Constraint-Driven Learning of Logic Programs

Rolf Morel
University of Oxford
rolf.morel@cs.ox.ac.uk

Abstract

Two fundamental challenges in program synthesis, i.e. learning programs from specifications, are (1) program correctness and (2) search efficiency. We claim logical constraints can address both: (1) by expressing strong requirements on solutions and (2) due to being effective at eliminating non-solutions. When learning from examples, a hypothesis failing on an example means that (a class of) related programs fail as well. We encode these classes into constraints, thereby pruning away many a failing hypothesis. We are expanding this method with failure explanation: identify failing sub-programs the related programs of which can be eliminated as well. In addition to reasoning about examples, programming involves ensuring general properties are not violated. Inspired by the synthesis of functional programs, we intend to encode correctness properties as well as runtime complexity bounds into constraints.

Introduction

The sub-field of program synthesis known as inductive logic programming (ILP) (Muggleton 1991) uses logical methods to learn logic programs from examples. ILP represents the examples, background knowledge (BK), and hypotheses as logic programs (sets of clauses, i.e. sets of logical rules). A typical ILP problem is to find a program composed of BK, that is, provided functions and predicates, such that this program is correct on a set of positive and negative examples. The fundamental challenge of ILP is to efficiently find a solution in a huge hypothesis space.

After a brief overview of related systems, we summarize our approach to facing this challenge. Subsequently, we discuss ongoing work on integrating failure explanation into our framework. As future work we intend to use constraints to go beyond example-based specifications. We look at refinement types as a way of encoding functional correctness guarantees. Finally, we consider the potential of runtime complexity bounds for eliminating intractable hypotheses.

Related Work

One approach to learning hypotheses is to learn them clause by clause, one at a time. Systems using this strategy typically use a set covering algorithm (Muggleton 1995; Srinivasan 2001; Ahlgren and Yuen 2013). By relying heavily on having been provided good examples, these systems tend to be quite efficient. This comes at the cost of learning overly specific solutions and struggling to learn recursive programs (Cropper, Dumancic, and Muggleton 2020).

Recently, encoding the ILP problem as a satisfiability problem has become popular (Law, Russo, and Broda 2014; Kaminski, Eiter, and Inoue 2018; Evans and Grefenstette 2018). Typically, it is possible for these systems to learn optimal and recursive programs. Their performance hinges on the effectiveness of conflict-driven clause learning in modern SAT solvers. However, these systems tend to scale badly, especially in terms of the size of the examples' domain.

Learning from Failures

In this past year, myself and Andrew Cropper have worked on introducing the *learning from failures* (LFF) approach to ILP (Cropper and Morel 2020). In LFF, an ILP system employs three distinct stages: *generate*, *test*, and *constrain*. In the generate stage, a logic program hypothesis is generated such that no *hypothesis constraint* (which restrict the syntactic form of hypotheses) is violated. In the test stage, the hypothesis is tested against training examples. A hypothesis *fails* when a negative example is entailed or when a positive example is not. Programs related by subsumption to a failing hypothesis provably fail as well. In the constrain stage, constraints are derived that eliminate these related hypotheses. If a negative example is entailed, the constraints prune generalisations of the hypothesis. If a positive example is not entailed, the constraints prune specialisations of the hypothesis. The three stages follow each other iteratively in a loop until a hypothesis is found entailing all positive examples and no negative example.

One way of understanding LFF is by seeing the learning of constraints as externalizing the way that clauses are learned from conflicts inside SAT solvers. We detect conflicts by testing hypotheses. My implementation of the *Popper* system embodies this idea. Popper maintains an answer set programming (ASP) formula whose models correspond to Prolog programs. In the first stage, a model is obtained and converted to a program. The program is tested on all positive and negative examples using Prolog. When a hypothesis fails, Popper adds hypothesis constraints to its ASP

formula, thereby narrowing down the space of hypotheses. Popper's scalability is in large part due to our ASP formula being ignorant of the examples' domain.

Popper supports infinite domains, arbitrary data structures, learning textually minimal programs, and learning recursive programs. The ILASP systems (Law, Russo, and Broda 2020) have a similar generate-test-and-constrain loop, though more tightly integrated. ILASP3's notion of *coverage constraints* is similar to our hypothesis constraints. However, whereas ILASP3 requires the entire space of clauses to be pre-computed to derive these constraints Popper does so from singular programs.

Experiments with number theory, robot strategies, and list transformation problems show that constraints can drastically improve learning performance, and that state-of-the-art ILP systems such as ILASP and Metagol (Cropper, Morel, and Muggleton 2019) are outperformed by Popper.

