Appears in *Proceedings of the Fifth International Workshop on Inductive Logic Programming (ILP-95),* pp.403-416, Leuven, Belgium, September 1995

Inducing Logic Programs without Explicit Negative Examples

John M. Zelle
Cynthia A. Thompson
Mary Elaine Califf
Raymond J. Mooney
Department of Computer Sciences
University of Texas
Austin, TX 78712
(512) 471-9589, 471-9558
(zelle,cthomp,mecaliff,mooney)@cs.utexas.edu

February 6, 1995

Abstract

This paper presents a method for learning logic programs without explicit negative examples by exploiting an assumption of *output completeness*. A mode declaration is supplied for the target predicate and each training input is assumed to be accompanied by all of its legal outputs. Any other outputs generated by an incomplete program implicitly represent negative examples; however, large numbers of ground negative examples never need to be generated. This method has been incorporated into two ILP systems, CHILLIN and IFOIL, both of which use intensional background knowledge. Tests on two natural language acquisition tasks, case-role mapping and past-tense learning, illustrate the advantages of the approach.

1 Introduction

Inductive logic programming (ILP) is a growing subtopic of machine learning that studies the induction of Prolog programs from examples in the presence of background knowledge (Muggleton, 1992; Lavrač & Džeroski, 1994). Most ILP systems require both positive and negative examples of ground instances of a predicate. However, explicit negative examples of a predicate are not always readily available. A standard solution is to automatically produce a large set of negative examples using a closed-world assumption, i.e. for an n-ary predicate, all n-tuples of terms chosen from a fixed set are generated and the positive examples are removed. However, it is frequently intractable to generate an adequate set of negative examples using this brute-force approach. For example, if one wants to learn a natural language parser from a corpus of realistic examples of the predicate `parse(Sentence, Representation)`, it is intractable to produce all possible pairs of sentences and representations.

This paper presents a general method for learning logic programs without explicit negative examples. An assumption of *output completeness* is used instead to implicitly determine if a hypothesized clause is overly-general and, if so, to quantify the degree of over-generality by simply estimating the number of negative examples covered. The target predicate is assumed to have a known mode (e.g. `parse(+,-)`), and a training set of positive examples is said to be output complete if for every set of inputs present in an example, each of the correct outputs for this input is represented by an example in the set. Therefore, any *other* outputs generated for an input can be assumed to represent negative examples.

This paper describes how this assumption can be exploited to learn logic programs without ever explicitly generating large sets of negative examples. It also describes how this approach has been implemented in two ILP systems: CHILLIN (Zelle & Mooney, 1994), a system that combines top-down and bottom-up methods and includes a method for inventing predicates, and IFOIL, a version of FOIL (Quinlan, 1990) that employs intensional background knowledge. [1] Finally, we present results demonstrating the advantage of this approach using two problems in natural language acquisition. CHILLIN

[1] IFOIL is derived from FOIDL which learns *first-order decision lists* using intensional background knowledge (Mooney & Califf, 1995).

is tested on mapping sentences to case-role representations (McClelland & Kawamoto, 1986; Zelle & Mooney, 1993), and IFOIL on generating the past tense of English verbs (Rumelhart & McClelland, 1986; Ling, 1994).

The remainder of the paper is organized as follows. Section 2 presents the basic approach to learning without explicit negatives. Section 3 describes how this approach has been implemented in IFOIL and presents results on the past-tense problem. Section 4 discusses how CHILLIN has been modified to incorporate implicit negatives and presents results on the case-mapping problem. Section 5 reviews related work, Section 6 discusses future work, and Section 7 presents our conclusions.

2 Learning with Implicit Negatives

2.1 Background: FOIL

Since IFOIL is based on FOIL, this subsection presents a brief review of this important ILP system; Quinlan (1990), Quinlan and Cameron-Jones (1993), and Cameron-Jones and Quinlan (1994) provide a more complete description. FOIL learns a function-free, first-order, Horn-clause definition of a *target* predicate in terms of itself and other *background* predicates. The input consists of extensional definitions of these predicates as tuples of constants of specified types. FOIL also requires negative examples of the target concept, which can be supplied directly or computed using a closed-world assumption.

Given this input, FOIL learns a program one clause at a time using a greedy-covering algorithm that can be summarized as follows:

Let *positives-to-cover* = positive examples.
While *positives-to-cover* is not empty
    Find a clause, C, that covers a preferably large subset of *positives-to-cover*
        but covers no negative examples.
    Add C to the developing definition.
    Remove examples covered by C from *positives-to-cover*.

