Appears in *Symbolic, Connectionist, and Statistical Approaches to Learning for Natural Language Processing,* Springer Verlag, 1996.
Also appears in *Working Notes of the IJCAI-95 Workshop on New Approaches to Learning for Natural Language Processing*Montreal, Quebec, Canada, August 1995.

Learning the Past Tense of English Verbs Using Inductive Logic Programming

Raymond J. Mooney and Mary Elaine Califf

Department of Computer Sciences, University of Texas
Austin,TX 78712-1188

Abstract. This paper presents results on using a new inductive logic programming method called FOIDL to learn the past tense of English verbs. The past tense task has been widely studied in the context of the symbolic/connectionist debate. Previous papers have presented results using various neural-network and decision-tree learning methods. We have developed a technique for learning a special type of Prolog program called a *first-order decision list,* defined as an ordered list of clauses each ending in a cut. FOIDL is based on FOIL [19] but employs intensional background knowledge and avoids the need for explicit negative examples. It is particularly useful for problems that involve rules with specific exceptions, such as the past-tense task. We present results showing that FOIDL learns a more accurate past-tense generator from significantly fewer examples than all previous methods.

1 Introduction

The problem of learning the past tense of English verbs has been widely studied as an interesting subproblem in language acquisition. Previous research has applied both connectionist and symbolic method to this problem [22, 12, 9]; however, previous efforts used specially-designed feature-based encodings that impose a fixed limit on the length of words and fail to capture the generativity and position-independence of the underlying transformation. We believed that representing the problem as constructing a logic program for the predicate `past(X,Y)` where `X` and `Y` are words represented as lists of letters (e.g `past([a,c,t],[a,c,t,e,d])`, `past([a,c,h,e],[a,c,h,e,d])`, `past([r,i,s,e],[r,o,s,e])`) would produce much better results.

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 [15, 8]. Due to the expressiveness of first-order logic, ILP methods can learn relational and recursive concepts that cannot be represented in the attribute/value representations assumed by most machine-learning algorithms. However, current ILP techniques make important assumptions that restrict their application. Many assume that background knowledge is provided *extensionally* as a set of ground literals. However, an adequate extensional representation of background knowledge for some problems is infinite or intractable large. Most techniques assume that explicit negative examples of the target predicate are available or can be computed using a closed-world assumption, but for

some problems explicit negative examples are not available, and an adequate set of negative examples computed using a closed-world assumption is infinite or intractably large. A third assumption is that the target program is expressed in "pure" Prolog where clause-order is irrelevant and procedural operators such as cut (!) are disallowed. However, a concise representation of many concepts requires the use of clause-ordering and/or cuts [2]. The currently most well-known and successful ILP systems, GOLEM [14] and FOIL [19], both make all three of these assumptions.

Due to these limitations, we were unable to get reasonable results on learning past tense from either FOIL or GOLEM. This paper presents a new ILP method called FOIDL (First-Order Induction of Decision Lists) which helps overcome these limitations. The system represents background knowledge *intensionally* as a logic program. It does not require explicit negative examples. Instead, an assumption of *output completeness* can be used to implicitly determine whether a hypothesized clause is overly-general and to quantify the degree of over-generality by estimating the number of negative examples covered. Finally, a learned program can be represented as a *first-order decision list*, an ordered set of clauses each ending with a cut. As its name implies, FOIDL is closely related to FOIL and follows a similar top-down, greedy specialization guided by an information-gain heuristic. However, the algorithm is substantially modified to address the three advantages listed above. The resulting system is able to learn the past tense of English more accurately and from fewer examples than any of the previous methods applied to this problem.

The remainder of the paper is organized as follows. Section 2 provides background material on FOIL and on the past-tense learning problem. Section 3 presents the FOIDL algorithm. Section 4 presents our results on learning the past-tense of English verbs. Section 5 discusses some related work, and Section 6 presents directions for future work. Section 7 summarizes and presents our conclusions.

2 Background

2.1 FOIL

Since FOIDL is based on FOIL, this section presents a brief review of this important ILP system; see articles on FOIL for a more complete description [19, 18, 4]. 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. The papers referenced above provide details and information on additional features.

