Appears in *New Generation Computing,* 16, 3, p. 263-281 (1998)

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 as a substitute for 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 generally more accurate results than a range of methods previously applied to this problem. 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.

1 Introduction

Inductive Logic Programming (ILP) is a research area at the intersection of machine learning and logic programming and concerns the synthesis of logic programs from sample input/output tuples for a given target predicate and background knowledge in the form of definitions of predicates that can be employed by the learned program (Muggleton, 1992; Lavrač & Džeroski, 1994). Mooney and Califf (1995) introduced FOIDL, a new ILP system motivated by problems encountered when applying existing ILP methods to the problem of learning to generate the past tense of English verbs from their base forms. Past-tense

*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.

learning is a touchstone problem in language acquisition that has been fairly extensively studied in developmental psycholinguistics and cognitive science (Rumelhart & McClelland, 1986; Bloom, 1994). FOIDL is based on the popular and successful FOIL ILP system (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 out-perform all previous learning methods applied to this problem (Mooney & Califf, 1995). However, although these novel characteristics of the algorithm were motivated by the past-tense problem, they are general properties that promised to be useful for other problems as well.

In this paper, we present results that show that FOIDL is a very general approach that is useful for a range of problems. First, we extended the system to learn recursive definitions and expanded its search to include *determinate literals* (Muggleton & Feng, 1990; Quinlan, 1991). We then tested the system on two ILP benchmarks: the finite element mesh design problem introduced by Dolsak and Muggleton (1992) and a selection of the list-processing programs from Bratko (1990) previously used by Quinlan and Cameron-Jones (1993). We compare FOIDL's performance to FOIL and to FFOIL (Quinlan, 1996), a recent specialized version of FOIL for learning single-output functions without explicit negative examples. First-order decision lists enable FOIDL to achieve accuracy on the finite element mesh design problem that is generally superior to a range of previous ILP systems. FOIDL's intensionality and use of implicit negatives also allow it to more successfully learn correct list-processing programs from small sets of randomly selected positive examples. We also discuss the problem of evaluating ILP systems and present a new methodology for measuring the accuracy of inducing recursive programs which we believe is preferable to previous methods.

The remainder of the paper is organized as follows. Section 2 provides background on FOIL and FFOIL. Section 3 summarizes the original FOIDL algorithm and its extensions. 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 Our Research in ILP

Our research group has been interested in ILP and its use in practical applications for some time, and several ILP systems have been developed due to that interest. The FORTE system applies theory refinement techniques to first-order Horn clause theories (Richards & Mooney, 1995). It identifies possible errors in the initial theory using a set of training examples and revises the theory using operators developed using a variety of methods such as propositional theory refinement, first-order induction, and inversion resolution. The CHILL system applies ILP to the problem of learning parsers for natural language (Zelle & Mooney, 1996). ILP is used to learn control rules for an initial overly general parser. The DOLPHIN system combines ILP with Explanation-Base Generalization (EBG) to learn search control rules which eliminate backtracking in Prolog programs (Zelle & Mooney, 1993). The SCOPE system builds on the ideas in DOLPHIN. It uses EBG and ILP to improve planning systems by constructing domain-specific control rules (Estlin & Mooney, 1997). The RAPIER system, while it does not use a logic representation, uses ILP techniques to learn patterns for information extraction (Califf & Mooney, 1997).[1]

2.2 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).[2] 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 a complete set of tuples of constants of specified types that satisfy the predicate. 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. The papers referenced above provide details and information on additional features.

2.3 FFOIL

FFOIL (Quinlan, 1996) is a descendant of FOIL with modifications, somewhat similar to FOIDL's, that specialize it for learning functional relations.[3] First, FFOIL assumes that the final argument of the relation is an *output argument* and that the other arguments of

[1] For further information on these systems and other work done in our research lab, see our web page at `http://net.cs.utexas.edu/users/ml/`.

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

[3] The development of FFOIL was partially motivated by our own work on FOIDL.

