Appears in the Proceedings of ICML 2000

Learning Horn Expressions with LogAn-H

Roni Khardon RONI@DCS.ED.AC.UK
Division of Informatics, University of Edinburgh, Edinburgh, EH9 3JZ Scotland

Abstract

The paper introduces LogAn-H — a system for learning first-order function-free Horn expressions from interpretations. The system is based on an interactive algorithm (that asks questions) that was proved correct in previous work. The current paper shows how the algorithm can be implemented in a practical system by reducing some inefficiencies. Moreover, the paper introduces a new algorithm based on it that avoids interaction and learns from examples only. We describe qualitative and quantitative experiments in several domains. The experiments demonstrate that the system can deal with varied problems, large amounts of data and large hypotheses, and that it achieves good classification accuracy.

1. Introduction

Work in Inductive Logic Programming (ILP) has established a core set of methods and systems that proved useful in a variety of applications (Muggleton & De Raedt, 1994). Theoretical results, however, identified strong limits to learnability when only examples are used. One recent strand of theoretical work has shown that larger classes of expressions are learnable if the learner is allowed to ask questions (Arimura, 1997; Reddy & Tadepalli, 1998; Rao & Sattar, 1998; Khardon, 1999b; Khardon, 1999a). The system LogAn-H (Logical Analysis for Horn expressions) implements an interactive algorithm based on Khardon (1999a) as well as an algorithm for learning from examples. There are no restrictions other than the requirement that expressions are function free: the system can learn multi-clause programs with recursive clauses, features that are not handled by many ILP systems.

In the interactive mode the system poses questions to the user, either about the correctness of its hypothesis or about the label of hypothetical examples. The system will converge on an expression that agrees with the user's responses. In the second mode the system acts as a batch system for learning from examples. The main idea here is to "simulate" the user by appealing to the data set. The batch mode can be alternatively interpreted as using a certain form of specific to general search over multi-clause hypotheses. Thus the contribution here can also be seen as introducing a new form of refinement search. In both cases we discuss algorithmic devices that contribute to efficiency.

We have experimented with both modes of the system and challenging learning problems. We describe qualitative experiments illustrating the performance of the system on list manipulation procedures (15-clause program including standard procedures) and a toy grammar learning problem. The grammar problem introduces the use of background information to qualify grammar rules that may be of independent interest for other settings. We also describe quantitative experiments with examples drawn for Bongard problems (De Raedt & Van Laer, 1995). These experiments study the performance of the system by varying the size of programs and example set as well as comparing it to other systems. In all these good performance is demonstrated by LogAn-H .

2. Learning from Interpretations

Learning from interpretations has seen growing interest in recent years (De Raedt & Dzeroski, 1994; De Raedt & Van Laer, 1995; Blockeel & De Raedt, 1998). Unlike the standard ILP setting, where examples are atoms, examples in this framework are interpretations of the underlying first order language (i.e. first order structures). We introduce the setup informally through examples. Formal definitions can be found in Khardon (1999a). The task is to learn a universally quantified function-free Horn expression, that is, a conjunction of Horn clauses. The learning problem involves a finite set of predicates whose names and arities are fixed in advance. In the examples that follow we assume

two predicates $p()$ and $q()$ both of arity 2. For example, $c_1 = \forall x_1, \forall x_2, \forall x_3, [p(x_1, x_2) \wedge p(x_2, x_3) \rightarrow p(x_1, x_3)]$ is a clause in the language. An example is an *interpretation* listing a domain of elements and the extension of predicates over them. The example e_1=$([1,2,3],[[p(1,2),p(2,3),p(3,1)],[q(1,3)]])$ describes an interpretation with domain $[1,2,3]$ and where the four atoms listed are true in the interpretation and other atoms are false. To improve readability, we separated the $p()$ atoms from $q()$ atoms but this has no special significance. The size of an example is the number of atoms true in it, so that $size(e_1)$=4. The example e_1 falsifies the clause above (substitute $\{1/x_1, 2/x_2, 3/x_3\}$), so it is a negative example. On the other hand e_2=$([a,b,c,d],[[p(a,b),p(b,c),p(a,c),p(a,d)],[q(a,c)]])$ is a positive example. We use standard notation $e_1 \not\models c_1$ and $e_2 \models c_1$ for these facts. The *batch algorithm* performs the standard supervised learning task: given a set of positive and negative examples it produces a Horn expression as its output.

