On avoiding redundancy in Inductive Logic Programming systems

Nuno Fonseca, Vitor S. Costa, Fernando Silva, Rui Camacho

Technical Report Series: DCC-2003-04

Departamento de Ciência de Computadores – Faculdade de Ciências

&

Laboratório de Inteligência Artificial e Ciência de Computadores

Universidade do Porto

Rua do Campo Alegre, 823 4150-180 Porto, Portugal

Tel: +351+22+6078830 – Fax: +351+22+6003654

http://www.dcc.fc.up.pt/Pubs/treports.html

On avoiding redundancy in Inductive Logic Programming systems

Nuno Fonseca[1], Vitor Santos Costa[2], Fernando Silva[1], Rui Camacho[3]

[1] DCC-FC & LIACC, Universidade do Porto
R. do Campo Alegre 823, 4150-180 Porto, Portugal
{nf,fds}@ncc.up.pt

[2] COPPE/Sistemas, Universidade Federal do Rio de Janeiro
Centro de Tecnologia, Bloco H-319, Cx. Postal 68511 Rio de Janeiro, Brasil
vitor@cos.ufrj.br

[3] Faculdade de Engenharia & LIACC, Universidade do Porto
Rua Dr. Roberto Frias, s/n 4200-465 Porto, Portugal
rcamacho@fe.up.pt

November 2003

Abstract

Inductive Logic Programming (ILP) is a subfield of Machine Learning that provides an excellent framework for learning in multi-relational domains and inducing first-order clausal theories. ILP systems perform a search through very large hypothesis spaces containing redundant hypotheses. The generation of redundant hypotheses may prevent the systems from finding good models and increase the time to induce them. In this paper we propose a classification of hypotheses redundancy. We show how expert knowledge can be provided to an ILP system to avoid the generation of redundant hypotheses. Preliminary results suggest that the the number of hypotheses generated and execution time are substancially reduced when using expert knowledge to avoid the generation of redundant hypotheses.

Keyworks: Machine Learning, Inductive Logic Programming, expert-assistance

1 Introduction

Inductive Logic Programming (ILP) [1, 2] is a form of supervised learning that aims at the induction of logic programs, or theories, from a given set of examples and prior knowledge (*background knowledge*), also represented as logic programs. ILP has been successfully applied to learning in multi-relational domains [3].

Like other Machine Learning approaches, ILP systems have to traverse a potentially infinite hypothesis space. At each search node an ILP system generates and then evaluates an hypothesis (represented as a clause). The evaluation of an hypothesis usually requires computing its coverage, that is, computing which and how many examples it explains. ILP systems therefore may have long execution times.

Research in improving the efficiency of ILP systems has thus focused in reducing their sequential execution time, either by reducing the number of generated hypotheses (see, e.g., [4, 5]), or by efficiently testing candidate hypotheses (see, e.g., [6, 7, 8, 9]). An alternative approach to improve the response time of ILP systems, besides improving their sequential efficiency, is through parallelization as recommended by Page [10], and confirmed by research results [11, 12, 13, 14, 15]. Our report contributes to the effort of improving the efficiency of ILP systems by identifying types of *redundancy* found in ILP search spaces and by proposing techniques for handling such redundancy.

In order to explain why ILP systems generate redundant hypotheses, we first observe that ILP systems may be seen as using refinement operators [16] to generate hypotheses. According to Van der Laag [17], ideal refinement operators should respect three properties: *properness*, i.e., a refinement operator should not generate equivalent (redundant) clauses; *local finiteness*; and *completeness*. He showed that ideal operators do not exist for unrestricted θ-subsumption ordered set of clauses, as used in most ILP systems. Hence, generic refinement operators for ILP cannot be ideal, and is usually the properness property that is sacrified. Thus is usual the generation redundant of hypotheses by ILP systems.

The efficiency of an ILP system may therefore be significantly improved if the number of redundant hypotheses is reduced. A first step to achieve this goal is to identify and classify the types of redundancy actually found in ILP systems search spaces. Based on this information one can envisage ways to try to avoid the generation of redundant hypotheses. We thus classify several types of redundancy in hypotheses. To the best of our knowledge this is the first time that someone presents a classification of hypotheses redundancy found in ILP systems search spaces. A second step is to deal with these forms of redundancy. Ideally, we would like to avoid redundancy automatically whenever possible. Alternatively, we present several strategies through which experts can easily provide relevant knowledge to help reduce redundancy. The exploitation of the human expertise is not novel in ILP. Recently, human expertise in providing a partial ordering on the sets of background predicates was exploited by Ashwin et al. [18] with good results.

The remainder of this report is organized as follows. Section 2 provides some background information. Section 3 presents a classification of hypotheses redundancy and in Section 4 we propose techniques to handle the identified types of redundancy. In Section 5 we present and discuss some preliminary results. We conclude in Section 6 pointing out future work.

