Technical Report, Artificial Intelligence Lab,
University of Texas at Austin, January 1996.

Advantages of Decision Lists and Implicit Negatives in Inductive Logic Programming*

Mary Elaine Califf and Raymond J. Mooney
Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712
{mecaliff,mooney}@cs.utexas.edu

January 20, 1996

Abstract

This paper demonstrates the capabilities of FOIDL, an inductive logic programming (ILP) system whose distinguishing characteristics are the ability to produce first-order decision lists, the use of an output completeness assumption to provide implicit negative examples, and the use of intensional background knowledge. The development of FOIDL was originally motivated by the problem of learning to generate the past tense of English verbs; however, this paper demonstrates its superior performance on two different sets of benchmark ILP problems. Tests on the finite element mesh design problem show that FOIDL's decision lists enable it to produce better results than all other ILP systems whose results on this problem have been reported. Tests with a selection of list-processing problems from Bratko's introductory Prolog text demonstrate that the combination of implicit negatives and intensionality allow FOIDL to learn correct programs from far fewer examples than FOIL.

Keywords: *inductive logic programming*

1 Introduction

New machine learning algorithms are often motivated by the difficulties of solving new problems with existings algorithms. While this can be an important source of progress, it could lead to a proliferation of overly specialized systems designed to handle a narrow range of learning problems. Therefore, it is important to test new systems with problems other than those which originally motivated the systems' development.

*This research was supported by a fellowship from AT&T awarded to the first author and by the National Science Foundation under grant IRI-9310819.

Mooney and Califf (1995) introduced FOIDL, a new inductive logic programming (ILP) system motivated by problems with applying previously existing ILP techniques to the problem of generating the past tense of English verbs from the base form of the verb. FOIDL is based on FOIL (Quinlan, 1990) but has three distinguishing characteristics: 1) it uses intensional background knowledge; 2) it avoids the need for explicit negative examples by using an output completeness assumption to create implicit negatives; and 3) it is able to create first-order decision lists (ordered lists of clauses, each ending in a cut). These characteristics allow FOIDL to perform well on the past tense problem, bettering all previous results (Mooney & Califf, 1995).

In this paper, we present results that show that the FOIDL algorithm is useful for more than the problem for which it was originally designed. We have tested the system on two standard ILP problems: the finite element mesh design introduced by Dolsak and Muggleton (1992) and a selection of the list-processing programs from Bratko (1990) used by Quinlan and Cameron-Jones (1993). We compare FOIDL's performance to FOIL and to FFOIL (Quinlan, submitted), a version of FOIL which learns single-output functions. The first order decision lists enable FOIDL to achiever better accuracy on the finite element mesh design problem than has previously been reported for an ILP system, and FOIDL's intensionality and use of implicit negatives allow it to learn correct programs for the list-processing examples from small sets of randomly selected examples.

The remainder of the paper is organized as follows. Section 2 provides background on FOIL and FFOIL. Section 3 summarizes the FOIDL algorithm. Section 4 presents our results on the finite element mesh design and list-processing problems. Section 5 discusses related work, and Section 6 summarizes and presents our conclusions.

2 Background

2.1 FOIL

Since FOIDL is based on FOIL, we present a brief review of this important ILP system; see articles on FOIL for a more complete description (Quinlan, 1990; Quinlan & Cameron-Jones, 1993; Cameron-Jones & Quinlan, 1994).[1] 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 $positives\text{-}to\text{-}cover$ = positive examples.
While $positives\text{-}to\text{-}cover$ is not empty
        Find a clause, $C$, that covers a preferably large subset of $positive\ s\text{-}to\text{-}cover$
                but covers no negative examples.
```

[1] FOIL is also available by anonymous FTP from ftp.cs.su.oz.au in the file pub/foil6.sh.

```
    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. The papers referenced above provide details and information on additional features.

2.2 FFOIL

FFoil is a descendant of Foil with modifications(Quinlan, submitted), somewhat similar to Foidl's, that specialize it for learning functional relations.[2] First, FFoil assumes that the final argument of the relation is an *output argument* and that the other arguments of the relation uniquely determine the output argument. This assumption is used to provide implicit negative examples: each positive example under consideration whose output variable is not bound by the clause under construction is considered to represent one positive and r - 1 negatives, where r is the number of constants in the range of the function. Second, FFoil assumes that each clause will end in a cut, so that previously covered examples can be safely ignored in the construction of subsequent clauses. Thus, FFoil, like Foidl, constructs first-order decision lists, though it constructs the clauses in the same order as they appear in the program, while Foidl constructs its clauses in the reverse order.

