Automated Debugging with Tractable Probabilistic Programming

Aniruddh Nath and **Pedro Domingos**
Department of Computer Science & Engineering
University of Washington
Seattle, WA 98195, U.S.A.
{*nath, pedrod*}*@cs.washington.edu*

Abstract

Probabilistic programming languages allow domain experts to specify generative models in a high-level language, and reason about those models using domain-independent algorithms. Given an input, a probabilistic program generates a distribution over outputs. In this work, we instead use probabilistic programming to explicitly reason about the distribution over *programs*, rather than outputs. We propose *Tractable Probabilistic Programs* (TPP), a language to represent rich probabilistic dependencies between different parts of a program; we make use of the recent work on sum-product networks to ensure that inference remains tractable. We explain how TPP can be applied to the problem of automated program debugging; given a corpus of buggy programs, a TPP model can be learned to capture a probability distribution over the location of the bug. The model can also incorporate additional sources of information, such as coverage statistics on test suites. We also briefly outline how TPP can be used to solve the more ambitious problem of *fault correction*, i.e. predicting the most probable true program conditioned on a buggy one. The ability to learn common patterns of bugs and incorporate multiple sources of information potentially makes TPP useful as a unifying framework for automated program debugging.

Introduction

The term 'probabilistic programming' refers to the practice of specifying probabilistic models in a high-level language, and reasoning about them using domain-independent algorithms. This paradigm decouples model design from algorithm design, simplifying the job of the domain expert (who no longer needs to be involved in algorithm design) and making algorithmic advances more widely applicable.

Existing probabilistic programming languages are designed to answer probabilistic queries about the program's output variables, conditioned on some input (evidence). A common design for probabilistic programming languages is to add stochastic primitive functions to an existing deterministic language (e.g. sampling from Bernoulli or Gaussian distributions). This allows users to easily define sophisticated probability distributions over the output variables.

However, probabilistic programs can also been seen as distributions over deterministic programs (e.g. Church (Goodman et al. 2008) programs are distributions over Scheme programs, and ProbLog (De Raedt, Kimmig, and Toivonen 2007) programs are distributions over Prolog programs). Seen through this lens, existing probabilistic programming languages define trivial distributions over deterministic languages, typically modeling a probabilistic program as a set of independent distributions over statements.

In this paper, we consider the design of probabilistic programming languages that allow probabilistic dependencies among statements. Specifically, we define *Tractable Probabilistic Programs* (TPP), a language for defining rich probability distributions over programs. TPP uses Sum-Product Networks (SPNs; Poon and Domingos 2011) to model these rich dependencies, while ensuring that probabilistic queries about the program can be computed in polynomial time. The design of TPP is not tied to the choice of a particular deterministic language; TPP allows the user to define a probabilistic version of any deterministic language.

This view of probabilistic programming is well-suited to tackling the problem of automated debugging, a challenging task of great practical importance. Automated debugging has been an active research area for several decades (Shapiro 1983); the bulk of the effort has gone towards the problem of *fault localization*. A well-established approach in the software engineering community is to use coverage statistics on test suites to compute 'suspiciousness scores' independently for each statement (e.g. Jones and Harrold 2005). In the following section, we outline how TPPs can incorporate this coverage data into a rich probabilistic model that captures dependencies between statements, rather than predicting the suspiciousness of each line independently.

Tractable Probabilistic Programming

A *TPP* for a deterministic language L defines a probability distribution over programs in L. The TPP contains a *rule SPN* for each production rule $\alpha \rightarrow \beta$ in L's grammar (α is a non-terminal symbol; β is a string of terminals and non-terminals). The leaves of this SPN are univariate distributions over terminals in β, and sub-SPNs for each non-terminal in β. The SPN may also include leaf distributions over additional variables, which we refer to as *rule attributes*. These rule attributes can be used to represent variables of interest that are not part of the program itself. In the fault localization setting, the important attribute is the vari-

able 'Buggy', which is true iff the line contains a bug. Other informative features may also be included as attributes; for instance, one or more coverage-based suspiciousness metrics (Jones and Harrold 2005) may be included for each line.

