

Learning the Past Tense of English Verbs:

An Extension to FOIDL

Ahmed Sameh, Tarek Radi, and Rana Mikhail

*Department of Computer Science,
The American University in Cairo,
P.O.Box 2511, Cairo, EGYPT
Email: sameh@aucegypt.edu*

Abstract

First-Order Induction of Decision Lists (FOIDL) is a new ILP method developed as a result of the failure of existing ILP methods when applied to the learning of past tense of English verbs. Using intentional knowledge representation, output completeness, and first-order decision lists, it can learn highly accurate rules for the past tense problem but with some drawbacks. In this paper, we present an extension to FOIDL that learns rules faster using a learning algorithm similar to version space learning.

1. Introduction

Inductive Logic programming (ILP) is one of the newest sub-fields in AI. It combines inductive methods with the power of first-order representations, concentrating in particular on the representation of theories as logic programs (Russell and Norving 1995). Over the last five years, it has become a major part of the research agenda in machine learning. ILP is a highly technical field, relying on some fairly advanced material from the study of computational logic.

ILP is a research area formed at the intersection of Machine Learning and Logic Programming. ILP systems develop predicate descriptions from examples and background knowledge. The examples, background knowledge and final descriptions are all described as logic programs. The theory of ILP is based on proof theory and model theory for the first order predicate calculus. ILP systems have been applied to various problem domains. Many applications benefit from the relational descriptions generated by the ILP systems. The ability of ILP systems to accommodate background knowledge is also fundamental. Some relationships learned in particular applications have been considered discoveries within those domains. 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 (e.g. Version Space).

The problem of learning the past tense of English verbs has been widely studied as an interesting problem in language acquisition. Previous research has applied both connectionist and symbolic methods to this problem. However, these efforts used specially-designed feature-based encoding that impose a fixed limit on the length of

words and fail to capture the generativity and position-independence of the underlying transformation. An example of the connectionist approach is Rumerlhart and McClelland's computational model of past tense learning, which was the first to use the classic perceptron algorithm and a special phonemic encoding of words (Mooney and Califf 1995, 1996). Their general goal was to show that connectionist models could account for interesting language-learning behavior that was previously thought to require explicit rules. This model was heavily criticized by opponents of the connectionist approach to language acquisition for the relatively poor results achieved and the heavily-engineered representations and training techniques employed. The best and most efficient method of tackling this learning problem has been the use of ILP.

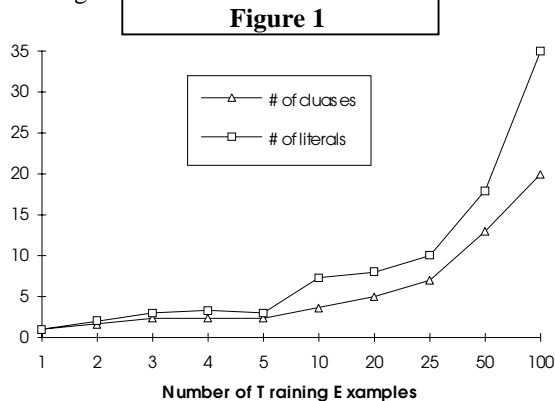
The current, most well-known, and successful ILP systems are GOLEM and FOIL (Muggleton and Feng 1990)(Quinlan 1990). These systems make assumptions that restrict their application and introduce significant limitations. A new ILP method called FOIDL (First-Order Induction of Decision Lists) overcomes these restrictions by incorporating new properties such as intentional knowledge representation, output completeness and first-order decision lists (Mooney and Califf 1997). FOIDL is related to FOIL and follows a top-down greedy specialization, guided by an information gain heuristic. The development of FOIDL was motivated by a failure observed when applying existing ILP methods to a particular problem, that of learning the past tense of English verbs. By overcoming FOIL's limitations, FOIDL is able to learn highly accurate rules for the past-tense problem using much fewer examples in its training set, compared to that, which was required by previous methods.

FOIDL has some drawbacks including the problem of local minimum, slow speed and consumption of large amount of memory (Mooney and Califf 1996).