The interactive mode captures Angluin's (1988) model of learning from equivalence queries (EQ) and membership queries (MQ) where we assume that a *target expression* – the true expression classifying the examples – exists and denote it by T. An EQ asks about the correctness of the current hypothesis. With an EQ the learner presents a hypothesis H (a Horn expression) and, in case H is not correct, receives a counter example, positive for T and negative for H or vice versa. With a MQ the learner presents an example (an interpretation) and is told in reply whether it is positive or negative for T. The learner asks such queries until H is equivalent to T and the answer to EQ is "yes".

3. The Algorithms

3.1 The Interactive Algorithm

The interactive algorithm is basically the algorithm *A2* from Khardon (1999a). We first describe some of the basic operations of the algorithm. Here again we illustrate concepts informally through examples.

Dropping domain elements: When dropping a domain element (object) we remove the element from the domain and all atoms referring to it from the extensions of predicates. Thus if we remove 2 from e_1 we get $e_1' = ([1,3],[[p(3,1)],[q(1,3)]])$.

Pairing: The pairing operation combines two examples to create a new example. When pairing two examples we utilise an injective mapping from the smaller domain to the larger one. The mapping $\{1/a, 2/b, 3/c\}$ can be used to pair e_1, e_2. We can then rename objects in one of the interpretations and form the intersection of the atoms in extensions of predicates. Using the above we get the pairing $e_p = ([1,2,3],[[p(1,2),p(2,3)],[q(1,3)]])$. Notice that $p(3,1)$ in e_1 has no equivalent $p(c,a)$ in e_2 and is therefore dropped. Similarly, $p(a,c), p(a,d)$ in e_2 do not have equivalent atoms in e_1 and are dropped. Clearly, any two examples have many possible pairings, one for each injective mapping of domain elements. Note that we can use domain element names from either of the interpretations in the pairing.

Candidate clauses: For an interpretation I, *rel-ant*(I) is a conjunction of positive literals obtained by listing all atoms true in I and replacing each object in I with a distinct variable. So $rel\text{-}ant(e_1) = p(x_1,x_2) \wedge p(x_2,x_3) \wedge p(x_3,x_1) \wedge q(x_1,x_3)$. Let X be the set of variables corresponding to the domain of I in this transformation. The set of candidate clauses *rel-cands*(I) includes clauses of the form $(rel\text{-}ant(I) \rightarrow p(Y))$, where p is a predicate, Y is a tuple of variables from X of the appropriate arity and $p(Y)$ is not in *rel-ant*(I).[1] For example, *rel-cands*(e_1) includes among others the clauses $[p(x_1,x_2) \wedge p(x_2,x_3) \wedge p(x_3,x_1) \wedge q(x_1,x_3) \rightarrow p(x_2,x_2)]$, and $[p(x_1,x_2) \wedge p(x_2,x_3) \wedge p(x_3,x_1) \wedge q(x_1,x_3) \rightarrow q(x_3,x_1)]$, where all variables are universally quantified. We refer to the variabilisation as *variabilise*(α) where according to context α may be just the antecedent or both antecedent and consequent.

Algorithm overview: Intuitively, the algorithm generates clauses from examples by using *rel-cands*(). It then uses dropping of domain elements and pairing in order to get rid of irrelevant parts of these clauses.

The algorithm maintains a sequence S of negative interpretations. These are initially used to generate the hypothesis by using *rel-cands*(s_i) for each $s_i \in S$. Once the hypothesis H is formed the algorithm asks an equivalence question: is H the correct expression? This is the main iteration structure which is repeated until the answer "yes" is obtained. On a positive counter example (e is positive but $e \not\models H$), wrong clauses (s.t. $e \not\models C$) are removed from H.

On a negative counter example (e is negative but $e \models H$), the algorithm first minimises the number of objects in the counter example as follows: The algorithm drops a domain element and if the resulting hypothetical example is negative (it asks a MQ) it continues with the smaller example; otherwise it retains

[1] The algorithm in Khardon (1999a) allows for empty consequents as well. The system can support this indirectly by explicitly representing a predicate *false* with arity 0 (which is false in all interpretations).