3 The FOIDL Algorithm

Foidl adds three major features to Foil: 1) Intensional specification of background knowledge, 2) Output completeness as a substitute for explicit negative examples, and 3) Support for learning first-order decision lists. We now describe the modifications made to incorporate these features.

As described above, Foil assumes background predicates are provided with extensional definitions; however, this is burdensome and frequently intractable. Providing an intensional definition in the form of general Prolog clauses is generally preferable. Intensional background definitions are not restricted to function-free pure Prolog and can exploit all features of the language.

Modifying Foil to use intensional background is straightforward. Instead of matching a literal against a set of tuples to determine whether or not it covers an example, the Prolog interpreter is used in an attempt to prove that the literal can be satisfied using the intensional definitions. Unlike Foil, Foidl does not maintain expanded tuples; positive and

[2]The development of FFoil was partially motivated by our work on Foidl.

negative examples of the target concept are reproved for each alternative specialization of the developing clause.

Learning without explicit negatives requires an alternate method of evaluating the utility of a clause. In FOIDL a mode declaration and an assumption of *output completeness* (that for every unique input pattern appearing in the training set, the training set includes all positive examples with that input pattern) together determine a set of implicit negative examples.

Consider the predicate, `last(Element,List)` which holds when `Element` is the last element of `List`. Providing the mode declaration `last(-,+)` indicates that the predicate should provide the correct final element when a list, the standard definition of last/2. Since the final element of a given list 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 an output completeness assumption, determining whether 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. `last([a,c,b], X)`). If any outputs are generated that are not positive examples, the clause still covers negative examples and requires further specialization. In addition, in order to compute the gain of alternative literals during specialization, the negative coverage of a clause needs to be quantified. Each ground, incorrect answer to an output query clearly counts as a single negative example (e.g. `last([a,c,b], [a])`). However, output queries will frequently produce answers with universally quantified variables. For example, given the overly-general clause `last(A,B) :- append(C,D,A).`, the query `last([a,c,b], X)` generates the answer `last([a,c,t], Y)`. This implicitly represents coverage of an infinite number of negative examples.

In order to quantify negative coverage, FOIDL uses a parameter u to represent 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 variable arguments in the answer and p is the number of positive examples with which the answer unifies. The u^v term stands for the number of unique ground outputs represented by the answer (e.g. the answer `append(X,Y,[a,b])` stands for u^2 different ground outputs) and the p term stands for the number of these that represent positive examples. This allows FOIDL to quantify coverage of large numbers of implicit negative examples without ever explicitly constructing them.

Unfortunately, this estimate is not sensitive enough. For example, in the past tense domain both clauses

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

where split(A,B,C) is equivalent to append(B,C,A) when B and C are non-empty lists cover u implicit negative examples for the output query `past([a,c,t], X)` since 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 the output to be the input with some suffix added. Since there are presumably more words 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 `split(B,A,C)` will be preferred.

The algorithm incorporating intensional background knowledge and implicit negatives is:

```
Initialize $C$ to $R(V_1, V_2, ..., V_k)$ :-. where $R$ is the target predicate with arity $k$.
Initialize $T$ to contain the examples in $positives\text{-}to\text{-}cover$ 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 subset of positive examples in $T$ that can still be prove d as instances
          of the target concept using the specialized clause, plus the output querie s in $T$
          that still produce incorrect answers.
      Replace $T$ by $T'$.
```

Since expanded tuples are not produced, the information-gain heuristic for picking the best literal is simply:

$$gain(L) = |T'| \cdot (I(T) - I(T')). \tag{1}$$

$|T|$ is computed as the number of positive examples in T plus the sum of the number of implicit negatives covered by each output query in T. This is the algorithm for IFOIL (Intensional FOIL), which is simply FOIDL with the decision list feature turned off, making the system useful for non-functional relations.

FOIDL's final feature is that it can produce first-order decision lists. As described above, these are ordered sets of clauses each ending in a cut. When answering an output query, the cuts simply eliminate all but the first answer produced when trying the clauses in order. Therefore, this representation is similar to propositional decision lists (Rivest, 1987), which are ordered lists of pairs (rules) of the form (t_i, c_i) where the test t_i is a conjunction of features and c_i is a category label and an example is assigned to the category of the first pair whose test it satisfies.