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

3.1 Review of the Basic 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, listing a complete set of ground tuples is burdensome and frequently intractable. Frequently this is handled by artificially limiting the domain of terms, such as only considering lists of length at most three composed from a fixed set of three atoms. 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 the 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 negative examples of the target concept are reproved for each alternative specialization of the developing clause. Of course, determining the coverage of a clause by theorem proving instead of database lookup incurs a reasonably significant computational cost.

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 given a list as input the predicate should provide the final element, 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 consider the 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. Both clauses 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-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 that can still be proved as instances of the target concept using the specialized clause, plus the output queries 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 (like a `cond` statement in LISP).

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-pro cessing 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.

3.2 Handling Recursion

In extensional systems such as FOIL, FFOIL, and GOLEM, it is assumed that a recursive predicate call can be evaluated by directly matching the set of extensional training examples. Since a recursive literal is assumed to be false unless it matches a training example, the training set must be specifically constructed to be complete for some restricted body of terms, such as all lists of three possible elements with length at most three. For example, if an extensional ILP system is given the positive training examples:

```
member(a,[c,b,a]),
member(a,[a])
```

but is *not* given the example

```
member(a,[b,a]),
```

then the call `member(a,[b,a])` will be incorrectly assumed to fail when invoked by a learned recursive definition. Therefore, it is difficult for extensional systems to learn recursive definitions from random examples because the coverage of recursive clauses cannot be properly determined since an incomplete extensional definition provides only a “noisy oracle” for the target predicate (Cohen, 1993).

In an intensional system such as FOIDL or CHILLIN (Zelle & Mooney, 1994; Zelle, 1995), a recursive call is evaluated intensionally using the current partially-learned definition. In FOIDL, in order to allow a recursive call to terminate before all of the proper base cases have been learned, the set of positive examples are initially included in the learned definition for use as possible base cases. During the construction of a new clause, before alternative specializations of the current clause are tested for their ability to cover a given example, the example is temporarily removed from the current definition so that it can not be used to prove itself. For example, consider the following positive examples:

```
member(a,[c,b,a]),
member(a,[a]),
member(b,[b]).
```

An initial proposed clause

```
member(X, [H | T]) :- member(X, T).
```

can be shown to cover the first example by invoking it twice recursively and then matching the second example as a temporary base case. However, the second example is not considered to cover itself. Subsequently, the correct base case:

```
member(X, [X]).
```

can be learned to cover the last two examples. If instead the correct base case is learned first, then it can alternatively be used to ground out the intensional interpretation of the recursive clause when it is subsequently proposed. FOIDL avoids infinitely recursive clauses by using a pre-specified depth bound when proving examples. Alternatively, techniques for proving termination could be used to prevent infinite recursion (Cameron-Jones & Quinlan, 1993).

Once a definition is learned that covers all of the positive examples, redundant clauses are greedily deleted in order to simplify the definition and eliminate any unnecessary positive examples initially added to the definition. Each clause is temporarily deleted to determine if all of the examples can still be proven without it. If a clause is deemed unnecessary, it is permanently deleted and the remaining clauses are checked for possible deletion from the resulting definition. For the `member` example, all of the initial positive examples would be deleted since all of them can be proved from the two learned clauses alone.

Using this approach, FOIDL can learn recursive definitions from small, random, incomplete sets of examples. The experiments on learning definitions for simple list manipulation predicates in section 4.2 demonstrate that the approach works better than previous extensional methods. The same basic approach was initially used by CHILLIN and versions have been subsequently employed by SKILIT (Jorge & Brazdil, 1996) and FOIL-I (Inuzaka, Kamo, Ishii, Seki, & Itoh, 1996).

3.3 Adding Determinate Literals

Determinate literals are literals that may have little or no gain, but add at least one new variable that 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 (Quinlan, 1991). Determinate literals are added when no literal has positive gain, in the hope that the new variables will allow for a literal that does have gain. For example, in learning list processing relations such as append, it is often necessary to look at the structure of a list. The literal that divides the list into head and tail generally provides no information gain, but it is determinate and it makes possible subsequent literals that do have gain. When during the construction of a clause there is no literal with positive gain, all possible determinate literals are added in order to provide a range of new variables. Afterwards, the final clause is typically pruned to remove all unnecessary literals, since there can be large numbers of added literals which end up not contributing to the coverage or accuracy of the clause.

