Appears in *Proceedings of the Eleventh International Workshop on Machine Learning*
pp. 343-351, Rutgers, NJ, July, 1994

Combining Top-down and Bottom-up Techniques in Inductive Logic Programming

John M. Zelle, Raymond J. Mooney and Joshua B. Konvisser
Department of Computer Sciences
University of Texas
Austin, TX 78712
zelle,mooney,konvissa@cs.utexas.edu

Abstract

This paper describes a new method for inducing logic programs from examples which attempts to integrate the best aspects of existing ILP methods into a single coherent framework. In particular, it combines a bottom-up method similar to GOLEM with a top-down method similar to FOIL. It also includes a method for predicate invention similar to CHAMP and an elegant solution to the "noisy oracle" problem which allows the system to learn recursive programs without requiring a complete set of positive examples. Systematic experimental comparisons to both GOLEM and FOIL on a range of problems are used to clearly demonstrate the advantages of the approach.

1 INTRODUCTION

Inductive Logic Programming (ILP) research investigates the construction of first-order, definite-clause logic programs from a set of examples and given background information. As such, it stands at the intersection of the traditional fields of machine learning and logic programming. The explosion of recent research in this area has clustered around two basic induction methods, bottom-up and top-down.

Bottom-up methods, drawing heavily on logic programming theory, search for program clauses by considering generalizations created by inverting logical resolution or, more generally, implication. A successful representative of this class is Muggleton and Feng's GOLEM (Muggleton and Feng, 1992). Top-down methods, in contrast, learn clauses by searching from general to specific in a manner analogous to traditional machine-learning approaches for inducing decision trees. Perhaps the best-known example is Quinlan's FOIL(Quinlan, 1990; Cameron-Jones and Quinlan, 1994) which uses an information heuristic to guide search through the space of possible program clauses.

While both GOLEM and FOIL have been successful, each of these approaches has weaknesses. GOLEM is based on the construction of relative least-general generalizations, *rlggs* (Plotkin, 1970) which forces the background knowledge to be expressed extensionally as a set of ground facts. This explicit model of background knowledge can be excessively large, and the clauses constructed from such models can grow explosively. A partial answer to the efficiency problem is the restriction of hypotheses to the so-called ij-determinate clauses. A related problem is sensitivity to the distribution of input examples. If only a random sample of positive examples is presented, the resulting model of the predicate to be learned is incomplete, and GOLEM may fail to create sufficiently general hypotheses, resulting in diminished performance.

FOIL also uses extensional background knowledge, but this requirement is for efficiency reasons; top-down algorithms can easily use intensionally defined background predicates to evaluate various competing hypotheses. A more fundamental weakness is that FOIL constructs clauses which are function-free. Any functions (e.g. list structures) must be handled by including explicit constructor predicates as part of the background knowledge. The proliferation of constructor predicates can significantly degrade FOIL's performance. In addition, FOIL suffers its own version of the incomplete model problem when trying to learn recursive predicates. Recursive hypotheses are evaluated by using the positive examples as a model of the predicate being learned. When the examples are incomplete, they provide a "noisy oracle" and FOIL has difficulty learning even simple recursive concepts (Cohen, 1993).

This paper describes a new ILP algorithm, CHILLIN[1], which combines elements of top-down and bottom-

[1]For CHILL, INduction algorithm. CHILL is a language acquisition system based on learning control-rules for logic programs (Zelle and Mooney, 1993).

up induction methods. This algorithm has been used as an element of a larger system which learns to parse natural language (Zelle and Mooney, 1993). CHILLIN's combination of techniques offers several advantages. CHILLIN learns with intensionally expressed background knowledge and can handle examples containing functions without explicit constructor predicates. Additionally, CHILLIN provides a simple, efficient solution to the noisy-oracle problem, resulting in better performance than GOLEM or FOIL when learning from random examples. Finally, CHILLIN provides a simple framework for demand-driven invention of new predicates.

The following section presents the CHILLIN algorithm and discusses the ways in which it improves on strictly top-down or bottom-up methods. Section 3 presents experimental results across a range of domains comparing CHILLIN to FOIL and GOLEM. Section 4 briefly discusses some related work. Section 5 outlines future work, and Section 6 presents our conclusions.

2 THE CHILLIN ALGORITHM

2.1 OVERVIEW

The input to CHILLIN is a set of positive and negative examples of a concept expressed as ground facts, and a set of background predicates expressed as definite clauses. CHILLIN constructs a definite-clause concept definition which entails the positive examples, but not the negative.

