Debugging Probabilistic Programs

Chandrakana Nandi
Dan Grossman
University of Washington, Seattle, USA
{cnandi, djg}@cs.washington.edu

Adrian Sampson
Cornell University, Ithaca, USA
asampson@cornell.edu

Todd Mytkowicz
Microsoft Research, Redmond, USA
toddm@microsoft.com

Kathryn S. McKinley
Google, USA
mckinley@cs.utexas.edu

Abstract

Many applications compute with estimated and uncertain data. While advances in probabilistic programming help developers build such applications, debugging them remains extremely challenging. New types of errors in probabilistic programs include 1) ignoring dependencies and correlation between random variables and in training data, 2) poorly chosen inference hyper-parameters, and 3) incorrect statistical models. A partial solution to prevent these errors in some languages forbids developers from explicitly invoking inference. While this prevents some dependence errors, it limits composition and control over inference, and does not guarantee absence of other types of errors. This paper presents the FLEXI programming model which supports constructs for invoking inference *in* the language and reusing the results in other statistical computations. We define a novel formalism for inference with a *Decorated Bayesian Network* and present a tool, DePP, that analyzes this representation to identify the above errors. We evaluate DePP on a range of prototypical examples to show how it helps developers to detect errors.

CCS Concepts • **Software and its engineering** → **Syntax**; Error handling and recovery; • **Mathematics of computing** → *Maximum likelihood estimation*

Keywords Probabilistic programming, debugging, program analysis, statistical inference

1. Introduction

Estimated data is consumed and produced by a wide range of applications spanning machine learning, sensors, and approximate hardware. Insufficient programming languages, tools, and statistical expertise mean that these probabilistic programs may produce unexpected results at best and incorrect results in the worst case. In addition to the usual logic and other programming errors, probabilistic programs add their own set of debugging challenges—inference tasks involve approximations which propagate through programs and lead to incorrect outputs. Recent work offers developers some help in reasoning about the correctness of probabilistic programs [33, 34]. For example, programmers can write probabilistic assertions [33], but unlike traditional assertions which must *always* be true, probabilistic assertions are specified with a *probability* of being true in a given execution. Even though these assertions fail when the program's results are unexpected, they do not give us any information about the *cause* of the failure. To help determine the cause of failure, we identify three types of common probabilistic programming defects.

Modeling errors and insufficient evidence. Probabilistic programs may use incorrect statistical models, e.g., using Gaussian (0.0, 1.0) instead of Gaussian (1.0, 1.0), where Gaussian (μ, σ) represents a Gaussian distribution with mean μ and standard deviation σ. On the other hand,even if the statistical model is correct, their input data (e.g., training data) may be erroneous, insufficient or inappropriate for performing a given statistical task.

Ignoring dependence. A probabilistic program can have a dependence bug if 1) random variables are incorrectly treated as independent, or 2) inference tasks are composed incorrectly or performed too early in the system.

Incorrect hyper-parameters. Incorrectly chosen hyper-parameters in inference algorithms (e.g., sample size) can lead to *approximation* bugs or, conversely, wasted effort (computation time).

Some probabilistic programming languages [5, 7, 33] prevent some of these defects by making inference inaccessible to the programmer. Uncertain⟨T⟩ [7], for example, implicitly invokes a hypothesis test whenever the program performs a conditional on a random variable, and the result is a "flat," non-statistical Boolean. This simplicity comes at a cost in expressiveness: many application scenarios need to invoke inference at arbitrary points while giving statistical semantics to the inference results. For example, to the best of our knowledge, no current programming system supports composition of inference results produced on different machines. If worker machines in a data center application run independent statistical computations, they need a way to send their results over the network. In a traditional probabilistic programming language, the workers would need to either serialize an entire Bayesian network representation or exit the probabilistic domain and communicate "flat" values. By instead running inference, each worker machine could produce a compact statistical representation to send to a combining machine.

