A Summary of “Pairwise saturations in inductive logic programming”

Max Robinson MAX.ROBINSON@JHU.EDU
Johns Hopkins University,
Baltimore, MD 21218 USA

1. Introduction

2. Related Work

The early ideas of inductive logic programming (ILP) can be tied back to Gordon Plotkin’s thesis (1971) on “Automatic methods of inductive inference”. The thesis explores an incremental algorithm for solving model inference problems, which are similar to problems in ILP. In his thesis, Plotkin develops a bottom-up generalization strategy called Relative least general generalizations (rlggs). This generalization was then built upon later.

Stephen Muggleton, who Drole and Kononenko reference often, later entered the field. The paper “Inductive Logic Programming” (1991) develops the concept of logic programming with machine learning, which is the foundation for this subfield. Muggleton from then on became one of the staples for the ILP field and as a result is referenced often by Drole and Kononeko.

Muggleton continued on to develop both ProGolem and Asymmetric Relative Minimal Generalisations (ARMGs) (Muggleton, Santos, & Tamaddoni-Nezhad, 2010). ProGolem is a framework written in Prolog for constructing bottom-clauses with Golem. Golem is a technique for constructing relative least general generalisations (rlggs) in an efficient way in order to conduct a search for a generalization that lead to a working hypothesis (?).

ARMG, as described by the authors, provides a bottom up approach to generalization without exponential growth of the size of the generalization. In bottom up approaches, this exponential growth of generalization size is one of the key hindrances of the approach. ARMG allowed for the bottom-up approach to be more viable.

3. Preliminaries

The following are eight definitions of concepts for inductive logical programming used to describe the pairwise saturations algorithm developed by the authors.

3.1 Definition 1 - Bottom Clause

The bottom clause $\perp_i$ of an example given background knowledge is the most specific clause for the hypothesis such that the background knowledge and the negation of the example entails the negation of the bottom clause. Symbolically, $B \wedge \neg e_i \models \neg \perp_i$

3.2 Definition 2 - Mode Declaration

Mode declaration is a way to assign roles to a predicate. For each argument position an argument is given a +, -, or # which corresponds to an input argument, output argument, or constant argument respectively. The point of mode declaration is to provide associations to the arguments for predicates about how instantiations of predicates can be constructed. Using an example, background knowledge, and mode declaration, a bottom clause can be found. The authors mention that Muggleton (1995) provided an algorithm to do so.

An example given of a bottom clause by the authors is:
$has\_daughter(ann) : -parent(ann, bob), parent(ann, cid), parent(ann, eve),$
$femail(ann), parent(bob, tina), male(bob), male(cid), femail(eve), femail(tina).$

3.3 Definition 3 - Depth of Values

The depth of values is calculated by the max depth of the literal's input arguments +1. Input values to the head clause have a depth of 0. Depth is the concept of how many input and output relations were made to produce a literal with those arguments. In other words, if H(A):- P(A,B), P'(B) , B has a depth of 1

3.4 Definition 4 - Depth of Literals

The depth of a literal in a clause is 0 if it is the head of the clause, and otherwise the max depth of a literal is the max depth of any of it's arguments + 1. Literals are then broken into layers based on their depth.

3.5 Definition 5 - Head-connected

A head connected clause means that all input values of a literal must show up in a previous layers output or as the input arguments to the head of the clause.

3.6 Definition 6 - Variabilization

Variabilization of a clause is the result of substituting values in input and output positions with variables. This just provides a generalization of the mode declaration rules and bottom clause.

3.7 Definition 7 - Asymmetric Relative Minimal Generalization (ARMG)

The ARMG of e_1 and e_2 is computed by creating the variabilization of e_1 and removing any predicates that are not covered in e_2. Covered means that the predicate does not hold true in the second example or does not exist in the second example. By removing these predicates the clause left is a general clause applying to both examples. ARMG is asymmetric meaning that $armg(e_1|e_2) = armg(e_2|e_1)$ is not true in general.

3.8 Definition 8 - Compatible Literals

Described by the authors, two literals are compatible if they conform to three conditions. One, they share the predicate symbol and arity. Two, they have the same values as their constant arguments. Three, for each input position in the preposition, the intersection of the instantiation sets for the argument is not the empty set. Symbolically, $IS[A_i] \cap IS[B_i] \neq \emptyset$

4. Pairwise saturations

The Pairwise saturation, developed by the authors, focuses on using constraints derived from AMRG to remove literals from the first example before the AMRG is computed. The constraints are developed from the idea that, for each literal that is an element in the hypothesis, there has to exist substitutions for that literal such that the literal is an element of bottom clause 1 and 2. The authors suggest that this will not be the case for all literals. There will be some literals that cannot be a part of a head connected hypothesis where a substitution exists for clause 2 for that literal. As a result, that literal can be removed. The constraints developed are constraints on constants and input variables.