2.2 Learning the Past Tense of English Verbs

The problem of learning the English past tense has been attempted by both connectionist systems [22, 12] and systems based on decision tree induction [11, 9]. The task to be learned in these experiments is: given a phonetic encoding of the base form of an English verb, generate the phonetic encoding of the past tense form of that verb. The task can also be done using the alphabetic forms forms of the verbs, and we use that form of the task for the examples in this paper. 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. Each output unit or separate decision tree is used to predict a character in the fixed-length output pattern from all of the input characters.

Although ILP methods seem more appropriate for this problem, our initial attempts to apply FOIL and GOLEM to past-tense learning gave very disappointing results [3]. Below, we discuss how the three problems listed in the introduction contribute to the difficulty of applying current ILP methods to this problem.

In principle, a background predicate for `append` is sufficient for constructing accurate past-tense programs when incorporated with an ability to include constants as arguments or, equivalently, an ability to add literals that bind variables to specific constants (called *theory constants* in FOIL). However, a background predicate that does not allow appending with the empty list is more appropriate. We use a predicate called `split(A, B, C)` which splits a list `A` into two non-empty sublists `B` and `C`. An intensional definition for `split` is:

```
split([X, Y | Z], [X] , [Y | Z]).
split([X | Y], [X | W], Z) :- split(Y,W,Z).
```

Providing an extensional definition of `split` that includes all possible strings of 15 or fewer characters (at least 10^{21} strings) is clearly intractable. However, providing a partial definition that includes all possible splits of strings that actually

appear in the training corpus is possible and generally sufficient. Therefore, providing adequate extensional background knowledge is cumbersome and requires careful engineering; however, it is not the major problem.

Supplying an appropriate set of negative examples is more problematic. 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 applied FOIL to the past tense problem [17], 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([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.

Although he solves the problem of providing negatives, Quinlan notes that his results are still hampered by FOIL's inability to exploit clause order [17]. For example, when using normal alphabetic encoding, FOIL quickly learns a clause sufficient for regular verbs:

```
past(A,B,C) :- B=[], C=[e,d].
```

However, since this clause still covers a fair number of negative examples due to many irregular verbs, it continues to add literals. As a result, FOIL creates a number of specialized versions of this clause that together still fail to capture the generality of the underlying default rule.

However, an experienced Prolog programmer would exploit clause order and cuts to write a concise program that first handles the most specific exceptions and falls through to more general default rules if the exceptions fail to apply. Such a program might be:

```
past(A,B) :- split(A,C,[e,e,p]), split(B,C,[e,p,t]), !.
past(A,B) :- split(A,C,[y]), split(B,C,[i,e,d]), !.
past(A,B) :- split(A,C,[e]), split(B,A,[d]), !.
past(A,B) :- split(B,A,[e,d]).
```

FOIDL can directly learn programs of this form, i.e., ordered sets of clauses each ending in a cut. We call such programs first-order decision lists due to the similarity to the propositional *decision lists* introduced by Rivest [21]. FOIDL uses the normal binary target predicate and requires no explicit negative examples. Therefore, we believe it requires *significantly* less representation engineering than all previous work in the area.

3 FOIDL Induction Algorithm

As stated in the introduction, 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. Intentional 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, expanded tuples are not maintained and positive and 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. A mode declaration and an assumption of output completeness together determine a set of implicit negative examples. The output completeness assumption indicates that for every unique input pattern in the training set, the training set includes all of the correct output patterns. Therefore, any other output which a programm produces for a given input pattern must be a negative example.

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(-,-,+)`, indicating 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., `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. 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., `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) :- split(A,C,D).`, the query `past([a,c,t], X)` generates the an-

swer `past([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 a bound on the 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. It is generally sufficient to estimate u as a fairly large constant (e.g., 1000), and empirically the method is not very sensitive to its exact value as long as it is significantly greater than the number of ground outputs ever generated by a clause.

Unfortunately, this estimate is not sensitive enough. For example, both clauses

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

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.

As described above, first-order decision lists 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 [21], 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 [21] and in CN2 [5], 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 Brkic [23] 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. It learns an ordered sequence of clauses in reverse order, resulting in a program which produces only the first output generated by the first satisfied clause.

