Formal Methods For Probabilistic Programming

Daniel Selsam, Percy Liang, David L. Dill
Stanford University

1 Motivation

Probabilistic programming systems are infamously difficult to test [5, 11]. Most properties of interest only hold probabilistically, and so on any particular set of runs a correct implementation may produce unexpected output while an incorrect implementation may produce the expected output. To make matters worse, the expected output is rarely known in the first place. While a small subset of a probabilistic programming language may permit exact inference by other means—for example, dynamic programming can be used for exact inference on polytrees—most of the complexity in a probabilistic programming system will likely only apply to programs that make use of more sophisticated language features, such as branching, recursion, and black-box simulators. There may be no way to know a priori how inference is supposed to behave on such programs.

2 A New Methodology

In Selsam et al. [13] we demonstrated a practical methodology for building machine learning systems that enables developers to find and eliminate implementation errors systematically without recourse to empirical testing using an *interactive proof assistant* [3, 4, 6, 9, 10, 1, 2]. In our approach, developers first state a formal theorem in the proof assistant defining what it means for their system to be correct. Once they implement a draft of the system, they try to prove the correctness theorem using the proof assistant. The process of interactive proving exposes implementations errors systematically by yielding impossible proof obligations, and once all implementation errors have been fixed, the developers can complete the formal proof and be certain that the implementation is correct.

Our methodology was motivated by the challenges faced when implementing and debugging the probabilistic programming system Venture [8]. As a case study, we implemented a simple probabilistic programming system, Certigrad, in the Lean Theorem Prover [2] and formally proved that the implementation satisfies many semantically deep correctness properties. Selsam et al. [13] discussed the challenges involved in designing the libraries of background mathematics, in developing formal specifications from informal ones, and in constructing formal proofs. Here we describe the semantics of Certigrad in more detail and discuss the verification of a probabilistic program transformation. The complete development can be found at `www.github.com/dselsam/certigrad`.

3 Semantics of Certigrad

The semantics of Certigrad are based on the abstraction of stochastic computation graphs (SCGs) [12]. It is designed for stochastic optimization only and does not support conditioning or inference. We represent an SCG by a list of nodes, where each node contains an identifier, a (possibly stochastic) operation, and a list of identifiers representing its parents in the graph. An SCG also includes a list of identifiers representing the inputs to the graph, and a list of identifiers representing the nodes that are considered losses that we want to minimize. We denote an SCG into a probability distribution

over scalars by interpretation, building an environment that maps identifiers to tensors as we go along, and returning the sum of the loss nodes in the final environment. Specifically, we recurse on the list of nodes in the graph, starting with an environment whose keys include the inputs to the graph. As a base case, when we reach the end of the list of nodes, we evaluate the sum of the loss nodes in the final environment and inject the result into the probability monad (as a δ distribution). There are two recursive cases: deterministic nodes and stochastic nodes. When we reach a deterministic node, we evaluate the node in the environment and inject the result into the probability monad. When we reach a stochastic node, we create the primitive probability distribution that samples from the node's operator given the current environment. In both cases, we also create a lambda with a fresh variable representing the value of the current node, map the node's identifier to the fresh variable in the environment, and then recurse on the remaining nodes. We then apply the *bind* constructor for probability distributions to the original distribution and the recursively computed one to yield a distribution that represents the entire subgraph. Since we implemented this denotation as an actual function in Lean, we can prove formal theorems about the distributions induced by Certigrad programs.

4 Verifying probabilistic program transformations

Semantics-preserving program transformations are an essential tool for optimizing programs written in traditional programming languages. Such transformations hold even more promise in probabilistic programming languages since they admit weaker notions of program equivalence and thus admit a broader class of acceptable transformations. Moreover, there are many interesting properties that we may wish to optimize for besides the resources consumed by a simulation, such as the *variance* of the simulation's outputs. In the case of Certigrad, we consider a program transformation sound even if it changes the induced distribution of the SCG as long as it preserves its expected value, and our primary goal for transforming the program is to lower the program's variance. Among the probabilistic program transformations we verified is one that integrates out the KL-divergence from a standard normal multivariate Gaussian distribution to an arbitrary multivariate Gaussian distribution. This transformation is used to derive the autoencoding variational Bayes model (AEVB) [7] from a naïve variational autoencoder, but can also be applied to any program that estimates the KL divergence empirically. We formally proved that the transformation preserves the expected loss, i.e. that $\forall g, \mathbf{E}[g] = \mathbf{E}[\text{integrateKL}(g)]$.[1] The proof consists of two parts: first we derive the closed form expression of the integral in question, and then we use that identity to prove the main result by induction.

5 Getting Started

Interactive proof assistants are sophisticated systems with many components that have no analogues in traditional programming languages. One of the main costs in applying our methodology is the one-time cost of learning how to use such a tool. Extensive tutorials for the Lean Theorem Prover can be found at `https://leanprover.github.io/documentation/`. Another cost of applying our methodology is developing formal proofs of the stated lemmas and theorems. Most theorems of interest are outside the known decidable fragments of logic and so cannot be reliably automated. However, pragmatic developers can pay this cost as desired by only formally proving the properties that they most fear do not hold. Given the difficulty of detecting implementation errors in probabilistic programming systems, we think developers of such systems may often prefer the cost of proving to the cost of testing, and to the ongoing stress of never knowing if the system in question is actually correct.

[1] We decided to verify this transformation after the author discovered that an incorrect sign had persisted in his implementation of AEVB for days without detection.

References

[1] Coq Development Team. *The Coq proof assistant reference manual: Version 8.5*. INRIA, 2015-2016.

[2] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. The Lean theorem prover (system description). In *Automated Deduction-CADE-25*, pages 378–388. Springer, 2015.

[3] Michael JC Gordon. Edinburgh lcf: a mechanised logic of computation. 1979.

[4] Michael JC Gordon and Tom F Melham. Introduction to hol a theorem proving environment for higher order logic. 1993.

[5] Roger B. Grosse and David K. Duvenaud. Testing MCMC code. *CoRR*, abs/1412.5218, 2014. URL `http://arxiv.org/abs/1412.5218`.

[6] John Harrison. Hol light: A tutorial introduction. In *International Conference on Formal Methods in Computer-Aided Design*, pages 265–269. Springer, 1996.

[7] D. P. Kingma and M. Welling. Auto-encoding variational Bayes. *arXiv*, 2014.

[8] Vikash Mansinghka, Daniel Selsam, and Yura Perov. Venture: a higher-order probabilistic programming platform with programmable inference. *arXiv preprint arXiv:1404.0099*, 2014.

[9] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer, 2002.

[10] Sam Owre, John M Rushby, and Natarajan Shankar. Pvs: A prototype verification system. In *Automated DeductionCADE-11*, pages 748–752. Springer, 1992.

[11] Alexey Radul. On testing probabilistic programs. `https://alexey.radul.name/ideas/2016/on-testing-probabilistic-programs/`. Accessed: 2017-10-16.

[12] John Schulman, Nicolas Heess, Theophane Weber, and Pieter Abbeel. Gradient estimation using stochastic computation graphs. In *Advances in Neural Information Processing Systems*, pages 3528–3536, 2015.

[13] Daniel Selsam, Percy Liang, and David L. Dill. Developing bug-free machine learning systems with formal mathematics. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 3047–3056, International Convention Centre, Sydney, Australia, 06–11 Aug 2017. PMLR. URL `http://proceedings.mlr.press/v70/selsam17a.html`.