Beam Search in Incremental Rule Learning

Gregory D. Weber
Department of Computer Science
Indiana University East
2325 Chester Boulevard
Richmond, IN 47374 USA
gdweber@indiana.edu

Abstract
This paper describes ICN, an incremental version of the CN2 rule learning system. Unlike other incremental rule learning systems which learn rules gradually, adding and removing conditions in a hill-climbing search, ICN learns or unlearns each rule “all at once,” using beam search as in CN2. In batch training and testing with the forest cover prediction problem, ICN performs nearly as well as CN2. ICN’s efficient incremental algorithm, however, allows it to learn from much more data. When trained and tested incrementally on the entire forest cover data set (581,012 instances), ICN’s performance exceeds that of the best known classifier for this problem.

Introduction
A learning system is incremental if it interleaves the development of its concept with the acquisition of examples; non-incremental, if it first acquires all of its training examples and only then begins to develop its concept. Autonomous agents that learn need to learn incrementally, since they cannot afford to wait to accumulate large quantities of experience before learning begins (Fisher 1987). Efficient incremental learning, with low memory requirements and fast processing time per example, may also be useful for knowledge discovery in large databases (“data mining”), because it enables learning from many more examples than could be stored at one time in memory.

Most incremental rule learning systems perform hill-climbing search. Hill climbing is a psychologically plausible model of human learning (Langley, Gennari, & Iba 1987). Such systems typically respond to a misclassification either by generalizing a rule (removing one or more conditions from it), specializing a rule (adding one or more conditions), adding a new rule, or deleting a rule. Some versions of hill-climbing simultaneously explore the space of rules, by adding and removing conditions of existing rules, and the space of rule sets (or lists), by adding or removing entire rules. Systems that take a hill-climbing approach to incremental rule learning include HILLARY (Iba, Wogulis, & Langley 1988), discrimination learning in PRISM (Langley 1987), ISC (Langley 1996), Complementary Discrimination Learning (CDL) (Shen 1994), and the rule-learning subsystem of CLARION (Sun & Peterson 1998).

Non-incremental or “batch” rule learning systems typically have two advantages over incremental systems. (1) Since they have access to all the training examples from the beginning to the end of learning, they can make more accurate estimates of statistics and use them to make better-educated guesses about what is the best rule to learn at each step. (2) They use more powerful and intensive methods of search, which are less likely than hill-climbing to get stuck in local optima. The non-incremental rule learning system CN2, for example, uses beam search to find the best rule. The first advantage follows from the nature of incremental and non-incremental learning: it is logically impossible for a learning system to acquire all the examples before learning and still be incremental. But the second advantage is only accidental, in the sense that it is logically unnecessary for incremental systems to use weaker methods of search such as hill climbing.

This paper presents ICN, an incremental rule learning system which uses the beam search of CN2. ICN is an efficiently incremental learning system, in the sense that its processing time per example and memory usage have constant bounds, no matter how many examples are processed.

The ICN Learning Algorithm
The ICN learning algorithm is an incremental version of CN2 (Clark & Niblett 1989; Clark & Boswell 1991). Both CN2 and ICN learn probabilistic rules of the form IF A1, A2, . . . , Am THEN F1, F2, . . . , Fk, where the Ai are attribute tests (conditions) and Fj is the frequency of class j (j = 1, 2, . . . , k). A set of probabilistic rules forms a probabilistic model suitable for a world characterized by uncertainty; probabilistic models have been successful in many areas of artificial intelligence, including natural language understanding and robotics (Thrun 2002). The main procedure of CN2 builds an unordered set of rules from a set of training examples as follows. For each class, it applies a sequential covering algorithm (Michalski 1969; Fünnkranz 1999), which repeatedly learns a rule that covers some instances of the class, then removes the covered instances from the training set. Each rule is learned by a general-to-specific beam search using the Laplacean accuracy estimate as an evaluation function; the accuracy estimate is (ni + 1)/Σi−1(ni + 1), where ni is the number of instances of class i matching the rule, c is the current class, and k is the number of classes. In addition, the search applies a chi-square test to determine if the “best” rule’s prob-
Begin CN2Learn()
  Ruleset ← ∅
  for each class C
    T ← all the training examples
    while (T contains instances of C and
      NewRule ← fi ndBestRule(C, T) succeeds)
      add NewRule to Ruleset
      remove from T all instances covered by NewRule
    end while
  end for
  add the default rule to Ruleset
  return Ruleset
end.