The basic operation of the algorithm is best illustrated by a concrete example. For alphabetic past-tense, the current algorithm easily learns the partial clause:

```
past(A,B) :- split(B,A,C), C = [e,d].
```

This clause still covers negative examples due to irregular verbs. However, it produces correct ground output for a subset of the examples. Therefore, it is best to terminate this clause to handle these examples, and add earlier clauses in the decision list to handle the remaining examples. The fact that it produces incorrect answers for other output queries can be safely ignored in the decision-list framework. The examples correctly covered by this clause are removed from *positives-to-cover* and a new clause is begun. The literals that now provide the best gain are:

```
past(A,B) :- split(B,A,C), C = [d].
```

covering the verbs that just add "d" since they end in "e". This clause also produces correct ground output for a subset of the examples; however, it is not complete since it produces incorrect output for examples correctly covered by a previously learned clause (e.g., `past([a,c,t], [a,c,t,d])`). Therefore, specialization continues until all of these cases are also eliminated, resulting in the clause:

```
past(A,B) :- split(B,A,C), C = [d], split(A,D,E), E = [e].
```

which is added to the front of the decision list. This approach ensures that every new clause produces correct outputs for some new subset of the examples but doesn't result in incorrect output for examples already correctly covered by previously learned clauses. This process continues adding clauses to the front of the decision list until all of the exceptions are handled and *positives-to-cover* is empty.

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 a 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'.

In many cases, this algorithm is able to learn accurate, compact, first-order decision lists for past tense, like the "expert" program shown in section 2.2. However, the algorithm can encounter local-minima in which it is unable to find any literals that provide positive gain while still covering the required minimum number of examples.[1] This was originally handled by terminating search and memorizing any remaining uncovered examples as specific exceptions at the top of the decision list. However, this can result in premature termination that prevents the algorithm from finding low-frequency regularities. For example, in the alphabetic version, the system can get stuck trying to learn the complex rule for when to double a final consonant (e.g., grab $\rightarrow$ grabbed) and fail to learn the rule for changing "y" to "ied" since this is actually less frequent.

The current version, like FOIL, tests if the learned clause meets a minimum-accuracy threshold, but only counts as errors incorrect outputs for queries correctly answered by previously learned clauses. If it does not meet the threshold, the clause is thrown out and the positive examples it covers are memorized at the top of the decision list. The algorithm then continues to learn clauses for any remaining positive examples.

When the minimum-accuracy threshold is met, the decision-list property is exploited in a final attempt to still learn a completely accurate program. If the negatives covered by the clause are all examples correctly covered by previously learned clauses, FOIDL treats them as "exceptions to the exception to the rule" and returns them to *positives-to-cover* to be covered correctly again by subsequently learned clauses. With the minimum clause-accuracy threshold set to 50%, FOIDL only applies this *uncovering* technique when it results in covering more examples than it uncovers, thereby guaranteeing progress towards fitting all of the training examples.

An implementation of FOIDL in Quintus Prolog is available by anonymous FTP from `ftp.cs.utexas.edu`.

4 Experimental Results

To test FOIDL's performance on the English past tense task, we ran experiments using data from Ling [9] which consist of 1390 pairs of base and past tense verb forms in alphabetic and UNIBET phonemic form. We ran three different experiments. In one we used the phonetic forms of all verbs. In the second we used the phonetic forms of the regular verbs only, because this is the easiest form of the task and because this is the only problem for which Ling provides

[1] Like FOIL, FOIDL includes a parameter for the minimum number of examples that a clause must cover (normally set to 2).

learning curves. Finally, we ran trials using the alphabetic forms of all verbs. 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.

In order to better separate the contribution of using implicit negatives from the contribution of the decision list representation, we also ran experiments with IFoil, a variant of the system which uses intensional background and the output completeness assumption, but does not build decision lists.

We ran our own experiments with Foil, Foidl, and IFoil and compared those with the results from Ling. The Foil experiments were run using Quinlan's representation described above. As in Quinlan [17], negative examples were provided by using a randomly-selected 25% of those which could be generated using the closed world assumption.[2] All experiments with Foidl and IFoil used the standard default values for the various numeric parameters. The differences among Foil, IFoil, and Foidl were tested for significance using a two-tailed paired t-test.

4.1 Results

The results for the phonetic task using all verbs are presented in Figure 1. The graph shows our results with Foil, IFoil, and Foidl along with the best results from Ling, who did not provide a learning curve for this task. As expected, Foidl out-performed the other systems on this task, surpassing Ling's best results with 500 examples with only 100 examples. IFoil performed quite poorly, barely beating the neural network results despite effectively having 100% of the negatives as opposed to Foil's 25%. This poor performance is due at least in part to overfitting the training data, because IFoil lacks the noise-handling techniques of Foil6. Foil also has the advantage of the three-place predicate, which gives it a bias toward learning suffixes. IFoil's poor performance on this task shows that the implicit negatives by themselves are not sufficient, and that some other bias such as decision lists or the three-place predicate and noise-handling is needed. The differences between Foil and Foidl are significant at the 0.01 level. Those between Foidl and IFoil are significant at the 0.001 level. The differences between Foil and IFoil are not significant with 100 training examples or less, but are significant at the 0.001 level with 250 and 500 examples.