2 Background

This section briefly presents some basic concepts and terminology used in the remaining of the report and is not meant as an introduction to the field of ILP. For such introduction we refer to [19, 20, 21].

From a logic perspective, the ILP problem can be defined as follows. Let E^+ be the set of positive examples, E^- the set of negative examples, $E = E^+ \cup E^-$, and B the background knowledge. In general, B and E can be arbitrary logic programs. The aim of an ILP system is to find a set of hypotheses (also referred to as a theory) H, in the form of a logic program, such that the following conditions hold:

- **Prior Satisfiability**: $B \wedge E^- \not\models \Box$
- **Prior Necessity**: $B \not\models E^+$
- **Posterior Satisfiability**: $B \wedge E^- \wedge H \not\models \Box$ (Consistency)
- **Posterior Sufficiency**: $B \wedge H \models E^+$ (Completeness)
- **Posterior Necessity**: $B \wedge h_i \models e_1^+ \vee e_2^+ \vee \ldots \vee e_n^+ (\forall h_i \in H, e_j \in E^+)$

The sufficiency condition is sometimes named *completeness* with regard to positive evidence, and the posterior satisfiability is also known as *consistency* with the negative evidence. Posterior necessity states that each hypothesis h_i should not be vacuous. The consistency condition is sometimes relaxed to allow hypotheses to be inconsistent with a small number of negative examples. This allows ILP systems to deal with noisy data (examples and background knowledge).

The ILP problem can be mapped into a search through a space of hypotheses. The states in the search space (designated as *hypothesis space*) are concept descriptions (hypotheses) and the goal is to find one or more states satisfying some quality criterion. The ILP problem can be solved by the use of general artificial intelligence techniques like generate and test algorithms. However, due to the large and, in most domains, even infinite size of the search space, this approach is too computationally expensive to be of interest. To tackle this problem the search space is structured by imposing a *generality order* upon the clauses. Such an order on clauses is usually denoted by $\preceq$, and the structured search space designated as generalization lattice. A clause C is said to be a generalization of D (dually: D is a specialization of C) if $C \preceq D$ holds. There are many generality orders, the most important are subsumption and logical implication. In both of these orders, the most general clause is the empty clause $\Box$. The subsumption order is the generality order most often used in ILP and is defined as follows:

Definition 1 *Let C and D be clauses. A clause C subsumes D, denoted by $C \preceq D$, if there exists a substitution θ such that $C\theta \subseteq D$.*

The search can be done in two ways: specific-to-general [22] (or *bottom-up*); or general-to-specific [16, 23, 24] (or *top-down*). Refinement operators generalize or specialize an hypothesis, thus generating more hypotheses. To restrict the generation of hypotheses is usual to impose further conditions to the refinement operators besides completeness. One of those conditions require that the generated hypothesis satisfy the language bias. Bias are the restrictions, mostly syntactic, imposed on candidate hypotheses.

3 Redundancy in Hypotheses

In which context is an hypothesis redundant? We classify the redundancy of an hypothesis in terms of its "location" as intrinsically redundant or contextually redundant. An hypothesis is *intrinsically redundant*, or simply redundant, if it includes redundant literals (e.g., $a(X) \leftarrow b(X), b(X)$). An hypothesis is *contextually redundant* if it is redundant when considered in some context (e.g., if it repeats a node of the search space).

A different approach to classify redundant hypotheses is by the way in which redundancy is detected. From this perspective, redundancy could be classified as semantic or syntactic. *Syntactic redundancy* can be verified through syntactic analysis (of the literals or clauses). On the other hand, *semantic redundancy* requires computing the model to determine equivalence between the clauses or literals, and thus depends on the background knowledge.

In general, even if an hypothesis is redundant, it is possible that some of its descendants will not be. We say that an hypothesis h is $\mathcal{R}*$ if all refinements of h are also redundant. Search may only be pruned at $\mathcal{R}*$ hypotheses, otherwise we would loose completeness.

We next formalize several types of intrinsic redundancy and contextual redundancy. In the definitions throughout this section we will use the following notation: C is a sequential definite clause in the form $L_0 \leftarrow L_1, \ldots, L_n$ $(n \geq 1)$; L_i $(1 \leq i \leq n)$ is a literal in the body of the clause and L_0 is the head literal of the clause; each literal L_i can be represented by $p_i(A_1, \ldots, A_{ia})$ where p_i is the predicate symbol with arity ia and $A_1, \ldots, A_{ia}$ are the arguments; and S is the set of hypotheses of the search space generated by an ILP system. The symbol $\models_B$ denotes the logical implication and $\equiv_B$ the logical equivalence considering the background knowledge provided (B). Since there is no doubt of the context of both logical relations we simplify the representation using only $\models$ and $\equiv$. Some logic definitions used in this section are provided in the appendix. For further definitions we refer the reader to [25].

