Extraction of Meta-Knowledge to Restrict the Hypothesis Space for ILP Systems

Eric McCreath* and Arun Sharma†
Department of Artificial Intelligence
School of Computer Science and Engineering
The University of New South Wales
Sydney NSW 2052, Australia
Email: {ericm,arun}@cse.unsw.edu.au

Abstract

Many ILP systems, such as GOLEM, FOIL, and MIS, take advantage of user supplied meta-knowledge to restrict the hypothesis space. This meta-knowledge can be in the form of type information about arguments in the predicate being learned, or it can be information about whether a certain argument in the predicate is functionally dependent on the other arguments (supplied as mode information). This meta-knowledge is explicitly supplied to an ILP system in addition to the data.

The present paper argues that in many cases the meta- knowledge can be extracted directly from the raw data. Three algorithms are presented that learn type, mode, and symmetric meta-knowledge from data. These algorithms can be incorporated in existing ILP systems in the form of a preprocessor that obviates the need for a user to explicitly provide this information. In many cases, the algorithms can extract meta-knowledge that the user is either unaware of, but which information can be used by the ILP system to restrict the hypothesis space.

Empirical evidence for the effectiveness of these algorithms is demonstrated by incorporating them in FOIL. Finally, the biasing effects of the different kinds of meta-knowledge on the hypothesis space are discussed.

Keywords: Inductive Logic Programming, Machine Learning.

1 Introduction

A fundamental problem in the design of Inductive Logic Programming (ILP) systems is the explosion in the size of the hypothesis space. A system is provided with a set of true facts about a relation, $\mathcal{E}^+$, and a set of false facts, $\mathcal{E}^-$, and definition of some background predicates, $\mathcal{B}$, either in intensional or extensional form. The system is required to find an intensional definition of the relation that explains every true fact in $\mathcal{E}^+$ and does not explain any false fact in $\mathcal{E}^-$. The system makes use of a hypotheses space induced by the predicates in the background knowledge and the predicate being learned (to allow for recursive definitions). In general, the size of this hypotheses space turns out to be huge. In order to discover a correct hypothesis feasibly, this space must be structured and searched

*Supported by an Australian Postgraduate Research Award.
†Supported by a grant from the Australian Research Council.

efficiently. Many ILP systems provide extra information or meta-knowledge [6, 8] about the relation being learned. This information renders a large number of hypotheses in the search space incompatible, and the system can safely exclude them from the search space. This compacts or biases the search space and improves the efficiency of the ILP system.

Table 1 shows the biasing properties employed by different ILP systems. All these systems require that the user explicitly provide this additional information. In some cases, the information is necessary to the operation of the system; in others, the biasing information is optional, and when added improves the performance.

Several of these properties are implicitly contained in the raw data. An ILP system may be modified by attaching a preprocessor of the data that acts as a property generator, as shown in Figure 1. Preprocessing of the raw data can yield different kinds of meta-knowledge, making it unnecessary for the user to explicitly provide it. This yields systems that learn with less assistance from the user. Moreover, if the user is not aware of certain properties of the data, or is unable to provide them, then this technique allows the system to perform more efficiently, and in some cases succeed in learning where it was previously unable to learn. Additionally, the same preprocessor can be used to learn properties of the background knowledge. This further improves the efficiency of the ILP system by creating a strong bias on the hypothesis space.

	Mode	Type	Symmetry	Exclusive Opposite
MISST [6]	√	√	√	√
Markus [5]	√	√	√	
Progol [11]	√	√		
MIS [12]	√	√		
GOLEM [10]	√			
FOIL [3]	√	√		
FLIP [1]	√			
SIERES	√			
MOBAL [9]		√		

Table 1 : ILP Systems using meta knowledge

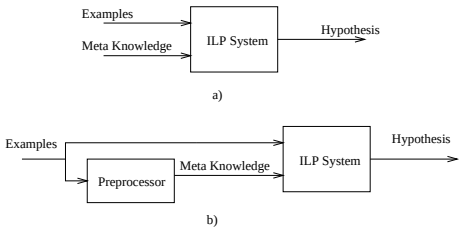

Figure 1 : (a) ILP System
(b) ILP System with property generator

Information such as rule models and predicate sets can not be learned by examining the examples of the relation. However, other properties, such as type, mode, symmetric, anti-symmetric, exclusive, transitive, and reflexive properties are implicitly present within the data. This paper describes a preprocessor that extracts type, functional dependency, and symmetry information from data. We briefly illustrate these properties next.