In the original algorithm of Rivest (1987) and in CN2 (Clark & Niblett, 1989), rules are learned in the order they appear in the final decision list (i.e. new rules are appended to the end of the list as they are learned). However, Webb and Brkič (1993) argue for learning decision lists in the reverse order since most preference functions tend to learn more general rules first, and these are best positioned as default cases towards the end. They introduce an algorithm, *prepend*, that learns decision lists in reverse order and present results indicating that in most cases it learns simpler decision lists with superior predictive accuracy. FOIDL can be seen as generalizing *prepend* to the first-order case for target predicates representing functions.

The resulting clause-specialization algorithm can now be summarized as follows:

```
Initialize $C$ to $R(V_1, V_2, ..., V_k)$ :-. where $R$ is the target predicate with arity $k$.
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 subset of positive examples in $T$ whose output query still produces
            a first answer that unifies with the correct answer, plus the output queries in $T$
            that either
                1) Produce a non-ground first answer that unifies with the correct answer, or
                2) Produce an incorrect answer but produce a correct answer using a
                   previously learned clause.
        Replace $T$ by $T'$.
```

To handle the list-processing tasks, FOIDL requires two features that were unnecessary for the past-tense task: recursion and determinate literals. Both of these are features of FOIL, but the change from an extensional system to an intensional one requires that the features be implemented somewhat differently.

To handle recursion, FOIDL begins by adding all of the positive examples in the training set to the current theory. When a new clause is being created, before each example is proved with the new clause, the example is temporarily removed from the theory so that it can't be used to prove itself. Once the clause is complete, FOIDL greedily removes examples from the theory permanently. FOIDL avoids infinitely recursive clauses by using a depth bound when proving examples.

Determinate literals are literals which may have little or no gain, but add at least one new variable which can only take on exactly one value for each positive tuple and and no more than one value for each negative tuple when extensional tuples are maintained as in FOIL(Cameron-Jones & Quinlan, 1994). Since FOIDL does not maintain the extensional tuples, it must recognize that there is exactly one possible proof of the literal given the bindings produced by the rest of the clause.

4 Experimental Results

4.1 Finite Element Mesh Design

The finite element mesh design application, introduced by Muggleton and Feng (1992), concerns the division of an object into regions for finite element simulation. The regions are created by cutting each edge of the object into a number of intervals. The ILP task is to learn to generate a suitable number of intervals for each edge: division into too many intervals leads to excessive computation in the simulation; division into too few leads to a poor approximation of the behavior of the object.

The tests described here use data concerning five objects with a total of 277 edges. The function to be learned is `mesh(A,B)` where `A` is an edge and `B` is the number of intervals that

Object	Edges	Correct			Error ≤ 1		
		FOIL	FFOIL	FOIDL	FOIL	FFOIL	FOIDL
A	54	16	21	21	21	33	34
B	42	9	13	15	14	20	24
C	28	8	11	11	10	14	13
D	57	12	22	22	15	27	27
E	96	16	33	39	45	66	73
Total	277	61 (22%)	100 (36%)	108 (39%)	105 (38%)	160 (58%)	186 (67%)

Table 1: Results for finite element mesh design data

the edge should be divided into. The background information consists of thirty relations describing various properties of the edges and their relationship to other edges in the object. We ran five trials; in each, the learning system was provided with the information about four of the objects, and the resulting program was tested on the edges from the remaining object.

Table 1 shows our results for FOIDL along with those reported in Quinlan (submitted) for FOIL and FFOIL. We report both the number of edges which exactly match the expert's and the number for which the system is no more that one away, since being close may be useful in this type of problem. FOIDL and FFOIL both exploit the functional nature of the problem using decision lists and produce results that are dramatically better than those of FOIL. FOIDL's results are slightly better than FFOIL's overall, but not significantly so. Both systems also perform much better than results reported for mFOIL and GOLEM (21% and 19% correct respectively) (Lavrač & Džeroski, 1994).

4.2 List-processing Programs

For another test of FOIDL's capabilities, we used a selection of the list-processing problems to which Quinlan and Cameron-Jones (1993) applied FOIL. These are a sequence of list-processing examples and exercises from Chapter 3 of (Bratko, 1990). For each problem, the background provided consists of the relations encountered previously. In his experiments, Quinlan uses two universes: all lists on three elements of length up to three and all lists on four elements of length up to four. For each problem, FOIL was provided with all positive examples of the relation in the specified universe and generates negatives using the closed world assumption. Because FOIL must either be provided with explicit negatives or be able to generate negatives using a closed world assumption, it cannot learn the various relations from a smaller set of examples if only positives are provided.