3.1 Intrinsic Redundancy

The problem of verifying where a clause has redundancy corresponds to the problem of verifying whether the clause is *condensed*, that is, if it does not subsume any proper subset of itself. Gottlob et al. [26] showed that the problem of verifying whether a clause is condensed is co-NP-complete. They also showed that it is undecidable to verify that a clause does not contain any proper subset that is implied by the clause.

We thus can only hope to address instances of the problem that are common in practice, and that are easy to detect. In this work we will be interested in $\mathcal{R}*$, so that we can benefit by pruning the search space.

The ILP process refines a clause by adding an extra literal or by binding variables in a clause. It is therefore natural to focus on redundant literals:

Definition 2 (Intrinsically Redundant Literal) *The literal L_i is an intrinsically redundant literal in a clause C if $(C \setminus L_i) \models C$*

Clauses which have a redundant literal are clearly *intrinsically redundant*. Consider for example the clause $a(X) \leftarrow b(X), b(X)$. It is an intrinsically redundant clause since it contains a redundant literal (e.g.,$L_2 = b(X)$).

3.1.1 Syntactic Redundancy

We next discuss several cases of syntactic redundancy. These cases are particularly interesting because they can be verified in polynomial time, and because they are surprisingly common in ILP systems. We start from the simplest case (we assume that the order of literals does not matter):

Definition 3 (Duplicate) *A literal L_i is duplicate in C if L_i occurs in C more than once.*

Consider for example the clause $a(X) \leftarrow b(X), b(X)$. As pointed out earlier, it contains a duplicate literal $b(X)$, and it is intrinsically redundant. It should be clear that a clause with a duplicate literal is $\mathcal{R}*$: all further refinements will have both duplicate literals.

Generalizations of duplicate literals are not necessarily $\mathcal{R}*$. Consider:

Definition 4 (Duplicate Up To Renaming) *A literal L_i is duplicate up to renaming if there is another literal L_j such that (i) there is a renaming σ of the free variables in L_i such that $L_i\sigma = L_j$, and (ii) all renamed variables only occur in L_i or L_j.*

As an example consider the clause $a(X) \leftarrow b(X,Y), b(X,Z)$. If the literal $b(X,Y)$ succeeds, the literal $b(X,Z)$ will also succeed, so it is sufficient to prove $a(X) \leftarrow b(X,Y)$. Unfortunately, as we said before, hypotheses with literals up to renaming are not necessarily $\mathcal{R}*$. For instance, $a(X) \leftarrow b(X,Y), b(X,Z), Y > Z$ is a not redundant but might be a valid refinement of the example clause. This form of redundancy therefore cannot always be used to improve the search space, but it has been successfully used to improve performance in coverage detection [8].

3.1.2 Semantic Redundancy

Often, the background knowledge may include structural information on a domain. We may know some degenerate cases when a literal is always or never satisfiable. We may also have extensional information on a predicate, say, whether it is reflexive, associative, or commutative. Last, we generalize this concept through the notion of entailment between sub-goals. It should be easy to see that all these cases are $\mathcal{R}*$: all properties we mention must hold for all instances of a literal, therefore any extension of the clause will also be redundant.

It is convenient to consider reflexivity as an example of two degenerate cases, a valid literal or an unsatisfiable literal:

Definition 5 (Tautology) *A literal L_i is tautologically redundant in C if L_i is always true.*

Consider for instance the "greater or equal" relation denoted by $\geq$. The literal $X \geq X$ is a tautologically redundant literal in the clause $a(X) \leftarrow X \geq X$.

Definition 6 (Contradiction) *A literal L_i is a contradiction in C if $C \setminus L_i$ is satisfiable and C is always inconsistent.*

Consider for instance the "greater than" relation denoted by $>$ and the "less than" relation denoted by $<$. The literal $X > Y$ is a contradiction redundant literal in the clause $a(X) \leftarrow X < Y, X > Y$.

Definition 7 (Commutativity) *The literal $L_i = p_i(A_1, \ldots, A_{ia})$ is commutative redundant in a clause C if there is a compatible literal $L_j = p_j(B_1, \ldots, B_{ja})$ such that:*

1. $L_j \neq L_i$

2. $\exists\, permutation((B_1, \ldots, B_{ja})) = (A_1, \ldots, A_{ia})$

3. $L_j \equiv L_i$

Consider the clause $C = r(X, Z) \leftarrow mult(X, 2, Z), mult(2, X, Z)$ where $mult(X, Y, Z)$ is true if $Z = X * Y$. Since multiplication is commutative, it is known that $mult(X, Y, Z) \equiv mult(Y, X, Z)$, thus $mult(Y, X, Z)$ is a commutative redundant literal.