CHILLIN is, at its core, a compaction algorithm which tries to construct a small, simple program that covers the positive examples. The algorithm starts with a most specific definition (the set of positive examples) and introduces generalizations which make the definition more compact. Compactness is measured by a CIGOL-like size metric (Muggleton and Buntine, 1988) which is a simple measure of the syntactic size of the program. The search for more general definitions is carried out in a hill-climbing fashion. At each step, CHILLIN considers a number of possible generalizations; the one producing the greatest compaction is implemented, and the process repeats.

Generalizations are produced under the notion of *empirical subsumption*. Intuitively, the algorithm attempts to construct a clause that, when added to the current definition, renders other clauses superfluous. The superfluous clauses are then eliminated to produce a more compact definition. Formally, we define empirical subsumption as follows: Given a set C of Clauses $\{C_1, C_2, \ldots, C_N\}$ and a set of positive examples E provable from C, a clause G empirically subsumes C_i iff $\forall e \in E : [(C - C_i) \cup G \vdash e]$. Throughout the rest of the paper, unless otherwise noted, the term "subsumption" should be interpreted in this empirical sense.

```
DEF := {E :- true | E ∈ Pos}
Repeat
   PAIRS := a sampling of pairs of clauses from DEF
   GENS := {G | G = build_gen(C_i,C_j,DEF,Pos,Neg)
                for ⟨C_i, C_j⟩ ∈ PAIRS}
   G := Clause in GENS yielding most compaction
   DEF := (DEF−(Clauses subsumed by G)) ∪ G
Until no further compaction
```

Figure 1: Basic Induction Algorithm

Figure 1 shows the basic compaction loop of CHILLIN. Like GOLEM, CHILLIN constructs generalizations from a random sampling of pairs of clauses in the current definition. The number of pairs chosen is a user-settable parameter which defaults to 15. The best generalization from these pairs is used to reduce `DEF`. The reduction is performed by adding `G` at the top of the definition and then using the standard Prolog proof strategy to find the first proof of each positive example; any clause which is not used in one of these proofs is then deleted from the definition.

2.2 CONSTRUCTING GENERALIZATIONS

The `build_gen` algorithm is shown in Figure 2. There are three basic processes involved. First is the construction of a simple generalization of the input clauses. If this generalization covers no negative examples, it is returned. If the initial generalization is too general, an attempt is made to specialize it by adding antecedents. If the expanded clause is still too general, it is passed to a routine which invents a new predicate that further specializes the clause so that it covers no negative examples. These three processes are explained in detail in the following subsections.

2.2.1 Constructing an Initial Generalization

The initial generalization of the input clauses is computed by finding the least-general-generalization (LGG) of the input clauses under theta-subsumption (Plotkin, 1970). The LGG of clauses C_1 and C_2 is the least general clause which subsumes (in the usual, non-empirical sense) both C_1 and C_2. The LGG is easily computed by "matching" compatible literals of the clauses; wherever the literals have differing structure, the LGG contains a variable. When identical pairings of differing structures occurs, the same variable is used for the pair in all locations.

For example, when learning the definition of `member`, the initial definition may contain clauses such as:

```
member(1, [1,2,3]) :- true.
member(2, [1,2,3]) :- true.
member(3, [3]) :- true.
```

The LGG of the first and third clauses is: `member(A,[A|B]):- true`. Here the new variable, `A`,

Function build_gen(C_i, C_j, **DEF**, **Pos**, **Neg**)
GEN := LGG(C_i,C_j) under θ-subsumption
CNEGS := Negatives covered by GEN
if CNEGS = {} **return** GEN

GEN := add_antecedents(Pos, CNEGS, GEN)
CNEGS := negatives covered by GEN
if CNEGS = {} **return** GEN

REDUCED := DEF - (Clauses subsumed by GEN)
CPOS := {e | e $\in$ Pos $\wedge$ REDUCED $\nvdash$ E }
LITERAL := invent_predicate(CPOS, CNEGS, GEN)
GEN := GEN $\cup$ LITERAL
return GEN

Figure 2: Build_gen Algorithm

has been used for both pairings of the constants `1` and `3`. This LGG is a valid generalization (it covers no negative examples of `member`) and, in fact, represents the correct base case for the usual definition. Of course, such generalizations are not always correct. The LGG of the second and third clauses is: `member(A,[B|C]) :- true` which asserts that anything is a member of a list which contains at least one element. This generalization requires further refinement to prevent coverage of negative examples.

Although the initial definitions in CHILLIN consist of unit clauses (the only antecedent being `true`), as the definition becomes more compact, the clauses from which LGGs are constructed may contain non-trivial conditions. LGG construction is still straight-forward; we need only be sure that consistent variable usage is maintained between as well as within literals. For example, consider two clauses defining the concept of `uncle`:

```
uncle(X,Y) :-
  sib(X,Z), parent(Z,Y), male(X).
uncle(X,Y) :-
  married(X,Z), sib(Z,W), parent(W,Y), male(X).
```

The LGG of these clauses yields:

```
uncle(A,B) :- sib(C,D), parent(D,B), male(A).
```

Here `A` replaces the pair `<X,X>`, `B` replaces `<Y,Y>`, etc.