Determinate literals are very easy to recognize in an extensional system such as FOIL, since it maintains tuples for all of the positive and negative examples with the possible values for each variable. An intensional system does not have this advantage. Earlier systems, such as CHILLIN and the original version of FOIDL instead allow for *weak* literals, literals that provide no gain but do add at least one new variable. However, the huge number of such literals prohibits adding all of them as is done with determinate literals, and adding only the first one may not be helpful. While allowing weak literals was sufficient for tasks such as the past-tense, it was not sufficient for the list processing tasks described in this paper.

Object	Edges	Correct					
		FOIL	FFOIL	FOIDL	mFOIL	GOLEM	FORS
A	54	16	21	21	22	17	22
B	42	9	15	15	12	9	12
C	28	8	11	11	9	5	8
D	57	12	22	22	6	11	16
E	96	16	54	39	10	10	29
Total	277	61 (22%)	123 (44%)	108 (39%)	59 (21%)	52 (19%)	87 (31%)

Table 1: Results for finite element mesh design data

To find determinate literals, FOIDL must perform an intensional equivalent of determining the number of extensional tuples produced by a literal. The tuples are equivalent to a set of bindings produced by the clause to which the literal is added. So FOIDL considers a literal determinate if, given each set of bindings produced by the clause for every example, there is exactly one possible proof of the literal if the example is positive, and no more than one possible proof of the literal if the example is negative. When a clause is complete, FOIDL greedily prunes literals that are not necessary to prove the positives covered or eliminate negatives.

4 Experimental Results

4.1 Finite Element Mesh Design

The finite element mesh design application, introduced by Dolsak and Muggleton (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 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 for several other systems. The numbers for FOIL and FFOIL are taken from (Quinlan, 1996). Those for mFOIL and GOLEM are from (Lavrač & Džeroski, 1994). FORS (Karalič, 1995), like FFOIL and FOIDL is a system specialized for learning functions. It is a first order regression system, which is particularly appropriate for this task, which is essentially a regression task. For the non-

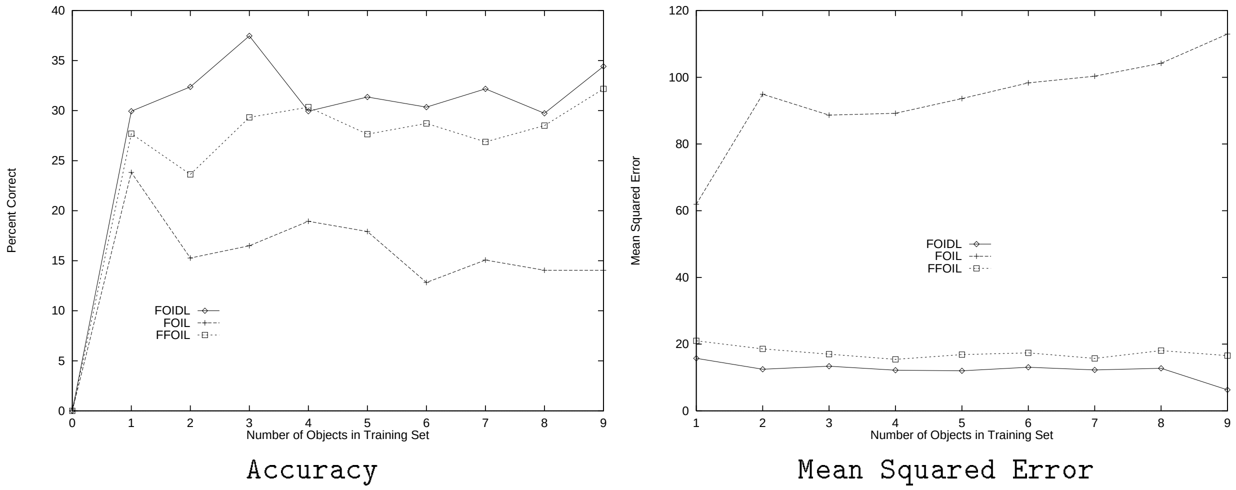

Figure 1: Results for full finite element mesh design data set

functional learning systems only the first answer returned by the learned definition is used to determine performance. The functional systems have a clear advantage on this problem, and FFOIL and FOIDL perform the best.