The "find a clause" step is implemented by a general-to-specific hill-climbing search that adds antecedents to the developing clause one at a time. At each step, it evaluates possible literals that might be added and selects one that maximizes an information-gain heuristic. The algorithm maintains a set

of tuples that satisfy the current clause and includes bindings for any new variables introduced in the body. The gain metric evaluates literals based on the number of positive and negative tuples covered, preferring literals that cover many positives and few negatives.

2.2 Counting Implicit Negatives

Learning without explicit negatives requires an alternate method of evaluating the utility of a clause. A mode declaration and an assumption of output completeness together determine a set of implicit negative examples.

Consider the predicate, `past(Present,Past)` which holds when `Past` is the past-tense form of a verb whose present tense is `Present`. Providing the mode declaration `past(+,-)` indicates that the predicate should provide the correct past-tense when provided with the present-tense form. Assuming the past form of a verb is unique, any set of positive examples of this predicate will be output complete. However, output completeness can also be applied to non-functional cases such as `append(-,-,+)`, meaning that all possible pairs of lists that can be appended together to produce a list are included in the training set (e.g. `append([],[a,b],[a,b])`, `append([a],[b],[a,b])`, `append([a,b],[],[a,b])`).

Given these assumptions, determining if a clause is overly-general is straightforward. For each positive example, an *output query* is made to determine all outputs for the given input (e.g. `past([a,c,t], X)`). If any outputs are generated that are not positive examples, the clause still covers negative examples and requires further specialization. When such specialization is needed, a computation must be done of the number of negatives covered. Each ground, incorrect answer to an output query clearly counts as a single negative example (e.g. `past([a,c,h,e], [a,c,h,e,e,d])`). However, output queries will frequently produce answers with universally quantified variables. For example, given the overly-general clause `past(A,B) :- append(C,D, A).`, the query, `parse([a,c,t], X)`, generates the answer `past([a,c,t], Y)`. This implicitly represents coverage of an infinite number of negative examples.

In order to quantify negative coverage, we employ a parameter u representing the total number of possible terms in the universe. The negative coverage represented by a non-ground answer to an output query is then estimated as $u^v - p$, where v is the number of uninstantiated output arguments in the answer and p is the number of positive examples with which

the answer unifies. The u^v term is an estimate of number of unique ground outputs represented by the answer and from this the number of these that represent positive examples is subtracted. This allows the coverage of large numbers of implicit negative examples to be quantified without ever explicitly constructing them.

This reasoning can be extended to handle the case of partially instantiated output arguments. For example, both clauses

```
past(A,B) :- append(C,D,A).
past(A,B) :- append(A,C,B).
```

produce outputs containing a single free variable for the output query `past([a,c,t], X)`. The first produces the answer `past([a,c,t], Y)`, and the second produces the answer `past([a,c,t], [a,c,t | Y])`. However, the second clause is clearly better since it at least requires input to be part of the output. Since there are presumably more words total than there are words that start with "a-c-t" (assuming the total number of words is finite), the first clause should be considered to cover more negative examples. Therefore, arguments that are partially instantiated, such as `[a,c,t | Y]`, are counted as only a fraction of a variable when calculating v. Specifically, a partially instantiated output argument is scored as the fraction of its subterms that are variables, e.g. `[a,c,t | Y]` counts as only $1/4$ of a variable argument. Therefore, the first clause above is scored as covering u implicit negatives and the second as covering only $u^{1/4}$. Given reasonable values for u and the number of positives covered by each clause, the literal `append(A,C,B)` will be preferred.

3 IFOIL

3.1 Implementation of IFOIL

IFOIL is closely related to FOIL, following a similar top-down, greedy specialization guided by an information-gain heuristic. However, the algorithm is modified to allow the use of intensional background and output completeness as a substitute for explicit negative examples.

Using intensional background in IFOIL is straightforward. Instead of matching a literal against a set of tuples to determine whether or not it covers an example, a theorem prover is used in an attempt to prove that the

literal can be satisfied using the intensional definitions. IFOIL's specialization algorithm, incorporating implicit negatives, is:

Initialize T to contain the examples in *positives-to-cover* and output queries for all positive examples.
While T contains output queries
    Find the best literal L to add to the clause.
    Let T' be the positive examples in T that can still be proved as instances of the target concept using the specialized clause, plus the output queries in T that still produce incorrect answers.
    Replace T by T'.