Table 1. The Interactive Algorithm

1. Initialise S to be the empty sequence.
2. Repeat until $H \equiv T$:
   (a) Let $H = variabilise(S)$.
   (b) Ask an equivalence query to get a counter example in case $H \not\equiv T$.
   (c) On a positive counter example I (s.t. $I \models T$) remove wrong clauses (s.t. $I \not\models C$) from H.
   (d) On a negative counter example I (s.t. $I \not\models T$):
      i. Minimise the number of objects in I using greedy procedure as described in text.
      ii. For $i = 1$ to m
      (where $S = ([s_1, c_1], \ldots, [s_m, c_m])$)
      For every pairing J of s_i and I
      If J's size is smaller than s_i's size and $J \not\models T$ (ask membership query) then
      A. Replace $[s_i, c_i]$ with $[J, c_i \cup (s_i \setminus J)]$.
      B. Quit loop (Go to Step 2a)
      iii. If no s_i was replaced then add $[I, rel\text{-}cands(I)]$ as the last element of S.

the previous example. This is performed once for each domain element. The minimisation ensures that the domain of examples is not unnecessarily large. The algorithm then tries to find a "useful" pairing of this counter example with one of the interpretations s_i in S. A useful pairing is a *negative* example that is also *smaller* than s_i. The search is done by trying all possible matchings of objects in the corresponding interpretations and asking membership queries. The interpretations in S are tested in increasing order and the *first* s_i for which this happens is replaced with the resulting pairing. The size constraint guarantees that measurable progress is made with each replacement. In case no such pairing is found for any of the s_i, the minimised counter example I is added to S as the *last* element. Note that the order of elements in S is used in choosing the first s_i to be replaced, and in adding the counter example as the last element. These are crucial for the correctness of the algorithm.

Some finer details: The description above is wasteful in the number of membership queries performed since the operation $rel\text{-}cands(s_i)$ generates consequents for s_i from scratch in each iteration. Instead, we attach to each s_i a set of possible consequents c_i. When s_i is first added to S, c_i is computed as in $rel\text{-}cands(s_i)$ but without variabilisation. Thereafter, however, an incremental update to c_i can be performed. Let $[s_i, c_i]$ be the situation before the pairing, and s_i' the pairing with a counter example. Then assuming s_i' uses object names from s_i we set $c_i' = c_i \cup (s_i \setminus s_i')$. Candidate clauses are generated from $[s_i', c_i']$ by variabilising consequents only from c_i'. For example, assume that s_1 is e_1 from above and that c_1 is $[p(1,1), q(3,1)]$. If the pairing with e_2 from above is performed (i.e. found to be negative by the membership query) then $s_1' = e_p = ([1,2,3], [[p(1,2), p(2,3)], [q(1,3)]])$ and $c_1' = [p(1,1), q(3,1), p(3,1)]$. The clauses generated by $[s_1', c_1']$ are $variabilise([s_1', c_1']) = [\alpha \rightarrow p(x_1, x_1)] \wedge [\alpha \rightarrow q(x_3, x_1)] \wedge [\alpha \rightarrow p(x_3, x_1)]$ where $\alpha = [p(x_1, x_2) \wedge p(x_2, x_3) \wedge q(x_1, x_3)]$.

The algorithm is summarised in Table 1 where T denotes the target expression. The reader is referred to Khardon (1999a) for proof of correctness and justification of the various steps. Intuitively, the idea is that two negative examples that match the same clause of the target will have a pairing that preserves the structure of that clause. If we find such a pairing then every atom deleted from the original interpretations is not needed in the target. The algorithm tries to identify such useful pairings by checking that the resulting examples are negative and small.

3.2 The Batch Algorithm

The batch algorithm is based on the observation[2] that we can answer the interactive algorithm's questions using a given set of examples E. Simulating equivalence queries is easy: given H we evaluate H on all examples in E. If it misclassifies any example we have found a counter example. Otherwise we found a consistent hypothesis. For membership queries we use:

Lemma 3.1 (Khardon, 1999a) *Let T be a function free Horn expression and I an interpretation over the same alphabet. Then $I \not\models T$ if and only if for some $C \in rel\text{-}cands(I)$, $T \models C$.*

Now, we can simulate the test $T \models C$ by evaluating C on all positive examples in E. If we find a positive example e with $e \not\models C$ then $T \not\models C$. Otherwise we assume that $T \models C$ and hence that $I \not\models T$.

A straightforward application will use the lemma directly whenever the algorithm asks a membership query (this is *A4* of Khardon, 1999a). However, a more careful look reveals that queries will be repeated many times. Moreover, with large data sets, it is useful to reduce the number of passes over the data. We therefore optimise the procedure as described below.

[2] Similar observations were made by Kautz et al. (1995) and Dechter and Pearl (1992) with respect to the propositional algorithm of Angluin et al. (1992).