Unlike the RLGGs used by GOLEM, LGGs are independent of any background knowledge and efficiently computable from the input clauses. At this point, GEN is guaranteed to be at least as general as either input clause, but may also cover negative examples. The process also effectively introduces relevant variables which directly decompose functional structures; this allows subsequent specialization without resort to explicit flattening through constructor predicates.

2.2.2 Adding Antecedents

As its name implies, `add_antecedents` attempts to specialize `GEN` by adding new literals as antecedents. The goal is to minimize coverage of negative examples while insuring that the clause still subsumes existing clauses. `Add_antecedents` employs a FOIL-like mechanism which adds literals derivable either from background or previously invented predicates. Antecedents are added one at a time using a hill-climbing process; at each step a literal is added that maximizes a heuristic gain metric.

The gain metric employed in CHILLIN is a modification of the FOIL information-theoretic gain metric. The count of positive tuples (loosely, the number of covered positive examples) in the FOIL metric is replaced with an estimate of the number of clauses in `DEF` which are subsumed by `GEN`. This estimate is obtained by partitioning the set of positive examples according to the first clause in `DEF` which covers a given example. The entire set of examples is proved once using `DEF`, and subsequent testing of `GEN` extensions is performed over the partitioned examples.

As an example of this process, consider learning the two-clause definition of `uncle` illustrated above. Initially, `DEF` contains unit clauses representing the positive examples. Constructing the LGG of two clauses, say `uncle(john, jay) :- true` and `uncle(bill sue) :- true` produces the unhelpful generalization `uncle(A,B) :- true` which covers all examples. Partitioning the positive examples according to the first covering-clause results in associating each example with the unit clause constructed from the example. Thus, the count of subsumed clauses for any given extension of GEN is just the count of positive examples covered by GEN; `add_antecedents` will perform analogously to FOIL, adding literals to `GEN` one at a time to maintain maximal coverage of positive examples and eliminate coverage of negatives. Depending on the distribution of examples, `add_antecedents` might produce either of the `uncle` clauses. When the outer loop of CHILLIN adds this clause to the definition, all of the unit clauses for the examples covered by the clause are removed.

In the next cycle the LGG of two remaining unit clauses may again produce the `GEN`, `uncle(A,B):-true`. This time, the partitioning in `add_antecedents` associates numerous examples with the first clause (the previously constructed generalization) and associates one example with each of the unit clauses remaining in `DEF`. In extending `GEN` at this point, the antecedents which had high gain previously will now look very poor; although they cover many positive examples, they only allow subsumption of one clause (namely the clause previously found). On the other hand, the antecedents of the second `uncle` disjunct will have very high gain as they allow the subsumption of the remaining unit clauses while eliminating negative examples. `Add_antecedents` will then learn the second clause, and the definition is complete. Of course CHILLIN will try one more round of compaction before discovering that the two disjuncts of `uncle` may not be combined to form an even more

compact definition.

This discussion has assumed that `add_antecedents` has predicates available which will allow it to completely discriminate between the positive and negative examples; however, this is not always the case. In such situations, `add_antecedents` may add a few antecedents and then be unable to extend the clause further because no literal has positive gain. This partially completed clause is then passed to `invent_predicate` for completion.

2.2.3 Inventing New Predicates

Predicate invention is carried out in a manner analogous to CHAMP(Kijsirikul et al., 1992). The first step is to find a minimal-arity projection of the clause variables such that the set of projected tuples (a tuple is a particular instantiation of variables appearing in the clause) from proofs of positive examples is disjoint with the projected tuples from proofs of negative examples. These ground tuple sets form the positive and negative example sets for the a new predicate. The top-level induction algorithm is recursively invoked to create a definition of the predicate.

CHAMP searches for a minimal projection of variables by greedily removing variables that are not necessary to maintain the disjointness of the tuple sets. CHILLIN differs in that it starts with no variables and greedily adds those variables which help separate the examples and also minimize the set of positive examples for the new predicate. Variables are chosen to maximize the number of negative examples eliminated per additional positive tuple created. The search terminates when all negative examples have been eliminated or there are no more variables to add.

