Combining Genetic Programming and Inductive Logic Programming using Logic Grammars

Man Leung Wong
Department of Computing and Decision Sciences
Lingnan University, Tuen Mun
Hong Kong
mlwong@ln.edu.hk

Kwong Sak Leung
Department of Computer Science and Engineering
The Chinese University of Hong Kong
Hong Kong
ksleung@cse.cuhk.edu.hk

Abstract

Genetic Programming (GP) and Inductive Logic Programming (ILP) have received increasing interest recently. Since their formalisms are so different, these two approaches cannot be integrated easily though they share many common goals and functionalities. A unification will greatly enhance their problem solving power. In this paper, a framework to combine GP and ILP is presented. The framework is based on a formalism of logic grammars and a system called LOGENPRO (the LOgic grammar based GENetic PROgramming system) is developed. It is so flexible that programs in different programming languages such as LISP, Prolog, and C can be induced.

The performance of LOGENPRO in inducing logic programs from noisy examples is also evaluated. A detailed comparison to FOIL and mFOIL has been conducted. The experiment demonstrates that LOGENPRO is a promising alternative to other inductive logic programming systems and sometimes is superior for handling noisy data.

1. Introduction

Genetic Programming [4] [5] and Inductive Logic Programming [8] have received increasing interest recently. Genetic Programming (GP) is a method of automatically inducing S-expression in LISP to perform specified tasks. Inductive Logic Programming (ILP) investigates the construction of logic programs from examples and background knowledge. Since their formalisms are so different, these two approaches cannot be integrated easily although their properties and goals are similar. If they can be combined in a common framework, then many of the techniques and theories obtained in one approach can be applied in the other one.

In this paper, a framework that can combine GP and ILP is presented. This framework is based on a formalism of logic grammars and a system called LOGENPRO (the LOgic grammar based GENetic PROgramming system) is developed. It is very flexible and programs in various programming languages such as LISP, Prolog, Fuzzy Prolog and C can be induced.

The next section presents the formalism of logic grammars. Section three is a description of LOGENPRO. The fourth section describes a combination of LOGENPRO and FOIL [8] that learns logic programs and is followed by a conclusion.

2. Logic grammars

The LOgic grammars based GENetic PROgramming system (LOGENPRO) can induce programs in various programming languages such as LISP and Prolog. It accepts logic grammars of different languages and produce programs in these languages. Logic grammars are context sensitive and are the generalizations of context free grammar (CFG). Their expressivenesses are much more powerful than that of CFG, but equally amenable to efficient execution. In this paper, logic grammars [6] will be described in a notation similar to that of definite clause grammars (DCG). The logic grammar in table 1 will be used throughout sections two and three. Instead of using a grammar for a logic programming language, a grammar for S-expression in Lisp is employed to illustrate flexibility that various programming languages can be specified by logic grammars. The grammar for a logic programming language can be found in section 4.

A logic grammar differs from a CFG in that the logic grammar symbols, whether terminal or non-terminal, may include arguments. The arguments can be any term in the grammar. A term is either a logical variable, a function or a constant. A variable is represented by a question mark ? followed by a string of letters and/or digits. A function is a grammar symbol followed by a bracketed n-tuple of terms and a constant is simply a 0-arity function. Arguments can be used in a logic grammar to enforce context-dependency. Thus, the permissible forms for a constituent may depend on the context in which that constituent occurs in the program. Another application of arguments is to construct tree structures in the course of parsing, such tree structures can provide a representation of the semantics (meaning) of the program.

The terminal symbols, which are enclosed in square brackets, correspond to the set of words of the language specified. For example, the terminal

`[(+ ?x ?y)]` creates the constituent `(+ 1.0 2.0)` of a program if ?x and ?y are instantiated respectively to 1.0 and 2.0. Non-terminal symbols are similar to literals in Prolog, `"exp-1(?x)"` in table 1 is an example of non-terminal symbols. Commas denote concatenation and each grammar rule ends with a full stop.