The finite element mesh design problem is clearly a difficult one, and most tests have used only the first five objects of the ten available, so we decided to examine the performance of FOIDL and FFOIL in more detail by running an experiment including all of the objects and running learning curves. The methodology used was to use leave-one-object-out cross-validation as in the other experiments, but also to run experiments using subsets of each training set in order to get the learning curves. The results from this experiment appear in Figure 1.

We looked both at the number of correct items and at the mean squared error between the correct and predicted number of intervals. The second metric is appropriate because this is a task where the distance from the correct answer is significant: being a little off may be acceptable, but being way off is not. Dividing an edge into 6 or 8 segments instead of 7 may still produce reasonable results without too much cost in computation, while using only 2 intervals would result in poor results and using 12 intervals might be far too costly.

In this experiment, the advantages of explicitly learning a functional definition are again very clear. Unlike with the five object data set, FOIDL fairly consistently outperforms FFOIL slightly, but the difference in accuracy is not significant. There seems to be a problem with over-fitting just looking at the accuracy graph, and FFOIL's noise handling probably helps improve its accuracy. However, FFOIL often fails to produce programs that cover the less common classes, while FOIDL's reverse order learning of clauses helps it to learn rules for those less common classes. This becomes apparent looking at the mean squared error. FOIDL outperforms FFOIL much more consistently on this metric, because it does learn clauses for the less common values.

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 the Prolog textbook by 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 (and other systems like GOLEM and PROGOL (Muggleton, 1995)) must either be provided with explicit negatives or be able to generate negatives using a closed world assumption, it cannot learn programs from smaller sets of examples if only positives are provided. Our hypothesis is that FOIDL's implicit negatives and intentional approach would enable it to learn a number of these list-processing relations from smaller sets of random positive examples.

4.2.1 Experimental Methodology

Due to the restrictions of extensional methods (see Section 3.2), many previous experiments on learning list-processing Prolog programs have employed specially constructed sets of examples that are guaranteed to be complete (Quinlan, 1990; Muggleton & Feng, 1990). However, ideally an ILP system should be able to learn such programs from random examples rather than carefully-selected sets. Some recent experiments have tested the ability of ILP systems to learn list-processing programs from small random samples of both positive and negative examples (Zelle & Mooney, 1994; Aha, Lapointe, Ling, & Matwin, 1994; Jorge & Brazdil, 1996; Inuzaka et al., 1996). For example, Inuzaka et al. (1996) present results showing that FOIL is usually more likely to learn correct programs in this case than PROGOL and that an intensional version of FOIL (FOIL-I) is consistently better than both FOIL and PROGOL.

However, supplying explicit negative examples of program I/O seems an unusual and burdensome requirement. Therefore, we believe that learning from random samples of positive examples with negatives expressed only implicitly by an assumption of a functional target predicate or output-completeness is more practical in many cases. Therefore, we compare several systems that do not require negative examples on their ability to learn from small sets of random positive examples.

Another issue in evaluating program induction is how to measure relative success. A few previous studies have produced standard learning curves which present the accuracy of a learned program at classifying random I/O tuples as positive or negative after training on various sized sets of random examples (Zelle & Mooney, 1994; Jorge & Brazdil, 1996). However, classifying random tuples is not the way a program is normally used. A more relevant measure of approximate accuracy of a program would be the average percentage of randomly selected inputs for which the program produces all and only the correct outputs (Mooney & Califf, 1995; Mooney, 1996). Other measures of accuracy typically used in information retrieval such as *recall* (percentage of the correct outputs which are produced) and *precision* (percentage of produced outputs which are correct) may also be useful. However, many may claim that an approximately correct list-processing program is not very useful and that an ILP system has only succeeded if it produces a *completely* correct program. Consequently, we have decided to present learning curves that measure the probability that a completely correct program is produced from random training sets of positive examples of various sizes. This probability was measured by running 20 independent trials in which the system is trained on random subsets of positive examples of a given size and determining the percentage of trials in which the system learns a completely correct definition that works for lists of arbitrary length (as determined by a manual inspection of the resulting programs). Inuzaka et al. (1996) presents similar results in tabular form for learning from random sets of positive and negative examples. We believe that this approach using only random positive I/O pairs is preferable to most previous methods for evaluating program induction and encourage others to adopt it as well.