Definition 8 (Transitivity) *The literal L_i is transitive redundant in a clause C if there are two compatible literals L_j and L_k in C such that $L_i \neq L_j \neq L_k$ and $L_j \wedge L_k \models L_i$*

Consider again the "greater or equal" relation: the literal $X \geq Z$ is transitive redundant in the clause $p(X, Y) \leftarrow X \geq Y, Y \geq Z, X \geq Z$.

Definition 9 (Proper Direct Entailment) *A literal L_i is proper direct entailed in a clause C if there is a compatible literal L_j in C such that $L_i \neq L_j$ and $L_j \models L_i$.*

For instance, the literal $X < 2$ is a proper direct entailed redundant in the clause $p(X) \leftarrow X < 1, X < 2$.

Definition 10 (Proper Direct Equivalence) *A literal L_i is properly direct equivalent in a clause C if there is a literal L_j in C such that $L_i \neq L_j$ and $L_i \equiv L_j$.*

Note that in the definition of proper direct equivalent redundant literal we drop the compatibility constraint on the literals. For instance, the literal $X < 1$ is equivalent redundant in the clause $p(X) \leftarrow 1 > X, X < 1$ since $1 > X \equiv X < 1$. This is another type of semantic redundancy.

Definition 11 (Direct Entailment) *A literal L_i is direct entailed redundant in C if there is a sub-sequence of literals SC from $C \setminus L_i$ such that $SC \models L_i$.*

For instance, consider the clause $p(X) \leftarrow X \leq 1, X \geq 1, X = 1$. The literal $X = 1$ is direct entailed redundant because there is a sequence of literals $(X \leq 1, X \geq 1)$ that imply L_i. In general, verifying whether a set of sub-goals entails another one requires solving a constraint system over some specific domain (the integers in the example).

3.2 Contextual Redundancy

When considering contextual redundancy we are manipulating sets of clauses (hypotheses) instead of sets of literals as in intrinsic redundancy:

Definition 12 (Contextual redundant clause) *The clause C is contextual redundant in $S \cup C$ if $S \models C$.*

The major types of contextual redundancy are obtained by generalizing over the cases of intrinsic redundancy:

Definition 13 (Duplicate) *A clause C is duplicate redundant in S if $C \in S$.*

For instance, consider that S contains the clause $C = p(X) \leftarrow a(X, Y)$. The clause C is duplicate redundant in S.

Definition 14 (Commutativity) *The clause C is commutative redundant in $S \cup C$ if there is a compatible clause $D \in S$ with the same literals of C but with a different ordering such that $C \equiv D$.*

For instance, $C = p(X) \leftarrow a(X, Y), a(X, Z)$ is a commutative redundant clause in S if S contains $p(X) \leftarrow a(X, Z), a(X, Y)$.

Definition 15 (Transitivity) *The clause C is transitive redundant in $S \cup C$ if there are two compatible clauses D and E in S such that*

1. *the body of C and D differ only in one literal (L_C and L_D respectively)*
2. *the body of E contains one more literal than D (L_E)*
3. $L_D \wedge L_E \models L_C$

For instance, consider the clause $p(X,Y,Z) \leftarrow X > Z$. Such clause is transitive redundant in a set S containing $p(X,Y) \leftarrow X > Y$ and $p(X,Y,Z) \leftarrow X > Y, Y > Z$.

Definition 16 (Direct Equivalence) *The clause C is directly equivalent redundant in $S \cup C$ if there is a compatible clause $D \in S$ such that*

1. *the body of C and D differ only in one literal (L_C and L_D respectively)*
2. *L_C is a proper equivalent redundant literal in D*

As an example consider $S = \{D = p(X) \leftarrow X > 1\}$. The clause $p(X) \leftarrow 1 < X$ is proper equivalent redundant in S since the clauses differ in one literal ($X > 1$ and $1 < X$) and $1 < X$ is a proper equivalent redundant literal in D.

Definition 17 (Direct Entailment) *The clause C is directly entailed redundant in $S \cup C$ if there is a literal L in C and compatible clause $D \in S$ such that L is a direct entailed redundant literal in D and $D \setminus SC = C \setminus L$.*

Consider $S = \{D = p(X) \leftarrow X \leq 1, X \geq 1\}$. The clause $C = p(X) \leftarrow X = 1$ contains a literal $X = 1$ that is a directly entailed redundant literal in D. Thus C is a directly entailed redundant clause.

4 Handling Redundancy

In the previous section we identified several types of redundancy and classified them as intrinsically redundant or contextually redundant. In this section we will show how and where such redundancy types can be efficiently eliminated in ILP systems that perform a search following a top-down approach.

