On Avoiding Redundancy in Inductive Logic Programming ⋆ ⋆⋆

Nuno Fonseca[1], Vítor S. Costa[2], Fernando Silva[1], and 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, UFRJ 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

Abstract. ILP systems induce first-order clausal theories performing a search through very large hypotheses spaces containing redundant hypotheses. The generation of redundant hypotheses may prevent the systems from finding good models and increases the time to induce them. In this paper we propose a classification of hypotheses redundancy and show how expert knowledge can be provided to an ILP system to avoid it. Experimental results show that the number of hypotheses generated and execution time are reduced when expert knowledge is used to avoid redundancy.

Keywords: **Redundancy, Expert-Assistance**

1 Introduction

Inductive Logic Programming (ILP) [1] 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. Like other Machine Learning approaches, ILP systems have to traverse a potentially infinite search 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 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 [2,3]; by efficiently testing candidate hypotheses [4,5,6]; or through

⋆ Appears in the Proceedings of 14th International Conference on Inductive Logic Programming (ILP 2004).

parallelism [7,8]. Arguably, best results can be achieved through a reduction of the search space. In this work, we start from the well-known observation that the search space for ILP can be highly *redundant*. It is known that redundancy cannot be completely eliminated on finite and complete systems [9]. Instead, we identify and classify several common forms of redundancy that are frequently found in ILP applications, and suggest techniques to address them. 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.

In order to explain why ILP systems generate redundant hypotheses, we first observe that ILP systems may be seen as using refinement operators [10] to generate hypotheses. According to Van der Laag [9], ideal refinement operators should respect three properties: *properness*, i.e., a refinement operator should not generate equivalent (redundant) clauses; *local finiteness*; and *completeness*. Van der Laag 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. Since guaranteeing completeness and local finiteness is fundamental, properness is usually sacrificed. Thus it is usual the generation of redundant hypotheses by ILP systems.

The efficiency of an ILP system may therefore be significantly improved if the number of redundant hypotheses is decreased. 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 avoid the generation of redundant hypotheses. We thus classify several types of redundancy. A second step is to deal with these forms of redundancy. We describe 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, Srinivasan et al. [11] obtained good results by using human expertise to provide a partial ordering on the sets of background predicates.

The remainder of this paper is organized as follows. Section 2 presents a classification of hypotheses redundancy and in Section 3 we describe how to handle the identified types of redundancy. In Section 4 we present and discuss some experimental results. We conclude in Section 5 pointing out future work.

2 Redundancy in Hypotheses

Our goal is to prune the search space by eliminating *redundant* hypotheses. Clearly, if an ILP system generates a single hypothesis, the hypothesis can never be redundant. Redundancy is therefore a property of a clause within a search space.

To guarantee completeness, we should prune the search at an hypothesis C_i if C_i is such that **(i)** C_i itself is redundant, and **(ii)** and C_i's *descendants* will be redundant, too. The first condition says that we want to prune repeated hypotheses. The second condition requires further explanation. Consider a clause obtained in an example run of Aleph [12], a top-down ILP system. In this example we consider the well-known carcinogenesis problem.We first reach a clause

saying that a molecule is active if it includes a Carbon 22:

$$\mathbf{1}.\ \ active(A) \leftarrow atm(A, B, c, 22, C)$$

The derivation eventually generates a clause, where D and B are Carbon 22s (they may refer to the same atom):

$$\mathbf{2}.\ \ active(A) \leftarrow atm(A, B, c, 22, C), atm(A, D, c, 22, C)$$

We may say that the second clause is redundant because if the first clause is satisfiable, the second is too, and vice-versa. Should we prune here? Aleph does not prune, and eventually derives a clause saying that the atoms B and D must have a bond:

$$\mathbf{3}.\ \ active(A) \leftarrow atm(A, B, c, 22, C), atm(A, D, c, 22, C), bond(A, D, B, 7)$$

This clause is not redundant: pruning an hypothesis is therefore wrong, if we do not consider descendants.