4.2.2 Experimental Results

The relations we considered naturally divide into two distinct sets of experiments since several of the relations are functional and others are not. Quinlan (1996) showed that FFOIL can learn the 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 using FOIDL, FFOIL, and IFOIL, which is FOIDL without decision lists.

The five functional relations we use are `last/2`, which takes a list as input and returns the last element, `conc/3`, which is the same as `append/3` and takes two lists as input and returns the concatenation of the lists, `reverse/2`, which takes as input a list and returns the list in reverse order, `shift/2`, which takes a list and shifts the elements one position to the left wrapping around front to back, and `translate/2`, which takes a list and returns a list in which all the members have been replaced by their substitutions, where the substitutions are defined by a predicate `means/2`. All three of the systems tested can learn correct definitions for these problems using a full universe of lists of three elements with lengths up to three.

On `reverse`, our results were not encouraging; it requires the entire universe of 40 examples when lists are limited to length three for all three of the systems. Figure 2 shows the results for `last`, `conc`, `shift` and `translate`. Both FOIDL and IFOIL are able to learn correct definitions from relatively small subsets of the positive examples. The results for `translate` do not convey quite how well FOIDL is doing on this problem. The base case for `translate` is `translate([],[]).`, and the system can't learn a definition for `translate` that covers the empty list when the empty list isn't in the set of training examples. FOIDL always learns the correct definition when the base case is in the training examples, and it usually learns a correct definition for all lists except the empty list. The lack of base cases is not a major problem for the other three predicates, because the base cases are a larger proportion of the data.

FFOIL's performance on these tasks points to one of the advantages of the FOIDL algorithm. The correct programs for `last`, `conc`, and `translate` are recursive, but `shift` does not require explicit recursion since it can be learned using `conc`. Note that `shift` is the only problem on which FFOIL performs better than FOIDL. FOIDL does better on the recursive