The relation for the length of a list has two arguments, the first is of type *List* and the second is of type *Integer*. This information may be described by a type declaration *Type*(*length*(*List, Integer*)). The relation for addition can be expressed as $add(t_1, t_2, t_3)$, where t_3 is the sum of t_1 and t_2. Now, t_3 is functionally dependent on t_1, t_2. That is, when t_1 and t_2 are bound, t_3 will also be bound. This information could be given to the ILP system in the form of a mode declaration $Mode(add(+, +, -))$. Additionally, *addition* is symmetric in its first two arguments; this information may be given to an ILP system in the form of a symmetry declaration $Symmetry(add(*, *, -))$.

The effectiveness of the preprocessor is demonstrated by combining it with FOIL [3]. We now proceed as follows. In Section 2, we illustrate the working of the algorithms for extracting type, mode, and symmetric information with the help of an example. In Section 3, we present and discuss experimental results demonstrating the speedup achieved as a result of the use of the preprocessor with FOIL. Finally, in Section 4, we briefly touch upon the usefulness of this approach in the design of an incremental

ILP system that dynamically modifies its hypothesis space based on feedback from a preprocessor that inspects the data for meta-knowledge.

2 An Illustration Of The Approach

We illustrate the proposed preprocessing technique in the context of a system that is required to learn an intensional definition of the predicate *merge*. This predicate, which takes three arguments, asserts that the third argument is a sorted list formed by merging the first two arguments that are also sorted lists. Suppose the system is also provided with background knowledge about the predicates *less*, *less-eq*, and *comp*. Predicates *less* and *less-eq* have the usual meaning over numbers; the predicate *comp* is true if the second and the third arguments are the *head* and *tail* of the list in the first argument. We assume that the background knowledge about these predicates is also provided extensionally as shown in Table 2. We now demonstrate how meta knowledge about type, mode, and symmetry can be extracted purely by examining the data. The three algorithms are sketched in Figure 2.

$\mathcal{E}^+$	$\mathcal{E}^-$	$\mathcal{B}$	
$merge([],[],[])$.	$merge([],[],[1])$.	$less(1,2)$.	$comp([2,1],2,[1])$.
$merge([],[1],[1])$.	$merge([1],[1],[])$.	$less\text{-}eq(1,2)$.	$comp([2,2],2,[2])$.
$merge([],[2],[2])$.	$merge([],[2],[1])$.	$less\text{-}eq(1,1)$.	$comp([1,1,1],1,[1,1])$.
$merge([],[1,2],[1,2])$.	$merge([],[1,2],[1,1,2])$.	$less\text{-}eq(2,2)$.	$comp([1,1,2],1,[1,2])$.
$merge([1],[],[1])$.	$merge([1],[2],[1])$.	$comp([1],1,[])$.	$comp([1,2,1],1,[2,1])$.
$merge([1],[1],[1,1])$.		$comp([2],2,[])$.	$\vdots$
$\vdots$	$\vdots$	$comp([1,1],1,[1])$.	$comp([2,2,1],2,[2,1])$.
$merge([1,2],[2],[1,2,2])$.	$merge([1,2],[1,2],[1,2])$.	$comp([1,2],1,[2])$.	$comp([2,2,2],2,[2,2])$.

Table 2: Raw information for learning *merge*

Below, the algorithms for functional dependency and symmetry extract correct information in the incremental (limiting) sense from complete data or under the closed world assumption. In batch mode, without the closed world assumption, the correctness of the information extracted depends on the "density" of the data. The algorithm for extracting type information, however, allways produces the the most specific type assignment implied by the given data.