Figure 2 presents accuracy results on the phonetic task using regulars only. The curves for SPA and the neural net are the results reported by Ling. Here again, Foidl out-performed the other systems. This particular task demonstrated one of the problems with using closed-world negatives. In the regular past tense task, the second argument of Quinlan's 3-place predicate is always the same: an empty list. Therefore, if the constants are generated from the positive examples, Foil will never produce rules which ground the second argument,

[2] We replicated Quinlan's approach since memory limitations prevented us from using 100% of the generated negatives with larger training sets.

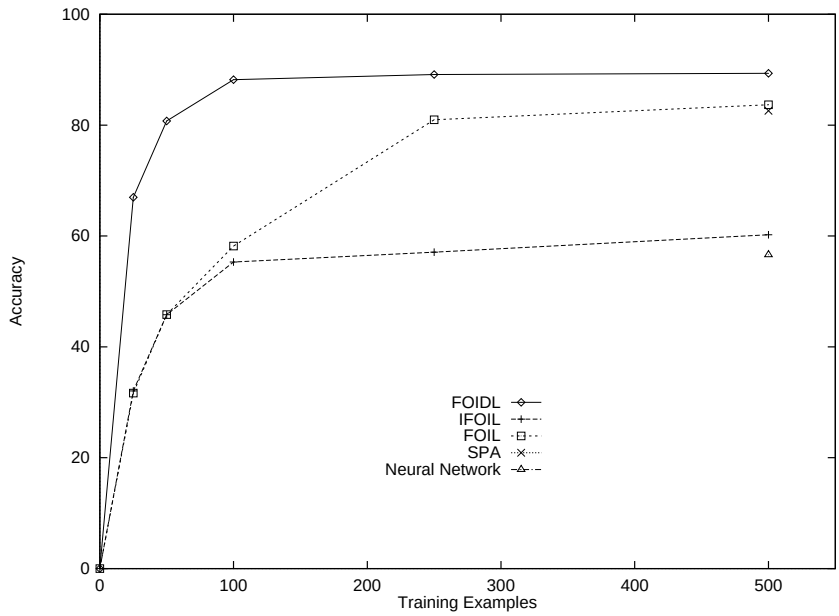

Fig. 1. Accuracy on phonetic past tense task using all verbs

since it cannot create negative examples with other constants in the second argument. This prevents the system from learning a rule to generate the past tense. In order to obtain the results reported here, we introduced extra constants for the second argument (specifically the constants for the third argument), enabling the closed world assumption to generate appropriate negatives. On this task, IFoil does seem to gain some advantage over Foil from being able to effectively use all of the negatives. The regularity of the data allows both IFoil and Foil to achieve over 90% accuracy at 500 examples. The differences between Foil and Foidl are significant at the 0.001 level, as are those between IFoil and Foidl. The differences between IFoil and Foil are not significant with 25 examples, and are significant at the 0.02 level with 500 examples, but are significant at the 0.001 level with 50-250 training examples.

Results for the alphabetic version appear in Figure 3. This is a task which has not typically been considered in the literature, but it is of interest to those concerned with incorporating morphology into natural language understanding systems which deal with text. It is also the most difficult task, primarily because of consonant doubling. Here we have results only for Foidl, IFoil, and Foil. Because the alphabetic task is even more irregular than the full phonetic task, IFoil again overfits the data and performs quite poorly. The differences between Foil and Foidl are significant at the 0.001 level with 25, 50, 250, and 500 examples, but only at the 0.1 level with 100 examples. The differences between IFoil and Foidl are all significant at the 0.001 level. Those between Foil and IFoil are not significant with 25 training examples and are significant only at