Note that there is a 1-1 correspondence between the ground clauses in $[s_i, c_i]$ and their variabilised versions. From now on we just use $[s_i, c_i]$ with the implicit understanding that the appropriate version is used.

The one-pass procedure: This procedure is used as a basic operation of the improved algorithm. Given a set $[s, c]$ *one-pass* iteratively tests clauses in $[s, c]$ against all examples in E. For each example e it removes *all* wrong consequents identified by e from c. If c is empty at any point then the process stops and $[s, \emptyset]$ is returned.

Minimisation: The minimisation procedure on example e first generates $[e, rel\text{-}cands(e)]$ and runs *one-pass* on it to get $[s, c]$. By the lemma above we are guaranteed that c is not empty. It then iteratively tries to drop domain elements. In each iteration it drops an object to get $[s', c']$, runs *one-pass* on $[s', c']$ to get $[s'', c'']$. If c'' is not empty it continues with it to the next iteration (assigning $[s, c] \leftarrow [s'', c'']$); otherwise it continues with $[s, c]$. This is done once for every object. The final result of this process is $[s, c]$ in which all consequents are correct with respect to E.

Pairing: When pairing, the interactive algorithm produces a candidate $[J, c_i \cup (s_i \setminus J)]$ if J is negative and smaller than s_i. When simulating from examples, we first check that J satisfies the size requirement. Then we run *one-pass* on $[J, c_i \cup (s_i \setminus J)]$ to get $[s, c]$. If c is not empty then J is negative and we can replace $[s_i, c_i]$ with $[s, c]$. Otherwise, the pairing fails.

The above simulation avoids repeated queries that result from direct use of the lemma as well as guaranteeing that no positive counter examples are ever found (since they are used before clauses are put into the hypothesis). This simplifies the description of the algorithm which is summarised in Table 2.

3.3 Other Algorithmic Issues

The algorithms described above are likely to repeat some questions. In particular, when pairing the same s_i with a new example we may produce an example that has been seen before. Therefore, the system uses a caching mechanism for examples. The caching is syntactic, that is, we check for identical interpretations including domain element names. This ensures that caching is not too costly and works well in practice since the pairing procedure uses the same names from previous s_i values. The batch algorithm also caches interpretations rather than clauses or clause sets (i.e only the s_i part of $[s_i, c_i]$). This allows us to avoid more calls of *one-pass* as the same s_i may be queried with different c_i sets.

Table 2. The Batch Algorithm

1. Initialise S to be the empty sequence.
2. Repeat until H is correct on all examples in E.
   (a) Let $H = variabilise(S)$.
   (b) If H misclassifies I (I is negative but $I \models H$):
      i. Let $[s, c] = one\text{-}pass([I, rel\text{-}cands(I)])$.
      ii. Minimise the number of objects in $[s, c]$ using *one-pass* as described in text.
      iii. For $i = 1$ to m
      (where $S = ([s_1, c_1], \ldots, [s_m, c_m])$)
      For every pairing J of s_i and I
      If J's size is smaller than s_i's size then
      let $[s, c] = one\text{-}pass([J, c_i \cup (s_i \setminus J)])$.
      If c is not empty then
      A. Replace $[s_i, c_i]$ with $[s, c]$.
      B. Quit loop (Go to Step 2a)
      iv. If no s_i was replaced then add $[s, c]$ as the last element of S.

The system also includes an optimisation that reduces the number of pairings that are tested without compromising correctness. Recall that the algorithm has to test all injective mappings between the domains of two interpretations. We say that a mapping is *live* if every paired 2-object appears in the extension of at least one atom in the pairing. One can show that if the target expression is range restricted (i.e. all variables in the consequent appear in the antecedent) then testing live mappings is sufficient. For example, the pairing e_p from above is live. On the other hand, the mapping $\{1/b, 2/a, 3/c\}$ can be used to pair e_1, e_2 resulting in the pairing $([1, 2, 3], [p(2, 3)])$ where the domain element 1 is not in the extension of any predicate. So, this pairing can be ignored. The system includes this optimisation as an option and it was used in some of the experiments below.

Another feature used is the ability to restrict the set of predicates allowed in consequents to reduce the number of irrelevant queries. The system can also support background knowledge in a natural way. Essentially, if some of the target is known then the system can use this as part of the hypothesis reducing complexity as there is no need to learn this part. This is not used in the experiments below so the details are omitted.