Table 1: A simple logic grammar

```
1:start   ->        [(*], exp(X), exp(X), [)].
2:start   ->        {member(?x,[X, Y])}, [(*], exp-1(?x),
                    exp-1(?x), [)].
3:start   ->        {member(?x,[X, Y])}, [(/], exp-1(?x),
                    exp-1(?x), [)].
4:exp(?x)    ->     [(+ ?x 0)].
5:exp-1(?x) ->      {random(0,1,?y)}, [(+ ?x ?y)].
6:exp-1(?x) ->      {random(0,1,?y)}, [(- ?x ?y)].
7:exp-1(?x) ->      [(+ (- X 11) 12)].
```

The right-hand side of a grammar rule may contain logic goals and grammar symbols. The goals are pure logical predicates for which logical definitions have been given. They specify the conditions that must be satisfied before the rule can be applied. For example, the goal `member(?x, [X, Y])` in figure 1 instantiates the variable ?x to either X or Y if ?x has not been instantiated, otherwise it checks whether the value of ?x is either X or Y. If the variable ?y has not been bound, the goal `random(0, 1, ?y)` instantiates ?y to a random floating point number between 0 and 1. Otherwise, the goal checks whether the value of ?y is between 0 and 1. The special non-terminal `start` corresponds to a program of the language. The number before each rule is a label for later discussions. It is not part of the grammar.

3. The LOgic grammars based GENetic PROgramming system (Logenpro)

The problem of inducing programs can be reformulated as a search for a highly fit program in the space of all possible programs in the language specified by the logic grammar. In LOGENPRO, populations of programs are genetically bred [2] [3] using the Darwinian principle of survival and reproduction of the fittest along with genetic operations appropriate for processing programs. LOGENPRO starts with an initial population of programs generated randomly, induced by other learning systems, or provided by the user. Logic grammars provide declarative descriptions of the valid programs that can appear in the initial population. A high-level algorithm of LOGENPRO is presented in table 2.

One of the contributions of LOGENPRO is in the representations of programs in different programming languages appropriately so that initial population can be generated easily and the genetic operators such as reproduction, mutation and crossover can be performed effectively. A program can be represented as a derivation tree that shows how the program has been derived from the logic grammar. LOGENPRO applies deduction to randomly generate programs and their derivation trees in the language declared by the given grammar. These programs form the initial population. For example, the program `(* (+ X 0) (+ X 0))` can be generated by LOGENPRO given the logic grammar in table 1. Its derivation tree is depicted in figure 1.

Table 2: The high level algorithm of LOGENPRO

1. Generate an initial population of programs.
2. Execute each program in the current population and assign it a fitness value according to the fitness function
3. If the termination criterion is satisfied, terminate the algorithm. The best program found in the run of the algorithm is designated as the result.
4. Create a new population of programs from the current population by applying the reproduction, crossover, and mutation operations. These operations are applied to programs selected by fitness proportionate or tournament selections.
5. Rename the new population to the current population.
6. Proceed to the next generation by branching back to the step 2.

Alternatively, initial programs can be induced by other learning systems such as FOIL [8] or given by the user. LOGENPRO analyzes each program and creates the corresponding derivation tree. If the language is ambiguous, multiple derivation trees can be generated. LOGENPRO produces only one tree randomly.

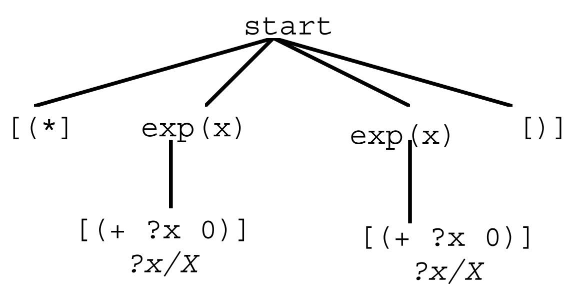

Fig. 1. A derivation tree of a program

Another contribution is the efficient algorithm for performing crossover between two derivation trees. The crossover is a sexual operation that starts with two parental programs and their corresponding derivation trees. Before performing any crossover, LOGENPRO check whether the offspring produced is valid according to the grammar. Thus, only valid offspring are produced and the genetic operation can be achieved effectively and efficiently.

4. Learning logic programs from imperfect data

Various approaches for ILP differ mainly in the search strategy and the heuristics used to guide the search. However, only a few ILP systems such as FOIL [8] and mFOIL [1] address the issue of learning from imperfect data which is significant in Knowledge discovery from databases [7]. Wong and Leung [9] have developed the Genetic Logic Programming System (GLPS) that can induce logic programs from noisy examples using Genetic Algorithms.