2.1 Type

A type assignment that is consistent with the predicate being learned and the predicates in the background knowledge can be found in one pass of the positive examples, negative examples, and the background knowledge (i.e., $\mathcal{E}^+ \cup \mathcal{E}^- \cup \mathcal{B}$). This may proceed as follows. Initially, set the types associated with each argument of each predicate to a different type. This may be represented as:

$$Type(merge(\tau_1, \tau_2, \tau_3), less(\tau_4, \tau_5), less\text{-}eq(\tau_6, \tau_7), comp(\tau_8, \tau_9, \tau_{10})).$$

Then begin to scan each member of $\mathcal{E}^+ \cup \mathcal{E}^- \cup \mathcal{B}$. An inspection of the very first fact, $merge([],[],[])$, implies that each argument of *merge* must have the same type, as the set of types partitions the Herbrand universe and each argument contains the same element $[]$.[1] The type assignment will now be updated to:

$$Type(merge(\tau_1, \tau_1, \tau_1), less(\tau_4, \tau_5), less\text{-}eq(\tau_6, \tau_7), comp(\tau_8, \tau_9, \tau_{10})).$$

[1]Note, the *type* extraction algorithm as described does not distinguish between types, one of which is a subset of the other; for example, the set of natural numbers and the set of integers.

The scan will pass over $less(1,2)$ having no effect on the type assignment as there is no evidence to show that 1 and 2 are of the same type. When $less\text{-}eq(1,2)$ is reached, and the same elements are seen as in the previous fact $less(1,2)$, it can be inferred that both the predicates must have the same type vector. Thus, the type assignment will now be updated to

$$Type(merge(\tau_1,\tau_1,\tau_1), less(\tau_4,\tau_5), less\text{-}eq(\tau_4,\tau_5), comp(\tau_8,\tau_9,\tau_{10})).$$

This process is continued until $\mathcal{E}^+ \cup \mathcal{E}^- \cup \mathcal{B}$ is completely scanned. The final type assignment will be

$$Type(merge(\tau_1,\tau_1,\tau_1), less(\tau_4,\tau_4), less\text{-}eq(\tau_4,\tau_4), comp(\tau_1,\tau_4,\tau_1)).$$

This type meta-knowledge would restrict the search space by excluding clauses that are inconsistent with this type assignment. For example, a clause such as

$$merge(X_1,X_2,X_3) \leftarrow comp(X_1,X_4,X_3), less(X_4,X_1)$$

is inconsistent with the induced type assignment, and hence can be excluded from the search space, thereby improving the performance of the system. We would like to note that type meta knowledge is learned in the MOBAL [9] system.

2.2 Functional Dependency

Extraction of *mode* or functional dependency information proceeds as follows. First, the *merge* predicate is considered. Initially, it is assumed that each possible grouping of arguments in *merge* has functional dependency. This premise is then checked against the data. If the premise is false, then the data will show a counterexample, and the preprocessor will discard that possible functional dependency from its list of functional dependencies. More specifically, suppose it is being checked whether the first argument in *merge* is functionally dependent on the second argument (expressed as $Mode(merge(+,-,\ ))$). For this the *merge* predicate is scanned for any counterexamples. If a counterexample is found then such a functional dependency does not exist. In this case the first two facts provide a counterexample, as the first fact maps [] to [] and the second maps [] to [1]. No such conflict is found for $Mode(merge(+,+,-))$. By repeating this process for other predicates, *less*, *less-eq*, and *comp*, the remaining mode information may be discovered. For the *merge* example this information will be:

$$Mode(merge(+,+,-), merge(+,-,+), merge(-,+,+),$$
$$less(-,+), less(+,-), comp(+,-,-), comp(-,+,+)).$$

It should be noted that a functional dependency is induced in the *less* relation, even though no such dependency exists. This is due to the limited number of facts about *less* in the background knowledge.

Once again this information can be used to restrict the hypothesis search space. For example, $merge(X_1,X_2,X_3) \leftarrow comp(X_1,X_2,X_3)$ is not consistent with the functional dependency information induced above.