3.4 Discussion

The batch algorithm simulates the interactive one by drawing counter examples from E. A subtle aspect concerning complexity is raised by this procedure. As-

sume the target expression T exists and that it uses at most k variables in each clause. It is shown in (Khardon, 1999a) that in such a case the minimisation procedure outputs an interpretation with at most k domain elements. As a result any clause C produced by the interactive algorithm has at most k variables which in turn guarantees that we can test whether $I \models C$ in time $O(n^k)$ where I has n domain elements. Unfortunately, we must simulate membership queries for the minimisation process itself. When doing this we generate clauses with as many variables as there are domain elements in I, which may lead to the complexity of *one-pass* growing with n^n if examples typically have n domain elements. This is a serious consideration that might slow down the batch system in practice. In the experiments we sorted the examples presenting smaller ones first thus improving the running time.

Several other systems use a covering method where the algorithm adds one clause at a time to the hypothesis and each clause is constructed by a general to specific refinement search (Quinlan, 1990; Muggleton, 1995; De Raedt & Van Laer, 1995). Refinement operations normally use the smallest possible refinement steps, e.g. adding a literal or specialising a term. On the other hand our system performs the search over complete hypotheses (cf. Bratko, 1999). It starts with a most specific clause and generalises it in two ways. First, dropping of objects effectively generalises by dropping a group of literals together. Thus it can be seen as a large refinement step compared with dropping a single literal. Second, we generalise the hypothesis by using pairing. This also offers large refinement steps, but this time with the added benefit of being directed by the examples. To summarise, the algorithm performs large (specific to general) refinement steps directed by the examples over multi-clause hypotheses.

Finally, it is easy to see that the batch algorithm works correctly if the data set is "complete". In fact, it suffices that all sub-models of negative examples that are positive have a positive "representative" in the data (where I' is a representative of I if it is isomorphic to I). It is not too hard to show that this guarantees that all answers to membership queries are correct, implying that the algorithm will work correctly and that the bounds proved in Khardon (1999a) apply. If this does not hold then the algorithm will still find a consistent hypothesis if one exists but the hypothesis may be large (i.e. it may over-fit the data). While this may be a strong condition to require or test, it can can serve as a rough guide for data preparation and evaluating whether the algorithm is likely to work well for a given application.

4. Experiments

The system is implemented in Sicstus Prolog and all experiments with LogAn-H were run on a Linux platform using a Pentium 2/366MHz processor. We first describe two experiments that illustrate the behaviour of the system on list manipulation programs and a toy grammar learning problem. The intention is to exemplify the scope and type of applications that may be appropriate. We then describe a set of quantitative experiments in the Bongard domain that illustrate applicability in terms of scalability and accuracy of the learned hypotheses as well as providing a comparison with other systems.

4.1 Learning List Manipulation Programs

To facilitate experiments with the interactive mode of the system, we have incorporated an "automatic-user mode" where (another part of) the system is told what is the expression to be learned (i.e. T). The algorithm's questions are answered automatically using T. Since implication for function-free Horn expressions is decidable this can be done reliably. In particular, Lemma 13 in Khardon (1999a) provides an algorithm that can test implication and construct counter examples to equivalence queries. Membership queries can be evaluated on T. We note that this setup is generous to our system since counter examples produced by the implemented "automatic user mode" are in some sense the smallest possible counter examples.

Using this mode we ran the interactive algorithm to learn a 15-clause program including a collection of standard list manipulation procedures. The program includes: 2 clauses defining $list(L)$, 2 clauses defining $member(I, L)$, 2 clauses defining $append(L1, L2, L3)$, 2 clauses defining $reverse(L1, L2)$, 3 clauses defining $delete(L1, I, L2)$, 3 clauses defining $replace(L1, I1, I2, L2)$, and 1 clause defining $insert(L1, I, L2)$ (via $delete()$ and $cons()$). The definitions have been appropriately flattened so as to use function free expressions. All these clauses for all predicates are learned simultaneously. This formalisation is not range-restricted so the option reducing the number of pairings was not used here. The system required 35 equivalence queries, 455 membership queries and about 8 minutes to recover the set of clauses exactly.

We have also run the batch algorithm on this problem using the examples from the interactive run. Note that if counter examples are produced in the same order then we should get the same behaviour as in the interactive mode, however, this was not done. The system found an expression consistent with all the examples

in about 7 minutes, using 18 equivalences queries, and 255 calls to *one-pass* . The hypothesis found included all the clauses in the intended program plus 3 wrong clauses. The set of examples was not rich enough to contradict these (no positive example satisfies their antecedent). The interesting point here is that despite this fact, all correct clauses were found. This happened in experiments in other domains as well.