This section describes the application of LOGENPRO to learn logic programs from noisy and imperfect training examples. An empirical comparison of LOGENPRO with FOIL (the publicly available version of FOIL, version 6.0 , is used in the experiment) and mFOIL [1] in the domain of learning illegal chess endgame positions from noisy examples is presented. mFOIL is based on FOIL that has adapted several features from the CN2 learning algorithm, such as the use of the Laplace and m-estimate as a search heuristic and the use of significance testing as a stopping criterion. Moreover, mFOIL uses beam search and can apply mode and type information to reduce the search space. In the experiment, the value of the m parameter of mFOIL is set to 0.01, the beam width is 5 and the significance threshold is 0. These values were found to be appropriate for the chess endgame problem.

In this experiment, LOGENPRO employs a variation of FOIL to find the initial population of logic programs. Thus, it uses the same noise handling mechanism of FOIL. The variation is called BEAM-FOIL because it uses a beam search method rather than the greedy search strategy of FOIL. BEAM-FOIL produces a number of different logic programs when it terminates and the best program among them is the solution of the problem. The logic programs created by BEAM-FOIL are used by LOGENPRO to initialize the first generation. In order to study the effect of genetic operations performed by LOGENPRO on the initial programs produced by BEAM-FOIL, a comparison between them is also presented.

Table 3: The logic grammar for the chess endgame problem

```
start       ->          clauses.
clauses     ->          clauses, clauses.
clauses     ->          clause.
clause      ->          consq, [:-], antes, [.].
consq       ->
            [illegal(WKf, WKr, WRf, WRf, BKf, BKr)].
antes       ->          antes, [,], antes.
antes       ->          ante.
ante        ->
            {member(?x,[WKf, WKr, WRf, WRf, BKf, BKr])},
            {member(?y,[WKf, WKr, WRf, WRf, BKf, BKr])},
            literal(?x, ?y).
literal(?x, ?y)     ->          [?x = ?y].
literal(?x, ?y)     ->          [ ~ ?x = ?y].
literal(?x, ?y)     ->          [ adjacent(?x, ?y) ].
literal(?x, ?y)     ->          [ ~adjacent(?x, ?y) ].
literal(?x, ?y)     ->          [ less_than(?x, ?y) ].
literal(?x, ?y)     ->          [ ~less_than(?x, ?y) ].
```

In the chess endgame, the setup is white king and rook versus black king [8]. The target predicate `illegal(WKf, WKr, WRf, WRr, BKf, BKr )` states whether the position where the white king is at `(WKf, WKr)`, the white rook at `(WRf, WRr)` and the black king at `(BKf, BKr)` is not a legal white-to-move position. The background knowledge is represented by two predicates, `adjacent(X, Y)` and `less_than(W, Z)`, indicating that rank/file `X` is adjacent to rank/file `Y` and rank/file `W` is less than rank/file `Z` respectively. LOGENPRO uses the logic grammar in table 3 for this problem. In the grammar, `[adjacent(?x, ?y)]` and `[less_than(?x, ?y)]` are terminal symbols. The logic goal `member(?x, [WKf, WKr, WRf, WRr, BKf, BKr])` will instantiate logic variable `?x` of the grammar to either `WKf`, `WKr`, `WRf`, `WRr`, `BKf`, or `BKr`, the logic variables of the target logic program.

The training set contains 1000 examples (336 positive and 664 negative examples). The testing set has 10000 examples (3240 positive and 6760 negative examples). Different amounts of noise are introduced into the training examples in order to study the performances of these systems in learning concepts from noisy environment. To introduce n% of noise into argument `X` of the examples, the value of argument `X` is replaced by a random value of the same type from a uniform distribution, independent to noise in other arguments. For the class variable, n% positive examples are labeled as negative ones while n% negatives examples are labeled as positive ones. Noise in an argument is not necessarily incorrect because it is chosen randomly, it is possible that the correct argument value is selected. In contrast, noise in classification implies that this example is incorrect. Thus, the probability for an example to be incorrect is $1-\{[(1-n\%)+n\%*\frac{1}{8}]^6*(1-n\%)\}$. In this experiment, the percentages of introduced noise are 5%, 10%, 15%, 20%, 30% and 40%. Thus, the probabilities for an example to be noisy are respectively 27.36%, 48.04%, 63.46%, 74.78%, 88.74% and 95.47%. Background knowledge and testing examples are not corrupted with noise.