2. Critique of FOIDL

In many cases, the FOIDL algorithm is able to learn accurate, compact, first-order decision lists for past tense. Those are, in fact, Prolog like programs that are ordered sets of clauses each ending in a cut (the !). Hence the most specific rules are placed first and then the next general. In order to check a given verb for its past tense,

one must check the produced rules in sequence, stopping at the first match of the representation of the verb in its present tense. FOIDL chooses those rules according to a specific 'gain' that it generates. It scans the literals



available in its training set, and finds the best literal (the one with largest information gain) that differentiates between this set of positive examples and the conflicting verbs (the negative examples). The rule with the highest gain is the most general rule, while the ones with the least gain are the most specific. Even though FOIDL requires significantly less representation engineering than all previous work in the area, yet due to the big amount of irregular verbs, 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 that a clause must cover. This was originally handled by terminating search and memorizing any remaining uncovered examples as specific exceptions at the top of the decision list, e.g. `past([a,r,i,s,e],[a,r,o,s,e])`. However, this can result in premature termination that prevents the algorithm from finding low-frequency regularities.

Despite its advantages, the use of intentional background knowledge in ILP incurs a significant performance cost, since examples must be continually reprocessed when testing alternative literals during specialization. FOIDL follows the Current-best-hypothesis search algorithm, similar to that described in (Russell and Norving 1995). This algorithm and many of its variants have been used in many machine learning systems, starting with Patrick Winston's "arch-learning" program. With a large number of instances and a large space, difficulties arise. Checking all the previous instances over again for conflicts on each modification is very expensive. It is also difficult to find good search heuristics, as repeated backtracking can consume a lot of time since the hypothesis space can be doubly exponentially large in its simplest case. Checking previous instances, accounts for most of the training time in FOIDL, and this is vivid when the conflicts are displayed during a FOIDL run. We have decided to give FOIDL some speed by changing it from a Current-best

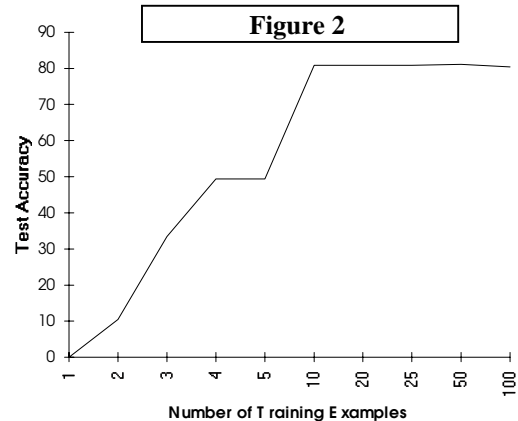
hypotheses algorithm to a least commitment algorithm still making use of the information content and gain heuristic.

3. Enhancing FOIDL

In this paper we introduce a method by which FOIDL was enhanced to run quicker and consume less memory space. First, we allow FOIDL to discover low frequency regularities. Second, we prevent it from reexamining old examples that are conflicting with the ones currently being evaluated during a FOIDL run. This reprocessing is the main reason for the large time delay and memory consumption in FOIDL. These two are our goals for enhancing FOIDL.

Experimentation with FOIDL has shown that the number of clauses and literals grows rapidly as we increase our training sample (see Figure 1). This can be explained by FOIDL's failure to find low-frequency regularities. It ends up adding exceptions such as `past([a,r,i,s,e],[a,r,o,s,e])` at the top of the list, and this is vivid from the rapid ascent of the curve in the above figure. It would be advantageous to learn such regularities rather than add rules as if FOIDL were a database.

Could this problem be solved simply by reducing the number of examples covered? It would generate less of such rules, but will these fewer rules be as effective? Verification of FOIDL has shown that the more examples the better the accuracy of classifying new and unseen verbs (see Figure 2). Thus, we need to use a large training set to increase accuracy and at the same time solve the problem of the unbearably slow generation of FOIDL rules. Backtracking arises because FOIDL's current-best-hypothesis approach has to choose a particular hypothesis as its best guess, based on a gain heuristic, generating a new rule that is placed before previously generated rules, and might classify examples covered by these old rules, thus changing their classification.



We have decided to use Version-Space learning as an alternative to FOIDL's learning. Version-space is a least-commitment search that tackles the above mentioned problems by avoiding backtracking and keeping only those hypotheses that are consistent with all the data so

far. Each new instance will either have no effect or will get rid of some of the hypotheses.

3.1 Version Space (VS) Learning

As our verification has shown, FOIDL experiences large increases in the number of clauses as the number of training samples increases due to its inability to capture low-frequency regularities. Besides looking for a solution to this weakness, we also want to attack the major lack of speed in FOIDL due to repeated reproofing of conflicts. A least-commitment search algorithm, if incorporated into FOIDL would tackle these two problems at once. One important property of this approach is that it is incremental and never goes back to reexamine already proved examples.