Suppose that we are trying to learn `uncle`, but do not have a definition of `male`. After `add_antecedents`, `GEN` might be: `uncle(A,B) :- sibling(A,C), parent(C,B)` which still covers some negative examples. Using this clause to prove some positive and negative examples might result in the set of bindings shown here in tabular form:

Set	A	B	C
Pos	john	jay	mary
	bill	sue	bruce
Neg	liz	jay	mary
	liz	sue	mary

Notice that variables `B` and `C` have overlap between positive and negative examples, while `A`'s values are disjoint. `Invent_predicate` will select the single variable `A` as the minimal projection and create positive examples `p1(john)` and `p1(bill)`, along with the negative example `p1(liz)`. Calling the top-level induction algorithm on these examples produces no compaction, so the learned definition of `p1` will just be a listing of the positive examples. Finally, `build_gen` completes its clause by adding the final literal `p1(A)` which is the newly invented predicate representing `male`.

Once a predicate has been invented and found useful for compressing the definition, it is made available for use in further generalizations. This enables the induction of clauses having multiple invented antecedents, something which is not possible in the strictly top-down framework of CHAMP.

2.2.4 Handling Recursion

When introducing clauses with recursive antecedents, care must be taken to avoid unfounded recursion. FOIL deals with this issue by attempting to establish an ordering on the arguments which may appear in a literal. CHILLIN takes a much simpler approach based on structure reduction: each recursive literal must have an argument that is a proper subterm of the corresponding argument in the head of the clause. For example, in the clause `member(A, [B|C]) :- member(A,C)`, the second argument of the recursive literal is structure reducing, and any recursive chaining of this clause must eventually "bottom-out." Well-founded recursion among multiple clauses is guaranteed by ensuring that every recursive literal has at least one argument that is structure reducing, and for all other recursive literals in the definition, the same argument is a (possibly improper) subterm of the corresponding argument in the head of the containing clause. This property is maintained by dropping any unsound recursive literals produced by the LGG operation and only considering addition of recursive antecedents which meet the structure-reducing conditions. These restrictions on recursive definitions are strictly stronger than those imposed by FOIL, but this simple approach works well on a large class of problems.

The evaluation of recursive clauses also requires some consideration. Testing the coverage of a non-recursive clause is easily achieved by unifying the head of a clause with an example and then attempting to prove the body of the clause using the background theory. Evaluation of a recursive clause, however, requires a definition of the concept being learned. As mentioned in the introduction, FOIL and GOLEM rely on the extensional definition provided by the positive examples, giving rise to the noisy-oracle problem when these examples are incomplete. CHILLIN, on the other hand, is able to use the current definition of the predicate being learned, which is guaranteed to be at least as general as the extensional definition. Coverage of recursive clauses is tested by temporarily adding the clause to the existing definition and evaluating the antecedents in the context of the background knowledge and the current (extended) definition. In this way, generalization of the original examples (say the discovery of the recursive base-case) can significantly improve the cov-

erage achieved by correct recursive clauses. This approach gives CHILLIN a significant advantage in learning recursive concepts from random examples.

This approach to recursion has proven effective in practice, although it is not without shortcomings. If a recursive clause is introduced and subsequent generalizations expand the coverage of the recursive call, the resulting definition could cover negative training examples; the current implementation does not check to insure that new generalizations maintain global consistency (although this would be easy to do). Such undesirable ordering effects do not often arise because recursive clauses do not generally show high gain until adequate base-cases have been constructed. Of course, if the data allow overly-general base-cases, then it is possible that the recursive clause may not be generated at all.

2.3 EFFICIENCY CONSIDERATIONS

The actual implementation of CHILLIN is somewhat more complicated than the abstract description presented so far. As the above discussion indicates, the process of constructing a generalization involves three steps: form an LGG, add antecedents and invent a new predicate. If this much effort was expended for a reasonable sampling of clause pairs on every iteration of the compaction loop, the algorithm would be intolerably slow. The current implementation provides two remedies for this problem.

First, the outer compaction loop is initially performed using only the LGG construction process to find generalizations. When no more compaction is found using simple LGGs, the more sophisticated refinement mechanisms are tried. Significant compaction is often obtainable in the initial phase, reducing the size of the theory on which the subsequent (more intensive) processing is done. Examples in the control-rule domains for which CHILLIN was designed are often highly structured, and this initial pass can reduce thousands of examples to a definition containing only tens of unit clauses.

A second conservation of effort is achieved by interleaving the building of generalizations. A given iteration of the compaction loop begins by gathering a sampling of clause pairs from which LGGs are immediately constructed. These generalizations form a pool of clauses which may need further refinement. CHILLIN proceeds by repeatedly removing the most promising clause and extending it with a single antecedent. The resulting clause is then returned to the pool and the process continues. If the selected clause is unextendable, it is set aside as a candidate for predicate invention. Predicate invention is invoked only if the pool of clauses has been exhausted without finding a valid generalization.

Using this more efficient approximation of the abstract algorithm, the current system, implemented in Quintus Prolog, running on a SparcStation 2, can handle induction problems involving thousands of examples.