A chosen level of noise is first introduced in the training set. Logic programs are then induced from the training set using LOGENPRO, FOIL, mFOIL, and BEAM-FOIL. Finally, the classification accuracy of the learned logic programs is estimated on the testing set. For BEAM-FOIL, the size of beam is ten and thus ten logic programs are returned by it. The population size for LOGENPRO is 10 and the maximum number of generations is 50. In fact, different population sizes have been tried and the results are still satisfactory even for a very small population. This observation is interesting and it demonstrates the advantage of combining inductive logic programming and genetic programming using the proposed framework. The fitness function evaluates the number of training examples misclassified by each individual in the population. Fifty runs of the above experiments are performed on different training examples. The results of the three systems are summarized in figure 2.

From this experiment, the classification accuracy of these systems degrades seriously as the noise level increases. The results were statistically evaluated using the paired *t*-test. For each noise level, each pair of systems was compared to determine if their difference in accuracy were statistically significant at the 99.95% confidence interval. The classification accuracy of LOGENPRO is better than that of FOIL. The differences are significant at the 99.95% confidence interval at all noise levels (except the noise level of 0%). The largest difference reaches 24% at the 20% noise level. Similarly, the accuracy of LOGENPRO is significantly better than that of mFOIL and BEAM-FOIL when the noise level is on or below 30%. On the other hand, the accuracy of mFOIL and BEAM-FOIL is better than that of LOGENPRO at the 40% noise level. But, the differences are not significant at the 99.95% confidence interval. It implies that the genetic operations of LOGENPRO can actually improve the logic programs provided by BEAM-FOIL.

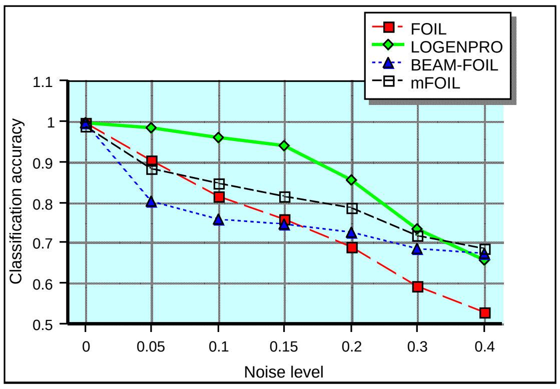

Fig. 2. Comparison between LOGENPRO, FOIL, mFOIL, and BEAM-FOIL

The result is surprising because LOGENPRO uses the same noise handling mechanism of FOIL. One possible explanation of the better performance of LOGENPRO is that the Darwinian principle of survival and reproduction of the fittest is a good noise handling method. It avoids overfitting noisy examples, but at the same time, it can finds interesting and useful patterns from noisy and imperfect examples.

5. Conclusion

We have proposed a framework for combining Genetic Programming and Inductive Logic Programming. This framework is based on a formalism of logic grammars. To implement the framework, a system called LOGENPRO (LOgic grammar based GENetic PROgramming) has been developed. The performance of LOGENPRO in inducing logic programs from imperfect training examples is evaluated using the chess endgame problem. A detailed comparison to FOIL and mFOIL has been conducted. Together with many other experiments we have completed with similar results, this experiment illustrates that LOGENPRO is a promising alternative to other inductive logic programming systems and sometimes is superior for handling noisy data.

6. Reference

[1] Dzeroski, S. and Lavrac, N. (1993). Inductive Learning in Deductive Databases. *IEEE Transactions on Knowledge and Data Engineering*, **5**, pp. 939-949.

[2] Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning*. MA: Addison-Wesley.

[3] Holland, J. H. (1975). *Adaptation in natural and artificial systems*. Ann Arbor: The University of Michigan Press.

[4] Koza, J. R. (1992). *Genetic Programming: on the Programming of Computers by Means of Natural Selection*. Cambridge, MA: MIT Press.

[5] Koza, J. R. (1994). *Genetic Programming II: Automatic Discovery of Reusable Programs*. Cambridge, MA: MIT Press.

[6] Pereira, F. C. N. and Warren, D. H. D. (1980). Definite Clause Grammars for Language Analysis - A Survey of the Formalism and a Comparison with Augmented Transition Networks. *Artificial Intelligence*, **13**, pp. 231-278.

[7] Piatetsky-Shapiro, G. and Frawley, W. J. (1991). *Knowledge Discovery in Databases*. Menlo Park, CA: AAAI Press.

[8] Quinlan, J. R. (1990). Learning logical definitions from relations. *Machine Learning*, **5**, pp. 239-266.

[9] Wong, M. L. and Leung, K. S. (1995). Genetic Logic Programming and Applications. *IEEE Expert*. Accepted to be published.