This paper presents a programming model called FLEXI (FLEXible Inference), in which inference is a first class citizen in the core language, similar to WebPPL [16] and Figaro [29]. Our implementation of inference samples from closed-form distributions (both discrete and continuous), or from computations that produce distributions. We give semantics to inference using the notion of a novel *Decorated Bayesian Network (DBN)*. To ensure that the resulting program does not have the previously mentioned bugs, we developed a debugger, DePP (Debugger for Probabilistic Programs) that exploits the DBN representation. To the best of our knowledge, DePP is the first tool of its kind. We implement some prototypical examples to evaluate DePP.

```
Uncertain<double> X = new Gaussian(0.0, 1.0);
Uncertain<double> Y = from x in X select x * 2;
Uncertain<double> Z = from y in Y
                      from x in X
                      select y + x;
if (Z < 6) {
  Console.Write("2-sigma rule holds for z.");}
```

Figure 1: Probabilistic program written in Uncertain⟨T⟩.

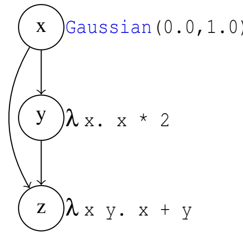

Figure 2: Bayesian network for entire program in Figure 1.

2. Background

This section provides an overview of Uncertain⟨T⟩ and probabilistic assertions on which our work is based.

2.1 Uncertain⟨T⟩

Uncertain⟨T⟩ is a probabilistic programming model built on top of C# and LINQ. It is a generic data type that can represent probability distributions over a numeric type, T. It overloads operators from LINQ such as `Select` and `SelectMany` for `Uncertain` types to construct Bayesian networks (which are directed acyclic graphs) where the nodes represent random variables and edges represent dependencies. A node can have parameters such as mean, variance, etc. that define the probability distribution being represented by the random variable. Figure 1 shows an example of a program written in Uncertain⟨T⟩. `x` $\sim$ Gaussian (0.0, 1.0), `y` $\sim$ Gaussian (0.0, 2.0) and `z` $\sim$ Gaussian (0.0, 3.0). Figure 2 shows its Bayesian network representations. The program creates a `Gaussian` distribution using a random primitive and performs arithmetic on it. It demonstrates the three kinds of operations allowed in Uncertain⟨T⟩: 1) creating a new uncertain value using a random primitive (line 1), 2) building up more complex uncertain values using operations on the underlying types (line 2-5), and 3) conditionals over the uncertain types (line 6).

2.2 Probabilistic Assertions

Since probabilistic programs may or may not produce correct outputs in all executions, they require a statistical view of correctness. Sampson et al. [33] introduced probabilistic assertions, or `passert`s, and an analysis to check them. Unlike classic program assertions which must be true in all program executions, `passert`s may only be true in some executions. We have added `passert`s to Uncertain⟨T⟩—we write them as `passert(e, p)` where `e` is an expression and `p` is the minimum probability with which the assertion must hold at 95% confidence level. The output of evaluating a `passert` in Uncertain⟨T⟩ is a boolean. Figure 3 presents the data obfuscation program from the passert [33] paper written in Uncertain⟨T⟩. The `passert` checks that the Gaussian noise added as obfuscation does not change the actual observation too much and returns `false` if it does.

3. Motivating Examples

This section illustrates how the three categories of failures described in the introduction arise in probabilistic programs.

3.1 Incorrect Models

In Figure 3, if we change the mean of the Gaussian from 0.0 to 1.0, the probability of the assertion being true reduces a lot. For a confidence of 95% and the same number of samples, the confidence interval for a mean of 1.0 is (0.49, 0.51) whereas the confidence interval for a mean of 0.0 is (0.84, 0.85). This example shows that choosing the right statistical model is crucial for ensuring that a probabilistic assertion passes.

3.2 Incorrect Data

Correct modelling cannot compensate for bad training data. Consider the temperature model in Figure 4 with a `passert` stating that the predicted temperature in Seattle, Washington should be less than 110 with probability 0.9. (The record high temperature for Seattle is 102°F set on July 29, 2009.)

```
double precise = 15493.5;
Uncertain<double> Noise = new Gaussian(0.0,1.0);
var Obfuscation = from n in Noise
                  select precise + n;
var Error = from o in Obfuscation
            select o - precise;
passert((Error < 1.0), 0.9);
```

Figure 3: Data obfuscation code.

```
double[] temperatures = load(training_data);
double[] weights = LearnWeights(temperatures);
double prediction = InferModel(test, weights);
passert (prediction < 110,  0.9);
```

Figure 4: Temperature prediction system.