We speculated that FOIDL's implicit negatives would enable it to learn several of the list-processing relations from smaller sets of positive examples. Previous experiments with learning list-processing Prolog programs have employed specially constructed sets of examples that are guaranteed to be complete in some sense. However, ideally an ILP system should be able to learn such programs from random examples rather than carefully-selected sets (Zelle & Mooney, 1994; Cohen, 1993). Consequently, we compared systems on randomly

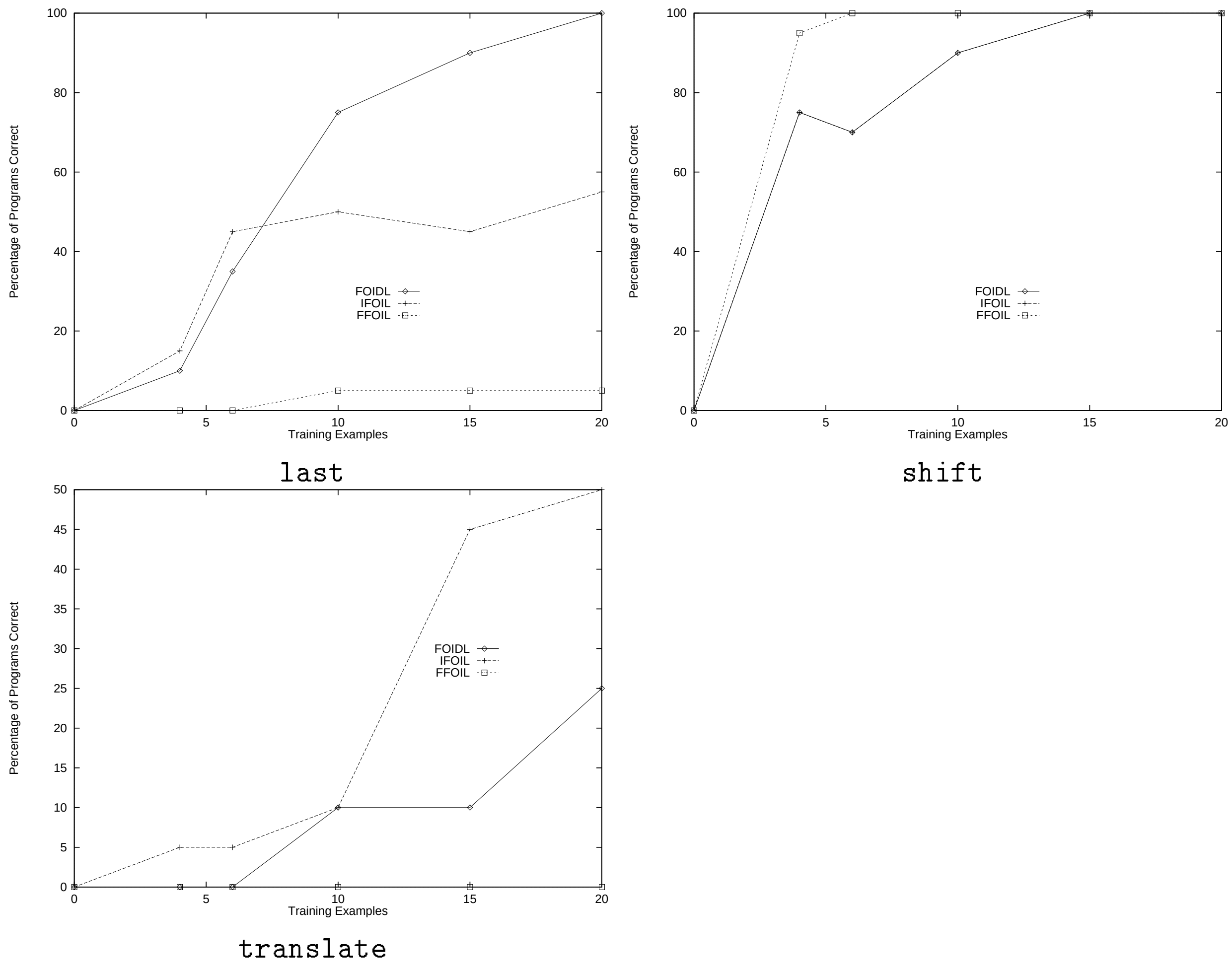

Figure 1: Results for functional list processing programs

selected examples. For each subset size, we ran 20 experiments, and we report the percentage of trials out of the 20 that the system is able to learn a correct definition of the relation for lists of arbitrary length. We use this definition of accuracy because in this type of domain it seems more approprate than a measure of the ability of incorrect programs to classify examples.

