

3 Previous Work

There has been a modest amount of previous work on improving probabilistic decision lists, as well as a fair amount of work in related fields, especially in transformation-based learning (Brill, 1995).

First, we note that non-probabilistic decision lists and transformation-based learning (TBL) are actually very similar formalisms. In particular, as observed by Roth (1998), in the two-class case, they are identical. Non-probabilistic decision lists learn rules of the form "If q_i then output y" while TBLs output rules of the form "If q_i and current-class is y', change class to y". Now, in the two class case, a rule of the form "If q_i and current-class is y', change class to y" is identical to one of the form "If q_i change class to y", since either way, all instances for which q_i is TRUE end up with value y. The other difference between decision lists and TBLs is the list ordering. With a two-class TBL, one goes through the rules from last-to-first, and finds the last one that applies. With a decision list, one goes through the list in order, and finds the first one that applies. Thus in the two-class case, simply by changing rules of the form "If q_i and current-class is y', change class to y" to "If q_i output y", and reversing the rule order, we can change any TBL to an equivalent non-probabilistic decision list, and vice-versa. Notice that our incremental algorithm is analogous to the algorithm used by TBLs: in TBLs, at each step, a rule is added that minimizes the training data error rate. In our probabilistic decision list learner, at each step, a rule is added that minimizes the training data entropy.

Roth notes that this equivalence does not hold in an important case: when the answers to questions are not static. For instance, in part-of-speech tagging (Brill, 1995), when the tag of one word is changed, it changes the answers to questions for nearby words. We call such problems "dynamic."

The near equivalence of TBLs and decision lists is important for two reasons. First, it shows the connection between our work and previous work. In particular, our new algorithm can be thought of as a probabilistic version of the Ramshaw and Marcus (1994) algorithm, for speeding up TBLs. Just as that algorithm stores the expected error rate improvement of each question, our algorithm stores the expected entropy improvement. (Actually, the Ramshaw and Marcus algorithm is somewhat more complex, because it is able to deal with dynamic problems such as part-of-speech tagging.) Similarly, the space-efficient algorithm using compound questions at the end of Section 2.2.1 can be thought of as a static probabilistic version of the efficient TBL of Ngai and Florian (2001).

The second reason that the connection to TBLs is important is that it shows us that probabilistic decision lists are a natural way to probabilize TBLs. Florian et al. (2000) showed one way to make probabilistic versions of TBLs, but the technique is somewhat complicated. It involved conversion to a decision tree, and then further growing of the tree. Their technique does have the advantage that it correctly handles the multi-class case. That is, by using a decision tree, it is relatively easy to incorporate the current state, while the decision list learner ignores that state. However, this is not clearly an advantage – adding extra dependencies introduces data sparseness, and it is an empirical question whether dependencies on the current state are actually helpful. Our probabilistic decision lists can thus be thought of as a competitive way to probabilize TBLs, with the advantage of preserving the list-structure and simplicity of TBL, and the possible disadvantage of losing the

dependency on the current state.

Yarowsky (1994) suggests two improvements to the standard algorithm. First, he suggests an optional, more complex smoothing algorithm than the one we applied. His technique involves estimating both a probability based on the global probability distribution for a question, and a local probability, given that no questions higher in the list were TRUE, and then interpolating between the two probabilities. He also suggests a pruning technique that eliminates 90% of the questions while losing 3% accuracy; as we will show in Section 4, our technique or variations eliminate an even larger percentage of questions while *increasing* accuracy. Yarowsky (2000) also considered changing the structure of decision lists to include a few splits at the top, thus combining the advantages of decision trees and decision lists. The combination of this hybrid decision list and the improved smoothing was the best performer for participating systems in the 1998 senseval evaluation. Our technique could easily be combined with these techniques, presumably leading to even better results. However, since we build our decision lists from last to first, rather than first to last, the local probability is not available as the list is being built. But there is no reason we could not interpolate the local probability into a final list. Similarly, in Yarowsky's technique, the local probability is also not available at the time the questions are sorted.

Our algorithm can be thought of as a natural probabilistic version of a non-probabilistic decision list learner which prepends rules (Webb, 1994). One difficulty that that approach has is ranking rules. In the probabilistic framework, using entropy reduction and smoothing seems like a natural solution.

4 Experimental Results and Discussion

In this section, we give experimental results, showing that our new algorithm substantially outperforms the standard algorithm. We also show that while accuracy is competitive with TBLs, two linear classifiers are more accurate than the decision list algorithms.

Many of the problems that probabilistic decision list algorithms have been used for are very similar: in a given text context, determine which of two choices is most appropriate. Accent restoration (Yarowsky, 1994), word sense disambiguation (Yarowsky, 2000), and other problems all fall into this framework, and typically use similar feature types. We thus chose one problem of this type, grammar checking, and believe that our results should carry over at least to these other, closely related problems. In particular, we chose to use exactly the same training, test, problems, and feature sets used by Banko and Brill (2001a; 2001b). These problems consisted of trying to guess which of two confusable words, e.g. "their" or "there", a user intended. Banko and Brill chose this data to be representative of typical machine learning problems, and, by trying it across data sizes and different pairs of words, it exhibits a good deal of different behaviors. Banko and Brill used a standard set of features, including words within a window of 2, part-of-speech tags within a window of 2, pairs of word or tag features, and whether or not a given word occurred within a window of 9. Altogether, they had 55 feature types. They used all features of each type that occurred at least twice in the training data.