3 EXPERIMENTAL EVALUATION

CHILLIN has been primarily used within a larger control-rule learning framework. However, we have undertaken a series of experiments to compare CHILLIN's performance with GOLEM and FOIL on some benchmark ILP tasks. We chose to compare against these systems because they are well-known, and arguably the most mature and efficient ILP platforms developed to date.

3.1 EXPERIMENTAL DESIGN

There is, as yet, no standard approach to the evaluation of ILP systems. We are primarily interested in the ability of systems to perform "realistic" learning tasks. That is, given some (random) sampling of examples how well do the learned hypotheses characterize the entire example space. Therefore, we have adopted an experimental strategy, common in propositional learning, of randomly splitting the example space into disjoint training and testing sets. The systems were trained on progressively larger portions of the training examples and the performance of the learned rules assessed on the independent testing set. This process of splitting, training and testing was repeated and the results averaged over 10 trials to produce learning curves for each of the systems on several benchmark problems. It is important to note that ILP systems are often tested using a set of complete or carefully chosen positive examples. We would not necessarily expect the systems to perform as well under the more realistic conditions of random selection used here.

The number of training examples in successive training sets was chosen experimentally to highlight the interesting parts of the learning curves. Except where indicated, enough training examples were provided so that the system having the best accuracy achieved a perfect score on the majority of the runs. The distribution of positive examples in many relational domains is quite sparse, and a relatively large number of positive examples are required for each of these ILP systems. In order to insure a reasonable number of positive training examples, training sets were always selected to be one-fifth positive and four-fifths negative examples. Testing sets included an equal number of positive and negative examples to test the ability of the resulting rules to recognize instances of the concept and reject non-instances.

Our experiments were performed using version 5.0 of FOIL and version 1.0 alpha of GOLEM both of which are written in C. All of the algorithms were run with default settings of the various parameters. No extra mode, type, or bias information was provided besides

the examples and background predicates. While all of the algorithms can make use of additional constraints, they do not necessarily do so in consistent ways; therefore, providing no extra information to any algorithm allows for a more direct comparison.

3.2 ACCURACY RESULTS

3.2.1 Learning Recursive Programs

The first three learning problems tested the ability to learn simple recursive concepts. We chose three problems widely used in the ILP literature: the list predicates `member` and `append` and the arithmetic predicate `multiply`.

For the list predicates, the data consisted of all lists of length 0-3 defined over three constants. The background information consisted of definitions of list construction predicates, `null` which holds for an empty list and `components` which decomposes a list into its head and tail. The results for these two problems were approximately the same. The learning curves for `member` are presented in the left-hand graph of Figure 3. As expected, with random examples, CHILLIN was able to learn accurate definitions with fewer examples than the other systems, and without using the background predicates.

The domain for the `multiply` problem consisted of integers in the range from zero to ten. The definition was to be learned in terms of background predicates: `plus`, `decrement`, `zero`, and `one`. We expected FOIL and GOLEM to do well on this problem as it is a standard benchmark which both systems have been shown capable of learning. CHILLIN, in its current form, is not able to formulate the correct recursive definition for this predicate, since the required recursive clause does not meet the structure-reducing conditions.

The learning curves, shown on the right-hand side of Figure 3, turned out to be quite surprising. None of the systems showed the ability to learn this concept accurately from random examples. CHILLIN quickly converged to definitions that were 90 percent correct for the limited domain, and was unable to improve. It's inaccurate definitions, however, were much better than those found by either of the other systems. Further experimentation showed that FOIL kept improving as the training set grew, but it was only reliable in generating correct definitions with nearly complete training sets. GOLEM was unable to learn the correct definition without additional guidance.

3.2.2 Learning with Nondeterminate Literals

Another traditional testbed for relational learners is the domain of family relationships. We performed experiments with an extended family tree in which the target predicate was either `grandfather` or `uncle` and the background consisted of facts concerning the relations: `parent`, `sibling`, `married`, `male` and `female`. This domain is interesting because it requires the use of literals which violate determinacy conditions used by GOLEM and other bottom-up ILP systems.

As expected, CHILLIN and FOIL do quite well on these problems, and GOLEM is unable to learn any reasonable definitions. On the `uncle` problem, both FOIL and CHILLIN learned accurate definitions from 100 training examples, with FOIL having a slight edge over the 10 trial average. Rather surprisingly, however, FOIL seemed to have more trouble on the simpler `grandfather` definition. As can be seen in the learning curves in the left-hand graph Figure 4, FOIL's performance takes a mysterious dip at 75 training examples before catching up with CHILLIN at 125 examples. Even at 175 examples where CHILLIN succeeds in finding a correct definition in all 10 trials, FOIL is only learning the correct definition half of the time. These experiments indicate that CHILLIN, like FOIL is able to learn definitions containing nondeterminate literals.