2.3 Symmetry

Meta knowledge about symmetry can also be derived from the data about the *merge* predicate. Clearly, *merge* is symmetric in its first two arguments. The generation of this information involves testing each pair of arguments for symmetry. This test requires two steps. First, a scan of the *merge* ground facts is made to show there are no examples contrary to merge being symmetric in the pair of arguments under consideration. When the last two arguments are being tested for symmetry the positive example $merge([1],[],[1])$ and the negative example $merge([1],[1],[])$ signal that no symmetry exists. Second, some positive confirmation that the symmetry exists is found. When the first two arguments

Algorithm 1 *Learning symmetric properties*

foreach *argument pair* i, j *in predicate* p **do**
  foreach $p(t_1, \ldots, t_i, \ldots, t_j, \ldots, t_n) \in \mathcal{E}^+$ **do**
    if $p(t_1, \ldots, t_j, \ldots, t_i, \ldots, t_n) \in \mathcal{E}^-$ **then**
      output
        p *is not symmetric in arguments* i, j
    fi
  od
  output p *is symmetric in arguments* i, j
od

Algorithm 2 *Learning type properties*

assign each predicate argument to a different type
assign each type to have an empty set of elements
foreach $c \in \mathcal{E}^+ \cup \mathcal{E}^- \cup \mathcal{B}$ **do**
  if c *signal the need to merge types* **then**
      merge types
  fi
  update type information
od
output *type information*

Algorithm 3 *Learning functional dependencies*

foreach *predicate name* $p \in \mathcal{E}^+$ **do**
  †**foreach** *mapping of sets of arguments*
        to an argument in p **do**
    reset_table(T)
    foreach *fact* $p(t_1, t_2, \ldots, t_n) \in \mathcal{E}^+$ **do**
      fun_input := *input terms of* $p(t_1, t_2, \ldots, t_n)$
      fun_output := *output term of* $p(t_1, t_2, \ldots, t_n)$
      if in_table(T, fun_input) **then**
        pre_output := get_value(T, fun_input)
        if fun_output $\neq$ pre_output **then**
          exit loop †
        fi
      else
        insert_element(T, fun_input, fun_output)
      fi
    od
    output *functional dependence exists in* p
  od
od

Figure 2: Algorithms for learning meta-knowledge

are being tested for symmetry the positive confirmation is first found in the examples $merge([],[1],[1])$ and $merge([1],[],[1])$.

It is easy to see that the information about symmetry can be used to restrict the hypotheses search space, as only one of the logically equivalent pair of clauses is considered. For example, the following two clauses are equivalent

$$merge(X_1, X_2, X_3) \leftarrow comp(X_1, X_4, X_5), merge(X_5, X_2, X_3)$$
$$merge(X_1, X_2, X_3) \leftarrow comp(X_1, X_4, X_5), merge(X_2, X_5, X_3)$$

because *merge* is symmetric in its first two arguments. So, there is no need to include both in the search space. Only MISST [6], LINUS [7], and mFOIL [4] use symmetry information.

3 Experimental Results

The meta-knowledge extraction algorithms have been incorporated into a preprocessor that generates meta-knowledge for Quinlan's FOIL[3]. FOIL was chosen because it is a stable ILP system and it makes use of *type* information and *key* information[2]. Two series of experiments were carried out. First, nine target relations from Bratko's Prolog book [2] were considered: *member*, *conc*, *del*, *sublist1*, *sublist2*, *permutation*, *add*, *gcd*, and *length*. These results, shown in Table 3, contrast learning with generated meta-knowledge and learning with no meta-knowledge. The second set of experiments considered five of the examples that came with FOIL: *member*, *ncm*, *hinton*, *ackermann*, and *qs44*. These experiments, summarized in Table 4, compare learning with preprocessor generated meta-knowledge, learning with user supplied meta-knowledge, and learning without any meta-knowledge. The experiments were carried out on a *DECstation 5000/33*. In the tables,

[2]The *key* information, as will be seen later, can be 'derived' from the mode information.

'$\surd$' denotes that FOIL returned a correct hypothesis, '$\times$' denotes that FOIL returned an incorrect hypothesis, '∞' denotes that FOIL didn't return ay hypothesis.