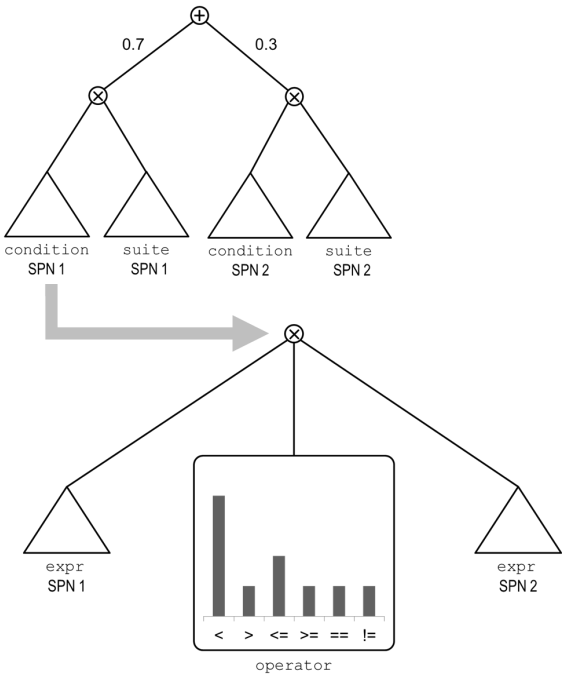
Intuitively, the sum nodes in the SPN can be thought of as choices between alternative subprogram distributions. The product nodes capture context-specific independence between subprograms.

Example 1. The following production rules are a fragment of the grammar of a Python-like language:

`while_stmt` → `'while' condition ':' suite`

`condition` → `expr operator expr`

The following are simple SPNs for these two rules:



Like many statistical relational representations, a TPP is a lifted specification of a probabilistic model, and must be *grounded* for a specific mega-example. In this case, a mega-example is the parse tree of a program, along with the corresponding rule attributes. Sub-SPNs for non-terminals are grounded recursively. The resulting ground SPN is linear in the size of the parse tree, and inference on the SPN is guaranteed to be tractable.

Learning

TPPs build on ideas similar to Relational Sum-Product Networks (Nath and Domingos 2014), and their structure can be learned similarly, using an extension of the LearnSPN algorithm (Gens and Domingos 2013). Alternatively, the rule SPN structure may be fixed, and the parameters trained by EM. A simple option is to use a PCFG-like structure, where each non-terminal is modeled by a sum over a fixed number of classes, where the sum node can have different parameters in each production rule that the symbol appears in. TPPs are not restricted to PCFG-like structures, and can compactly represent some high-treewidth models (due to their ability to compactly represent context-specific independence).

An implementation of the above learning algorithm is currently in development, and we are in the process of running experiments on a corpus of buggy Python programs. We use the TARANTULA score (a coverage-based metric; Jones and Harrold 2005) as a rule attribute. We hypothesize that the learned TPPs will capture context-specific dependencies between the coverage-based metrics and the probability of a bug (e.g. a high TARANTULA score is more informative inside a top-level `if` statement than inside a `for` loop).

Current Directions and Future Work

The above discussion deals with the use of TPPs for fault localization. TPPs can enrich existing fault localization methods by learning recurring patterns of errors from a corpus of buggy programs, and combining multiple sources of information (e.g. different suspiciousness metrics).

TPPs can also be used for the more ambitious problem of fault *correction*. In the simplest case, the correct and buggy program have identical parse trees, and TPPs can be used to correct local bugs in the terminal symbols (e.g. incorrect operators or variable names). In this case, the buggy terminals are included as rule attributes, and the correct program is predicted by querying the MAP state of the true terminals.

A more challenging case is when the parse tree of the true program is unknown a priori. In this setting, fault correction can be done by searching over parse trees when grounding a TPP, instead of taking a fixed parse tree as input. To retain tractability, the search space must be restricted (e.g. reordering blocks of code, or allowing insertions or deletions of bounded size).

Combining these (and potentially other) approaches in a single unifying framework for automated debugging could have significant practical importance, given the large economic cost of software errors. TPPs may also have applications in natural language processing, and other domains that use grammars.

Acknowledgments

This research was partly funded by ARO grant W911NF-08-1-0242, ONR grants N00014-13-1-0720 and N00014-12-1-0312, and AFRL contract FA8750-13-2-0019.

References

De Raedt, L.; Kimmig, A.; and Toivonen, H. 2007. Problog: A probabilistic Prolog and its application in link discovery. In *IJCAI*.

Gens, R., and Domingos, P. 2013. Learning the structure of sum-product networks. In *ICML*.

Goodman, N. D.; Mansinghka, V. K.; Roy, D. M.; Bonawitz, K.; and Tenenbaum, J. B. 2008. Church: a language for generative models. In *UAI*.

Jones, J. A., and Harrold, M. J. 2005. Empirical evaluation of the Tarantula automatic fault-localization technique. In *ASE*.

Nath, A., and Domingos, P. 2014. Learning tractable statistical relational models. In *StaR-AI*.

Poon, H., and Domingos, P. 2011. Sum-product networks: A new deep architecture. In *UAI*.

Shapiro, E. Y. 1983. *Algorithmic Program Debugging*. MIT Press.