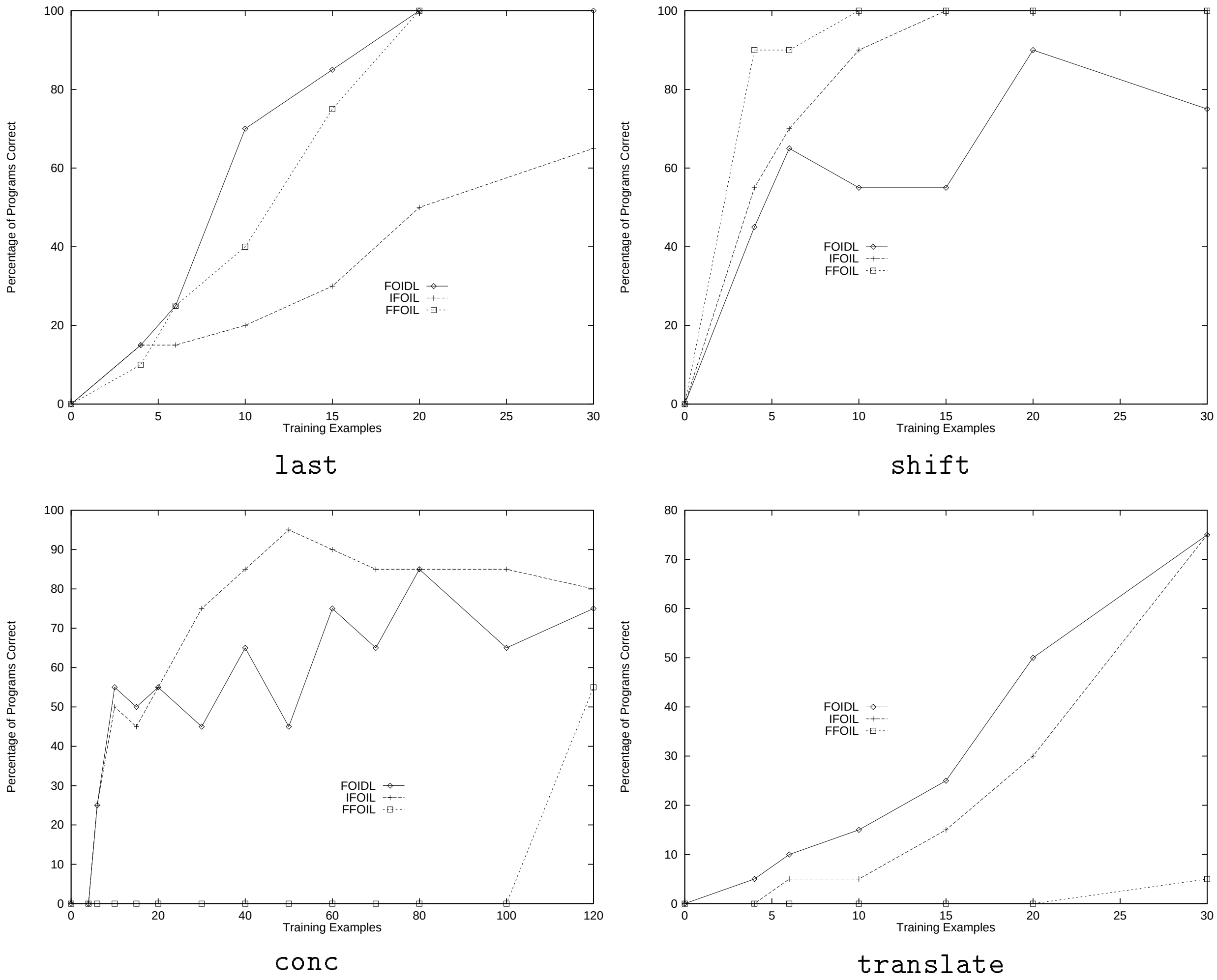

Figure 2: Results for functional list processing programs

problems 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 intensionally interprets both the background and learned rules, it requires *any* base example or clause, not the immediately preceding example. Thus, in order 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.

FOIDL and IFOIL vary quite a bit on these tasks, with FOIDL consistently better on `last` and `translate`, but consistently worse on `shift` and `conc`. FOIDL's reverse approach to learning decision lists hurt it to some extent on both `shift` and `conc`, with a tendency to create bad default clauses with skewed example sets. On `last` and `translate`, however, the decision list bias seems to help.

Besides demonstrating the performance of FOIDL on functions, we wished to examine the usefulness of implicit negatives in non-functional applications. So we also ran an experiment with three of the non-functional list-processing problems: `del/3`, `insert/3`, and `member/2`. The predicate `del` takes an element and a list, and returns a list with one instance of that element removed from the list. More than one output may be possible:

```
?- del(1, [1, 2, 1], X).
 X = [2, 1] ;
 X = [1, 2]
```

The predicate `insert/3` takes an element and a list and returns the result of inserting that element somewhere in the list. The relationship to `del` is obvious, and is typically exploited in defining `insert`:

```
insert(A, B, C) :- del(A, C, B).
```

The predicate `member/2` is standard list membership, taking a list as input and returning each member.

Decision lists are not appropriate for these tasks, since the cuts at the end of each clause prevent backtracking to find additional solutions. Therefore, we use only IFOIL on these 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 IFOIL on each subset to determine whether it produced a correct program for lists of arbitrary length. Because of the requirement that the training data 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 tuples with those inputs.