Relation	FOIL Processing Times		Speed up	Preprocessing
	with learned meta-knowledge	no meta-knowledge		
member	0.13$\surd$	0.20$\surd$	1.5	1.2
conc	0.87$\surd$	3.70$\surd$	4.6	3.4
del	0.22$\surd$	0.25$\surd$	1.1	1.5
sublist1	1.77$\times$	8.52$\surd$		13.5
sublist2	43.87$\surd$	47.98$\surd$	1.1	78.0
permutation	1.90$\surd$	8.65$\surd$	4.6	2.5
add	1.76$\surd$	1.77$\surd$	1.0	2.1
gcd	704.60$\surd$	718.32$\surd$	1.0	9.4
length	0.15$\surd$	2.60$\surd$	17.3	1.5

Table 3: Processing Times — Examples from Bratko's PROLOG book

Examples from Bratko's Book: Addition of preprocessor generated meta-knowledge to FOIL improved the performance of FOIL on seven out of nine relations as shown in Table 3. On some relations the improvement in performance was significant. For one of the predicates, *sublist1*, addition of generated meta-knowledge actually caused FOIL to learn an incorrect hypothesis. The reason for this is explained below.

FOIL receives type information, specifying the types used, the elements that constitute the types, and the type of each argument in each relation. FOIL also receives a set of *keys* for each relation. Each *key* is a string where the characters map to the arguments of the relation. If the character is '`#`' then the corresponding argument must be bound; otherwise the character is '`-`' and the corresponding argument may be bound or unbound. FOIL only adds literals to the growing clause that are consistent with a *key* binding. If no *keys* are given for a relation, then any binding is valid. This information forms a semantic constraint on the clauses generated by FOIL. Information about *key*s may be induced from the functional dependency (mode) information learned from the raw data. When the preprocessor is used on a relation that contains functional dependencies, a set of *keys* is induced. However, this relation may be intended for use in a non-determinate manner. This can be seen in the *sublist1* example; the clause to be induced is:

$$\text{sublist1}(S, L) \leftarrow \text{conc}(X, Y, L), \text{conc}(S, Z, Y)$$

Since the *key* information for the concatenation relation, *conc*, is `##-/#-#/-##`, the above clause can not be induced by FOIL as X and Y are not bound in the literal $conc(X, Y, L)$. This is a significant problem, as the hypothesis search space has been over reduced, thereby excluding the correct hypothesis. One way this may be overcome is to only use *key* information as an initial restriction; removing it when no sensible logic program is generated. In *sublist2* example, the background knowledge includes both concatenation and composition (*conc* and *comp*) predicates, and this problem does not arise. Hence, *sublist2* can be induced with the generated meta-knowledge.

Examples that came with FOIL: The experiments on the examples that came with FOIL turn out to be a bit more tricky (Table 4). For all these examples, the preprocessor is successful in inducing the intended *type* meta-knowledge. Also, it is able to induce a super set of the *key* meta-knowledge. One would expect that if FOIL could

Relation	FOIL Execution Times					Pre-processing Time
	raw data, no m.k.[a]	with learned m.k.	m.k. explicitly provided[b]	*key* m.k. only	*type* m.k. only	
member	0.32√	0.18√	0.18√	0.30√	0.22√	3.8
ncm	263.93×	100.02√	18.48√	18.35√	272.83×	10.7
hinton	4.52√	5.32√	4.52√	4.52√	4.53√	4.1
ackermann	53.50√	50.92×	54.81√	55.03√	∞	3.6
qs44	∞	11771.53×	125.20√	155.30√	47332.80×	39.0

[a] meta-knowledge

[b] These are the unmodified examples that come with FOIL

Table 4: Processing Times — FOIL's given examples

learn an example without any *key* meta-knowledge then FOIL would be able to learn the example, perhaps even learn faster, with a super set of the correct *key* meta-knowledge. Unfortunately, there are problems with this expectation. This can be seen in the context of *ackermann* example, where correct meta-knowledge prevents learning. The extra information misdirects FOIL's greedy search technique. In the *member* and *ncm* examples, using generated meta-knowledge improves the performance of FOIL over not using meta-knowledge. In fact, *ncm* can't be learned without this meta-knowledge. In *hinton*, introducing meta-knowledge has a slightly negative effect. FOIL could not learn *qs44* with generated meta-knowledge, as it does not create a sufficient restriction on the hypothesis search space. It should be noted that FOIL is able to learn *qs44* with user supplied meta-knowledge, but that is because after trial and error a user can just give the right bit of meta-knowledge that does not throw FOIL off. Hence, in such situations, the preprocessor can be used as a guide to the user in selecting the right bit of meta-knowledge.