Figure 1: The CN2 main learning algorithm, CN2Learn.

ability distribution is significantly different from the default rule’s; if it is not, the rule is rejected and the search fails. The main algorithm and the search algorithm, fi ndBestRule, are shown in Figures 1 and 2.

ICN learns in three ways: by updating class frequencies in existing rules, by memorizing new instances, and by forming new rules. (Memorizing instances affects behavior only indirectly, through the effect of the instance memory on rule learning.) To keep memory usage in bounds, ICN forgets old instances and bad rules. ICN differs from most incremental rule learning systems in that: (1) it uses beam search instead of hill-climbing; (2) it learns only by specialization, not also by generalization; and (3) its basic learning step is to learn a complete rule, not a partial rule. ICN never modifies an existing rule by adding or removing conditions.

ICN uses the fi ndBestRule procedure of CN2 in an incremental main procedure, which maintains a rule set RS, a good set GS of instances that are correctly classifi ed by at least one non-default rule in RS, and a bad set BS of instances that are not correctly classifi ed by any of the non-default rules. The default rule is a rule with no conditions. ICN starts with RS = {the default rule}, GS = BS = ∅. Each time ICN receives an example, it attempts to classify it using the rules in RS and learns by updating the class frequencies of all matching rules. There is always at least one matching rule since the default rule matches every possible instance. ICN then “learns” by remembering the instance, storing it in GS if it is correctly classifi ed by one or more non-default rules, in BS otherwise. ICN begins learning new rules only when BS accumulates “enough” examples of any class (say, 10); that class then becomes a “learnable” class. If there are any learnable classes, ICN randomly selects one of them, which we will call C, and attempts to learn one rule for class C using the beam search of CN2 and the examples in GS and BS. (More precisely, the training set, as in CN2, contains all the negative instances in GS and all the instances in BS.) If ICN is able to discover a rule which is statistically signifi cant, it stores the rule in RS, removes the instances from BS which are correctly classifi ed by the new rule, and stores them in GS. ICN learns at most one rule before receiving the next instance. If, after receiving the next instance, there are still learnable classes, it will try to learn another rule.

If ICN is implemented naively, the sets RS, GS, and BS can grow without bound. The program can then run out of memory, or its performance can become unacceptably slow. To prevent this, ICN limits the sizes of RS, GS, and BS. Consequently, GS is only a sample, not the entire set, of instances correctly classifi ed by non-default rules. Similarly, BS is a sample of the instances not correctly classifi ed. By keeping GS and BS reasonably small, we can guarantee that learning one rule is reasonably fast. By keeping RS reasonably small, we can guarantee that classifying an instance is reasonably fast. By keeping RS, GS, and BS reasonably small, we can guarantee that ICN uses reasonable space. These guarantees together imply that ICN is an effi cient incremental learning and classifying system.

The limits on the sizes of RS, GS, and BS are achieved by interpreting the word “store” (as in “store an example or rule in a set”), in the above description of the algorithm, in a somewhat non-obvious sense. To store element X into set Z means: if Z is full, fi rst remove an element to make room; then insert X into Z. Various policies can be used to select the element to be removed, e.g., fi rst in fi rst out, least recently used, randomly selected, or, in the case of rules, remove the worst rule according to the Laplacean accuracy estimate or some other criterion of rule goodness. Other policies might aim to retain the same number of examples of each class, or to retain examples of each class in proportion to the class frequencies. It is probably desirable always to have a default rule, so the rule removal policy should re-
move only non-default rules.

This reinterpretation of “store” leads to a few complications which are necessary to keep the sets RS, GS, and BS consistent:

1. The program keeps a record of which non-default rules correctly classify which examples. This record can be implemented in a number of ways. For purposes of exposition, assume that the record is stored as a relation CC containing (rule, instance) tuples; a tuple \((r, i) \in CC\) means that non-default rule \(r\) correctly classified instance \(i\).

2. When ICN stores a new non-default rule \(r\) into RS, it inserts a tuple \((r, i)\) into CC for each instance \(i \in GS \cup BS\) which is correctly classified by \(r\). Any such instances in BS are moved to GS.

3. When ICN removes a non-default rule \(r\) from RS, for each tuple \((r, i) \in CC\), ICN also removes \((r, i)\) from CC and, if \(i\) is not correctly classified by any other non-default rule, moves \(i\) from GS to BS.

4. When ICN adds an instance \(i\) to GS, for each non-default rule \(r\) which correctly covers \(i\), it inserts the tuple \((r, i)\) into CC.

5. When ICN removes an instance \(i\) from GS, it removes each tuple \((r, i)\) from CC.

6. When adding or removing instances in BS, there is no need to update CC, since, by the very definition of BS, if \(i \in BS\), then \(i\) is not covered correctly by any non-default rule, and so there are no \((r, i)\) tuples in CC.