Next we describe the set of prunable hypotheses as $\mathcal{R}*$. Our goal in this work can be phrased as trying to obtain the best possible approximation to $\mathcal{R}*$. The reader may have noticed that whether a new hypothesis C_i is in $\mathcal{R}*$ depends **(a)** on the hypotheses $C_j, j < i$ that have been generated so far, and **(b)** on the hypotheses $C_k, k > i$ that are to be generated. The structure of $\mathcal{R}*$ thus depends on the ordering we use when we build the search space. Here, we shall consider the popular top-down approach where clauses are refined by introducing new substitutions and literals.

Detecting whether a clause is redundant and detecting whether all refinements of a clause are also redundant is hard. We would like an efficient approach, that could detect interesting cases with little computational overhead. Our observation is that we can be confident that two clauses will have the same refinements, if the clauses always generate the same bindings for their variables. More precisely, given a clause C, the set of variables $\mathcal{V} = \{V : V \in Vars(C)\}$[4], and the set of success substitutions θ from all variables in $\mathcal{V}$ to ground terms that satisfy C, we say that two clauses C_1 and C_2 are interchangeable if there is a mapping ω from the variables in $\mathcal{V}_1$ to the variables in $\mathcal{V}_2$ such that if $\forall\theta_1 \exists\theta_2, \theta_1 = \omega\theta_2$.

If the two clauses generate the same set of substitutions for all their variables, they have the same answers for the head variables, and must thus cover the same number of examples. One is therefore *redundant*. Refining in top-down approaches consists of introducing extra goals and/or constraints, say C'. But it is straightforward to show that if $(C_1 \wedge C')\sigma$ is satisfiable so must be $(C_2 \wedge C')\omega\sigma$, hence $(C_2 \wedge C')\sigma$. Therefore, all refinements to C_2 would be redundant (if the search for C_1 is complete).

Next, we address how to recognise cases of redundancy that address the conditions discussed above. We will use the following notation: C is a *sequential*

[4] $Vars(C)$ is the set of variables in a clause C.

definite clause [9], i.e., a sequence of literals, 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; two literals are *compatible* if they have the same predicate symbol and sign; two clauses are compatible if they have the same head literal; and $\mathcal{S}$ is the multiset of hypotheses of the search space generated by an ILP system. The symbol $\vDash_B$ 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 $\vDash$ and $\equiv$. For further definitions on logic we refer the reader to [13]. For an introduction to ILP we refer the reader to [14].

2.1 Self-Redundant Clauses

Ideally we would like to verify redundancy *syntactically*, that is, just through scanning the literals or clauses. On the other hand, we sometimes require *semantic* information, that is, prior knowledge on the model, to determine equivalence between clauses or literals. We also distinguish the interesting cases of self-redundant clauses i.e., where by just looking at a clause we know that the clause must repeat a parent in the search tree, naming the remainder cases *context-derived*.

The problem of verifying whether a clause h is self-redundant corresponds to the problem of verifying whether the clause is *condensed*, that is, whether it does not subsume any proper subset of itself. Gottlob et al. [15] 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 specific instances of the problem.

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 1 (Self-Redundant Literal) *The literal L_i is a self-redundant literal in a clause C if $(C \setminus L_i) \vDash C$*

Clauses which have a self-redundant literal are clearly *self redundant*: they can be reduced back to a simpler parent. Consider for example the clause $a(X) \leftarrow b(X), b(X)$. All solutions and refinements to $a(X) \leftarrow b(X), b(X)$ are solutions and refinements to $a(X) \leftarrow b(X)$. A case where we can prune clauses through syntactic analysis is therefore the basic case where a literal appears duplicate in a clause.

Semantic Redundancy Other examples of self-redundant clauses can be found using background knowledge. We may know of some degenerate cases when a literal is always or never satisfiable. We may also have extensional information

on a predicate stating whether it is reflexive, associative, or commutative. Last, we generalize this concept through the notion of entailment between sub-goals.

We consider reflexivity as an example of two degenerate cases, a valid literal or an unsatisfiable literal:

Definition 2 (Tautology) *A literal L_i is tautologically redundant in C if L_i is true for all possible instantiations.*

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 3 (Contradiction) *A literal L_i is a contradiction in C if $C \setminus L_i$ is satisfiable and C is 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 4 (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 $L_j = p_i(B_1, \ldots, B_{ja})$ 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 $mult(2, X, Z)$ is a commutative redundant literal.