The information-gain heuristic for picking the best literal is identical to the FOIL metric except that the counts of covered positive and negative tuples are replaced with the count of covered positive examples and the estimate of implicit negative examples.

3.2 Learning the English Past Tense

The problem of learning the English past tense has been to test connectionist systems (Rumelhart & McClelland, 1986; MacWhinney & Leinbach, 1991) or decision tree induction (Ling & Marinov, 1993; Ling, 1994). All of this work encodes the problem as fixed-length pattern association and fails to capture the generativity and position-independence of the true regular rules such as "add 'ed'," instead producing several position-dependent rules. Recently, ILP methods have also been applied to the task, with some success (Quinlan, 1994; Mooney & Califf, 1995).

The past-tense problem is valuable for demonstrating the value of implicit negatives since it is very difficult to supply an appropriate set of explicit negative examples for this problem. Accuracy for this domain should be measured by the ability to actually generate correct output for novel inputs, rather than the ability to correctly classify novel ground examples. Using a closed-world assumption to produce all pairs of words in the training set where the second is not the past-tense of the first tends to produce clauses such as:

```
past(A,B) :- split(B,A,C).
```

which is useless for producing the past tense of novel verbs. However supplying all possible strings of 15 characters or less as negative examples of the past tense of each word is clearly intractable.

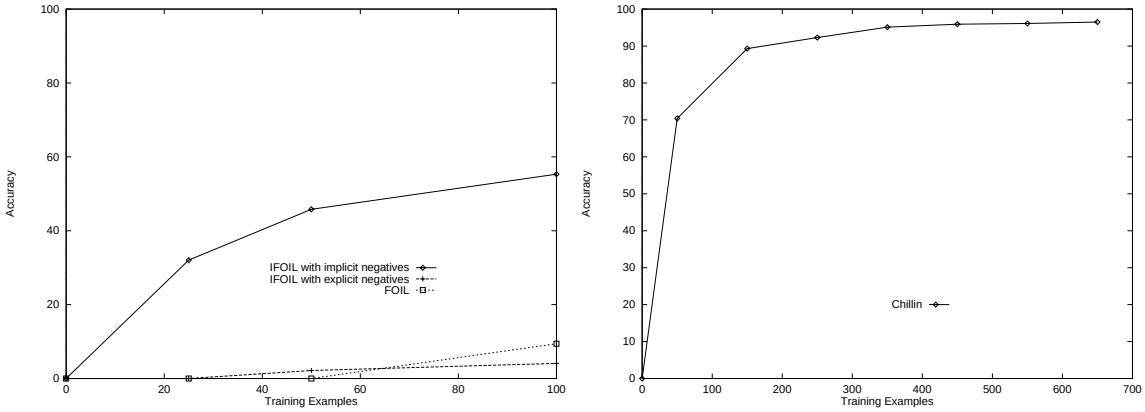
When Quinlan (1994) applied FOIL to the past tense problem, he used a three-place predicate `past(X,Y,Z)` which is true iff the input word `X` is transformed into past-tense form by removing its current ending `Y` and substituting the ending `Z`; for example: `past([a,c,t], [], [e,d])`, `past([a,r,i,s,e], [i,s,e], [o,s,e])`. This method allows the generation of useful negatives under the closed world assumption, but relies on an understanding of the desired transformation. We hypothesized that implicit negatives would allow IFOIL to learn the normal two argument version of the problem.

For comparison, we ran tests with FOIL and IFOIL using explicit negatives and with IFOIL using implicit negatives. The data for these experiments came from Ling (1994) and consists 1390 pairs of base and past tense verb forms in UNIBET phonemic form. The training and testing followed the standard paradigm of splitting the data into testing and training sets and training on progressively larger samples of the training set. All results were averaged over 10 trials, and the testing set for each trial contained 500 verbs.

The background predicate provided to the systems is `split` which splits a list into two non-empty sublists. Providing an extensional definition of `split` that includes all possible strings of 15 or fewer characters (at least 10^{21} strings) is intractable. However, providing a partial definition that includes all possible splits of strings that actually appear in the training corpus is sufficient, so that partial definition was provided for FOIL. The explicit negatives provided to IFOIL and FOIL were those formed by combining the base form of each verb with the past tense of each of the other verbs in the training set.