4.1 Constraints on constants

The constraint for constants is based on the idea that a constant is not effected by substitutions for the hypothesis, and as a result, the position for a constant value in a predicate matters a lot. If a literal in the first bottom clause, $P(..., c_k, ...)$, has a constant c_k at position k, than if they hypothesis covers examples 1 and 2, than $P(..., c_k, ...)$ must exist in the second bottom clause.

No amount of substitution would fix the fact that there is a literal that does not match in both clauses because of a constant value and position. As a result, any literal of that form can be removed from bottom clause 1 when computing ARMG, since the literal cannot be part of the hypothesis.

The limitation of this method is that it does not hold for constraint logic programming. This is logic programming that allows for constraints such as less than or greater than. These less than or greater than constraints changes the fact that the constant could be different for a literal, but a valid general hypothesis can exist. A simple example is for inequalities. $a < 50$ and $b < 45$, a generalization is that b must be less than 50. This would not be found using constraints on constraints.

4.2 Constraints on input variables

The authors develop constraints on input variables using two main concepts: instantiation sets of values, and the compatibility of literals. An instantiation set of a value is the set of occurrences for that value in the bottom clause as either an input argument of the clause's head or an output argument of literals at the previous layers of the clause. For each occurrence, the predicate symbol is stored along with the position of that value in the predicates arguments, and the layer of the literal. An example instantiation set, from the authors, might be $IS(A) = \{< easbound/1/0 >\}$. This would denote that value A was the first positional argument for the head of the bottom clause. From here, the authors prove the following theorem.

Theorem 1
If a literal $L \in \perp_1$ has no compatible literal in $\perp_2$, its generalization cannot appear in $ARMG(e_2|e_1)$.

The limitation on this theorem is that the bottom clause must be "complete". In practice bottom clauses are constrained. As a result, there are cases where these constraints can cause literals to be omitted incorrectly from pairwise saturation. While the authors give examples of when this might occur, the authors do not discuss the impact this could have on different applications of pairwise saturation. It is left as an exercise to the reader too figure out if this use case applies to their data sets or implementation of pairwise saturation.

An algorithm that employs using these constraints prior to running ARMG using two example clauses is said to use pairwise saturation. The authors develop and provide such an algorithm. In providing this algorithm, the authors make the first mention that the algorithm for pairwise saturation uses caching of instantiation sets.

The authors then provide an extension of pairwise saturation, n-wise saturation. Instead of $L \in \perp_1$ only being tested for compatibility with literals in clause two, $\perp_2$, it is checked with literals in n clauses for compatibility. If no compatibility is found in one of the n clauses, then the literal can not pear in the generalized hypothesis from ARMG.

In order for n-wise saturation to work, it is assumed that all randomly selected clauses for compatibility checks are all covered by the same clause in the solution. This assumption means that the accuracy of the solution will likely decrease as the probability of this assumption being false increases. The chances of this assumption being false tends to increase as n increases.

4.3 Time complexity of pairwise saturation

As the main goal of the authors is to reduce the runtime of doing bottom up ILP through pairwise saturation, the authors provide a time complexity analysis for the pairwise saturation algorithm. The time complexity is broken down into three parts: constructing the instantiation sets, compatibility checking, and the full pairwise saturation algorithm.

Constructing instantiation sets must be done for each bottom clause, and is done as data is being read into the pairwise saturation algorithm and cached. Letting i be the arity of the predicate with the highest arity, and l be the maximal number of literals in a bottom clause. il insertion operation then must be performed to construct the set. The authors use a red-black tree the in their implementation and thus claim that insertion is $O(logn)$. As a result the total time complexity is $O(illog(il))$. Notice that the time complexity is data structure dependent. This time complexity might not hold if the same data structure is not used.

The complexity for compatibility was shown to be $O(i^2l)$ for the same meanings of i and l. Again, this complexity is based on the underlying structure of a red-black tree to give an intersection computation time that can be computed in $O(il)$.

Finally the entire pairwise saturation algorithm is analyzed. For pairwise saturation, each literal in clause 1 is checked for a matching literal in clause 2. Given the worse case where the number of literals in each clause is l this is an $O(l^2)$ operation. Each check then requires a compatibility check, thus making the final algorithm $O(l^3i^2)$.

Notice that the time to load the data into the cache is not included in the calculation for run complexity calculation for pairwise saturation. This means that it is a small part, but it is also not considered part of the core algorithm, but provides large benefits, as shown in the results.

5. Analysis and Discussion of Results

References

Drole, M., & Kononenko, I. (2017). Pairwise saturations in inductive logic programming. *Artificial Intelligence Review*, *47*(3), 395–415.

Muggleton, S., Santos, J., & Tamaddoni-Nezhad, A. (2010). Progolem: A system based on relative minimal generalisation. In De Raedt, L. (Ed.), *Inductive Logic Programming*, pp. 131–148, Berlin, Heidelberg. Springer Berlin Heidelberg.