3.2.3 Control-rule Learning

The previous experiments concerned learning well-defined concepts containing only one or two clauses. CHILLIN was originally designed for learning control rules from structured examples where the definition of the correct concept is not necessarily simple, and certainly is not known *a priori*. For the last experiment we wanted to compare the performance of these systems on this type of problem. We chose a relatively simple task of determining when a shift-reduce parser should perform a shift operation in parsing a simple, regular corpus of active sentences.[2] CHILLIN typically learns a five or six clause definition for this concept.

The data for this problem was modified slightly so that the only logical functions appearing in the examples are list constructions. GOLEM and CHILLIN can both handle these structures without explicit constructor predicates. Unfortunately, it is not possible to run FOIL on this data. FOIL requires extensionally expressed constructor predicates; the `components` relation over lists of the required size (up to 8) constructible from the set of 34 constants appearing in these examples would require trillions of background facts. This illustrates the difficulties posed by the extensional background requirement.

The right-hand graph of Figure 4 shows the learning curves. On this problem CHILLIN tends to invent new predicates. For direct comparison, we performed the experiments with two versions of CHILLIN; the curve labeled "CHILLIN–" is CHILLIN with predicate invention turned off. The learning curves show that CHILLIN rapidly converges to very good definitions.

[2] This data is derived from a framework for parsing sentences from (McClelland and Kawamoto, 1986) as described in (Zelle and Mooney, 1993).