Definition 5 (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 $L_j \wedge L_k \vDash L_i$*

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

The previous declarations are easy to compute and do capture some of the most common cases of redundancy. The price is that we are restricted to some very specific cases. We next move to more general definitions:

Definition 6 (Direct Equivalence) *A literal L_i is 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 direct equivalent redundant literal we drop the compatibility constraint on the literals. For instance, the literal $X \leq 1$ is equivalent redundant in the clause $p(X) \leftarrow 1 > X, X \leq 1$ since $1 > X \equiv X \leq 1$. This is another type of semantic redundancy.

We can move one step further and consider entailment.

Definition 7 (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 8 (Direct Entailment) *A literal L_i is direct entailed redundant in C if there is a sub-sequence of literals SC in $C \setminus L_i$ 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).

Our definitions do not guarantee that all the refinements of a redundant clause with a redundant literal L_i are themselves redundant. In the case of equivalence, a sufficient condition to guarantee that the clause belongs to $\mathcal{R}*$ is that all variables in the right hand of the equivalence also appear in the left-hand side L_i, and vice-versa: if two literals are equivalent, all their instances will also be equivalent. In the case of entailment, it is sufficient that all variables in the left-hand side appear in L_i. Otherwise we could refine the entailing literal to a more specific set of literals which would not entail L_i.

2.2 Context-Derived Redundancy

When considering contextual redundancy we are manipulating the set of clauses (hypotheses) found so far $\mathcal{S}$, instead of sets of literals as in self redundancy:

Definition 9 (Context-derived redundant clause) *The clause C is contextual redundant in $\mathcal{S} \cup \{C\}$ if $\mathcal{S} \models C$.*

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

Definition 10 (Duplicate) *A clause C is duplicate redundant in $\mathcal{S} \cup \{C\}$ if $C \in \mathcal{S}$.*

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

Definition 11 (Commutativity) *The clause C is commutative redundant in $\mathcal{S} \cup \{C\}$ if there is a compatible clause $D \in \mathcal{S}$ with the same literals of C but with a different ordering such that $C \equiv D$.*

For instance, $C = p(X) \leftarrow a(c, X), a(d, X)$ is a commutative redundant clause in $\mathcal{S}$ if $\mathcal{S}$ contains $p(X) \leftarrow a(d, X), a(c, X)$.

Definition 12 (Transitivity) *The clause C is transitive redundant in $\mathcal{S} \cup \{C\}$ if there are two compatible clauses D and E in $\mathcal{S}$ such that*

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

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

Definition 13 (Direct Equivalence) *The clause C is directly equivalent redundant in $\mathcal{S} \cup \{C\}$ if there is a compatible clause $D \in \mathcal{S}$ 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 direct equivalent literal in D*

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

Definition 14 (Direct Entailment) *The clause C is directly entailed redundant in $\mathcal{S} \cup \{C\}$ if there is a literal L_C in C and a compatible clause $D \in \mathcal{S}$ such that*

1. *L_C is a direct entailed redundant literal in D (for some SC)*
2. *$D \setminus SC = C \setminus L_C$.*

Consider $\mathcal{S} = \{D = p(X) \leftarrow X \leq 1, X \geq 1\}$. 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.

The types of redundancy just described, both self and contextual redundancy, form an hierarchy as illustrated in Figure 1, in the Appendix.

3 Handling Redundancy

In this section we describe how and where the redundancy types identified in the previous section can be eliminated in ILP systems that follow a top-down search approach.

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

3.1 Possible approaches

We envisage several approaches to incorporate the information provided by the expert in ILP systems to avoid the generation of redundant hypotheses. In a nutshell, either we can take advantage of user-defined constraint mechanisms, or user-defined pruning, or we can improve the refinement operator.