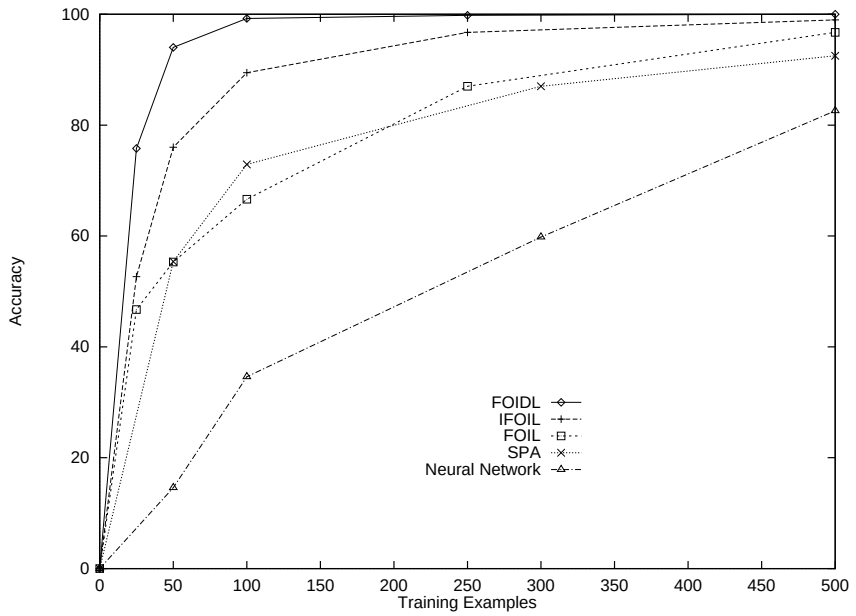

Fig. 2. Accuracy on phonetic past tense task using regulars only

the 0.01 level with 50 training examples, but are significant at the 0.001 level with 100 or more examples.

For all three of these tasks, FOIDL clearly outperforms the other systems. A sufficient set of negatives is necessary, and all five of these systems provide them in some way: the neural network and SPA both learn multiple-class classification tasks (which phoneme belongs in each position); FOIL uses the three-place predicate with closed world negatives; and IFOIL and FOIDL, of course, use the output completeness assumption. The primary importance of the implicit negatives is not that they provide an advantage over propositional and neural network systems, but that they enable first order systems to perform this task at all. Without them, some knowledge of the task is required. FOIDL's decision lists give it a significant added advantage, though this advantage is less apparent in the regular phonetic task, where there are no exceptions.

FOIDL also generates very comprehensible programs. The following is an example program generated for the alphabetic version of the task using 250 examples (excluding the memorized examples).

```
past(A,B) :- split(A,C,[e,p]), split(B,C,[p,t]),!.
past(A,B) :- split(A,C,[y]), split(B,C,[i,e,d]),
             split(A,D,[r,y]),!.
past(A,B) :- split(A,C,[y]), split(B,C,[i,e,d]),
             split(A,D,[l,y]),!.
past(A,B) :- split(B,A,[m,e,d]), split(A,C,[m]),
```

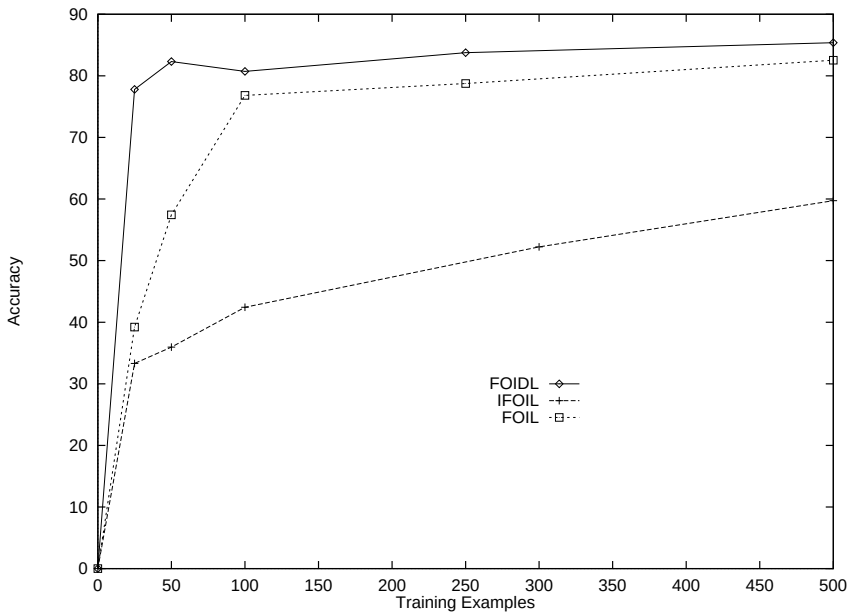

Fig. 3. Accuracy on alphabetic past tense task

```
              split(A,[s],D),!.
past(A,B) :- split(B,A,[r,e,d]), split(A,C,[u,r]),!.
past(A,B) :- split(B,A,[d]), split(A,C,[e]),!.
past(A,B) :- split(B,A,[e,d]),!.
```