The incremental character of ICN requires one further complication. In CN2, a rule is learned to predict a particular class, and the predicted class normally has the highest frequency in the rule’s class frequencies.\(^1\) In ICN, too, the rule will normally have highest frequency for the class it predicts, when the rule is learned. But the frequencies can “go bad” as experience accumulates, so that after some time, the rule’s predicted class has a lower frequency than some other class. In this case, ICN considers the rule to be a bad rule, removes it from RS, and makes appropriate corrections to GS, BS, and CC. Similarly, if the rule’s most-likely class frequency drops below a user-determined threshold, the rule is considered bad and removed. Without these forms of rule pruning, ICN would be unable to recover from bad rules learned from early, non-representative examples of the data set.

These considerations lead to the ICN-Learn-and-Predict algorithm for incremental learning and predicting (see Figure 3).

Since the membership of GS and BS changes only gradually as new examples are received, it is unlikely that the system will succeed in learning a new rule if it has recently attempted to do so and failed. It is certain that time can be saved by not trying to learn a rule. To speed up processing, the ICN program allows the user to set the parameter WaitBetweenRules to specify how long it will wait between attempts to learn a new rule, for example, every 10 or every 100 training examples.

---

\(^1\)This agreement between predicted class and frequencies can be forced by setting CN2’s “maximum class forcing” parameter.

Algorithm ICN-Learn-and-Predict

Parameters: \(MaxRS\), \(MaxGS\), \(MaxBS\) = maximum sizes for sets RS, GS, BS; MinLearn = minimum number of instances of a class to enable learning a new rule; Alpha = chi-square significance level; DegradeFactor = factor used to set threshold for rule pruning, \(0 \leq \text{DegradeFactor} < 1\).

Inputs: ES= a stream of examples; Classes = list of classes which are possible.

Begin

\(RS \leftarrow \{\text{the default rule}\}\)
\(GS \leftarrow \emptyset; BS \leftarrow \emptyset; CC \leftarrow \emptyset\)

while \(ES \neq \emptyset\) do

\(E \leftarrow \text{next instance from } ES\)
\(\text{Attrs} \leftarrow \text{attributes}(E)\)
\(\text{Matches} \leftarrow \text{matching-rules}(RS, \text{Attrs})\)
\(\text{Pred} \leftarrow \text{predictions}(\text{resolve-conflicts}(\text{Matches}), \text{Attrs})\)
\(\text{AClass} \leftarrow \text{class}(\text{Pred})\) (i.e., the actual class)
\(\text{CMatches} \leftarrow \{\text{non-default rules in Matches which predicted AClass as the most likely class}\}\)

if \(\text{CMatches} \neq \emptyset\) then

store \(E\) in GS; update CC

else

store \(E\) in BS

end if

for each rule \(r\) in Matches, add 1 to \(r\)’s class frequency \(F_{\text{AClass}}\)

end for

for each non-default rule \(r\) in Matches

if \(r\) no longer predicts its “majority class,”

or \(r\)’s accuracy is below its pruning threshold, then

forget-rule\((r, RS, GS, BS, CC)\)

end if

end for

\(\text{Learnable} \leftarrow \{\text{classes with at least MinLearn examples in BS}\}\)

if \(\text{Learnable} \neq \emptyset\) then

\(C \leftarrow \text{randomly chosen class in Learnable}\)

SampleSet \(\leftarrow GS \cup BS\)

TrainingSet \(\leftarrow \text{SampleSet} \setminus \{\text{instances of } C \text{ in } GS\}\)

\(NR \leftarrow \text{niBestRule}(C, \text{TrainingSet})\)

if \(NR\) is a rule (i.e., learning succeeded) then

set \(NR\)’s class frequencies \(F_C\) by counting matches in SampleSet

set \(NR\)’s pruning threshold as \(\text{DegradeFactor} \cdot F_C\)

store \(NR\) in \(RS\); update \(GS, BS, \text{and CC}\)

end if

end while

end.

---

Figure 3: ICN main algorithm
Performance

This section compares the performance of ICN and CN2, using accuracy and other measures. For clarity we distinguish between two versions of the CN2 algorithm and two implementations of CN2. We designate the original published CN2 algorithm (Clark & Niblett 1989) as CN2-A, and the "improved" published CN2 algorithm (Clark & Boswell 1991) as CN2-B. We refer to the "offical" implementation of CN2-B, in the C programming language (Clark et al. 1996), as CN2-C; and to our re-implementation of CN2-B, in the Scheme programming language, as CN2-S.