A first approach could be the extension of user-defined constraint mechanisms available in some systems (e.g., Progol [16], Aleph [12], Indlog [17]). The constraints are added by a user in the form of clauses that define when an hypothesis should not be considered. Integrity constraints are currently used to prevent the generation of self redundant clauses containing contradiction redundant literals. Note that an "... integrity constraint does not state that a refinement of a clause that violates one or more constraints will also be unacceptable." [12]. Thus the constraint mechanism is not suitable for our needs since we want to discard refinements of redundant clauses.

Another approach to eliminate redundant literals could be through user-defined pruning declarations that some ILP systems accept (e.g., Progol [16], Aleph [12], IndLog [17]). 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 a user to provide such rules defining when an hypothesis should be discarded (pruned). The use of pruning greatly improves the efficiency of ILP systems since it leads to a reduction of the size of the search space. User-defined pruning declarations to encode meta-knowledge about redundancy has the advantage that they can easily be incorporated into ILP systems that support user-defined pruning. However, in our opinion, one should try to eliminate the redundancy as a built-in procedure of the refinement operator instead of using mechanisms like user-defined pruning since the first option should be more efficient.

A third approach is the modification of the refinement operator and the literal generation procedure to allow the use of information provided by the expert. This is the approach followed and is next described.

3.2 Redundancy declarations

To eliminate the generation of redundant hypotheses we modified the refinement operator and literal generation procedure to exploit redundancy meta-knowledge provided by an expert. We modified the April [18] ILP system to accept the redundancy declarations that we next describe. The declarations are provided to the system as background knowledge in the form of Prolog rules. We chose the April system due to our knowledge regarding its implementation but we should note that the work described is also applicable to other ILP systems.

We start by describing the declarations that a user may pass to the literal generation procedure. Duplicate, commutative, and 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. This information helps the ILP system to avoid the generation of hypotheses with commutative redundant literals. As an example consider that we inform the ILP system that the predicate adj(X,Y) is commutative (e.g., :-commutative(adj(X,Y),adj(Y,X))). That information can be used to prevent the generation of commutative equivalent literals such as $adj(X, 2)$ and $adj(2, X)$.

The equiv declaration allows an 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.

An ILP system using equivalence declarations needs only to consider one literal of each equivalence class. There is redundancy in having commutative and equiv declarations, since we can define commutativity using only the equiv declaration. The main reason for this is to keep compatibility with other systems (e.g., Aleph). We point out that the described declarations allow the ILP system to avoid the generation of several types of self redundant hypotheses and contextual redundant hypotheses (direct equivalent redundant clauses).

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

The declaration contradiction can be used to avoid the generation of contradiction redundant hypotheses. The declaration has the form of contradiction([$L_1, \ldots, L_k$]), where $L_1, \ldots, L_k$ is the conjunction of literals that when appearing together in a clause makes it inconsistent. For instance, $contradi - ction([X < Y, X > Y])$ states that both literals can not occur in an hypothesis because they would turn it inconsistent.

The generation of transitive redundant literals and clauses can be avoided by providing information 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,Y),lt(Y,Z) will not be generated by the refinement operator.

Proper direct entailment redundant literals can be avoided with the knowledge that a literal implies another. The knowledge can be provided using declarations such as 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 the generation of hypotheses containing a literal like $lt(A, 2)$ followed by a literal like lt(A,1) (e.g.,p(X)← lt(A,2),lt(A,1)).

Direct entailed redundant literals or clauses can be prevented from being generated if we state that there is a set of literals such that each literal in the

Redundancy Type	Handled	Declaration	Example
Self / Tautological	literal generation	`:-tautology('≥'(X,X)).`	p(X)←X≥X
Self / Contradiction	refinement	`:-contradiction( ['>'(X,Y),'<'(X,Y)]).`	p(X)←X<2,X>2
Self / Commutative	literal generation	`:-commutative( mult(X,Y,R),mult(Y,X,R))`	p(X)← mult(X,3,R), mult(3,X,R)
Self & Contextual Transitivity	refinement	`:-transitive('>'(X,Y), '>'(Y,Z), '>'(X,Z))`	p(X)←X>Y,Y>Z
Self / Proper Direct Entailment	refinement	`semantic_rule('<'(A,B), '<'(A,C)):- C<B`	p(X)← X<0,X<2
Self & Contextual Direct Equivalence	literal generation	`:-equiv('<'(X,Y), '≥'(Y,X))`	p(X,Y)←X<Y,Y≥X
Self & Contextual Direct Entailment	refinement	`:-d_entail(['≤'(X,Y), '≥'(X,Y))],'='(X,Y)`	p(X,Y)←X≤Y,X≥Y, X=Y