The generation of hypotheses in top-down ILP systems can be seen as being composed of the following two steps. First, an hypothesis is selected to be refined. Then a literal is selected (or generated) to be added to the end of the clause's body. We advocate that almost all types of redundancy previously described could be efficiently eliminated if the expert provides meta-knowledge information to ILP systems about predicates' properties and relations among the predicates found in the background knowledge. Such information can be used by the literal generation procedure or by the refinement procedure to avoid the generation of intrinsic and contextual redundant hypotheses.

4.1 Possible approaches

We envise several approaches to incorporate the information provided by the expert in ILP systems to avoid the generation of redundant hypotheses.

A possible solution is the modification of the refinement operator and the literal generation procedure to allow the use of information provided by the expert.

Another approach is the extension of user-defined constraint mechanisms available in some systems (e.g., Progol [27], Aleph [28], Indlog [9]). The constraints are added by the user in the form of clauses that define when an hypothesis should not be considered. Integrity constraints are currently used to eliminate the generation of intrinsic redundant clauses containing contradiction redundant literals. Note that a "...integrity constraint does not state that a refinement of a clause that violates one or more constraints will also be unacceptable." [28]. We are of the opinion that one should try to eliminate the redundancy as a built-in procedure of the refinement operator instead of using mechanisms like constraints since the first option should be more efficient.

Another possible way to eliminate redundant literals could be through the use of user-defined pruning. Pruning is used to exclude clauses and their refinements from the search. It is very useful to state which kinds of clauses should not be considered in the search. Some ILP systems allow the user to provide such rules defining when a hypothesis should be discarded (pruned). Before discarding the clauses they are evaluated against the examples. The use of pruning greatly improves the efficiency of ILP systems since it leads to a reduction of the size of the search space. However, since it involves evaluating a redundant hypothesis against the examples before discarding it, we do not consider it has an ideal solution.

4.2 Implementation

To eliminate the generation of redundant hypotheses we modified the refinement operator and literal generation procedure to exploit the meta-knowledge information provided by the expert. We modified the April [29] ILP system to accept the redundancy declarations that we describe next. The main reason for choosing April is our knowledge regarding its implementation. The declarations are provided to the system as background knowledge in the form of Prolog rules.

We start by describing the declarations that the user may pass to the literal generation. Duplicate, commutative and properly direct equivalent redundant literals can be eliminated during literal generation.

The user may inform the April system of literals' properties through declarations such as `tautology`, `commutative`, and `equiv`. For instance, the declaration `:- tautology('=<'(X,X))` informs the system that literals of the form `'<='(X,X)` are tautological redundant literals. With this information the ILP system avoids the generation of such redundant literals.

The `commutative` declaration indicates that a given predicate is commutative and helps to avoid the ILP system to generate hypotheses with commutative redundant literals. As an example consider that an ILP system is informed that the predicate `adj(X,Y)` is commutative through the declaration `:-commutative(adj/2)`. That information is used to prevent the generation of commutative equivalent literals such as $adj(X, 2)$ and $adj(2, X)$.

The `equiv` declaration allows the expert to indicate that two predicates, although possibly different, generate equivalent literals. For instance, the declaration `:-equiv('<='(X,Y),'>='(Y,X))` informs that the literals like `'<='(X,1)` and `'>='(1,X)` are equivalent.

The `commutative` and `equiv` declarations allow the expert (user) to pass knowledge about the equivalence of literals. Using the information provided the ILP system only needs to consider one literal of each equivalence class. We point out that these declarations allow the ILP system to avoid the generation of several types of intrinsically redundant hypotheses and contextual redundant hypotheses (direct equivalent redundant clauses).

The remaining types of redundancy are eliminated in the refinement operator. The generation of commutative redundant clauses or clauses containing duplicate literals is automatically avoided by the refinement operator of the April system without the need of extra information provided by the user.

To avoid the generation of contradiction redundant hypotheses we used the mechanism of constraints. The constraints are defined by rules of the form `constraint(HypothesysHead, HypothesisBody):-Body.` *`Body`* is a set of literals that specify the condition(s) that should not be violated by hypotheses found by April. For instance, the rule

```
constraint(p(X),Body):-body_contains(Body, (X<Y)), body_contains(Body,(X>Y)).
```

is evaluated as true if `Body` contains, for example, $X < 1$ and $X > 1$.

The generation of transitive redundant literals and clauses can be avoided by the use of information provided by the expert indicating which predicates are transitive. For instance, the rule `:- transitive(lt(X,Y),lt(Y,Z),lt(X,Z))` informs that the `lt` (less than) predicate is transitive.

With such information, a redundant hypothesis containing the literals `lt(X,1),lt(1,Y),lt(X,Y)` will not be generated by the refinement operator.