Classification accuracy is a legitimate measure of performance, but not sufficient by itself when the predictions are probabilistic (Weber 2003). We include three additional measures of performance. Extrinsic confidence (EC) is the average predicted probability of the actual class. Intrinsic confidence (IC) is the average maximum predicted probability. Entropy is well-known as a measure of uncertainty. Lower entropy values are better; for the other measures of performance, higher values are better.

The forest cover problem is to predict the type of forest cover (spruce fir, lodgepole pine, ponderosa pine, cottonwood/willow, aspen, Douglas fir, or krummholz) from cartographic variables, such as elevation, slope, distance from water and roadways, and soil type. There are 10 quantitative variables and 44 binary variables. The data set, available from the UCI KDD Repository (Bay 1999), contains 581,012 instances. Blackard reported 70% accuracy using a neural network with backpropagation and 58% using a linear discriminant. These results were obtained by using the first 11,340 instances as training data, the next 3,780 instances as validation data, and the remaining instances for testing (Blackard 1998).

Both CN2-C and CN2-S were trained using the combination of Blackard’s training and validation sets (i.e., the first 15,120 instances) and tested with the remaining 565,892 instances. Both programs used the default parameters of CN2-C except that maximum class forcing was set "on" and the chi-square critical value was set for a 1% significance level. Although CN2-C achieved 74.0% accuracy on the training set, its accuracy on the test set was only 55.8%, suggesting that in spite of the chi-square significance test it was overfitting. CN2-S achieved similar accuracy, 54.7%, on the test set.

An incremental learning system can be trained and tested in "batch" mode by running it through the training data with learning turned "on" and then running it through the test data with learning turned "off." In a way, this method of training and testing maximizes comparability between incremental and non-incremental systems, since it is clear that they both learn from the same set of examples. When ICN was trained and tested in batch mode with sufficient large instance and rule memories, it achieved nearly the same quality of predictions while consuming significantly less memory. ICN’s accuracy was 53.2%. Other measures of prediction quality were also close to those of CN2-S. When ICN’s instance and rule memories were shrunk, its predictions grew worse. Table 1 shows performance details. ICN used about the same amount of CPU time as CN2-S, but used only about a fifth as much memory for instance storage and half as much memory for rule storage. Table 2 shows details of resource usage.

However, batch training and testing does not permit the incremental learner to use its main strength, i.e., its capacity to learn from many more examples than a non-incremental system. For an incremental learner to compete in batch training and test mode is like boxing with a hand tied behind one’s back. As an efficient incremental learner, ICN is capable of simultaneously (1) training on all the data and (2) testing on all the data. Each instance serves, first, as a test example, for interpretation, only 22 times greater than CN2-C. It performed much better than CN2-S and, despite the slowdown from incremental training and testing maximizes comparability between incremental and non-incremental systems, since it is clear that they both learn from the same set of examples. When ICN was trained and tested in batch mode with sufficient large instance and rule memories, it achieved nearly the same quality of predictions while consuming significantly less memory. ICN’s accuracy was 53.2%. Other measures of prediction quality were also close to those of CN2-S. When ICN’s instance and rule memories were shrunk, its predictions grew worse. Table 1 shows performance details. ICN used about the same amount of CPU time as CN2-S, but used only about a fifth as much memory for instance storage and half as much memory for rule storage. Table 2 shows details of resource usage.

However, batch training and testing does not permit the incremental learner to use its main strength, i.e., its capacity to learn from many more examples than a non-incremental system. For an incremental learner to compete in batch train

### Table 1: Prediction quality. Due to the randomness of the ICN algorithm, results shown are the averages from three independent runs. Notes: (b) Batch training and testing.

<table>
<thead>
<tr>
<th>Inducer</th>
<th>Accuracy</th>
<th>EC</th>
<th>IC</th>
<th>Entropy</th>
</tr>
</thead>
<tbody>
<tr>
<td>CN2-C (b)</td>
<td>55.8</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>CN2-S (b)</td>
<td>54.7</td>
<td>39.4</td>
<td>49.2</td>
<td>1.87</td>
</tr>
<tr>
<td>ICN (b, 1)</td>
<td>38.0</td>
<td>34.2</td>
<td>58.5</td>
<td>1.54</td>
</tr>
<tr>
<td>ICN (b, 2)</td>
<td>53.2</td>
<td>39.1</td>
<td>55.5</td>
<td>1.70</td>
</tr>
<tr>
<td>ICN (i, 3)</td>
<td>75.1</td>
<td>67.9</td>
<td>81.5</td>
<td>0.80</td>
</tr>
</tbody>
</table>