The relations we considered naturally divide into two distinct sets of experiments. Several of them are functional, making them appropriate candidates for tests with systems that learn decision lists. Quinlan (submitted) showed that FFoil can learn these functional relations more quickly than Foil, but he did not explore the possibility of learning from fewer examples. We ran parallel trials on the functional relations from the set using Foidl, FFoil, and IFoil, which is Foidl without decision lists. On two of the functions: reverse and conc, our results were not encouraging. Reverse requires the entire universe of 40 examples when lists are limited to length three for all three of the systems. IFoil and Foidl begin to be able to learn correct definitions for conc with subsets of 40 examples from the 182 total.

Figure 1 shows the results for last, shift and translate. Foidl, both with and without decision lists, is able to learn correct definitions from relatively small subsets of the positive examples. FFOIL's performance on these tasks points to one of the advantages of the FOIDL algorithm. The correct programs for both last and translate are recursive, but shift does not require recursion, and shift is the only one of these functions which FFOIL performs well on. This is because the random subsets of examples rarely provide a sequence of examples of the type that the extensional algorithm requires in order to learn recursive rules. Since FOIDL interpret both the background and the rules being learned intensionally, it requires *any* base example or clause, not the immediately preceding example. Thus, to learn the rule

```
last(A,[B|C]) :- last(A,C).
```

FFOIL would require sequences of examples such as:

```
last(3,[1,2,3]).
last(3,[2,3]).
last(3,[3]).
```

while FOIDL would need only the last item and one of the other two from the sequence. FOIDL and IFOIL perform identically on shift, but quite differently on the other two tasks. IFOIL performs better with fewer examples on last and throughout on translate, but FOIDL catches up and surpasses IFOIL's performance on last. This seems to be because having very few examples tends to lead the decision list to create bad clauses as the default clause, leading to incorrect definitions. Once there are enough examples, however, the decision list bias becomes helpful on last.

Besides demonstrating the performance of FOIDL on functions, we wished to examine the usefulness of the implicit negatives in non-functional applications. So we also ran an experiment with four of the non-functional list-processing problems. Again we randomly selected subsets of positive examples from the universe of lists of length up to three on three elements (20 subsets of each size), and we ran FOIDL (without decision lists, of course) on each subset to determine whether it produced a correct program for lists of arbitrary length. Because of the requirement that the input be output complete, the subsets were chosen not by randomly selecting the examples, but by randomly selecting a set of inputs and then providing the system with all of the positive examples with those inputs. For the relations tested, the average number of examples per input varies from barely over 1 for deleting an item from a list to 5 for sublist.

The results for the non-functional relations tested appear in Figure 2. Although it learns some relations better than other, FOIDL clearly can exploit its ability to use implicit negative examples to learn these relations from far fewer positive examples than FOIL requires (75 for member, 81 for del, 81 for insert, and 202 for sublist).

5 Related Work

The first learning system to focus on learning functions only was FILP (Bergadano & Gunetti, 1993). It assumes that the target and all background relations are functional and uses this