Explaining Failures

Central to the scientific method is experimentally testing hypotheses. A failing hypothesis prompts scientists to rule out extensions of it. Moreover, a scientist will try to *explain* the failure in order to eliminate even more hypotheses. In a paper to be submitted in the upcoming months, we introduce failure explanation techniques for program synthesis. Given a logic program hypothesis, we test it on examples. When a hypothesis fails, we identify clauses and literals responsible for the failure. The identified sub-programs allow us to eliminate more hypotheses that also provably fail. I have developed an automatic failure explanation algorithm based on analysing SLD-trees. Unlike the ILASP and ProSynth (Raghothaman et al. 2020) systems, my $Popper_X$ system can identify responsible literals within clauses.

I will show that identifying sub-programs is effective in eliminating more hypotheses. I experimentally evaluate the introduction of failure explanation to the Popper ILP system. My results indicate that explaining failures can drastically reduce the size of the hypothesis space as well as learning times.

Future Work

Examples, e.g. in the form of test cases, are not the sole guide towards correct programs. Often programmers ensure that a program under construction does not violate important properties. In the future I want to encode refinement types (Polikarpova, Kuraj, and Solar-Lezama 2016; Feng et al. 2018) as constraints. These types serve as (over-approximating) correctness properties of predicates, e.g. that the input and output of *reverse* have the same length. Functional program synthesis by Frankle et al. (2016) stands out in supporting both refinement types as well as examples, casting the latter into the framework of the former. My envisioned approach is vastly simpler, as these properties, expressed as atoms in a background theory, can be directly integrated with our hypothesis constraints.

Knoth et al. (2019) use a strategy similar to that of Frankle et al. for incorporating resource bound specifications into a type-based synthesis system. A runtime complexity bound might say that a solution for *reverse* should take at most linear time with respect to its input argument. The above proposed approach should apply here as well: the relevant background atoms, representing resource bounds, can be enabled and disabled based on the hypothesis the solver is currently building up.

The hope is that the refinement type project should take no more than half a year. With most of the machinery in place, the runtime complexity bounds project should take less than half a year. In all, I anticipate showing that straightforward, primarily declarative algorithms can solve synthesis problems too difficult for the state-of-the-art, in ILP and beyond.

References

Ahlgren, J.; and Yuen, S. Y. 2013. Efficient program synthesis using constraint satisfaction in inductive logic programming. *J. Mach. Learn. Res.* 14(1): 3649–3682.

Cropper, A.; Dumancic, S.; and Muggleton, S. H. 2020. Turning 30: New Ideas in Inductive Logic Programming. *IJCAI 2020*.

Cropper, A.; and Morel, R. 2020. Learning programs by learning from failures. *Machine Learning* In Press, available http://arxiv.org/abs/2005.02259.

Cropper, A.; Morel, R.; and Muggleton, S. 2019. Learning higher-order logic programs. *Machine Learning* .

Evans, R.; and Grefenstette, E. 2018. Learning Explanatory Rules from Noisy Data. *J. Artif. Intell. Res.* 61: 1–64.

Feng, Y.; Martins, R.; Bastani, O.; and Dillig, I. 2018. Program synthesis using conflict-driven learning. *PLDI 2018*.

Frankle, J.; Osera, P.; Walker, D.; and Zdancewic, S. 2016. Example-directed synthesis: a type-theoretic interpretation. *POPL 2016*.

Kaminski, T.; Eiter, T.; and Inoue, K. 2018. Exploiting Answer Set Programming with External Sources for Meta-Interpretive Learning. *TPLP* 18(3-4): 571–588.

Knoth, T.; Wang, D.; Polikarpova, N.; and Hoffmann, J. 2019. Resource-guided program synthesis. *PLDI 2019*.

Law, M.; Russo, A.; and Broda, K. 2014. Inductive Learning of Answer Set Programs. *JELIA 2014*.

Law, M.; Russo, A.; and Broda, K. 2020. The ILASP system for Inductive Learning of Answer Set Programs. *CoRR* abs/2005.00904.

Muggleton, S. 1991. Inductive Logic Programming. *New Generation Comput.* 8(4): 295–318.

Muggleton, S. 1995. Inverse Entailment and Progol. *New Generation Comput.* 13(3&4): 245–286.

Polikarpova, N.; Kuraj, I.; and Solar-Lezama, A. 2016. Program Synthesis from Polymorphic Refinement Types. *POPL 2016*.

Raghothaman, M.; Mendelson, J.; Zhao, D.; Naik, M.; and Scholz, B. 2020. Provenance-guided synthesis of Datalog programs. *POPL 2020*.

Srinivasan, A. 2001. The ALEPH manual. *Machine Learning at the Computing Laboratory, Oxford University* .