4.2 A Toy Grammar Learning Problem

Consider the problem of learning a grammar for English from examples of parsed sentences. In principle, this can be done by learning a Prolog parsing program. In order to test this idea we generated examples for the following grammar, which is a small modification of ones described by Pereira and Shieber (1987).

```
s(H2,P0,P)   :- np(H1,P0,P1),vp(H2,P1,P),
                number(H1,X),number(H2,X).
np(H,P0,P)   :- det(HD,P0,P1),n(H,P1, P),
                number(HD,X),number(H,X).
np(H,P0,P)   :- det(HD,P0,P1),n(H,P1, P2),
                rel(H2,P2,P),number(H,X),
                number(H2,X),number(HD,X).
np(H,P0,P)   :- pn(H,P0,P).
vp(H,P0,P)   :- tv(H,P0,P1),np(_,P1,P).
vp(H,P0,P)   :- iv(H,P0,P).
rel(H,P0,P)  :- relw(_,P0,P1),vp(H,P1,P,L1).
```

Note that the program corresponds to a chart parser where we identify a "location parameter" at beginning and end of a sentence as well as between every two words, and true atoms correspond to edges in the chart parse. Atoms also carry "head" information for each phrase that is used to decide on number agreement. The grammar allows for recursive sub-phrases. A positive example for this program is a sentence together with its chart parse, i.e. all atoms that are true for this sentence.

We note that the grammar learning problem as formalised here is interesting only if external properties (such as $number()$) are used. Otherwise, one can read-off the grammar rules from the structure of the parse tree. It is precisely because such information is important for the grammar but normally not supplied in parsed corpora that this setup may be useful. Of course, it is not always known which properties are the ones crucial for correctness of the rules as this implies that the grammar is fully specified. In order to model this aspect in our toy problem we included all the relations above and in addition a spurious property of words *confprop*() (confuse property) whose values were selected arbitrarily. For example a chart parse of the sentence "sara writes a program that runs" is represented using the positive example:

```
([sara,writes,a,program,that,runs,
  alpha,beta,singular,0,1,2,3,4,5,6],
 [[s(writes,0,4),s(writes,0,6)],
  [np(sara,0,1),np(program,2,4),np(program,2,6)],
  [vp(writes,1,4),vp(runs,5,6),vp(writes,1,6)],
  [rel(runs,4,6)],  [pn(sara,0,1)],
  [n(program,3,4)], [iv(runs,5,6)],
  [tv(writes,1,2)], [det(a,2,3)],
  [relw(that,4,5)],
  [number(runs,singular),number(program,singular),
   number(a,singular),number(writes,singular),
   number(sara,singular)],
  [confprop(runs,beta),confprop(program,alpha),
   confprop(writes,beta),confprop(sara,alpha)]]).
```

Note also that one can generate negative examples from the above by removing implied atoms (of $s(), np(), vp(), rel()$) from the interpretation. It may be worth emphasising here that negative examples are not non-grammatical sentences but rather partially parsed strings. Similarly, a non-grammatical sequence of words can contribute a positive example if all parse information for it is included. For example "joe joe" can contribute the positive example

```
([joe,0,1,2,alpha,singular],[pn(joe,0,1),
  pn(joe,1,2),np(joe,0,1),np(joe,1,2),
  number(joe,singular),confprop(joe,alpha)])
```

or a negative one such as

```
([joe,0,1,2,alpha,singular],[pn(joe,0,1),
  pn(joe,1,2),np(joe,0,1),
  number(joe,singular),confprop(joe,alpha)]).
```

A simple analysis of the learning algorithm and problem setup shows that we must use non-grammatical sentences as well as grammatical ones and we have done this in the experiments. A final issue to consider is that for the batch algorithm to work correctly we need to include positive sub-structures in the data set. While it is not possible to take all sub-structures we approximated this by taking all *continuous* substrings. Given a sentence with k words, we generated from it all $O(k^2)$ substrings, and from each we generated positive and negative examples as described above.

We ran two experiments with this setup. In the first we hand picked 11 grammatical sentences and 8 non-grammatical ones that "exercise" all rules in the grammar. With the arrangement above this produced 133 positive examples and 351 negative examples. The batch algorithm ran for about 1 minute recovering an exact copy of the grammar, making 12 equivalence queries and 88 calls to *one-pass*.