Avoiding the generation of hypotheses with proper directed entailed redundant literals can be achieved if the expert provides knowledge that a literal implies another. The knowledge can be provided using declarations on the form of $semantic\_rule(L1, L2){:}-RuleBody$, meaning that $L1$ implies $L2$ if the `RuleBody` is evaluated as true. For instance, the rule `semantic_rule(lt(A,B), lt(A,C)):-C<B` allows the refinement operator to avoid generating hypotheses containing a literal like $lt(A, 2)$ followed by a literal like `lt(A,1)` (e.g.,`p(X)`$\leftarrow$ `lt(A,2),lt(A,1)`).

Direct entailed redundant literals or clauses can be prevented from being generated if the expert provides knowledge that a sequence of literals implies another literal. Such information can be provided using the following declaratione `d_entail(`$[L_1, \ldots, L_k]$`,L)`. With such information the refinement operator will not generate clauses containing L together with any of the L_i $(1 \geq i \geq k)$ and clauses containing all L_i. For instance, the clauses `p(X)`$\leftarrow$ `X<=1,X=1` or `p(X)`$\leftarrow$ `X<=1,X>=1` would not be generated if the expert provides a declaration like `d_entail([X<=1,X>=1],X=1)`.

Redundancy Type	**Handled**	**Declaration**	**Example**
Intrinsic / Tautological	literal generation	`:-tautology('>='(X,X)).`	p(X)$\leftarrow$ X>=X
Intrinsic / Contradiction	refinement	`constraint(p(X),Body):- contains(Body,'>'(X,Y)), contains(Body,'<'(X,Y)).`	p(X)$\leftarrow$ X>Y, X<Y
Intrinsic / Commutative	literal generation	`:-commutative(mult(X,Y,R), mult(Y,X,R))`	p(X)$\leftarrow$ mult(X,3,R), mult(3,X,R)
Intrinsic & Contextual Transitivity	refinement	`:-transitive('>'(X,Y), '<'(Y,Z), '<'(X,Z))`	p(X)$\leftarrow$ X>1, 1>Z, X>Z
Intrinsic / Proper direct Entailment	refinement	`semantic_rule('<'(A,B), '<'(A,C)):- C<B`	p(X)$\leftarrow$ X<0,X<2
Intrinsic & Contextual Direct Equivalence	literal generation	`:-equiv('<'(X,Y), '>='(Y,X))`	p(X)$\leftarrow$ X<Y, Y>=X
Intrinsic & Contextual Direct Entailment	refinement	`d_entail(['<='(X,Y), '>='(X,Y))],'='(X,Y)`	p(X)$\leftarrow$ X<=1,X>=1,X=1

Table 1: Redundancy Declarations

Table 1 summarizes the types of redundancy handled in our implementation. For each type of redundancy it is shown where it is handled, an example of a redundancy declaration, and an example of redundant hypothesis.

5 Experiments and Results

The impact of using the redundancy declarations presented in the previous section was empirically evaluated in four datasets. We selected datasets where we were able to identify redundancy on the background knowledge. There are far more interesting datasets available, but a domain expert is required in order to detect possible types of redundancy. Unfortunately, since we did not have access to such experts we selected datasets where our expertise was sufficient to detect redundancy.

The aim of the experiments was to evaluate if the redundancy declarations improve the efficiency of ILP systems. The experiments were made on an AMD Athlon XP 1400+ processor PC with 512MB of memory, running the Linux RedHat (kernel 2.4.20) operating system. The ILP system used was the April [29] system version 0.5. The Prolog compiler used was YAP [30] version 4.3.23.

The datasets used were downloaded from the Machine Learning repositories of the Universities

Dataset	Characterization			April's Settings	
	$\mid E^+ \mid$	$\mid E^- \mid$	$\mid B \mid$	**i**	**noise**
krki I	342	658	1	1	10
krki II	3240	6760	1	1	10
multiplication	9	15	3	2	0
range	19	14	1	1	0

Table 2: Settings used in the experiments

of Oxford[1] and York[2], and from Camacho's[3] home page. Table 2 characterizes the datasets in terms of number of positive and negative examples as well as background knowledge size. Furthermore, it shows the April settings used with each dataset. The i-depth corresponds to the maximum depth of a literal with respect to the head literal of the hypothesis [31]. Finally, the parameter *noise* defines the maximum number of negative examples that an hypothesis may cover in order to be accepted. No limit on the number of hypotheses generated was imposed, and thus an exhaustive search was performed.