As the left-hand graph in Figure 1 demonstrates, it is all but impossible for FOIL and IFOIL to learn a useful, generative concept with this representation using the explicit negatives. However, using implicit negatives, IFOIL is able to achieve an accuracy of 55% with 100 positive examples. This is comparable to the performance of FOIL(58%) exploiting the three-place predicate and a closed-world assumption (Mooney & Califf, 1995). [2]

[2] This system is not the final word on learning past tense with ILP. To achieve the best accuracy, clause ordering and cuts are required, as in Mooney and Califf (1995).



Past Tense Case-role Mapping

Figure 1: Experimental Results

4 Implicit Negatives in CHILLIN

The utility of our technique for handling implicit negative examples is not limited to its IFOIL incarnation. We have used the same technique in a version of the CHILLIN ILP system to attack induction problems which are very difficult for purely FOIL-like approaches.

4.1 Overview of CHILLIN

CHILLIN (Zelle & Mooney, 1994) is an ILP algorithm developed in the context of learning control rules for natural language parsing. It combines elements of both top-down and bottom-up induction techniques including a mechanism for demand-driven predicate invention. At its heart is a basic compaction algorithm as follows:

DEF := {E :- true | E ∈ Positives}
Repeat
  PAIRS := a sampling of pairs of clauses from DEF
  GENS := {G | G = build_gen(C_i,C_j,DEF,Positives,Negatives) for $\langle C_i, C_j \rangle \in$ PAIRS}
  G := Clause in GENS yielding most compaction
  DEF := (DEF−(Clauses subsumed by G)) ∪ G
Until no further compaction

CHILLIN starts with a most specific definition (the set of positive examples) and introduces generalizations which make the definition more compact (as measured by a CIGOL-like size metric (Muggleton & Buntine, 1988)). The search for more general definitions is carried out in a hill-climbing fashion. At each step, a number of possible generalizations are considered; the one producing the greatest compaction of the theory is implemented, and the process repeats.

The `build_gen` algorithm attempts to construct a clause which empirically subsumes some clauses of `DEF` without covering any of the negative examples. The first step is to construct the least general generalization (LGG) (Plotkin, 1970) of the input clauses. If the LGG does not cover any negative examples, no further refinement is necessary. If the clause is too general, an attempt is made to refine it using a FOIL-like mechanism which adds literals derivable either from background or previously invented predicates. If the resulting clause is still too general, it is passed to a routine which invents a new predicate to discriminate the positive examples from the negatives which are still covered.

4.2 Incorporating Implicit Negatives

Since CHILLIN uses top-down specialization of overly-general clauses, an estimate of implicit negatives may be used just as in the IFOIL algorithm described above. Although the gain metric employed in CHILLIN is not identical to that of FOIL, it is modified in exactly the same way, replacing a count of covered explicit negatives with an estimate of implicit coverage.

Unlike FOIL, however, CHILLIN also performs demand-driven predicate invention. The original CHILLIN algorithm performs invention in a manner analogous to CHAMP (Kijsirikul, Numao, & Shimura, 1992). The first step is to find a subset of variables appearing in the clause whose instantiations are sufficient to discriminate covered positive examples from covered negatives. The instantiations of these variables when covering the positive and negative examples generate positive and negative tuples for a new predicate which can be used to specialize the clause. The top-level induction algorithm is recursively invoked with these derived tuples as positive and negative examples to create a definition for the predicate.

There are two cases to consider when modifying predicate invention to use implicit negative examples. The simpler case occurs when all output variables

in the head of the clause for which the predicate is being invented also appear in the body of the clause. In this situation, output queries always produce ground results. The fact that the clause still covers negative examples means that there are some queries for which the ground outputs are simply incorrect. These incorrect outputs become a set of explicit negative examples for the normal predicate invention algorithm, and the recursive invocation of the top-level algorithm is made using explicit negative examples. The version of CHILLIN used for the experiments reported below used only this mechanism for predicate invention. Generalizations which still contained free variables were simply discarded.

When some of the output variables appearing in the clause head do not appear in the clause body, it is impossible to generate a set of ground negative examples. However, it is still possible to perform demand-driven predicate invention. Since the remaining free variables must be bound to prevent coverage of implicit negatives, they must be included as output variables of the new predicate. The input variables are chosen by finding a small subset of the bound variables that are sufficient to *determine* the values of the output variables when covering the positive examples. Once chosen, the instantiations of these variables when the clause is used to cover positive examples produce a set of derived tuples which can be used as positive examples for the new predicate. The implicit negative version of the top-level induction algorithm can then be invoked using these examples and the associated mode declaration. This predicate invention mechanism may be added to CHILLIN in the future.