In the second experiment we produced all grammatical sentences with "limited depth" restricting arguments of $tv()$ to be $pn()$ and allowing only $iv()$ in relative clauses. This was done simply in order to restrict the number of sentences, resulting in 120 sentences and

386 sub-sentences. We generated 614 additional random strings to get 1000 base strings. These together generated 1000 positive examples and 2397 negative examples. With this setup the batch algorithm found a hypothesis consistent with all the examples, in about 2 minutes, using 12 equivalence queries and 81 calls to *one-pass*. The hypothesis included all correct grammar rules plus 2 wrong rules, yielding a similar behaviour to the one we had for the list experiments.

The experiments demonstrate that it is possible to apply our algorithms to problems of this scale even though the setup and source of examples is not clear from the outset. They also show that it may be possible to apply our system (or other ILP systems) to the NLP problem but that data preparation will be an important issue in such an application.

4.3 Bongard Problems

Our quantitative experiments were done with artificial data akin to Bongard problems, as illustrated previously in the ICL system (De Raedt & Van Laer, 1995). In this domain an example is a "picture" composed of objects of various shapes (triangle, circle or square), triangles have a configuration (up or down) and each object has a colour (black or white). Each picture has several objects (the number is not fixed) and some objects are inside other objects. For our experiments we generated random examples, where each parameter in each example was chosen uniformly at random. In particular we used between 2 and 6 objects, the shape colour and configuration were chosen randomly, and each object is inside some other object with probability 0.5 where the target was chosen randomly amongst "previous" objects to avoid cycles. Note that since we use a function free representation the domain size in examples is larger than the number of objects (to include: *up*, *down*, *black*, *white*). In order to label examples we arbitrarily picked 4 target Horn expressions of various complexities. Table 3 gives an indication of target complexities. The first 3 targets have 2 clauses each but vary in the number of atoms in the antecedent and the fourth one has 10 clauses of the larger kind making the problem more challenging. The numbers of atoms and variables are meant as a rough indication as they vary slightly between clauses. To illustrate the complexity of the targets, one of the clauses in IV is
$circle(X)in(X,Y)in(Y,Z)colour(Y,B)colour(Z,W)$
$black(B)white(W)in(Z,U) \rightarrow triangle(Y)$

We ran the batch algorithm on several sample sizes. Table 4 summarises the accuracy of learned expressions as a function of the size of the training set (200 to 3000) when tested on classifying an independent set of 3000 examples. The last column in the table gives the majority class percentage (marked *bl* for baseline), and the one but last line in the table gives the average running times for training on target IV in minutes. Each entry is an average of 10 independent runs where a new set of random examples is used in each run.

Table 3. Complexity of Targets

Target	*Clauses*	*Atoms*	*Variables*
I	2	4	2
II	2	6	4
III	2	9	6
IV	10	9	6

Table 4. Performance Summary

Sys	Tgt	200	500	1000	2000	3000	*bl*
LogAn	I	99.7	100	100	100	100	64.4
LogAn	II	97.6	99.4	99.9	99.9	100	77.8
LogAn	III	90.8	97.2	99.3	99.9	99.9	90.2
LogAn	IV	85.9	92.8	96.8	98.4	98.9	84.7
time	IV	1.1	3.7	7.0	14.9	26.4	min
ICL	IV	85.2	88.6	89.1	90.2	90.9	84.7

Clearly, the algorithm is performing very well on this problem setup. Note that the baseline is quite high, but even in terms of relative error the performance is good. We also see a reasonable learning curve obtained for the more challenging problems. Notice that for target I the task is not too hard since it is not unlikely that we get a random example matching the antecedent of a rule exactly (so that discovering the clause is easy) but for the larger targets this is not the case. We have also run experiments with up to 10 shapes per example with similar performance.

In order to put these experiments in perspective we applied the systems ICL (De Raedt & Van Laer, 1995) and Tilde (Blockeel & De Raedt, 1998) to the same data. Exploratory experiments suggested that ICL performs better on these targets so we focus on ICL. We ran ICL to learn a Horn CNF and otherwise with default parameters. ICL uses a scheme of declarative bias to restrict its search space. With a general pattern implying little bias, success was limited. We thus used a bias allowing up to 4 shapes and identifying the relation between $config$ and *up*, *down* and similarly *colour* and *black*, *white*. Interestingly, for ICL the change in performance from target I to IV was less drastic than in LogAn-H. This may well be due to the fact that LogAn-H builds antecedents directly from examples. The last line in Table 4 gives the performance of ICL on target IV. As can be seen the system performs less well than LogAn-H on this problem. The comparison shows that LogAn-H indeed does something complementary to other systems and

establishes that the learning tasks considered are non-trivial. To summarise we have seen that LogAn-H performs very well in this domain, learning complex and large expressions over large data sets.