Dataset	| Hypotheses |			Time (sec.)		
	normal	**red-decl**	(%)	**normal**	**red-decl**	(%)
krki I	7,281	911	12.51	3.13	0.97	30.99
krki II	2,103,988	192,911	9.16	5,991.73	194.70	3.24
multiplication	839	478	56.97	49.20	11.86	24.10
range	4,203	579	13.77	7.08	1.27	17.93

Table 3: Impact of using redundancy declarations (**red-decl**) on April

Table 3 summarizes the performance of the April system using redundancy declarations and not using them. It shows the total number of hypotheses generated, execution time, and the impact in number of generated hypotheses and execution time (given as a ratio between using redundancy declarations and not using them). For the purposes of this study we do not present accuracies of the models generated because they do not differ in both runs of April for each dataset. For all datasets considered, one can observe that a significant reduction on the execution time and on the number of hypotheses generated has occurred.

These results suggest that the performance of the April system is significantly improved if the expert provides redundancy knowledge. It is important to remember that since the redundancy information provided is used to eliminate $\mathcal{R}*$ redundant hypotheses, the accuracy of the models found is not affected negatively.

6 Conclusions

This work contributes to the effort of improving the efficiency of ILP systems by classifying major forms of redundancy found in the search space of ILP applications, and by designing a mechanism that allows pruning of redundant hypotheses. In our approach, a domain expert provides meta-knowledge about the redundancy types by describing high-level properties of the relations in the background knowledge. Experiments with the modified system show substantial performance improvements, up to an order of magnitude. Our experimental results have two limitations: only four datasets were used; and the datasets are relatively small. However, we believe that they suggest that we can achieve even more significant performance gains for larger datasets. The major thrust of our work is to make

[1] `http://www.comlab.ox.ac.uk/oucl/groups/machlearn/`
[2] `http://www.cs.york.ac.uk/mlg/index.html`
[3] `http://www.fe.up.pt/~rcamacho/datasets/datasets.html`

ILP systems able to learn from larger datasets. Most ILP systems are configured to generate a limited number of hypotheses. Therefore, avoiding redundant hypotheses may lead to the generation of good hypotheses that otherwise would be lost. We hope that this may result in an improvement of the quality of the induced models. Further experiments are required to confirm or refute these claims. Also, we have not considered all forms of redundancy in ILP learning. Work is thus necessary to continue on the discovery of major sources of redundancy in hypotheses, namely through experience with more applications.

Acknowledgments

The work presented in this report has been partially supported by project APRIL (Project POSI/SRI/-40749/2001) and funds granted to *LIACC* through the *Programa de Financiamento Plurianual, Fundação para a Ciência e Tecnologia* and *Programa POSI*. Nuno Fonseca is funded by the FCT grant SFRH/BD/7045/2001.

References

[1] S. Muggleton. Inductive logic programming. In *Proceedings of the 1st Conference on Algorithmic Learning Theory*, pages 43–62. Ohmsma, Tokyo, Japan, 1990.

[2] S. Muggleton. Inductive logic programming. *New Generation Computing*, 8(4):295–317, 1991.

[3] Ilp applications. http://www.cs.bris.ac.uk/ ILPnet2/Applications/.

[4] C. Nédellec, C. Rouveirol, H. Adé, F. Bergadano, and B. Tausend. Declarative bias in ILP. In L. De Raedt, editor, *Advances in Inductive Logic Programming*, pages 82–103. IOS Press, 1996.

[5] Rui Camacho. Improving the efficiency of ilp systems using an incremental language level search. In *Annual Machine Learning Conference of Belgium and the Netherlands*, 2002.

[6] Hendrik Blockeel, Luc Dehaspe, Bart Demoen, Gerda Janssens, Jan Ramon, and Henk Vandecasteele. Improving the efficiency of Inductive Logic Programming through the use of query packs. *Journal of Artificial Intelligence Research*, 16:135–166, 2002.

[7] Vítor Santos Costa, Ashwin Srinivasan, and Rui Camacho. A note on two simple transformations for improving the efficiency of an ILP system. *Lecture Notes in Computer Science*, 1866, 2000.

[8] Vítor Santos Costa, Ashwin Srinivasan, Rui Camacho, Hendrik Blockeel, and Wim Van Laer. Query transformations for improving the efficiency of ilp systems. *Journal of Machine Learning Research*, 2002.

[9] Rui Camacho. *Inducting models of human control skills unsing ML algorithms.* PhD thesis, Univerity of Porto, 2000.

[10] David Page. ILP: Just do it. In J. Cussens and A. Frisch, editors, *Proceedings of the 10th International Conference on Inductive Logic Programming*, volume 1866 of *Lecture Notes in Artificial Intelligence*, pages 3–18. Springer-Verlag, 2000.

[11] L. Dehaspe and L. De Raedt. Parallel inductive logic programming. In *Proceedings of the MLnet Familiarization Workshop on Statistics, Machine Learning and Knowledge Discovery in Databases*, 1995.

[12] T. Matsui, N. Inuzuka, H. Seki, and H. Itoh. Comparison of three parallel implementations of an induction algorithm. In *8th Int. Parallel Computing Workshop*, pages 181–188, Singapore, 1998.