The *ncm* (n-choose-m) example is interesting. First, only integers are used so introducing type meta-knowledge has no effect. Second, *key* information is necessary for FOIL to induce the correct clauses, when this type information is induced by the preprocessor, FOIL can induce the correct hypothesis. Third, the *key* information for decrement predicate is the only difference between the data with user supplied meta-knowledge and data with preprocessor generated meta-knowledge. The unmodified data only allows *dec* to be used as the *decrement* function:

```
*dec(M,M) #-
```

whereas when this meta-knowledge is induced by the preprocessor, `dec` may be used either as *decrement* function or as *increment* function

```
*dec(M,M) #-/-#.
```

Note how this small change has a significant impact on the hypothesis search space, and hence, a significant effect in the performance of the ILP system.

In the case of *ackermann*, *key* information misdirects the greedy search algorithm. Hence, FOIL is unable to learn a correct hypothesis. Even-though, the *key* information is consistent with the target hypothesis. Note the 'raw data no m.k.' and 'm.k. explicitly provided' input files are identical, as there is no key information in the *ackermenn* example provided with FOIL.

The preprocessor was implemented in the functional programming language Gofer. When it is re-implemented in C and included within FOIL, its effect on the overall system performance would be insignificant. So, preprocessing time is not included in the

calculation of speed-up.

4 Discussion

The preprocessor described in this paper has successfully demonstrated the usefulness of extracting three kinds of meta-knowledge from raw data. The capability of the preprocessor is being expanded to further extract anti-symmetric, exclusive, transitive, and reflexive properties from relational data. Due to lack of space, we postpone a disscusion of the theoretical properties of the algorithms to the full version of the paper.

The development of algorithms for extracting various kinds of meta-knowledge is aimed at design of an incremental ILP system that dynamically changes its hypothesis generation strategy based on the feedback provided from the meta-knowledge discovered. We illustrate this with the help of an example. Suppose, an ILP system is learning a predicate, and it begins with a conservative hypothesis generation heuristic by assuming that the predicate is symmetric with respect to certain arguments. In case, its assumption is true, the system continues to use this heuristic. However, if a new fact provides a counterexample to the symmetry assumption, the system widens its hypothesis generation heuristic. Such a system can be seen to make more prudent search of the hypothesis space.

ACKNOWLEDGEMENTS: We thank Ross Quinlan and the reviewers for helpful comments.

References

[1] F. Bergadano and D. Gunetti. An interactive system to learn functional logic programs. In *Machine Learning*, 1994.

[2] I. Bratko. *Prolog Programming for Artificial Intelligence*. Addison-Wesley, Workingham, 1990. second edition.

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

[4] S Džeroski. Handling noise in inductive logic programming. Master's thesis, Faculty of Electrical Engineering and Computer Science, University of Ljubljana, Slovenia, 1991.

[5] Marko Grobelnik. Induction of prolog programs with markus. Jožef Stefan Institute, 1994.

[6] M. Kirschenbaum and L. Sterling. Refinement strategies for inductive learning of simple prolog programs. In *Proc. of the 9th inter. conference on Artificial Intelligence*, 1991.

[7] N. Lavrač and S. Džeroski. *Inductive Logic Programming: Techniques and Applications*. Ellis Horwood (Simon and Schuster), 1994.

[8] Nada Lavrač and Sašo Džeroski. Background knowledge and declarative bias in inductive concept learning. In K. P. Jantke, editor, *Analogical and Inductive Inference*, Lecture Notes in Artificial Intelligence. Springer-Verlag, October 1992.

[9] K. Morik. Balanced cooperative modelling. In *Proc. First International Workshop on Multistrategy Learning*, pages 65–80, George Mason University, Fairfax, VA, 1991.

[10] S. Muggleton and C. Feng. Efficient induction of logic programs. In *Proceedings of the First Conference on Algorithmic Learning Theory, Tokyo*, pages 368–381. Ohmsa Publishers, 1990. Reprinted by Ohmsa Springer-Verlag.

[11] Stephen Muggleton. Inverse entailment and progol, May 1995.

[12] E. Shapiro. Inductive inference of theories from facts. Technical Report 192, Computer Science Department, Yale University, 1981.