4.3 Experiments with Case-role Mapping

Case-role mapping is a typical problem in natural language processing. Traditional case theory (Fillmore, 1968) decomposes a sentence into a proposition represented by the main verb and various arguments such as agent, patient, and instrument, represented by noun phrases. The basic mapping problem is to decide which sentence constituents fill which roles. The ILP task considered here is the induction of a program for the relation `parse(Sentence, Representation)` which holds when `Representation` is the correct case-role mapping for `Sentence`. Furthermore, we desire that the resulting program be generative, given a suitable goal such as `parse([the,man,ate,the pasta],X)`, we want the program to produce a binding `X = [ate, agent:[man,`

`det:the], patient:[pasta, det:the]]`. Thus, the desired mode is `parse(+,-)`, effectively producing a case-role parser.

The experiments were conducted using a set of 1475 sentence/case-structure pairs originally from (McClelland & Kawamoto, 1986) The corpus was produced from a set of 19 sentence templates generating sentences such as, `the HUMAN ate the FOOD with the UTENSIL`, where the capitalized items were replaced by words of the appropriate category. The sample actually comprises 1390 unique sentences, some of which allow multiple analyses. Thus, the relation `parse/2` is not a function. Training was done considering each unique sentence as a single example. If a particular sentence was chosen for inclusion in a training or testing set, the pairs representing all correct analyses of the sentence were included in that set to provide for output completeness.

Training and testing followed the standard paradigm of first choosing a random set of test examples (in this case 740) and then inducing programs using increasingly larger subsets of the remaining examples. During testing, the program was used to enumerate all analyses for a given test sentence. Parsing of a sentence can fail in two ways: an incorrect analysis may be generated, or a correct analysis may not be generated. In order to account for both types of inaccuracy, a metric was introduced to calculate the "average correctness" for a given test sentence as follows: $Accuracy = (\frac{C}{P} + \frac{C}{A})/2$ where P is the number of distinct analyses produced, C is the number of the produced analyses which were correct, and A is the number of correct analyses possible for the sentence. This result is an average of what are sometimes called the "precision" and "recall" accuracy for a given sentence.

The case-role mapping problem presents two difficulties for ILP algorithms such as FOIL. First, the space of negative examples is very large. Clearly it is intractable to generate the complete set of all possible incorrect analyses of the input sentences; in making a closed-world assumption to generate explicit negative examples, only a fractional subset may be considered. A second difficulty is represented by the rich structure of the examples. FOIL constructs clauses which are function-free. In order to inspect functional structure FOIL must make use of background predicates such as `components/3`. The single case-role example, `parse([the,man,ate,the,pasta], [ate,agent:[man,det:the],patient:[pasta,det:the]])`, requires 12 literals just to access relevant parts of the structures. Obviously, it is difficult to learn such complex clauses. Given these difficulties, it is not surprising that we were unable to get FOIL to learn any definition of `parse/2`. IFOIL

also failed on this task, suggesting that the main difficulty lies with the function-free representation.

The bottom-up component of CHILLIN is able to decompose functional structures directly, and experiments with CHILLIN were run without any background predicates except those invented by CHILLIN itself. CHILLIN without implicit negatives is hampered by the lack of appropriate negative examples. Some experiments were performed using the training set to generate a small fraction of the explicit closed-world negatives. Specifically, each input sentence of the training set was paired with each output analysis appearing for different sentences in the training examples. Using this technique, a sample of 250 sentences generates over 36,000 negative examples. CHILLIN had no problem inducing definitions for these examples. Training sets of 150 examples or more produced rules with over 99% *recognition* accuracy on a separate testing set consisting of 740 positive examples and a random set of 1000 negatives generated from them in a manner analogous to that used for the training examples. Unfortunately, the resulting programs were not generative. Given an output query, they produced results with unbound variables. Hence, they were not useful as case-role parsers.

CHILLIN using implicit negatives, however, did produce good case-role parsers for this data. The righthand graph of Figure 1 shows an average learning curve over three independent trials. Training on 650 examples produced programs with over 97% accuracy in producing correct case-role parses for novel sentences. The addition of implicit negatives to CHILLIN's existing capabilities for direct functional decomposition render this a tractable ILP problem.