5. Conclusion

The paper introduced the system LogAn-H implementing new algorithms for learning function free Horn expressions. The system is based on algorithms proved correct in Khardon (1999a) but includes various improvements in terms of efficiency as well as a new batch algorithm that learns from examples only. Moreover, we have argued that the batch algorithm can be seen as performing a refinement search over multi-clause hypotheses. The main difference from other algorithms is in using different operations for refining the clauses. These in turn are based on dropping objects and on the pairing procedure. We demonstrated through qualitative and quantitative experiments that the system performs well in several domains on tasks that may well be beyond the scope of other ILP systems. Of course, the system will not always be better. In particular, if we have a small number of examples each with a large domain then LogAn-H is not likely to perform very well. We have, however, briefly mentioned properties that should guarantee good performance. In particular, if the data set includes all sub-structures of examples, or if this is approximated to some degree then the system should perform well.

There is also plenty of scope for further improvements. For example, facilities for handling noise and dealing with numerical attributes may be necessary for some applications. Further improvements in efficiency may be obtained by utilising information about the types of objects in interpretations. Extensions allowing for function symbols may be obtained by following recent theoretical developments. The model inference problem (Shapiro, 1983) is sufficiently different so that our algorithms do not immediately apply to it. New algorithms for this problem will be of interest.

Acknowledgements

This work was partly supported by EPSRC grant GR/M21409. I am grateful to several people for their help: Stephen Kitt wrote some of the code for the system, Hendrik Blockeel provided the Tilde system as well as a generator for Bongard examples, Win Van Laer provided the ICL system, and Mark Steedman made useful comments on earlier drafts.

References

Angluin, D. (1988). Queries and concept learning. *Machine Learning*, *2*, 319–342.

Angluin, D., Frazier, M., & Pitt, L. (1992). Learning conjunctions of Horn clauses. *Machine Learning*, *9*, 147–164.

Arimura, H. (1997). Learning acyclic first-order Horn sentences from entailment. *Proceedings of the Conference on Algorithmic Learning Theory* (pp. 432–445). LNAI 1316.

Blockeel, H., & De Raedt, L. (1998). Top down induction of first order logical decision trees. *Artificial Intelligence*, *101*, 285–297.

Bratko, I. (1999). Refining complete hypotheses in ILP. *Proceedings of the Workshop on Inductive Logic Programming* (pp. 44–55). LNAI 1634.

De Raedt, L., & Dzeroski, S. (1994). First order *jk*-clausal theories are PAC-learnable. *Artificial Intelligence*, *70*, 375–392.

De Raedt, L., & Van Laer, W. (1995). Inductive constraint logic. *Proceedings of the Conference on Algorithmic Learning Theory* (pp. 80–94). LNAI 997.

Dechter, R., & Pearl, J. (1992). Structure identification in relational data. *Artificial Intelligence*, *58*, 237–270.

Kautz, H., Kearns, M., & Selman, B. (1995). Horn approximations of empirical data. *Artificial Intelligence*, *74*, 129–145.

Khardon, R. (1999a). Learning function-free Horn expressions. *Machine Learning*, *37*, 241–275.

Khardon, R. (1999b). Learning range-restricted Horn expressions. *Proceedings of the European Conference on Computational Learning Theory* (pp. 111–125). LNAI 1572.

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

Muggleton, S., & De Raedt, L. (1994). Inductive logic programming: Theory and methods. *Journal of Logic Programming*, *20*, 629–679.

Pereira, F., & Shieber, S. (1987). *Prolog and natural-language analysis.* Stanford : Center for the Study of Language and Information.

Quinlan, J. R. (1990). Learning logical definitions from relations. *Machine Learning*, *5*, 239–266.

Rao, K., & Sattar, A. (1998). Learning from entailment of logic programs with local variables. *Proceedings of the Conference on Algorithmic Learning Theory* (pp. 143–157). LNAI 1501.

Reddy, C., & Tadepalli, P. (1998). Learning first order acyclic Horn programs from entailment. *Proceedings of the Conference on Inductive Logic Programming* (pp. 23–37). LNAI 1446.

Shapiro, E. Y. (1983). *Algorithmic program debugging.* Cambridge, MA: MIT Press.