In this example, if the training data mistakenly includes readings from Scottsdale, Arizona where high temperatures regularly reach 110°F in the summer, then no matter how accurate the statistical learning algorithms are, the resulting prediction will not be correct and the assertion will fail.

3.3 Ignoring Dependence by Treating Random Variables as Independent

Consider an implementation of a monitoring system involving a thermometer and a hygrometer in Figure 5. The monitoring system

```
List<Uncertain<double>> Temperature;
List<Uncertain<double>> Humidity;
var test = from t in Temperature.ElementAt(0)
           from h in Humidity.ElementAt(0)
           select (h > 90 & t > 77);
passert (test, 0.9);
```

Figure 5: Treating dependent data as independent.

records both temperature and humidity at certain intervals and sends them through a communication channel as (t,h) pairs. If the channel randomly induces noise ε in some of the readings such that ε for t and h within any given pair is equal, then the values of temperature and humidity received at the other end of the channel will not be

independent. In this program however, temperature and humidity are treated as independent random variables. Consequently, it may cause the `passert` to fail because the correlation between temperature and humidity is ignored.

3.4 Ignoring Dependence by Performing Inference Too Early

Consider the code in Figure 6 which is similar to Figure 1 but with a `passert`. In this program, both `y` and `z` depend on the *same* values of `x`. We can verify that the `passert` is true. Now consider

```
Uncertain<double> X = new Gaussian(0.0, 1.0);
Uncertain<double> Y = from x in X select x * 2;
Uncertain<double> Z = from y in Y
                      from x in X
                      select y + x;
passert(z < 6, 0.9)
```

Figure 6: Program from Figure 1 with a `passert`.

adding an explicit inference call invoking MCMC [15] sampling to approximate the distribution of `y` using 1000 independent executions as shown in Figure 7. Due to this operation, the information regarding the dependence of `z` and `y` on the same `x` is lost. The outcome of performing inference on `z` as `mcmc_sample(z, 1000)` would thus give us 1000 samples from `z` computed using separate sets of values drawn independently from `y` and `x`.

```
Uncertain<double> X = new Gaussian(0.0, 1.0);
Uncertain<double> Y = from x in X select x * 2;
Uncertain<double> Ys= Y.mcmc_sample(1000)
Uncertain<double> Z = from y in Ys
                      from x in X
                      select y + x;
passert(z < 6, 0.9)
```

Figure 7: Running inference early on an uncertain type leads to loss of dependence information.

Because the dependence between `z` and `y` is ignored, the `passert` may fail. This example shows that a call to an inference method (`mcmc_sample`) too early in the Bayesian network can lead to incorrect program output or failure of a `passert`. The potential for these types of errors is high in systems that for practical reasons, want to compose models and communicate results between them, rather than having one giant model. An example is a machine learning model with a very large and diverse training dataset.

3.5 Incorrect Hyper-parameters

Inference tasks involve *hyper-parameters* such as sample size, top-k etc. Their values impact the result of the inference procedure. Further, the choice of the values depends on how the model obtained after inference is going to be used. As an example, in a program with multiple inference procedures, where the outcome of one inference is used in another, insufficient samples can degrade the final outcome of the program, but lots of samples can substantially slow the program down with no gains in accuracy. Below are some experimental results illustrating this.

Insufficient sampling For evaluating `passert`s, Sampson et al. [33] use the Chernoff bound—an over-approximate bound which does not consider the quality of the samples drawn so far and gives a sample size purely based on the confidence level and accuracy. Evaluating a `passert` is a form of inference similar to MCMC in Section 3.4 but not used in any further computations. We ran their data obfuscation code [33] using 200 samples (instead of 8321 samples as suggested by the Chernoff bound) and observed that for the same confidence level, the interval was wider, (0.78, 0.88) and the `passert` failed due to this imprecision.

Wasteful sampling On the other hand, we observed that changing the number of samples from 8321 to 2000 has a very small effect in the output—the confidence intervals returned were (0.83, 0.85) and (0.83, 0.86) respectively.

4. The FLEXI Programming Model

This section formalizes the imperative FLEXI programming language that extends Uncertain⟨T⟩ with inference operators and probabilistic assertions.