[13] Hayato Ohwada and Fumio Mizoguchi. Parallel execution for speeding up inductive logic programming systems. In *Lecture Notes in Artificial Intelligence*, number 1721, pages 277–286. Springer-Verlag, 1999.

[14] Hayato Ohwada, Hiroyuki Nishiyama, and Fumio Mizoguchi. Concurrent execution of optimal hypothesis search for inverse entailment. In J. Cussens and A. Frisch, editors, *Proceedings of the 10th International Conference on Inductive Logic Programming*, volume 1866 of *Lecture Notes in Artificial Intelligence*, pages 165–173. Springer-Verlag, 2000.

[15] Y. Wang and D. Skillicorn. Parallel inductive logic for data mining. In *Workshop on Distributed and Parallel Knowledge Discovery, KDD2000*, Boston, 2000. ACM Press.

[16] E.Y. Shapiro. *Algorithmic Program Debugging.* The MIT Press, 1983.

[17] P.R.J. van der Laag. *An analysis of refinement operators in inductive logic programming.* PhD thesis, Erasmus Universiteit, Rotterdam, the Netherlands, 1995.

[18] A. Srinivasan, R.D. King, and M.E. Bain. An empirical study of the use of relevance information in inductive logic programming. *Machine Learning Research (to appear)*, 2003.

[19] S. Muggleton and L. De Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19/20:629–679, 1994.

[20] N. Lavrac and S. Dzeroski. *Inductive Logic Programming: Techniques and Applications.* Ellis Horwood, 1994.

[21] S.-H. Nienhuys-Cheng and R. de Wolf. *Foundations of Inductive Logic Programming*, volume 1228 of *Lecture Notes in Artificial Intelligence.* Springer-Verlag, 1997.

[22] S. Muggleton and C. Feng. Efficient induction of logic programs. In *Proceedings of the 1st Conference on Algorithmic Learning Theory*, pages 368–381. Ohmsma, Tokyo, Japan, 1990.

[23] J. R. Quinlan and R. M. Cameron-Jones. FOIL: A midterm report. In P. Brazdil, editor, *Proceedings of the 6th European Conference on Machine Learning*, volume 667, pages 3–20. Springer-Verlag, 1993.

[24] S. Muggleton. Inverse entailment and Progol. *New Generation Computing, Special issue on Inductive Logic Programming*, 13(3-4):245–286, 1995.

[25] Cristopher John Hogger. *Essentials of Logic Programming.* Oxford University Press, 1990.

[26] G. Gottlob and C.G. Fermiiler. Removing redundancy from a clause. *Artificial Intelligence*, 61(2):263–289, June 1993.

[27] Stephen Muggleton and John Firth. Relational rule induction with cprogol4.4: A tutorial introduction. In Saso Dzeroski and Nada Lavrac, editors, *Relational Data Mining*, pages 160–188. Springer-Verlag, September 2001.

[28] Aleph. http://web.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/.

[29] Nuno Fonseca, Rui Camacho, Fernando Silva, and Vítor S. Costa. Induction with April: A preliminary report. Technical Report DCC-2003-02, DCC-FC, Universidade do Porto, 2003.

[30] V. S. Costa, L. Damas, R. Reis, and R. Azevedo. *YAP Prolog User's Manual.* Universidade do Porto, 1989.

[31] S. Muggleton and C. Feng. Efficient induction in logic programs. In S. Muggleton, editor, *Inductive Logic Programming*, pages 281–298. Academic Press, 1992.

A Definitions from Logic

A **literal** is an atom (**positive literal**) or the negation of an atom (**negative literal**). A **clause** C is a finite (possibly empty) set of literals that represent the disjunction of literals. The empty clause is denoted by $\square$ and is always false (it has no true element). A clause $\{L_1,\ldots,L_i,\neg L_{i+1},\ldots,\neg L_n\}$ can be represented as $L_1 \vee \ldots \vee L_i \vee \neg L_{i+1} \vee \ldots \vee \neg L_n$ or, equivalently, as $L_1 \vee \ldots \vee L_i \vee \leftarrow L_{i+1} \wedge \ldots \wedge L_n$ A clause is a **definite clause** if it has exactly one positive literal. The positive literal is called the **head** of the clause. The set of all negative literals is called the **body** of the clause. A **Horn clause** (Prolog clause) is a clause with at most one positive literal. Two literals are **compatible** if they have the same predicate symbol and sign. Two clauses are **compatible** if they have the same head literal. The traditional definition of clause does not take into account the internal ordering and repetition of literals, which are relevant in the context of Prolog programs. To cope with this problem we will used the sequential clause definition [17]. A **sequential clause**, denoted by $\vec{C},\vec{D},\ldots$ is a sequence of literals. Unless otherwise stated, all clauses in this report are sequential definite clauses.