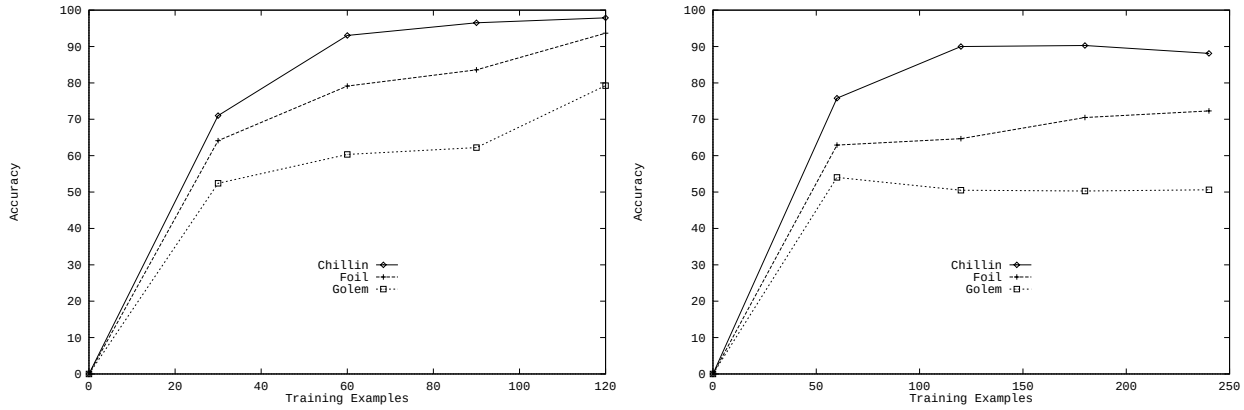

Figure 3: Accuracy on Recursive Definitions

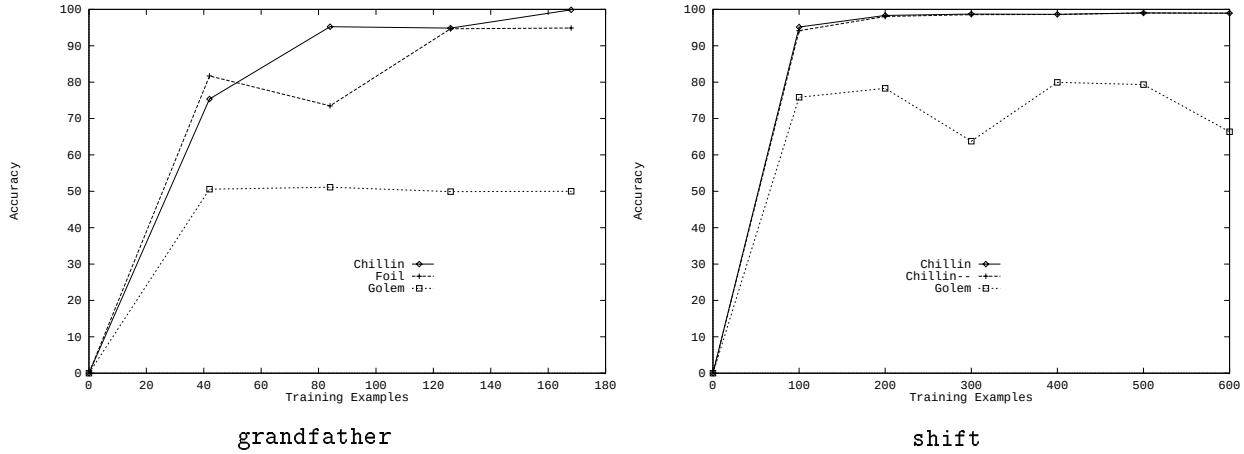

Figure 4: Concept Accuracy for `grandfather` and control rule

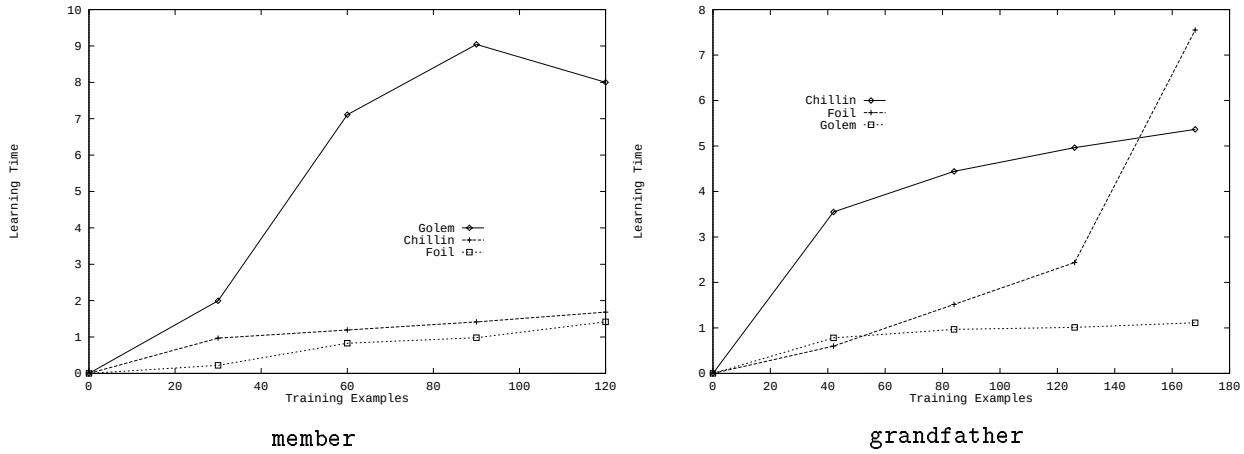

Figure 5: Timing Results

Disabling predicate invention had only a minor impact (1%) in accuracy with smaller training sets, and no difference was detectable for larger sets. GOLEM, on the other hand, never achieves greater than 80% accuracy and displays erratic learning behavior in this domain.

3.3 TIMING RESULTS

Given the differences in implementation, we expected FOIL and GOLEM to be considerably faster than CHILLIN. However, this was not the case. On all problems where GOLEM was learning useful rules, it was significantly slower than CHILLIN often by a factor of 10 or more. While FOIL tended to be a bit faster than CHILLIN, the learning times for the two systems were generally comparable. The timing curves for the `member` experiments shown in the left-hand graph of Figure 5 are typical. This graph shows the time in seconds required to learn a set of rules as a function of training set size. In the experiments where FOIL had more difficulty learning accurate rules such as `multiply` and `grandfather`, CHILLIN actually ran faster than FOIL at some data points. The left-hand graph of Figure 5 shows timing results for `grandfather`. Note that the run-time for GOLEM is lower here because it is not learning a definition, but rather, just memorizing the examples.

4 RELATED RESEARCH

Like CHILLIN, SERIES (Wirth and O'Rorke, 1991) and, later INDICO (Stahl et al., 1993) make use of LGGs of examples to construct clause heads containing functions. However, both of these systems precompute a set of clause heads for which bodies are subsequently induced. The approach taken by CHILLIN interleaves the bottom-up and top-down mechanisms, handling a larger class of concepts.

A number of recent investigations have considered the noisy-oracle problem in the induction of recursive definitions (Cohen, 1993; Lapointe and Matwin, 1992; Muggleton, to appear). However, the proposed mechanisms either severely limit the class of learnable programs (e.g. to single clause, linearly recursive) or rely on computationally expensive matching of subterms, or both. None has yet been implemented and tested in a system for large-scale induction over hundreds or thousands of examples.

Predicate invention is also an area of considerable interest. Like CHILLIN and CHAMP, SERIES and INDICO employ demand-driven predicate invention. These systems differ significantly in the heuristics used to select arguments for the new predicate. Another approach to invention is the use of the intra–construction operator of inverse-resolution (Muggleton and Buntine, 1988; Wirth, 1988; Rouveirol, 1992; Banerji, 1992). In this approach, new predicates are invented through a restructuring of an existing definition, usually to make it more compact. Unfortunately, we are not aware of any work that has systematically evaluated the competing approaches or the practical utility of predicate invention.