4.1 Language

Figure 8 shows the core language syntax of FLEXI. FLEXI builds on a standard imperative language with variables, assignment, sequencing, and control flow constructs. It adds distributions d, which programs can draw from using distribution assignments $v \leftarrow d$. The grammar includes a special inference expression $I\ (v,\ List(e))$ that takes an arbitrary `Uncertain<T>` value v and zero or more hyper-parameters e and produces a new primitive `Uncertain<T>` value for the same `T`. FLEXI provides different inference algorithms: `enumeration`, `mcmc_sample`, maximum likelihood estimation (`MLE`), maximum a aestimation (`MAP`), etc. The `ulist` expression converts a list of uncertain values $List(Uncertain(T))$ to an uncertain list of values $UList(T)$. FLEXI programs end with a single probabilistic assertion.

4.2 Decorated Bayesian Network

We represent probabilistic programs as Decorated Bayesian Networks(DBNs). Figure 9 (a) shows the network at line 3 of Figure 7, after calling `mcmc_sample`. The Bayesian network has no node corresponding to the variable `y`. Due to `mcmc_sample`, the sub-graph was replaced with a single, independent node that contains the result of the MCMC inference process.

The `mcmc_call` call analyzes the *representation* of the Bayesian network rooted at the node where it was called (in this case, the node `y`) to infer the distribution and draws samples from it. Ordinary lifted operations such as $+$ and $*$ on `Uncertain` types cannot inspect the *representation* to draw samples. This difference sets inference operators apart from other operators.

To define the semantics for programs using inference operations, we extend the Bayesian network representation to include both ordinary nodes and *inference* nodes, $I(x, List(e))$, where x is a value of an uncertain type and $List(e)$ is a list of hyper-parameters. We call this new representation a *Decorated Bayesian Network* (DBN). Graphically, the entire program in Figure 7 produces the DBN shown in Figure 9 (b), where the shaded circle is the `mcmc_sample` inference operation. An inference node has exactly one incoming edge—the distribution being inferred—and any number of out-going edges indicating where the result is used. Here, the `mcmc_sample` node has the sample-size as a hyper-parameter that is 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 sub-graph, and replaces the inference node with the resulting distribution node.

Depicting a probabilistic program's data flow using a DBN helps reveal the advantages and pitfalls of using inference. Adding inference helps satisfy engineering constraints in large applications. By compacting a complex sub-graph in a Bayesian network to a simple histogram representation, inference can make subsequent operations more efficient, or it can let the program serialize a distribution for storage on disk or transmission over a network. On

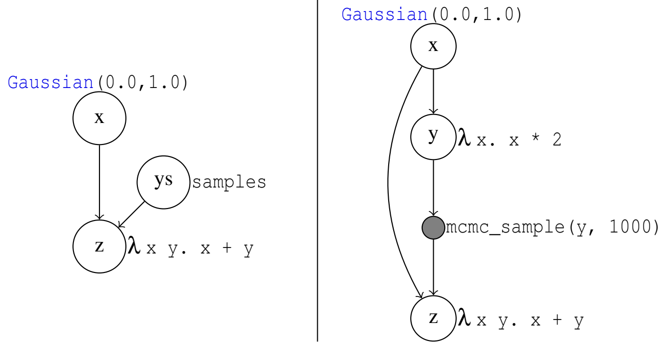

Figure 9: (a) Bayesian network calling inference. (b) The DBN for Figure 7.

$$
\begin{aligned}
Pr &::= s\ ;\ \texttt{passert}\ c \\
s &::= v \leftarrow e \mid v \leftarrow d \mid s\,;s \mid \texttt{skip} \mid \texttt{if}\ c\ s_1\ s_2 \mid \texttt{while}\ \ c\ s \\
c &::= e < e \mid e = e \mid e \wedge e \mid e \vee e \mid \neg\, e \\
e &::= I\ (v,\ List(e)) \mid e \oplus e \mid v \mid r \mid \texttt{ulist}\ (e) \\
I &::= \texttt{enumerate} \mid \texttt{mcmc_sample} \mid \texttt{MLE} \mid \texttt{MAP} \mid \ldots \\
T &::= P \mid Uncertain\ (\ P\ ) \mid UList\ (\ T\ ) \\
P &::= \texttt{int} \mid \texttt{float} \mid \texttt{double}
\end{aligned}
$$

$$
\begin{aligned}
r \in \mathbb{R},\ v &: \text{identifiers}, \\
d \in &\ \text{Probability distributions}, \\
\oplus &: \text{primitive operations}
\end{aligned}
$$

Figure 8: Syntax of FLEXI.

the other hand, adding inference introduces two potential sources of error: *dependence* errors and *approximation* errors.

