Reasoning about Inference in Probabilistic Programs

Chandrakana Nandi
University of Washington, Seattle, WA
cnandi@cs.washington.edu

Adrian Sampson
Cornell University, Ithaca, NY
asampson@cs.cornell.edu

Dan Grossman
University of Washington, Seattle, WA
djg@cs.washington.edu

Todd Mytkowicz, Kathryn S. McKinley
Microsoft Research, Redmond, WA
{toddm, mckinley}@microsoft.com

Abstract

Complex applications such as search, robotics, and speech recognition operate over uncertain inputs using one or more uncertain machine learning models. While previous work shows that probabilistic programming languages can be used to model such systems efficiently [1], the above mentioned sources of uncertainty make debugging probabilistic programs challenging.

Some of the fundamental errors that occur in probabilistic programs are due to incorrect statistical models, ignoring dependence between random variables, or, using wrong hyper-parameters (such as sample size) for inference. To prevent these errors, the solution most probabilistic programming languages adopt is to keep inference *external* to the core language semantics. While this limitation avoids potential pitfalls, it prevents programmers from invoking inference at arbitrary points in the program and composing the results. For example, in the Uncertain⟨T⟩ programming model, inference happens implicitly when a conditional statement involving an Uncertain⟨T⟩ type is encountered. However, designers of large systems often need to use inference for practical reasons, such as, to efficiently transmit a probability distribution across the network or to decompose an intractable inference problem into smaller pieces.

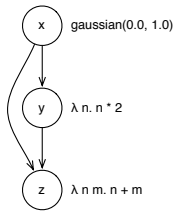
We present a programming model, FLEXI, that extends a typical probabilistic programming language [1, 4] with new constructs for invoking inference. To debug problems that arise from the use of inference, we use a *Decorated Bayesian network* to represent the program's distribution and its inference operations. The representation lets us detect and explain bugs from two categories: approximation and dependence.

Ignoring dependence

Ordinary probabilistic programming languages [1, 2, 4] all share a common structure: the program builds up probability distributions as values, and statistical inference is *external* to the language semantics. In FLEXI, in contrast, we make inference a part of the core language. Consequently, programmers can run statistical inference anywhere in the program and compose the results. To understand the semantics of statistical inference, first consider the semantics of ordinary operations on random variables in a probabilistic language [1].

```
Uncertain<double> x = gaussian(0.0, 1.0);
Uncertain<double> y = x * 2;
Uncertain<double> z = y + x;
```

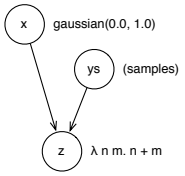
This short program demonstrates the two kinds of operations in an ordinary probabilistic language: creating new uncertain values using a random primitive (here, `gaussian`) and building up more complex uncertain values using operations on the underlying type. The latter feature corresponds to the *bind* operation on a probabilistic monad [3]: the language *lifts* a computation on **`double`** values to work on `Uncertain<`**`double`**`>` values instead. In our example, the expression `x * 2` creates a new `Uncertain<`**`double`**`>` value that encapsulates a doubling function over **`double`**s and `y + x` lifts the addition operator over **`double`**s to apply to pairs of `Uncertain<`**`double`**`>`s which produces this Bayesian network:



Now consider a program that uses MCMC sampling to estimate a distribution using 1000 independent executions:

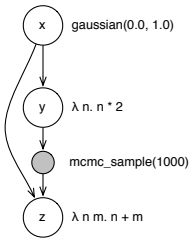
```
Uncertain<double> ys = mcmc_sample(y, 1000);
Uncertain<double> z = ys + x;
```

The type of `ys` is also `Uncertain<`**`double`**`>`. However the operation is very different: `mcmc_sample` executes the deferred computation in `y` multiple times, tabulates the results, and produce a "flat" value to encapsulate the result. The Bayesian network for this program is



This network no longer has a node corresponding to the variable `y`. Sampling replaced a subgraph with a single, independent node that contains the result of the MCMC inference process. This behavior—collapsing a distribution's Bayesian network representation into a "flat" representation—is not unique to MCMC sampling: it describes *any* inference procedure. Simple exhaustive enumeration, for example, inspects a Bayesian network to compute all the possible outputs from a distribution and produces a weighted list of possible outcomes. Sampling strategies from basic rejection sampling to more sophisticated Metropolis–Hastings implementations, all build an empirical depiction of a distribution from randomly selected outcomes.