5 Related Work

Bergadano, Gunetti, and Trinchero (1993) allow the user to supply an intensional definition of negative examples that covers a large set of ground instances (e.g `(past([a,c,t],X), not(equal(X,[a,c,t,e,d])))`); however, to be equivalent to output completeness, the user would have to explicitly provide a separate intensional negative definition for each positive example. Also, they do not provide a way of quantifying implicit negative coverage.

INDICO (Stahl, Tausend, & Wirth, 1993) employs a related method for learning from positive examples only; however, it is only applicable to func-

tional predicates. Output completeness is a more general assumption that allows for non-functional predicates such as `parse`, which allows multiple outputs in order to handle truly ambiguous sentences. Also, INDICO uses a special-purpose heuristic to evaluate literals rather than estimating negative coverage based on overly-general answers containing free variables.

6 Future Work

The idea of output completeness and implicit negatives could be potentially added to other ILP methods. An important constraint is that the system interpret partially learned clauses intensionally so that general output queries can be used to produce potentially non-ground answers. Tests should also be conducted on a wider variety of problems in which it is intractable to generate negative examples using a closed-world assumption. For problems where it is possible to generate a sufficient set of examples using a closed world assumption, using implicit negatives has the potential of being more efficient. Tests demonstrating this efficiency gain are another area for future research.

7 Conclusion

We have presented a method for learning logic programs without explicit negative examples by exploiting the assumption of output completeness to determine whether a clause is overly-general. This method has been incorporated into two ILP systems, CHILLIN and IFOIL, both of which use an estimate of the number of implicit negative examples covered to guide the top-down specialization of a clause. CHILLIN also includes a method for inventing predicates in the context of implicit negatives. Each system was tested on a different natural language acquisition task, past-tense learning or case-role mapping, to illustrate the advantage of the approach.

Acknowledgements

Support for this research was provided in part by grant IRI-9310819 from the National Science Foundation and an MCD fellowship from the University of

Texas awarded to the third author.

References

Bergadano, F., Gunetti, D., & Trinchero, U. (1993). The difficulties of learning logic programs with cut. *Journal of Artificial Intelligence Research*, *1*, 91–107.

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

Fillmore, C. J. (1968). The case for case. In Bach, E., & Harms, R. T. (Eds.), *Universals in Linguistic Theory*. Holt, Reinhart and Winston, New York.

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

Lavrač, N., & Džeroski, S. (Eds.). (1994). *Inductive Logic Programming: Techniques and Applications*. Ellis Horwood.

Ling, C. X. (1994). Learning the past tense of English verbs: The symbolic pattern associator vs. connectionist models. *Journal of Artificial Intelligence Research*, *1*, 209–229.

Ling, X., & Marinov, M. (1993). Answering the connectionist challenge: A symbolic model of learning the past tense of English verbs. *Cognition*, *49*(3), 235–290.

MacWhinney, B., & Leinbach, J. (1991). Implementations are not conceptualizations: Revising the verb model. *Cognition*, *40*, 291–296.

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

Mooney, R. J., & Califf, M. E. (1995). Induction of first-order decision lists: Results on learning the past tense of English verbs. *Journal of Artificial Intelligence Research*, *??* submitted.

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

Muggleton, S. H. (Ed.). (1992). *Inductive Logic Programming*. Academic Press, New York, NY.

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

Quinlan, J. R. (1994). Past tenses of verbs and first-order learning. In Zhang, C., Debenham, J., & Lukose, D. (Eds.), *Proceedings of the Seventh Australian Joint Conference on Artificial Intelligence*, pp. 13–20 Singapore. World Scientific.

Quinlan, J. R., & Cameron-Jones, R. M. (1993). FOIL: A midterm report. In *Proceedings of the European Conference on Machine Learning*, pp. 3–20 Vienna.

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

Rumelhart, D. E., & McClelland, J. (1986). On learning the past tense of English verbs. In Rumelhart, D. E., & McClelland, J. L. (Eds.), *Parallel Distributed Processing, Vol. II*, pp. 216–271. MIT Press, Cambridge, MA.

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

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

Zelle, J. M., & Mooney, R. J. (1994). Combining top-down and bottom-up methods in inductive logic programming. In *Proceedings of the Eleventh International Conference on Machine Learning* New Brunswick, NJ.