The Version Space algorithm is usually used for a small domain where the closed world assumption is feasible. In the past tense case, Mooney's experience with FOIL has shown that intentional background knowledge is more manageable than explicit negatives in a closed world. The typical version space algorithm also assumes a training set of positive and negative examples, with positive examples being used to update the S set by generalizing it, and negatives examples being used to update the G set by specializing it (Russell and Norving 1995). Positive examples are also used to remove non-matching G_i 's and negative examples are used to remove non-matching S_i 's. Making use of Mooney's experience that providing positive and negative examples to completely describe a closed world is a bad idea, we have decided to implement a modified version space learning algorithm by using intentional knowledge and positive examples only (see section 3.3).

Let us assume that the target decision list to be learnt is as follows:

```
past([g, r, a, b], [g, r, a, b, e, d])!
past(A, B) :- split(B, A, [d]), split(A, D, [e])!
past(A, B) :- split(B, A, [e, d]).
```

The point we must be aware of here is that these clauses are ordered from top to bottom. When we ask $Past(Q1, Q2)$ for example, the first clause this pair matches will be the rule to use, even though it might also match later clauses. This ordering constraint is what causes conflicting groups to appear, in which a verb already matching a clause near the bottom, also incorrectly matches some other clause placed higher in the list, causing incorrect classification. In our implementation, this has been avoided by the use of buckets, with the bucket size determining how general a clause is (see later).

3.2 Applying Standard VS

If we had used positive and negative examples in a normal version space training and without buckets, and used either the intentional knowledge (such as split) of exception

examples (such as $past(flog, flogged)$), then a typical S and G training would be as follows:

Initialization with a positive example:

$G = T$ (most general clause)

$S = past([jump], [jumped])$

After Addition of Another Positive example:

$G = T$ (no change)

$S = past([jump], [jumped]) \vee$

$past([employ], [employed])$

After Generalizing S :

$G = T$

$S = past(A, B) :- split(B, A, [e, d]).$

After First Negative example:

$G = T \wedge 'past([die], [died])$

$S = past(A, B) :- split(B, A, [e, d]) \wedge 'A=[die]$

As one can see, S has been generalized early, and thus if the "ed" clause was of low frequency, a generalization would be already introduced, and this is a big advantage over FOIDL. The disclosure of the clauses is generated in an ordered manner just as in decision lists, and the first clause from the left should match the topmost Prolog clause generated by FOIDL. But, as more negative examples similar to the $past([face], [faced])$ typical error are introduced, S and G become:

$G = T \wedge '(past(A,B):- split(B, A, [d]) \wedge$
 $split(A, D, [e]))$

$S = past(A, B) :- split(B, A, [e, d]) \wedge$
 $'(past(A,B):- split(B, A, [d]) \wedge$
 $split(A, D, [e]))$

It can be noticed that we are heading to a major problem now, since we are developing a rule that states there is no past tense to a verb that ends in "e" generated by the addition of a "d", which is incorrect. Thus, negative example learning is not useful, besides the possibility of an infinitely large negative example set in the domain which might overwhelm the learnt positive examples.

3.3 Adapted VS Learning

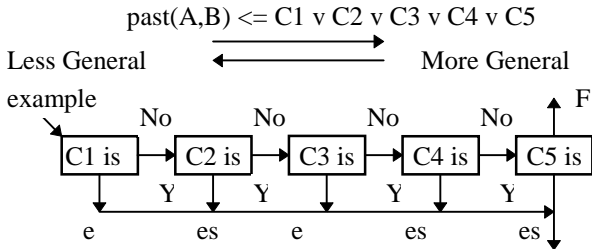
An alternative technique would be to reverse engineer the Prolog clauses into the ordered clause of disjunction required to be learnt and produced by our version space algorithm. The equivalent of the previous set of Prolog clauses in ordered First Order Predicate notation would be

```
past(A,B) <=
(A=[g, r, a, b] ^ B=[g, r, a, b, e, d]) v
(split(B, A, [d]) ^ split(A, D, [e])) v
(split(B, A, [e, d]))
```

This can be even simplified further to be represented in literals actually derived from intentional background knowledge (the split predicate).