Dependence errors occur when an inference operation compacts information, removing dependence edges from a graph. In Figure 7, replacing the sub-graph rooted at `y` with the sampled node `ys` eliminates the dependence between `x` and `y`. As a consequence, sampling values at `z` can lead to incorrect behavior because the correlation between the two `y` and `z` is ignored.

Approximation errors arise from the selection of the hyper-parameters. For example, if `mcmc_sample` uses too few samples, the resulting `z` might not faithfully approximate the sum.

Both problems arise when the program's result does not match a version with inference removed. Intuitively, inference operations should be semantic no-ops, i.e., $I(x, List(e)) \equiv x$ for a suitable definition of $\equiv$. Therefore, we can detect an inference-related bug as follows. If a program fails to meet its statistical specification (its `passert`), and removing an inference node from the program's DBN makes it satisfy the `passert`, then we blame the inference operation for the bug.

A DBN is a generalization of a Bayesian network: i) every Bayesian network is a DBN, and ii) every DBN without the inference nodes is a Bayesian network. The first part is trivially true because a Bayesian network is a DBN with *no* inference nodes. To prove the second part, we argue that every DBN can be converted to an equivalent Bayesian network. Consider the semantics of an inference node—it is an additional node in the Bayesian network of a program that summarizes the sub-graph rooted at it by inferring its distribution, then drawing samples and compacting them. Removing this node does not affect any of the ordinary nodes or the structure of the original Bayesian network. Hence, every DBN can be converted to a Bayesian network by removing the inference nodes.

5. DePP

DePP uses a combination of static and dynamic analyses to identify the causes of errors in terms of the systematic classification from Section 3.

5.1 Ignoring Dependence

Programs can ignore dependence in two ways: 1) by treating random variables as independent when they are not, and 2) by executing inference too early leading to an early approximation which in turn causes loss of dependence information.

Treating random variables as independent when they are not. To detect dependent random variables, DePP computes a correlation matrix among all random variables in a probabilistic program. The correlation coefficient is a number between -1 and $+1$ where values close to $\pm$ 1 indicate highly positive or negative correlation and values close to 0 indicate negligible correlation.

DePP finds the correlations by statically analyzing the Bayesian network of the program and tracing its way up to the leaf nodes corresponding to random variables. If the correlation coefficient between two random variables is more than a threshold τ or less than $-\tau$, DePP marks them as correlation-dependent. DePP uses the Spearman's rank correlation coefficient [8], r_s to compute the correlation matrix. r_s can determine any monotonic relationship between two variables. Computation of r_s is based on ranking of the values of the two variables [8] which DePP obtains by sampling. If there are k random variables in the program, DePP returns a $k \times k$ matrix of correlation coefficients. DePP reports a bug if random variables represented as independent nodes in the Bayesian network are correlation-dependent. For instance, it detects if two distinct inputs are dependent.

Premature inference. DePP statically analyzes the Bayesian network of a program to detect whether dependence information is lost due to wrong inference calls. For every inference call, DePP adds an *inference* node to the Bayesian network to generate a DBN. It then traverses the DBN to detect dependence errors, treating it as a graph problem. A DBN should be rejected if there exists any node Q that has two or more distinct paths to some ancestor P and any node (other than P or Q) on any of those paths is an inference node. This relationship is an error because inference calls on any of the intermediate nodes would lead to loss of information about their dependence on the common parent P. Consequently, estimating the distribution of the common child node Q will lead to incorrect samples.

```
Gaussian noise = new Gaussian(0.0, 1.0);
var noisy_t_h = from n in noise
                from t in (Uncertain<double>)t1
                from h in (Uncertain<double>)h1
                select Tuple.Create(n+t, n+h);
controlHvacSystem(noisy_t_h);
```

Figure 10: Correct implementation of noisy communication channel adding noise to a (t, h) pair.