Table 1. Redundancy Declarations

set implies the redundant literal. Such information can be provided with the declaration `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 ($1 \leq i \leq k$) and clauses containing all L_i. For instance, the clauses `p(X)`← `X`≤`1,X=1` or `p(X)`← `X`≤`1,X`≥`1` would not be generated if the expert provides a declaration like `d_entail([X`≤`Y,X`≥`Y],X=Y)`.

The unforeseen types of redundancy can be handled by the declaration `redundant(Literal,HypothesisBody):-Body`, where `Literal` is the literal to be added to the `HypothesisBody` and `Body` is a conjunction of literals that specify the conditions that make an hypothesis redundant. All types of redundancy could be handled by using this generic declaration. There are two main reasons that lead us to provide a set of declarations instead of a single declaration. The first reason is efficiency - ILP system implementors can devise more efficient ways of handling the redundancy described by each declaration than through a generic declaration of redundancy like `redundant`. The second reason is legibility - declarations with names that indicate the type of redundancy simplify the reading and the understanding of the background knowledge.

Table 1 summarizes the types of redundancy supported in our implementation, where redundancy is handled, examples of redundancy declarations and redundant hypotheses.

A final note on the implementation. Most of the declarations described (e.g., `transitive`, `contradiction`, `commutative`, `equiv`, and `d_entail`) are implemented by performing a matching between the literals in a declaration and the ones in a clause. The declaration `semantic_rule` is the only exception to this schema, since it involves the evaluation of the `body` of the semantic rule. The test if a declaration is applicable to a clause involves comparing (matching) the

Dataset	Characterization			April's Settings	
	$\mid E^+ \mid$	$\mid E^- \mid$	$\mid B \mid$/wrd	**i**	**noise**
krki I	342	658	1/1	1	10
krki II	3240	6760	1/1	1	10
mutagenesis	114	57	21/7	2	5
multiplication	37	24	3/2	2	0
range	19	14	1/1	1	0
proteins	848	764	24/8	2	90
ackermann	51	119	4/2	2	0

Table 2. Datasets characteristics and main experiment settings

literals in the clause with the ones in the declaration. The cost of such a test is linear on its size (number of literals), thus having a low computational cost.

4 Experiments and Results

The utility of the redundancy declarations presented in the previous section is empirically evaluated by determining the impact of their use in the execution time and number of hypotheses generated. Six datasets were used in the experiments. We selected datasets for which our knowledge, regarding the predicates in the background knowledge, enabled us to identify redundancy. The datasets were gathered from the Machine Learning repositories of Oxford[5] and York[6], and from one of the authors' home page[7].

Table 2 characterizes the datasets in terms of number of positive and negative examples as well as background knowledge size (number of predicates) and number of predicates **w**ith **r**edundancy **d**eclarations (*wrd*). 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 [19]. Finally, the parameter *noise* defines the maximum number of negative examples that an hypothesis may cover in order to be accepted. In all datasets the search was constrained to find hypotheses with a body containing a maximum of seven literals. The only exception was the *mutagenesis* dataset with a maximum number of literals set to 2. No limit was imposed on the number of hypotheses generated, and thus an exhaustive search within language restrictions was performed. In the appendix, Table 4, we show examples of redundancy declarations used in each dataset. The experiments were made on an AMD Athlon XP 1400+ processor PC with 512MB of memory, running the Linux Fedora (kernel 2.4.25) operating system. The ILP system used was the April [18] system version 0.9 together with the YAP Prolog engine version 4.4.