$past(A,B) \Leftarrow$
 $(A=[g, r, a, b] \wedge B=[g, r, a, b, b, e, d]) \vee$
 $(split(?B, ?A, ?C) \wedge (= ?C (D)) \wedge split(?A, ?D, ?E) \wedge (= ?E (E))) \vee$
 $(split(?B, ?A, ?C) \wedge (= ?C (E D)))$

This in fact is our target clause to be learnt by the new version space algorithm. The question now is whether to impose an order or not. If we give each conjunction of literals above a name then we have



Just like Prolog statements generalize after each cut operator (!) from top to bottom, the clause here generalizes from left to right since they tend to cover more examples. The generalization here is a keyword allowing the possibility of setting G_1 to the rightmost conjunction (here C5) and S_1 to the leftmost conjunction (here C1), where G_1 is one of the hypotheses in G and S_1 is one of the hypotheses in S . The training for all these positive examples would be carried out by the following algorithm.

Whenever a hypotheses in G is specialized and a complete rule such as $(split(?B ?A ?C) \wedge (= ?C (ED)))$ is generated and an identical one is also generated by the generalization of S , then the two frontiers have met, and the concatenation of S and G will be the learnt clause equivalent to the desired Prolog clause. Thus S is producing rules starting from C1 and G is producing rules starting from C5, and both will meet when S and G generate C3 for example.

This technique is very well from the version space point of view, but implementation wise, we still run into the possible situation in which a verb may match two different clauses and would have to be reprovved. A solution to this is not impose any order on the clauses while generating them, and only assign an order later on depending on the percentage of training examples covered. Assuming a large training set, then the disregard of order while generation (to gain speed) and then regenerating order depending on how general the clause is can prove effective. This would require us to store with every clause all the verbs that it satisfies. This is the bucket concept, the details of which are described below.

3.4 The Bucket Concept

When rules are generated in G to specialize it, or in S to generalize it, we have the option to keep the instances that are covered by these rules linked to each one of them. This can help us know how many verbs are covered by that

rule after the generation of the rules comes to an end. This group of instances is referred to as a “bucket”. The size of the bucket can be used to enforce an order on the clauses generated. Those with a larger bucket are more general rules, since they actually cover more of the training samples. As examples are seen n at a time, each one either ends up in the bucket of an already existing clause or in a newly created clause that is specific to it, called a “solo” clause (since the bucket size is 1).

In the tracing of the algorithm provided below, we see G_1 specializing by changing its single clause (with 3 verbs in its bucket) to 2 clauses. The examples ([employ] [employed]) and ([jump] [jumped]) are removed from the original clause and are covered by the new specialized rule: $(Split(?B ?A ?C) \wedge (= ?C [ED]))$ while the non matching verb ([grab][grabbed]) is kept as it is in the old bucket of the clause $(Split(?B ?A ?C))$. Each of these two buckets must be kept for future specialization of G and in S for its future generalization. As more examples are seen, they are distributed onto the clauses that they match, and stored in the corresponding buckets.

The idea is that solo clauses such as $(past[grab],[grabbed])$ can be developed by the addition of other positive examples such as $([stab],[stabbed])$ to generate rules describing low frequency regularities, such as $(Split(?B ?A ?C) \wedge (= ?C (BED)))$.

As positive verbs are added to G , the clauses need to be specialized into more specific rules. This is where the stored reserved instances come into play. We use them to specialize and generate a new rule. The following table shows a predicted typical trace along with the buckets of each clause for one cycle of training. A training set increment of 3 verbs per cycle was used in this example, and show quick promising results for only two such cycles.

From this example, the Prolog rules obtained after vector space learning (ordered by bucket size in ascending order) would be

$Past(A B) :-$ $split(B, A, [bed]) !$
 $split(B, A, [d]) !$
 $split(B, A, [ed])$

Thus we have arrived at an ordered decision list just like FOIDL, but without the three drawbacks.