5 FUTURE WORK

The top-down component of CHILLIN could clearly be improved by adding more of FOIL's current features such as exploiting mode and type information, dealing with noise, and checking termination of recursive clauses (Cameron-Jones and Quinlan, 1994). Ehancements are also needed for multi-predicate learning (De Raedt et al., 1993), particularly for inventing predicates useful in the definition of multiple concepts. Finally, further experimental evaluation on a wider range

of more realistic problems is needed. In particular, ablation studies on the specific utility of predicate invention are indicated.

6 CONCLUSION

The CHILLIN ILP algorithm attempts to integrate the best aspects of existing ILP methods into a coherent, novel framework that includes both top-down and bottom-up search, predicate invention, and a solution to the noisy-oracle problem. Our current experimental results indicate that it is a robust and efficient system which can learn a range of logic programs (including recursive and nondeterminate ones) from random examples more effectively than current methods such as GOLEM and FOIL. It has also recently been used to learn natural language parsers from real text corpora requiring induction over thousands of complex, structured examples (Zelle and Mooney, 1994). Consequently, we believe it provides an important foundation for continued progress on robust and efficient induction of complex relational and recursive concepts.

Acknowledgments

Thanks to Ross Quinlan and Mike Cameron-Jones for FOIL, and Stephen Muggleton and Cao Feng for GOLEM. This research was supported by the National Science Foundation under grant IRI-9102926 and the Texas Advanced Research Program under grant 003658114.

References

Banerji, R. B. (1992). Learning theoretical terms. In Muggleton, S., editor, *Inductive Logic Programming*, 93–110. New York, NY: Academic Press.

Cameron-Jones, R. M., and Quinlan, J. R. (1994). Efficient top-down induction of logic programs. *SIGART Bulletin*, 5(1):33–42.

Cohen, W. W. (1993). Pac-learning a resticted class of recursive logic programs. In *Proceedings of National Conference on Artificial Intelligence*, 86–92. Washington, D.C.

De Raedt, L., Lavrac, N., and Dzeroski, S. (1993). Multiple predicate learning. In *Proceedings of the Thirteenth International Joint conference on Artificial intelligence*, 1037–1042. Chambery, France.

Kijsirikul, B., Numao, M., and Shimura, M. (1992). Discrimination-based constructive induction of logic programs. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, 44–49. San Jose, CA.

Lapointe, S., and Matwin, S. (1992). Sub-unification: A tool for efficient induction of recursive programs. In *Proceedings of the Ninth International Workshop on Machine Learning*, 273–281. Aberdeen, Scotland.

McClelland, J. L., and Kawamoto, A. H. (1986). Mechnisms of sentence processing: Assigning roles to constituents of sentences. In Rumelhart, D. E., and McClelland, J. L., editors, *Parallel Distributed Processing, Vol. II*, 318–362. Cambridge, MA: MIT Press.

Muggleton, S. (to appear). Inverting implication. *Artificial Intelligence*.

Muggleton, S., and Buntine, W. (1988). Machine invention of first-order predicates by inverting resolution. In *Proceedings of the Fifth International Conference on Machine Learning*, 339–352. Ann Arbor, MI.

Muggleton, S., and Feng, C. (1992). Efficient induction of logic programs. In Muggleton, S., editor, *Inductive Logic Programming*, 281–297. New York: Academic Press.

Plotkin, G. D. (1970). A note on inductive generalization. In Meltzer, B., and Michie, D., editors, *Machine Intelligence (Vol. 5)*. New York: Elsevier North-Holland.

Quinlan, J. (1990). Learning logical definitions from relations. *Machine Learning*, 5(3):239–266.

Rouveirol, C. (1992). Extensions of inversion of resolution applied to theory completion. In Muggleton, S., editor, *Inductive Logic Programming*, 63–86. New York, NY: Academic Press.

Stahl, I., Tausend, B., and Wirth, R. (1993). Two methods for improving inductive logic programming systems. In *Machine Learning: ECML-93*, 41–55. Vienna.

Wirth, R. (1988). Learning by failure to prove. In *Proceedings of EWSL 88*, 237–51. Pitman.

Wirth, R., and O'Rorke, P. (1991). Constraints on predicate invention. In *Proceedings of the Eighth International Workshop on Machine Learning*, 457–461. Evanston, Ill.

Zelle, J. M., and Mooney, R. J. (1993). Learning semantic grammars with constructive inductive logic programming. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, 817–822. Washington, D.C.

Zelle, J. M., and Mooney, R. J. (1994). Inducing deterministic Prolog parsers from treebanks: A machine learning approach. In *Proceedings of National Conference on Artificial Intelligence*. Seattle, WA.