INTERACTIVE WRITING AND DEBUGGING OF BAYESIAN PROBABILISTIC PROGRAMS

JAVIER BURRONI, ARJUN GUHA, DAVID JENSEN

College of Information and Computer Sciences
University of Massachusetts Amherst

1. Introduction

In systems for Bayesian probabilistic programming such as BLOG, a user defines the generative model, presents the observations, and specifies the queries of interest. The program is then executed to obtain the desired results: posterior distributions over one or more queries. Obtaining results for a different query requires re-running the entire program. However, neither the data nor the generative model are changed by the new query, so performing inference in this case is inefficient. We show how to eliminate this inefficiency while simultaneously facilitating a more fluid exploration of the probabilistic model specified by the program. Our system is an implementation of BLOG [Milch et al., 2005] that allows users to interact with the full posterior distribution. This system is also the first interactive debugger for probabilistic programs.

One internal result of the execution of a BLOG program is the creation of a posterior distribution over possible worlds. Each possible world is a model structure of first-order logic that can be constructed from the given generative process [Milch et al., 2005]. We propose a system that exposes the posterior distribution of worlds to the user, providing the ability to query any element of the full posterior. When the worlds are restricted to ancestors of the observations, they are complete worlds, but removing this restriction may create partial instantiations (some variables may not be generated). Once the user takes control of the process, they may create and evaluate any number of queries. If the user provides a query that includes objects not yet instantiated, the system can instantiate that part of the world while paying only the cost of *forward inference*.

For the case when a joint probability distribution that can be represented as a Bayesian network, both *backward inference* and *forward inference* may be necessary to estimate a posterior distribution. *Backward inference* refers to the task of inferring the distribution of a set of variables given all their descendents, which is required to infer the value of an observation's *ancestors*. In contrast, *forward inference* refers to computing the posterior distribution of a set of variables given observations of all their *ancestors*[1]. This difference is relevant because *backward inference* is usually the more costly inference mechanism.[2] Our proposed system highlights this as it runs *backward inference* once and required *forward inferences* are executed in a lazily.

Date: December 19, 2017.

[1]It is important to notice that Bayesian learning happens only during *backward inference*. *Forward inference* is a way of forward simulation once the value of the variable's *ancestors* are determined.

[2]To use this idea in the context of BLOG, the idea must be extended to Contingent Bayesian Networks (CBN) with *self supporting instantiations*. See Theorem 1 of [Milch and Russell, 2012] for more details.

2. Inspect and query.

We introduce a new operator—`inspect`—and a modified operator—`query`—that allows BLOG users to explore the posterior distribution of worlds. Given a world ω, the `inspect(expr)` statement evaluates the BLOG expression `expr` in ω.Therefore, a user can use `inspect` to expose the value of any constant, function, or relation defined in ω. On the other hand, the usual semantics of `query(expr)` can be thought of as the application of `inspect(expr)` over a sample of the posterior distribution of worlds.

A step-by-step debugger can be built on top of `inspect`. Using the recursive definition of expressions, recursive application of `inspect` will act as step-by-step debugging. In that way, any expression can be traced to primitive random choices earlier in execution.

As the generative model is itself made of BLOG expressions, it can also be inspected. After randomly selecting a world ω, debugging the generative model is equivalent to recursively inspecting the composition of expressions that end in the `observe` statements. This will reveal the choices made by nature in ω to produce these observations. Additionally, the researcher may insert breakpoints to debug starting at a predefined location in the code. In that way, locally defined symbols—function arguments—can be inspected to gain insight regarding the generation of a specific output.

Unlike debugging non-probabilistic programs, users who debug probabilistic programs face a distribution over traces (possible worlds). The most obvious approach would be to randomly choose one world and perform step-by-step debugging. However, in addition to moving forward or backward in a specific trace, our approach supports *lateral* movements: moves to a randomly chosen ω_p such that $\omega_p \models p$ for a suitable predicate p. For example, if a user wants to understand why a given branch of an `if`-statement is taken, they can move to one or more worlds where the condition holds and explore each of those worlds using both `inspect` and `step`.

3. The impact of observations

With the ability to interactively execute any query, the user also gains the ability to inspect the effect of information—encoded through the use of `obs` statements—on any possible `query`. Given a model with an observation set $\mathcal{O}_n$ of size n, the system will take subsets $\mathcal{O}_0 \subseteq \mathcal{O}_k \subseteq \cdots \subseteq \mathcal{O}_n$, where k indicates the size of the subset and compute a posterior distribution over each subset. For instance, the posterior distribution over $\mathcal{O}_0$ will be equal to the prior distribution. That will show the sensitivity of posterior distributions of any `query` to observations. In Figure 1, the impact of observations can be seen in the "grades" model of Heckerman et al. [2004]. As more data is used, the probability for a student to get A or B decreases.

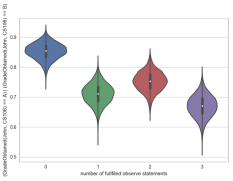

Figure 1. For the grades example from Heckerman et al. [2004], posterior distribution of the query `(GradeObtained(John, CS106) == A) | (GradeObtained(John, CS106) == B)` for the prior (zero observations) and when using at most three observe statements.

4. Future Work.

Although our implementation is for the BLOG PPL, it could be applied to other PPLs using information that they currently compute. Take for instance Anglican [Wood et al., 2014]. For its black-box variational Bayes algorithm, it stores information regarding random choices using `addresses` that uniquely identify checkpoints: sample and observe statements. With that information, the idea presented on this paper can be implemented. A more recent PPL, ProbTorch [Siddharth et al., 2017] exposes a `Trace` structure that should allow the efficient implementation of this debugger.

References

David Heckerman, Christopher Meek, and Daphne Koller. Probabilistic models for relational data. Technical report, Technical Report MSR-TR-2004-30, Microsoft Research, 2004.

Brian Milch and Stuart Russell. General-purpose MCMC inference over relational structures. *arXiv preprint arXiv:1206.6849*, 2012.

Brian Milch, Bhaskara Marthi, Stuart Russell, David Sontag, Daniel L. Ong, and Andrey Kolobov. BLOG: Probabilistic models with unknown objects. 2005.

N. Siddharth, Brooks Paige, Jan-Willem van de Meent, Alban Desmaison, Noah D. Goodman, Pushmeet Kohli, Frank Wood, and Philip Torr. Learning disentangled representations with semi-supervised deep generative models. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 5927–5937. Curran Associates, Inc., 2017.

Frank Wood, Jan Willem Meent, and Vikash Mansinghka. A new approach to probabilistic programming inference. In *Artificial Intelligence and Statistics*, pages 1024–1032, 2014.