5 Related Work

5.1 Related Work on Past-Tense Learning

The shortcomings of most previous work on past-tense learning were reviewed in section 2.2, and the results in section 4 clearly demonstrate the generalization advantage FOIDL exhibits on this problem.

Most of the previous work on this problem has concerned the modelling of various psychological phenomenon, such as the U-shaped learning curve that children exhibit for irregular verbs when acquiring language. This paper has not addressed the issue of psychological validity, and we make no specific psychological claims based on our current results.

However, humans can obviously produce the correct past tense of arbitrarily-long novel words, which FOIDL can easily model while fixed-length feature-based representations clearly cannot. Ling also developed a version of SPA that eliminates position dependence and fixed word-length [10] by using a sliding window. A large window is used which includes 15 letters on either side of the current

position (padded with blanks if necessary) in order to always include the entire word for all the examples in the corpus. The results on this approach are significantly better than normal SPA but still inferior to FOIDL's results.

5.2 Related Work on ILP

Although each of the three features mentioned in the introduction distinguishes FOIDL from most work in Inductive Logic Programming, a number of related pieces of research should be mentioned. The use of intensional background knowledge is the least distinguishing feature, since a number of other ILP systems also incorporate this aspect. FOCL [16], MFOIL [[8]], GRENDEL [6], FORTE [20], and CHILLIN [25] all use intensional background to some degree in the context of a FOIL-like algorithm.

The use of implicit negatives is significantly more novel. Bergadano et al. [2] allows the user to supply an intensional definition of negative examples that covers a large set of ground instances; 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 [7] 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. FLIP [1] provides a method for learning functional programs without negative examples by making an assumption equivalent to output completeness for the functional case only.

The notion of a first-order decision list is unique to FOIDL. The only other ILP system that attempts to learn programs that exploit clause-order and cuts is that of Bergadano et al. [2]. Their paper discusses learning arbitrary programs with cuts, and the brute-force search used in their approach is intractable for most realistic problems. FOIDL is tailored to the specific problem of learning first-order decision lists, which use cuts in a very stylized manner that is particularly useful for functional problems that involve rules with exceptions.

6 Future Work

One obvious topic for future research is FOIDL's cognitive modelling abilities in the context of the past-tense task. Incorporating over-fitting avoidance methods may allow the system to model the U-shaped learning curve in a manner analogous to that demonstrated by Ling and Marinov [11]. Its ability to model human results on generating the past tense of novel psuedo-verbs (e.g., spling $\rightarrow$ splang) could also be examined and compared to SPA and connectionist methods.

Although first-order decision lists represent a fairly general class of programs, currently our only convincing experimental results are on the past-tense problem. The decision list mechanism in general should be applicable to other language

problems (as evidenced by the use of propositional decision lists for problems such as lexical disambiguation [24]. Many realistic problems consist of rules with exceptions, and experimental results on additional applications are needed to support the general utility of this representation.

7 Conclusions

Learning the past tense of English is a small by interesting subproblem in language acquisition which captures some of the fundamental problems such as the generative ability to handle arbitrarily long input and the ability to learn exceptions as well as underlying regularities. Compared to feature-based approaches such as neural-network, decision tree, and statistical methods, inductive logic programming offers the advantage of generativity in being able to handle arbitrarily long input. In addition, the use of first-order decision lists allow one to easily represent exceptions as well as general default rules. Our results clearly demonstrate that an ILP system for learning first-order decision lists can outprerform both the symbolic and the neural-network systems previously applied to the past-tense task. Since the issues of generativity and exceptions and defaults are ubiquitous in language acquisition, we believe this approach will also be useful for other language learning problems.

Acknowledgements Most of the basic research for this paper was conducted while the first author was on leave at the University of Sydney supported by a grant to Prof. J.R. Quinlan from the Australian Research Council. Thanks to Ross Quinlan for providing this enjoyable and productive opportunity and to both Ross and Mike Cameron-Jones for very important discussions and pointers that greatly aided the development of FOIDL. Partial support was also provided by grant IRI-9310819 from the National Science Foundation and an MCD fellowship from the University of Texas awarded to the second author. A fuller discussion of this research appears in [13].