To illustrate, consider again the programs in Figures 1 and 7. As described earlier, `y` and `z` depend on the same `x` but due to the call to `mcmc_sample` in Figure 7, this dependence information is lost, as illustrated in Figure 9(b). The correct graphical representation of the

```
Gaussian noise = new Gaussian(0.0, 1.0);
var noisy_t_h = from n in noise
  from t in (Uncertain<double>)t1.mcmc_sample(1000)
  from h in (Uncertain<double>)h1.mcmc_sample(1000)
  select Tuple.Create(n+t, n+h);
controlHvacSystem(noisy_t_h);
```

Figure 11: Incorrect implementation of noisy communication channel adding noise to a (t_1, h_1) pair.

program is the graph in Figure 2. The nodes for `y` and `z` both have an incoming edge from node `x`, implying that to infer the distribution of `z`, the same set of sample values from `x` should be used that are used for inferring `y`. In other words, `y` and `z` both depend on `x`. DePP would detect the incorrect call to `mcmc_sample` on `y`.

5.2 Incorrect Hyper-parameters

DePP identifies approximation errors by exploring the hyper-parameter space in a systematic way and suggesting good ones using a dynamic *meta-inference* technique. The meta-inference uses inference on the hyper-parameters that are themselves used in inference algorithms. This algorithm lets the developer fix a prior distribution over the hyper-parameter to be inferred and maximizes its likelihood *while* ensuring that the correctness or performance requirement of the actual program is satisfied. The hyper-parameter for the same inference algorithm might require re-tuning depending on the application. The overall correctness criterion is given by the following instantiation Bayes' rule. Let us assume that a program involves the hyper-parameter h and let the correctness criterion be C.

$$\begin{aligned} &P(\text{program is correct}) \\ &= P(\text{C is met} \wedge h \text{ is good}) \\ &= P(\text{C is met} \mid h \text{ is good}) * P(h \text{ is good}) \end{aligned}$$

Similarly, if the goal is to attain a certain performance, say, D, then the rule is given by:

$$\begin{aligned} &P(\text{program satisfies performance criterion}) \\ &= P(\text{D is met} \wedge h \text{ is good}) \\ &= P(\text{D is met} \mid h \text{ is good}) * P(h \text{ is good}) \end{aligned}$$

where "h is good" implies that the value of h that is chosen maximizes its likelihood function.

We implemented meta-inference as an extensible library with algorithms for three types of hyper-parameters: sample size n, learning rate α, and top-k. Whenever the programmer has to select a hyper-parameter, she can call the respective meta-inference method from the library and it will select the hyper-parameter for her. For every hyper-parameter, DePP has some built-in prior probability distributions but the programmer can also add new distributions and/or change the parameters of the existing ones.

5.2.1 Meta-inference for Sample Size

Definition. Let us consider k samples, each of size n being drawn from a distribution. We define the *best sample* of size n to be the one that has minimum sample variance. Further, if there are t best samples of sizes $n_1, n_2, \ldots n_t$, the best sample among those is the one with least in-sample variance.

In most real applications, the posterior distribution from which the samples are drawn may not have a closed form, and so knowing the population parameters may not be possible. Hence, DePP uses the above empirical measure to determine the goodness of a sample of a given size. If however more information about the posterior is available, such as the mean, then DePP also minimizes the difference between the sample mean and population mean.

DePP currently uses a Truncated Geometric prior for inferring the sample sizes. The probability density function is given as:

$$f_{tg}(x \mid p, lo, up) = \begin{cases} \frac{p(1-p)^x}{F(up)-F(lo)} & lo \leq x \leq up \\ 0 & elsewhere \end{cases}$$

where $F(x)$ is the cumulative density function of the Geometric distribution, defined as: $F(x) = 1-(1-p)^{x+1}$. We chose a Truncated Geometric because it is discrete and allows flexible upper and lower bounds for drawing samples from. The form of the prior is such that DePP is bound to select lower sample sizes with higher probability in order to maximize its likelihood. The programmer can use default values or specify lo, up and p of the prior, where lo and up are respectively the lower and upper bounds on the sample size. The optimal value of n found by DePP is then given by:

$$\arg\max_n \left(\frac{L(p, lo, up \mid n)}{\text{var}(S_n)} \right) \tag{1}$$

where $L(p, lo, up \mid n) = f_{tg}(x \mid p, lo, up)$ is the likelihood of the prior over n and $\text{var}(S_n)$ is the sample variance of the n samples drawn from the underlying distribution.