We ran our comparisons using 7 different algorithms. The first three were variations on the standard probabilistic decision list learner. In particular, first we ran the standard sorted decision list learner, equivalent to the algorithm of Figure 3, with a threshold of negative infinity. That is, we included all rules that had a predicted entropy at least as good as the unigram distribution, whether or not they would actually improve entropy on the training data. We call this "Sorted: $-\infty$." Next, we ran the same learner with a threshold of 0 ("Sorted: 0"): that is, we included all rules that had a predicted entropy at least as good as the unigram distribution, and that would at least improve entropy on the training data. Then we ran the algorithm with a threshold of 3 ("Sorted: 3"), in an attempt to avoid overfitting. Next, we ran our incremental algorithm, again with a threshold of reducing training entropy by at least 3 bits.

In addition to comparing the various decision list algorithms, we also tried several other algorithms. First, since probabilistic decision lists are probabilistic analogs of TBLs, we compared to TBL (Brill, 1995). Furthermore, after doing our research on decision lists, we had several successes using simple linear models, such as a perceptron model and a maximum entropy (maxent) model (Chen and Rosenfeld, 1999). For the perceptron algorithm, we used a variation that includes a margin requirement, τ (Zaragoza

$\overline{w_j} = 0$
for 100 iterations or until no change
  for each training instance x_j, y_j
    if $\overline{q}(x_j) \cdot \overline{w_j} \times y_j < \tau$
      $\overline{w_j}$ += $\overline{q}(x_j) \times y_j$

Figure 4: Perceptron Algorithm with Margin

		1M	10M	50M
Sorted:	$-\infty$	14.27%	8.88%	6.23%
Sorted:	0	13.16%	8.43%	5.84%
Sorted:	3	10.23%	6.30%	3.94%
Incremental:	3	10.80%	6.33%	4.09%
Transformation		10.36%	5.14%	4.00%
Maxent		8.60%	4.42%	2.62%
Perceptron		8.22%	3.96%	2.65%

Figure 5: Geometric Mean of Error Rate across Training Sizes

		1M	10M	50M
Sorted:	$-\infty$	1065	10388	38893
Sorted:	0	831	8293	31459
Sorted:	3	45	462	1999
Incremental:	3	21	126	426
Transformation		15	77	244
Maxent		1363	12872	46798
Perceptron		1363	12872	46798

Figure 6: Geometric Mean of Model Sizes across Training Sizes

		1M	10M	50M
Sorted:	$-\infty$	0.91	0.70	0.55
Sorted:	0	0.81	0.64	0.47
Sorted:	3	0.47	0.43	0.29
Incremental:	3	0.49	0.36	0.25
Maxent		0.44	0.27	0.18

Figure 7: Arithmetic Mean of Entropy across Training Sizes

and Herbrich, 2000). Figure 4 shows this incredibly simple algorithm. We use $\overline{q}(x_j)$ to represent the vector of answers to questions about input x_j; $\overline{w_j}$ is a weight vector; we assume that the output, y_j is -1 or +1; and τ is a margin. We assume that one of the questions is TRUE, eliminating the need for a separate threshold variable. When $\tau = 0$, the algorithm reduces to the standard perceptron algorithm. The inclusion of a non-zero margin and running to convergence guarantees convergence for separable data to a solution that works nearly as well as a linear support vector machine (Krauth and Mezard, 1987). Given the extreme simplicity of the algorithm and the fact that it works so well (not just compared to the algorithms in this paper, but compared to several others we have tried), the perceptron with margin is our favorite algorithm when we don't need probabilities, and model size is not an issue.

Most of our algorithms have one or more parameters that need to be tuned. We chose 5 additional confusable word pairs for parameter tuning and chose parameter values that worked well on entropy and error rate across data sizes, as measured on these 5 additional word pairs. For the smoothing discount value we used 0.7. For thresholds for both the sorted and the incremental learner, we used 3 bits. For the perceptron algorithm, we set τ to 20. For TBL's minimum number of errors to fix, the traditional value of 2 worked well. For the maxent model, for smoothing, we used a Gaussian prior with 0 mean and 0.3 variance. Since sometimes one learning algorithm is better at one size, and worse at another, we tried three training sizes: 1, 10 and 50 million words.