### Table 2: Computational efficiency. Time is (training time + testing time) in CPU minutes, on a 750-MHz AMD Athlon processor (i686-compatible) with 128 MB RAM. Most of the time is for training. CN2-S (interpreted Scheme) was 10 times slower than CN2-C (compiled C); but it used only 10 MB of memory for training, while CN2-C used 63 MB. CN2-C was unable to load all of the test data into memory at once; testing required splitting the test data file into several smaller files, testing on each of them in succession, and averaging the results. ICN(i, 3) trained on 38 times as many examples as CN2-S and CN2-C, but its running time is only 2.4 times that of CN2-S and, despite the slowdown from interpretation, only 22 times greater than CN2-C.

...
ICN in batch train and test mode (see Table 1). Furthermore, incrementally training and testing ICN used less memory than batch training and testing for any of the learning systems, and it used only 2.4 times as much CPU time as batch CN2-S, in spite of using 38 times as much training data (see Table 2).

Figure 4 shows learning curves for ICN. All four measures of prediction quality improve sharply during the first 50,000 training examples and continue to improve until they reach an optimum level after about 150,000 examples. Most remain level from then onwards, although the entropy curve rises slightly.

Figure 5 shows resource usage curves for ICN. Memory usage (number of rules, number of conditions, and rule space) rises sharply during the first few thousand training examples up to the maximum number of rules allowed. Although CPU time per example rises gradually during the first 200,000 examples, it gradually declines thereafter. Thus, resource usage remains within constant limits.

**Conclusions**

ICN performed as well as could be hoped in batch learning tests, and far better than expected in incremental learning, on the forest cover prediction problem.

It may seem, to some readers, that comparing ICN’s performance under an incremental training and testing regime, to the performance of other inducers (and of ICN itself) under a batch training and testing regime, is at best irrelevant, at worst unfair. Is it not like comparing two athletes, A and B, by observing that A can swim a mile in 30 minutes, but B can run the same distance in only 4 minutes? Certainly, the change of training and testing methods from batch to incremental is a change of task, and it is to be expected that performance of a learning system will be different for the two tasks. Nevertheless, it may be a relevant and fair comparison, if the new task is at least as well suited to the problem as the old. Changing the task is sometimes productive, as when the German army decided to go around, not through, the Maginot Line. For another example, suppose an urgent message must be delivered to a post a mile away, which can be reached both by land and by water routes. If A can only deliver the message by swimming, but B can deliver it by running, B is better for the job, even if B’s swimming is no better than A’s. On the other hand, if the only route is by water, then running speed becomes irrelevant.

The question comes down to this: is incremental learning sometimes a task which is at least as appropriate as batch learning? Both incremental and batch learning methods enable the problem solver to predict the class of previously unseen examples. In the case of rule learners, the batch learners do this with a static set of rules (it does not change after learning), while incremental learners employ a dynamically changing set of rules (learning never stops). A static rule set can have value. For example, it is possible to use a static rule set to explain why the predicted classifications are as they are, and such explanations may be more difficult or impossible if the rule set is dynamic. On the other hand, if the only goal is to make the best possible predictions, then incremental learning may be at least as appropriate as batch learning. Since neural networks provide the power predict but not the power to explain the predictions (at least not easily), it seems that incremental rule learning would be suitable in every case where neural networks are suitable. In all such cases, therefore, a change of task from batch to incremental learning seems fair and reasonable.

The question remains, why is ICN’s performance so
good? Is it simply because ICN is able to learn from far more examples than any of the programs in batch mode? Or does ICN benefit from its ability, as an incremental learner, to track a "drifting" concept? In other words, as we move through the file of forest cover examples from beginning to end, do the characteristics of the different classes change, so that a good description of the classes from one part of the file is a poor description of the classes from another part? Or maybe ICN was just lucky on this one problem, and it will perform badly on everything else.

It should be possible to answer these questions by further experimentation. Testing ICN on a variety of learning problems will reveal whether its high performance is generally applicable, or just the luck of the draw. To test whether there is a concept drift benefit, ICN might be run for the first 150,000 examples with learning on, then run through the remaining examples with learning off. Alternatively, ICN might be tested with random reorderings of the example file. If concept drift is a significant factor, we should expect ICN’s performance to be worse under either of these conditions. If ICN performs well on a variety of data sets and shows no indications of being aided by concept drift, then the conclusion will have to be that its strength is simply its ability to learn from a huge number of examples.

References