5.2.2 Meta-inference for α

The learning rate α is a hyper-parameter that trades off the speed of learning in algorithms such as MAP and MLE [19] and data fitting. A higher learning rate ensures faster learning but a slower learning rate ensures that there is less over-fitting [20]. To balance this trade-offs, a programmer sets a prior probability distribution so that different learning rates have probabilities of being selected associated with them. For example, in linear regression, it could be desirable that the error during learning drops below a certain threshold after a certain number of iterations. The task of DePP is then to choose the value of α that maximizes the likelihood of the prior while also ensuring that the error drops below the threshold, ε within the required number of iterations. To that end, DePP seeks an α optimizing $h(\alpha)$, defined as:

$$h(\alpha) = \left(\frac{f(\alpha)}{\text{number of iterations until } \Delta e \leq \varepsilon} \right) \tag{2}$$

where $\Delta e = |e_i - e_{i+1}|$, i.e., the change in error during training between two iterations and $f(\alpha)$ is the likelihood of α.

5.2.3 Meta-inference for Top-k

Top-k is a hyper-parameter used in systems that require the highest probability values from a discrete distribution. An example of such a system is a search engine. A search algorithm typically takes a query q and a set of documents and return the best k matches. This approach entirely rules out the possibility of showing a result that is slightly lower ranked but might be useful for the client. A different approach that explicitly exposes the uncertainty in the search model would return top scoring k' documents and includes some additional k'' documents that match q, but are not in the top-k.DePP finds the value of k using meta-inference. In the implementation of meta-inference algorithm for top-k, we chose the prior for k as: $f_{tg}(k \mid 0.001, 1,$ `matches.Count`$)$, where `matches.Count` represents the total number of documents matching q. A simple correctness criterion which we implemented in DePP is that the best ranked document should *always* be returned as part of the k documents. In other words, DePP draws a sample from the list of documents that matched q and checks whether the sample at least contains the best scored document.

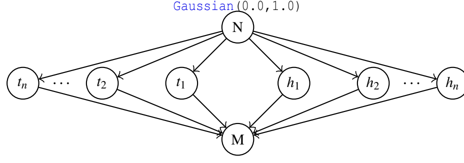

Figure 12: Bayesian network showing the correct dependence between temperature, humidity and noise. n=100.

variables	# noisy data	significance	analysis time
(t_i, h_i)	40	high	25.60 s
$t_1, t_2, \ldots$	27	low	9.20 s
$h_1, h_2, \ldots$	54	low	8.91 s

Table 1: Results of using DePP for detecting dependence among random variables on a dataset of 100 pairs of temperature and humidity readings [4]. We mark correlation ≥ 0.7 as high.

5.3 Incorrect Statistical Model and/or Poor Data

If all the previous approaches for detecting the cause of error fail, then DePP ultimately tries to check for the last cause of failure—an incorrect statistical model and/or inaccurate or insufficient data. DePP does this dynamically. It runs the program multiple times in order to ensure that there are sufficient samples from the posterior distribution represented by the program. This determines if the error is due to insufficient sampling. If a very large sample size also leads to failure of the `passert`, then DePP concludes that the statistical model itself is wrong or the data (if any) is bad. This approach exploits the *Law of Large Numbers (LLN)*, which guarantees that as the sample size $n \longrightarrow \infty$, the sample mean, $\widehat{\mu}$, converges to the population mean, μ, with high probability (weak LLN) or almost surely (strong LLN).

6. Case Studies

This section presents case studies illustrating different bugs related to dependence and approximation errors and the use of DePP in detecting them. We ran all experiments on a machine running 64-bit Windows 10 with 2.3 GHz, Core i3 processor.

6.1 Dependence Error

This section illustrates the use of DePP for finding dependence bugs, focusing on the two types of dependence bugs described in Section 3—incorrectly treating random variables as independent, and incorrectly performing inference at wrong program points.