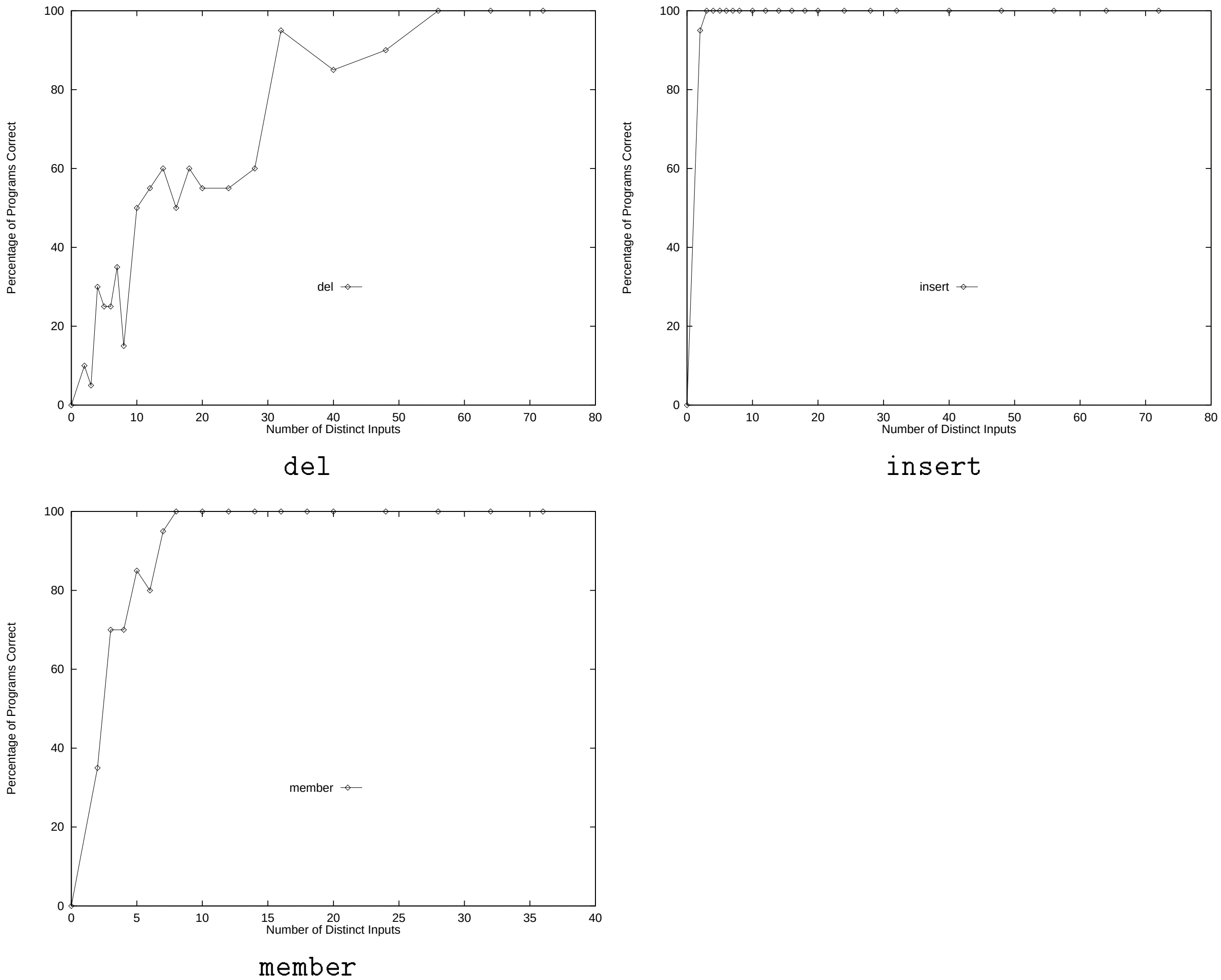

Figure 3: IFoil's performance on non-functional list processing programs

The results for the non-functional relations tested appear in Figure 3. Although it learns some relations faster than others, IFOIL 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, and 81 for insert) and no explicit negatives.

5 Related Work

The first ILP system to focus specifically on learning functions was FILP (Bergadano & Gunetti, 1993). It assumes that the target and all background relations must be functional and uses this knowledge to limit its search for literals. However, these assumptions prevent its application to many of the tasks considered here because they typically involve non-functional background relations. Also, although FILP assumes that its target relation is a function, the definitions learned consist of unordered sets of clauses.

Other systems, most notably LOPSTER(Lapointe & Matwin, 1992) and CRUSTACEAN (Aha et al., 1994), have addressed learning recursive 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 crucially exploit the assumption that the target relation has this particular structure. More recently, Jorge and Brazdil (1996) has used SKILIT to learn recursive programs from small sets of randomly selected positive and negative examples with some success. However, they provide the system with a grammar defining the structure of the recursive predicates, and measure accuracy on classifying tuples rather than the more stringent requirement of completely correct programs. FOIDL can require a few more examples than these other systems, but this is because it is a more general system which has solved a number of other types of problems as well, and because FOIDL does not make assumptions about the structure of the solution, only about the nature of the training data.