[5] `http://web.comlab.ox.ac.uk/oucl/research/areas/machlearn/`
[6] `http://www.cs.york.ac.uk/mlg/index.html`
[7] `http://www.fe.up.pt/~rcamacho/datasets/datasets.html`

Dataset	pwrd	Hypotheses			Time (sec.)		
		normal	red-decl	(%)	normal	red-decl	(%)
krki I	100	2,671	411	15	4.8	1.39	29
krki II	100	4,999	1,341	27	67.72	28.65	42
mutagenesis	30	3,823	3700	97	31,133	30,975	99
multiplication	66	2,163	1,032	47	2.03	0.96	47
range	100	5,257	303	6	6.3	0.15	2
proteins	33	1,470,742	1,303,351	88	499,638	457,582	91
ackermann	50	38,197	31,656	82	70.69	60,77	85

Table 3. Impact of using redundancy declarations (**red-decl**).*pwrd* is the percentage of predicates in the background knowledge with redundancy declarations.

Table 3 summarizes the performance of the April system using redundancy declarations and not using them. It shows the **p**ercentage of predicates **w**ith **r**edundancy **d**eclarations (*pwrd*), the total number of hypotheses generated, execution time, and the impact in the 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 for each dataset. However, it is important to remember that the accuracy of the models is not affected negatively since the redundancy information provided is used to eliminate $\mathcal{R}*$ redundant hypotheses. Note that the performance results for the *mutagenesis* dataset are very preliminary. In this dataset, due to time limitations, we had to constrain the search to find clauses using only up to two literals in the body. Nevertheless, the results show that even for such small search space we can obtain a slight improvement. For the remaining datasets considered, one can observe that redundancy declarations reduced the execution time and the number of hypotheses generated. The reductions were more substantial in the datasets with greater *pwrd*.

5 Conclusions

This work contributes to the effort of improving the efficiency of ILP systems. We studied the major forms of redundancy found in the search space of ILP applications and described how to avoid redundancy. 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. Experimental results show that performance improvements can be obtained by using redundancy declarations.

The major thrust of our work is to make ILP systems able to learn from large datasets. Most ILP systems are configured to generate a limited number of hypotheses. Therefore, avoiding redundant hypotheses may also lead to the generation of better hypotheses that otherwise would be lost due to the search

limit. We hope that this may result in an improvement of the quality of the induced models. Further experiments are required to better assess this claim, and to understand what redundancy declarations and how often each contributes to the reduction on the search space. It would be also interesting to see which types of redundancy can be detected automatically, perhaps in a pre-processing stage, in order to automatically generate the redundancy declarations.

Acknowledgments

We thank the anonymous reviewers for their useful comments. The work has been partially supported by project APRIL (Project POSI/SRI/40749/2001) and *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. M. Sebag and Rouveirol C. Tractable induction and classification in first order logic via stochastic matching. In *15th Int. Join Conf. on Artificial Intelligence (IJCAI'97)*, pages 888–893. Morgan Kaufmann, 1997.
3. 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.
4. H. Blockeel, L. Dehaspe, B. Demoen, G. Janssens, J. Ramon, and H. Vandecasteele. Improving the efficiency of Inductive Logic Programming through the use of query packs. *Journal of Artificial Intelligence Research*, 16:135–166, 2002.
5. V.S. Costa, A. Srinivasan, R. Camacho, H. Blockeel, and W. Van Laer. Query transformations for improving the efficiency of ilp systems. *JMLR*, 2002.
6. Rui Camacho. As lazy as it can be. In P. Doherty B. Tassen, P. Ala-Siuru and B. Mayoh, editors, *The Eighth Scandinavian Conference on Artificial Intelligence (SCAI'03)*, pages 47–58. Bergen, Norway, November 2003.
7. 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.
8. 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.
9. P.R.J. van der Laag. *An analysis of refinement operators in inductive logic programming*. PhD thesis, Erasmus Universiteit, Rotterdam, the Netherlands, 1995.
10. E.Y. Shapiro. *Algorithmic Program Debugging*. The MIT Press, 1983.
11. A. Srinivasan, R.D. King, and M.E. Bain. An empirical study of the use of relevance information in inductive logic programming. *JMLR*, 2003.
12. Ashwin Srinivasan. Aleph manual, 2003.
13. C. J. Hogger. *Essentials of Logic Programming*. Oxford University Press, 1990.

