A Summary of "Pairwise saturations in inductive logic programming"

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

1. Introduction

Inductive logic programming (ILP) (1991) is a subdomain of machine learning first developed explored in 1971 (Plokin, 1971). ILP deal a lot with concept learning. In other words, the goal is to generate a hypothesis or a concept given a set of positive examples, negative examples, and a set of background knowledge. The hypothesis is described in a logical program using the predicates in the background knowledge. The goal of the hypothesis is to be true for all positive examples and false for all negative example.

One large appeal for the use of ILP is that all data used to generate a solution, and the solution are all human readable. All pieces follow the typical logical programming paradigm. This makes it easier to encode domain-specific knowledge and to check that a solution produced makes sense if checked by domain experts. An unfortunate downside to ILP are the long run times typically associated with finding a solutions.

In the paper "Pairwise saturation in inductive logic programming" (Drole & Kononenko, 2017), the authors explore a technique for reducing the running time of systems that use asymmetric relative minimal generalizations (ARMG) (Muggleton et al. 2010) called pairwise saturation. With Pairwise saturation, the authors explore analyzing the bottom clauses of examples with the goal of removing literals that can be identified as not being part of the ARMG of the example. The removing of literals prior to computing the ARMG aims to reduce the run time of calculating hypotheses.

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. Muggleton (1994) developed 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 (Muggleton & Feng, 1990).

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 $\bot_i$ 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, $B \wedge \neg e_i \models \neg \bot_i$

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 shows that it is a small part, considered to almost be preprocessing, but provides large benefits, as shown in the results.

5. Analysis and Discussion of Results

The authors main goal was to show if the addition of pairwise saturation to the ProGolem speeds up the rate at which hypothesis are found with minimal loss to accuracy. The authors used three metrics to characterize their results: achieved accuracy, time needed to arrive at solution, and the size of the bottom clause which was used as the basis for the solution. Note, no configuration, computer, load, OS, or other details about the system on which the experiments were run was provided.

A comparison for results is made between ProGolem, ProGolem with n-wise saturation for n between 2 and 5 called ProParGolem, and ProGolem with caching of the bottom clause. The experiments used "real-world" datasets which come from different chemical domains. Data for eleven chemicals were used in the experiments. The goal is to predict the effects of the different compounds in different domains. The authors only reference the dataset through a url, http://www.doc.ic.ac.uk/jcs06/GILPS/ datasets.tar.bz2, which is now defunct. Experiments are performed in general by using 10x tenfold cross validation except in special cases due to data size.

The experiments reported average run time to produce a hypothesis and the accuracy of the hypothesis each with standard deviations, showing ProGolem being significantly slower than ProParGolem. The authors claim no significant difference between ProParGolem with n=2 and ProGolem. However, their claims are left to the reader to analyze the data in the table provided.

The authors then compare the ratio of time taken above ProParGolem where n=2, for ProGolem and ProGolemCaching. The authors conclude that caching creates an average speed up of 1.22 and caching plus pairwise saturation has a 1.36 times speed up on average. No detailed information about standard deviations is provided.

Comparison of the average length of the clause used to generate further hypotheses from shows that typically for more n used in n-wise saturation, the length of the clause submitted decreases. This is correlated by the authors with a decrease in the run time taken as n increased.

The paper concludes by claiming that an algorithm for reducing the size of the initial hypothesis in a system using ARGM has been shown. Pairwise saturation is explained to lead to faster induction because the solution is dependent on the size of the initial clause for generalization. In addition, it is noted that for the datasets that the additional time needed to compute the pairwise saturation does not exceed the time gained due to a shorter saturation.

The experiments run showed primarily metrics for the amount of time taken for each algorithm to run, on average. In addition to not providing information about the system on which the experiments were run, the authors spend their detailed comparison between ProGolem and ProParGolem. This makes it more difficult to understand what the role of caching had in overall performance and made the comparison seem stilted. Individual data points were also lightly analyzed, and the average times were used to highlight the desired results. There were few mentions of the data points which showed that there was little to no difference between ProGolemCaching and ProParGolem.

Little analysis was done on separating the running time taken for pairwise saturation and the amount of time then taken by the rest of the ProGolem algorithm. While this is not strictly necessary, it would have provided two important data points. One would be to show experimentally that the time complexity for pairwise saturation was correct. Second, the authors could then have done an analysis of exactly how much a smaller initial clause effected the time needed to construct a hypothesis. The experiments show that a smaller clause is important to run time, but they do not show how much. As stated by the authors it is clearly enough to offset the time it takes to run the pairwise saturation.

The author's claims that they have developed a system that decreases the runtime to find a hypotheses using ARMG, are loosely founded. The experiments show that for their particular system, whatever it may have been, that they were able to improve run times. The ability to reproduce these results seem shaky. Producing an algorithm that will produce shorter bottom clauses from which ARMG can then run was well developed. The proof and constraints created by the authors show that there is the ability to apply n-wise saturation to decrease the size of the bottom clause.

Ultimately, pairwise saturation looks to be a promising algorithm. The choice in runtime as a metric seems well intentioned by misguided and poorly executed. The foundation of the algorithm was built well, however. Pairwise saturation provides an interesting staring point for continued research into bottom clause size reduction, and its effects.

References

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

Muggleton, S., & de Raedt, L. (1994). Inductive logic programming: Theory and methods. *The Journal of Logic Programming*, *19-20*, 629 – 679. Special Issue: Ten Years of Logic Programming.

Muggleton, S., & Feng, C. (1990). Efficient induction of logic programs. In *Proceedings of the first conference on Algorithmic Learning Theory*, pp. 368 – 381.

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.

Plokin, G. (1971). *Automatic methods of inductive inference.* Ph.D. thesis, Edinburgh University, U.K.