In Figure 5, we show the error rates of each algorithm at different training sizes, averaged across the 10 words in the test set. We computed the geometric mean of error rate, across the ten word pairs. We chose the geometric mean, because otherwise, words with the largest error rates would disproportionately dominate the results. Figure 6, shows the geometric mean of the model sizes, where the model size is the number of rules. For maxent and perceptron models, we counted size as the total number of features, since these models store a value for every feature. For Sorted: $-\infty$ and Sorted: 0, the size is similar to a maxent or perceptron model – almost every rule is used. Sorted: 3 drastically reduces the model size – by a factor of roughly 20 – while improving performance. Incremental: 3 is smaller still, by about an additional factor of 2 to 5, although its accuracy is slightly worse than Sorted: 3. Figure 7 shows the entropy of each algorithm. Since entropy is logarithmic, we use the arithmetic mean.

Notice that the traditional probabilistic decision list learning algorithm – equivalent to Sorted: $-\infty$

– always has a higher error rate, higher entropy, and larger size than Sorted: 0. Similarly, Sorted: 3 has lower entropy, higher accuracy, and smaller models than Sorted: 0. Finally, Incremental: 3 has slightly higher error rates, but slightly lower entropies, and 1/2 to 1/5 as many rules. If one wants a probabilistic decision list learner, this is clearly the algorithm to use. However, if probabilities are not needed, then TBL can produce lower error rates, with still fewer rules. On the other hand, if one wants either the lowest entropies or highest accuracies, then it appears that linear models, such as maxent or the perceptron algorithm with margin work even better, at the expense of producing much larger models.

Clearly, the new algorithm works very well when small size and probabilities are needed. It would be interesting to try combining this algorithm with decision trees in some way. Both Yarowsky (2000) and Florian et al. (2000) were able to get improvements on the simple decision list structure by adding additional splits – Yarowsky by adding them at the root, and Florian *et al*. by adding them at the leaves. Notice however that the chief advantage of decision lists over linear models is their compact size and understandability, and our techniques simultaneously improve those aspects; adding additional splits will almost certainly lead to larger models, not smaller. It would also be interesting to try more sophisticated smoothing techniques, such as those of Yarowsky.

We have shown that a simple, incremental algorithm for learning probabilistic decision lists can produce models that are significantly more accurate, have significantly lower entropy, and are significantly smaller than those produced by the standard sorted learning algorithm. The new algorithm comes at the cost of some increased time, space, and complexity, but variations on it, such as the sorted algorithm with thresholding, or the techniques of Section 2.2.1, can be used to trade off space, time, and list size. Overall, given the substantial improvements from this algorithm, it should be widely used whenever the advantages – compactness and understandability – of probabilistic decision lists are needed.

References

M. Banko and E. Brill. 2001a. Mitigating the paucity of data problem. In *HLT*.

M. Banko and E. Brill. 2001b. Scaling to very very large corpora for natural language disambiguation. In *ACL*.

E. Brill. 1995. Transformation-based error-driven learning and natural language processing: A case study in part-of-speech tagging. *Comp. Ling.*, 21(4):543–565.

Stanley F. Chen and Joshua Goodman. 1999. An empirical study of smoothing techniques for language modeling. *Computer Speech and Language*, 13:359–394.

S.F. Chen and R. Rosenfeld. 1999. A gaussian prior for smoothing maximum entropy models. Technical Report CMU-CS-99-108, Computer Science Department, Carnegie Mellon University.

R. Florian, J. C. Henderson, and G. Ngai. 2000. Coaxing confidences out of an old friend: Probabilistic classifications from transformation rule lists. In *EMNLP*.

M. Kearns and R. Schapire. 1994. Efficient distribution-free learning of probabilistic concepts. *Computer and System Sciences*, 48(3):464–497.

W. Krauth and M. Mezard. 1987. Learning algorithms with optimal stability in neural networks. *Journal of Physics A*, 20:745–752.

R.J. Mooney and M.E. Califf. 1995. Induction of first-order decision lists: Results on learning the past tense of English verbs. In *International Workshop on Inductive Logic Programming*, pages 145–146.

G. Ngai and R. Florian. 2001. Transformation-based learning in the fast lane. In *NA-ACL*, pages 40–47.

L. Ramshaw and M. Marcus. 1994. Exploring the statistical derivation of transformational rule sequences for part-of-speech tagging. In *Proceedings of the Balancing Act Workshop on Combining Symbolic and Statistical Approaches to Language*, pages 86–95. ACL.

R. Rivest. 1987. Learning decision lists. *Machine Learning*, 2(3):229–246.

Dan Roth. 1998. Learning to resolve natural language ambiguities: A unified approach. In *AAAI-98*.

G. Webb. 1994. Learning decision lists by prepending inferred rules, vol. b. In *Second Singapore International Conference on Intelligent Systems*, pages 280–285.

David Yarowsky. 1994. Decision lists for lexical ambiguity resolution: Application to accent restoration in spanish and french. In *ACL*, pages 88–95.

David Yarowsky. 2000. Hierarchical decision lists for word sense disambiguation. *Computers and the Humanities*, 34(2):179–186.

Hugo Zaragoza and Ralf Herbrich. 2000. The perceptron meets reuters. In *Workshop on Machine Learning for Text and Images at NIPS 2001*.