14. S.-H. Nienhuys-Cheng and R. de Wolf. *Foundations of Inductive Logic Programming*, volume 1228 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 1997.
15. G. Gottlob and C.G. Fermuller. Removing redundancy from a clause. *Artificial Intelligence*, 61(2):263–289, June 1993.
16. 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.
17. R. Camacho. *Inducing Models of Human Control Skills using Machine Learning Algorithms*. PhD thesis, Department of Electrical Engineering and Computation, Universidade do Porto, 2000.
18. 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.
19. S. Muggleton and C. Feng. Efficient induction in logic programs. In S. Muggleton, editor, *Inductive Logic Programming*, pages 281–298. Academic Press, 1992.

Appendix

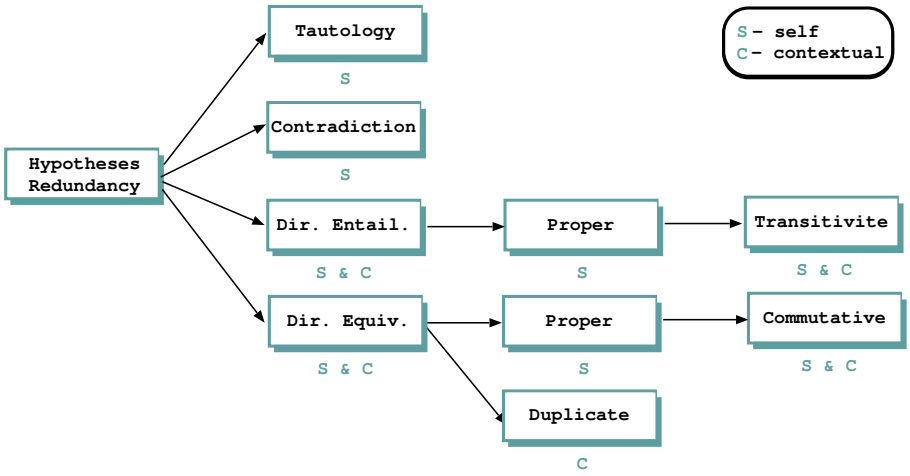

Fig. 1. Redundancy hierarchy

Datasets	**Declarations**
Krki I & II	:-commutative(adj/2). :-tautology(adj(X,X)).
multiplication	:- equiv(mult(A,B,C),mult(B,A,C)). :- equiv(plus(A,B,C),plus(B,A,C)).
range	semantic_rule(lt(A,B),lt(A,C)):- number(B),number(C), C<B. semantic_rule(lt(A,B),lt(C,B)):- number(A),number(C), C<A.
proteins	:-transitive(lth). :-transitive(ltv). semantic_rule(lth(A,B),ltv(B,A)). semantic_rule(ltv(A,B),lth(B,A)). :-d_entail([very_hydrophobic(A)],aromatic_or_very_hydrophobic(A)). :-d_entail([aromatic(A)],aromatic_or_very_hydrophobic(A)). :-d_entail([small(A)],small_or_polar(A)). :-d_entail([polar(A)],small_or_polar(A)). :-d_entail([aromatic_or_very_hydrophobic(A),not_very_hydrophobic(A)],aromatic(A)). :-d_entail([small_or_polar(A),large(A)],polar(A)).
ackermann	:-transitive(incr(A,B),decr(B,A),'='(A,B)).
mutagenesis	:-tautology(gteq(X,X)). :-tautology(lteq(X,X)). semantic_rule(gteq(A,B),gteq(A,C)):- number(B),number(C), B>C. semantic_rule(gteq(A,B),gteq(C,B)):- number(A),number(C), A>C. :-transitive(lteq). :-transitive(gteq). :-commutative(connected/2). :-d_entail([benzene(X,Y)],ring_size_6(X,Y)). :-d_entail([carbon_5_aromatic_ring(X,Y)],ring_size_5(X,Y)). :-d_entail([carbon_6_ring(X,Y)],ring_size_6(X,Y)). :-d_entail([hetero_aromatic_6_ring(X,Y)],ring_size_6(X,Y)). :-d_entail([hetero_aromatic_5_ring(X,Y)],ring_size_5(X,Y)).

Table 4. Examples of redundancy declarations used on the experiments