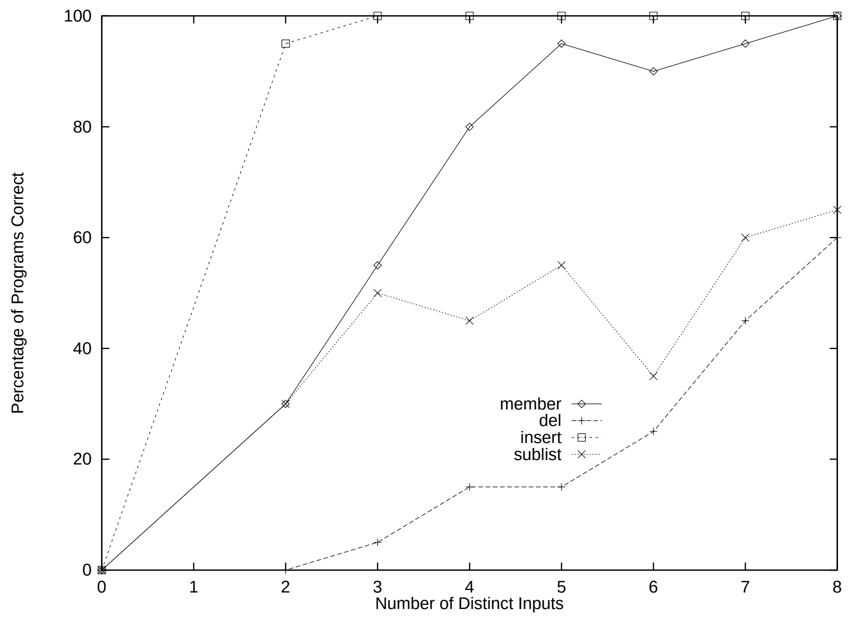

Figure 2: IFOIL's performance on non-functional list processing programs

knowledge to limit its search for literals. However, these assumptions prevent its use on the tasks considered here because they typically involve non-functional background relations. Also, although FILP assumes that its target relation is function, the definitions learned consist of unordered sets of clauses.

Other systems, most notably LOPSTER(Lapointe & Matwin, 1992) and CRUSTACEAN (Aha, Ling, Matwin, & Lapointe, 1993), have addressed learning relations from very small sets of examples. These systems can learn certain recursive relations from even fewer examples than FOIDL, but they can only learn relations with a particular recursive structure, and they use the assumption that the target relation has this particular structure. FOIDL requires a few more examples, but this is because it is a more general system, working on a number of other types of problems, and because FOIDL does not make assumptions about the structure of the solution, only about the nature of the data.

Two other systems use intensional negative examples, but neither is as general and flexible as FOIDL. Bergadano, Gunetti, and Trinchero (1993) allows 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. The non-monotonic semantics used to eliminate the need for negative examples in CLAUDIEN (De Raedt & Bruynooghe, 1993) has the same effect as an output completeness assumption in the case where all arguments of the target relation are outputs. However, output completeness permits more flexibility by allowing some arguments to be specified as inputs and only counting as negative examples those extra outputs generated for specific inputs in the training set.

The system most similar to FOIDL is FFOIL. Both systems use decision lists to learn functions, and both provide implicit negatives. However, there are several important difference between the systems. The two most obvious are the intensional versus extensional background and the different order in which they learn clauses. Also significant, though more subtle, are the differences in their handling of implicit negatives. FFOIL determines the number of negatives covered based on the range of the function as specified in the provided constants. It does not allow for the possibility that some, but not all of those constants might be covered, and it does not allow for the difficulties of generating all of the constants for a function an intractably large range, such as the standard past tense task. For experiments with FFOIL on the past tense task, Quinlan still uses a special formulation of the problem invented to allow FOIL some success at the task, which exploits the knowledge that the English past tense is formed using suffixes in order to greatly reduce the number of constants required in order to produce negatives. The final distinction between the two systems is FOIDL's greater flexibility. The implicit negatives can be used without decision lists to handle non-functional relations.

6 Conclusions

We have shown that FOIDL, an ILP system originally designed to address a particular problem in natural language processing, has important advantages outside of natural language processing. FOIDL's results on the finite element mesh design problem are better than any others reported for an ILP system, and it also performs well on several list-processing tasks. We believe that these results indicate that FOIDL's innovations–decision lists and implicit negatives generated using the output completeness assumption–will prove useful for a variety of machine learning tasks.

References

Aha, D. W., Ling, C. X., Matwin, S., & Lapointe, S. (1993). Learning singly recursive relations from small datasets. In *Workshop on Inductive Logic - IJCAI93*, pp. 47–58.

Bergadano, F., & Gunetti, D. (1993). An interactive system to learn functional logic programs. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, pp. 1044–1049 Chambery, France.

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

Bratko, I. (1990). *Prolog Programming for Artificial Intelligence.* Addison Wesley, Reading:MA.

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

Clark, P., & Niblett, T. (1989). The CN2 induction algorithm. *Machine Learning*, *3*, 261–284.

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

De Raedt, L., & Bruynooghe, M. (1993). A theory of clausal discovery. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, pp. 1058–1063 Chambery, France.

Dolsak, B., & Muggleton, S. (1992). The application of inductive logic programming to finite-element mesh design. In Muggleton, S. (Ed.), *Inductive Logic Programming*, pp. 453–472. Academic Press, New York.

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

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

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*, *3*, 1–24.

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

Quinlan, J. R. (submitted). Learning first-order definitions of functions. *Artificial Intelligence.*

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.

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

Webb, G. I., & Brkič, N. (1993). Learning decision lists by prepending inferred rules. In *Proceedings of the Australian Workshop on Machine Learning and Hybrid Systems*, pp. 6–10 Melbourne, Australia.

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*, pp. 343–351 New Brunswick, NJ.