In FLEXI, we define an *inference operation* as one that takes an arbitrary `Uncertain<T>` value, inspects its Bayesian network representation, and produces a new primitive `Uncertain<T>` value for the same `T`. The permission to inspect a value's *representation* sets inference operations apart from typical lifted operations on `Uncertain<T>` types, which do not have access to the underlying representation. To define the semantics for programs using inference operations, we extend the Bayesian network representation to include both normal nodes and inference nodes. The new representation, called a *decorated Bayesian network* (DBN), is a generalization: every Bayesian network is a DBN, and every DBN with no inference nodes is a Bayesian network. Graphically, the program above produces this DBN:



where the shaded circle is an inference node corresponding to the `mcmc_sample` inference operation. An inference node has exactly one in-edge—the distribution being inferred—and any number of out-edges indicating where the result is used. Inference nodes can have hyper-parameters: here, the `mcmc_sample` node has a sample-size parameter set to 1000. To execute a DBN, an evaluator traverses the inference nodes in topological order, extracts the sub-graph rooted at a given inference node, executes the inference node on that Bayesian network, and replaces the node with its resulting primitive distribution node. Depicting a probabilistic program's data flow using a DBN helps reveal the advantages and pitfalls of using inference. First, adding inference helps satisfy the engineering constraints in statistical software. By "flattening" a complex sub-graph in a Bayesian network to a simple data structure, inference can make subsequent operations more efficient, or it can let the program serialize a distribution for storage on disk or transmission on a network.

In contrast, adding inference introduces two potential sources of error: *dependence* errors and *approximation* errors. Dependence errors occur when an inference operation's "flattening" effect removes dependence edges from a graph. In our above example, replacing the sub-graph rooted at `y` with the primitive sampled node `ys` eliminates the dependence between `x` and `y`. As a consequence, the sum `z` may not match its original value. Approximation errors arise from the selection of hyper-parameters: if the `mcmc_sample` call uses too few samples, for example, the resulting `z` might not faithfully approximate the intended sum.

We can use the DBN representation to define both kinds of bugs. Both problems arise when the program's result does not match a version with inference removed. Intuitively, inference operations should be "semantic no-ops": all inference operations take in a given distribution and return a different representation of the same distribution that should be smaller and more efficient. Therefore, we can detect an inference-related bug by replacing an inference node in a DBN with a lifted identity function: a normal Bayesian network node whose result is the same as its argument. If a program fails to meet its statistical specification (i.e., a **`passert`**), and removing an inference node fixes a program's bad behavior, then we can "blame" that inference operation for the bug.

Conclusions

Programmers need to use statistical inference in probabilistic programs for practical reasons such as efficiency, but inference can introduce subtle correctness problems. We have identified a new category of bugs in probabilistic programming that arise from inference operations. To help automatically diagnose and fix these bugs, we propose a new programming model and tool that measures the correctness impact of inference operations.

References

[1] J. Bornholt, T. Mytkowicz, and K. S. McKinley. Uncertain<T>: A first-order type for uncertain data. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

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

[3] N. Ramsey and A. Pfeffer. Stochastic lambda calculus and monads of probability distributions. In *POPL*, 2002.

[4] A. Sampson, P. Panchekha, T. Mytkowicz, K. S. McKinley, D. Grossman, and L. Ceze. Expressing and verifying probabilistic assertions. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2014.