S_1	G_1
(employ employed) (grab grabbed) (jump jumped)	$Split(?B ?A ?C)$ (employ employed) (grab grabbed) (jump jumped)
(employ employed) (jump jumped) $Split(?B ?A ?C)$ $\wedge (= C [bed])$ (grab grabbed)	$Split(?B ?A ?C) \wedge (= C [ed])$ (employ employed) (jump jumped) $Split(?B ?A ?C)$ (grab grabbed)

(lie lied) (die died) (book booked) (employ employed) (jump jumped) split (?B ?A ?C) ^ (=C [bed]) (grab grabbed)	Split (?B ?A ?C) ^ (= C [ed]) (employ employed) (jump jumped) (book booked) Split (?B ?A ?C) (grab grabbed) (die died) (lie lied)
(book booked) (employ employed) (jump jumped) split (?B ?A ?C) ^ (=C [d]) (die died) (lie l) split (?B ?A ?C) ^ (=C [bed]) (grab grabbed)	split (?B ?A ?C) ^ (= C [ed]) → (employ employed) (jump jumped) (book booked) split (?B ?A ?C) ^ (=C [d]) → (die died) (lie lied) split (?B ?A ?C) → (grab grabbed)
Split (?B ?A ?C) ^ (= C [ed]) (book booked) (employ employed) (jump jumped) split (?B ?A ?C) ^ (=C [d]) (die died) (lie lied) split (?B ?A ?C) ^ (=C [bed]) (grab grabbed)	Split (?B ?A ?C) ^ (= C [ed]) (book booked) (employ employed) (jump jumped) split (?B ?A ?C) ^ (=C [d]) (die died) (lie lied) split (?B ?A ?C) ^ (=C [bed]) (grab grabbed)

4. Conclusion

FOIDL was created mainly to tackle the problem of learning to generate the past tense of English verbs. It can also be used to learn phonetics Arabic verbs, and to learn the integration and differentiation rules of calculus. In other domains, where intentional background knowledge is more of a graph with relations, rather than just two simple rules as used in the past tense domain, relational pathfinding can be used to avoid the local minimum and plateau problems of FOIDL. In this paper the ideas behind the enhancement of FOIDL for solving the past tense problem and detailed explanation of our new algorithm that runs faster and is capable of learning rules for low frequency regularities were described. Traces for this new algorithm has proved to be good and fast enough compared to the original FOIDL. We are currently working on modifying FOIDL to learn conjugation of Arabic verbs.

5. References

-Mooney, R.J. and 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).

-Mooney, R.J. and Califf. M.E. 1996. Learning the Past Tense of English Verbs Using Inductive Logic Programming, Aymbolic, Connectionis, and Statistical Approaches to Learning for Natural Language Processing, Springer Verlag.

-Mooney, R.J. and Califf. M.E. 1997. Advantages of Decision lists and implicit negatives in Inductive Logic Programming, Journal of Artificial Intelligence Research, 5 (2).

-Quinlan. J.R. 1990. Learning logical definitions from relations., Machine Learning, 5(3).

-Russell S., and Norvig P. 1995. Artificial Intelligence: A Modern Approach, Prentice Hall.

-Muggleton S. , and C. Feng C. 1990. Efficient Induction of Logic Programs, Proceedings of the First Conference on Algorithmic Learning Theory, Tokyo, Japan.

Implemented Algorithm

- 1) Initialize S with the first positive example and initialize G with the most general clause, e.g. $G_i = \text{split} (?B ?A ?C)$ adding that positive example to the bucket of this rule.
- 2) While examples still exist
 - a) For every n examples (where n is small compared to the training set) do
 - * Add *posex* to all S_i 's as a disjunction if no rule covers it otherwise, add it to the bucket of the rule that satisfies it
 - * Add *posex* to all G_i 's as a disjunction if no rule covers it otherwise, add it to the bucket of the rule that satisfies it
 - b) For each G_i , if there is one or more non-complete intentional disjunctions in G_i , try to specialize G_i . This intentional rule is generalized by the examples in its bucket into a new rule with a new bucket containing matching examples, while the old rule is kept with the remaining examples. The specialization should have a reasonably *large* gain. e.g. before specialization
 $G_i = (\text{split} (?B ?A ?C) \{ \text{bucket} = \text{employed, grabbed, jumped} \})$ after specialization
 $G_i = (\text{split} (?B ?A ?C) ^ { (=C (ED)) \{ \text{bucket} = \text{employ, jump} \} } \vee \text{split} (?B ?A ?C) \{ \text{bucket} = \text{grabbed} \})$
 - c) For Each S_i , if there are one or more non-intentional disjunctions, try to generalize S_i by replacing these disjuncts with a complete intentional rule with a reasonably small gain to capture less frequent regularities. The matching instances for this new rule are placed in the attached bucket. This way, already generated rules in G are not generated in S. If there are more than one generalizations, create a new S instance for each.
- 3) After all training examples have been covered, keep generalizing and specializing until an S_j matches a G_j .