Two other systems use the general idea of implicit 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 employ implicit negatives. However, there are several important differences. 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 with an intractably large range, such as the past-tense task. For experiments with FFOIL on past-tense, 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 since it can employ implicit negatives without decision lists to learn non-functional relations.

6 Conclusions

We have shown that FOIDL, an ILP system originally designed to address a particular problem in the acquisition of morphology, has important advantages outside of natural-language processing. FOIDL's accuracy results on the finite element mesh design problem are not significantly different from FFOIL's (the current best performer in a variety of ILP systems applied to this problem) and better in terms of mean squared error. The system also performs well on a variety of list-processing tasks, learning completely correct programs from small random sets of only positive examples. 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 relational machine-learning tasks.

References

Aha, D. W., Lapointe, S., Ling, C. X., & Matwin, S. (1994). Learning recursive relations with randomly selected small training sets. In *Proceedings of the Eleventh International Conference on Machine Learning*, pp. 12–18 New Brunswick, NJ.

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.

Bloom, P. (1994). Overview: Controversies in language acquisition. In Bloom, P. (Ed.), *Language Acquisition: Core Readings*, pp. 5–48. MIT Press, Cambridge, MA.

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

Califf, M. E., & Mooney, R. (1997). Applying ilp-based techniques to natural language information extraction: An experiment in relational learning. In *Working Notes of the IJCAI-97 Workshop on Frontiers in Inductive Logic Programming*.

Cameron-Jones, R. M., & Quinlan, J. R. (1993). Avoiding pitfalls when learning recursive theories. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, pp. 1050–1055 Chambery, France.

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.

Estlin, T. A., & Mooney, R. (1997). Learning to improve both efficiency and quality of planning. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, pp. 1227–1232.

Inuzaka, N., Kamo, M., Ishii, N., Seki, H., & Itoh, H. (1996). Top-down induction of logic programs from incomplete samples. In *Proceedings of the Sixth International Workshop on Inductive Logic Programming*, pp. 119–136.

Jorge, A., & Brazdil, P. (1996). Architecture for iterative learning of recursive definitions. In De Raedt, L. (Ed.), *Advances in Inductive Logic Programming*. IOS Press, Amsterdam.

Karalič, A. (1995). *First Order Regression*. Ph.D. thesis, University of Ljubljana,Slovenia.

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. (1994). *Inductive Logic Programming: Techniques and Applications*. Ellis Horwood.

Mooney, R. J. (1996). Inductive logic programming for natural-language processing. In *Proceedings of the Sixth International Workshop on Inductive Logic Programming*, pp. 205–224.

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. (1990). Efficient induction of logic programs. In *Proceedings of the First Conference on Algorithmic Learning Theory* Tokyo, Japan. Ohmsha.

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

Muggleton, S. (1995). Inverse entailment and Progol. *New Generation Computing Journal*, *13*, 245–286.

Quinlan, J. R. (1991). Determinate literals in inductive logic programming. In *Proceedings of the Eighth International Workshop on Machine Learning* Evanston, IL.

Quinlan, J. R. (1996). Learning first-order definitions of functions. *Journal of Artificial Intelligence Research*, *5*, 139–161.

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.

Richards, B. L., & Mooney, R. J. (1995). Automated refinement of first-order Horn-clause domain theories. *Machine Learning*, *19*(2), 95–131.

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

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.

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. (1995). *Using Inductive Logic Programming to Automate the Construction of Natural Language Parsers*. Ph.D. thesis, University of Texas, Austin, TX. Also appears as Artificial Intelligence Laboratory Technical Report AI 96-249.

Zelle, J. M., & Mooney, R. J. (1993). Combining FOIL and EBG to speed-up logic programs. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, pp. 1106–1111 Chambery, France.

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.

Zelle, J. M., & Mooney, R. J. (1996). Learning to parse database queries using inductive logic programming. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence* Portland, OR.