References

1. F. Bergadano and D. Gunetti. An interactive system to learn functional logic programs. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, pages 1044–1049, Chambery, France, 1993.
2. F. Bergadano, D. Gunetti, and U. Trinchero. The difficulties of learning logic programs with cut. *Journal of Artificial Intelligence Research*, 1:91–107, 1993.
3. M. E. Califf. Learning the past tense of English verbs: An inductive logic programming approach. Unpublished project report, 1994.
4. R. Mike Cameron-Jones and J. Ross Quinlan. Efficient top-down induction of logic programs. *SIGART Bulletin*, 5(1):33–42, Jan 1994.
5. P. Clark and T. Niblett. The CN2 induction algorithm. *Machine Learning*, 3:261–284, 1989.

6. W.W. Cohen. Compiling prior knowledge into an explicit bias. In *Proceedings of the Ninth International Conference on Machine Learning*, pages 102–110, Aberdeen, Scotland, July 1992.
7. L. De Raedt and M. Bruynooghe. A theory of clausal discovery. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, pages 1058–1063, Chambery, France, 1993.
8. N. Lavrač and S. Džeroski, editors. *Inductive Logic Programming: Techniques and Applications*. Ellis Horwood, 1994.
9. C. X. Ling. Learning the past tense of English verbs: The symbolic pattern associator vs. connectionist models. *Journal of Artificial Intelligence Research*, 1:209–229, 1994.
10. C. X. Ling, 1995. Personal communication.
11. C. X. Ling and M. Marinov. Answering the connectionist challenge: A symbolic model of learning the past tense of English verbs. *Cognition*, 49(3):235–290, 1993.
12. B. MacWhinney and J. Leinbach. Implementations are not conceptualizations: Revising the verb model. *Cognition*, 40:291–296, 1991.
13. R. J. Mooney and M. E. Califf. Induction of first-order decision lists: Results on learning the past tense of English verbs. *Journal of Artificial Intelligence Research*, 3:1–24, 1995.
14. S. Muggleton and C. Feng. Efficient induction of logic programs. In *Proceedings of the First Conference on Algorithmic Learning Theory*, Tokyo, Japan, 1990. Ohmsha.
15. S. H. Muggleton, editor. *Inductive Logic Programming*. Academic Press, New York, NY, 1992.
16. M. Pazzani and D. Kibler. The utility of background knowledge in inductive learning. *Machine Learning*, 9:57–94, 1992.
17. J. R. Quinlan. Past tenses of verbs and first-order learning. In C. Zhang, J. Debenham, and D. Lukose, editors, *Proceedings of the Seventh Australian Joint Conference on Artificial Intelligence*, pages 13–20, Singapore, 1994. World Scientific.
18. J. R. Quinlan and R. M. Cameron-Jones. FOIL: A midterm report. In *Proceedings of the European Conference on Machine Learning*, pages 3–20, Vienna, 1993.
19. J.R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5(3):239–266, 1990.
20. B. L. Richards and R. J. Mooney. Automated refinement of first-order Horn-clause domain theories. *Machine Learning*, 19(2):95–131, 1995.
21. R. L . Rivest. Learning decision lists. *Machine Learning*, 2(3):229–246, 1987.
22. D. E. Rumelhart and J. McClelland. On learning the past tense of English verbs. In D. E. Rumelhart and J. L. McClelland, editors, *Parallel Distributed Processing, Vol. II*, pages 216–271. MIT Press, Cambridge, MA, 1986.
23. G. I. Webb and N. Brkič. Learning decision lists by prepending inferred rules. In *Proceedings of the Australian Workshop on Machine Learning and Hybrid Systems*, pages 6–10, Melbourne, Australia, 1993.
24. David Yarowsky. Decision lists for lexical ambiguity resolution: Application to accent restoration in Spanish and French. In *Proceedings of the 32nd Annual Meeting of the Association for Computational Linguistics*, pages 88–95, Las Cruces, NM, 1994.
25. J. M. Zelle and R. J. Mooney. Combining top-down and bottom-up methods in inductive logic programming. In *Proceedings of the Eleventh International Conference on Machine Learning*, pages 343–351, New Brunswick, NJ, July 1994.