6.1.1 Correlation between Random Variables

Consider a monitoring system in a home which records temperature and relative humidity and sends them over a noisy communication channel to a central hub [36]. The data from the hub is used to control an HVAC system.

The communication channel induces `Gaussian(0.0, 1.0)` noise on temperature and humidity values at random. To simulate the system, we used 100 pairs of readings from a real dataset [4] that records temperature and relative humidity among other quantities from a monitoring system and implemented the noisy communication channel in FLEXI, as Figure 10 shows. Due to the noise, the data is *lifted* to `uncertain` types at the other end of the channel. These `uncertain` temperature and humidity pairs cannot be treated as independent random variables—they are correlated because the noise added to them is the same.

We used DePP to perform two types of correlation analyses: 1) between every pair of temperature and humidity readings, 2) between a sequence of temperature readings, and between a sequence of humidity readings. Table 1 summarizes the results.

6.1.2 Incorrect Inference Call Leading to Loss of Dependence Information

Consider again the monitoring system described in Section 6.1.1. As mentioned previously, the monitor sends pairs of (t_i, h_i) values over a `Gaussian(0.0, 1.0)` channel such that the noise added to t_i is the same as that added to h_i, for $i \in 1, 2, ..., 100$. The Bayesian network for this program is shown in Figure 12. t and h are used in the hub for further computations such as controlling an HVAC system [36]. This adds another node, M to the Bayesian network as shown in Figure 12. Making calls to inference methods such as `mcmc_sample` at t_i or h_i, shown in Figure 11 would ignore their dependence on the *same* noise which is a bug because according to the design of the noisy channel, for a pair of (t_i, h_i) values, the noise should be the same. It would end up creating independent noise variables for temperature and humidity as shown in the Bayesian network in Figure 14. We ran DePP on this incorrect implementation of the monitoring system and it successfully detected each of these early inference calls in less than 8 milliseconds.

6.2 Approximation Error

This section describes using DePP for dynamically selecting hyper-parameters for inference.

6.2.1 Sample Size

Consider the probabilistic program shown below which draws samples from an exponential posterior distribution with parameter 2.0.

```
Func<int, Uncertain<double>> F = (n)
=> mcmc_sample(new Exponential(2.0), n);
```

The correctness condition for this program is simply that the sample should be *good enough*, i.e., sufficient to describe the distribution. In this case, since the mean of the population is known (Exponential(λ) has mean $1/\lambda$, which is 0.5 in this case), DePP finds the best sample size and the corresponding sample by minimizing the difference between sample mean and population mean together with minimizing sample variance as explained earlier. To use DePP's meta-inference, the programmer invokes `DebugSampleSize`$(F, 0.5, f_{tg}(p, lo, up))$, a method from the meta-inference library, which optimizes Equation 1. Figure 13 shows the measure of overall goodness of the sample drawn against the different values of n when (p, lo, up) is set to (0.001, 75, 1000). DePP suggests the best value, $n = 178$ and also returned the best sample of this size. The two red circles in Figure 13 compare the values of the ratio of likelihood to variance shown in Equation 1 indicating that the value picked by DePP was the best in the given range.

6.2.2 Learning Rate, α for Linear Regression

We used DePP to infer the value of α by maximizing Equation 2, for a linear regression model we trained on a binary classification dataset obtained from the LIBSVM website [2, 30]. The dataset has 2 classes, 22 features and 49,990 training examples. It represents time series data from a 10-cylinder internal combustion engine which was used in a neural network competition at the Ford Research Laboratory [30]. Figure 15 shows the variations in the value of $h(\alpha)$ as α changes.

Query	top-k	# of matches
algorithm	6	10
artificial	4	10
machine	6	9
inference	4	6
statistical	7	5

Table 2: Search queries, final top-k and total number of matched documents. As explained in section 5.2.3, the top-k results returned by our search engine always contains the top ranked document.

Benchmark	p	time with MI	time without MI
LR	α	0.04	0.05
SE	top-k	100.70	62.10
Exp	n	17.40	0.10

Table 3: Execution times (in s) for the benchmarks for Meta-Inference(MI). LR SE and Exp are shorthand for Linear Regression, Search Engine and the Exponential distribution example for n.

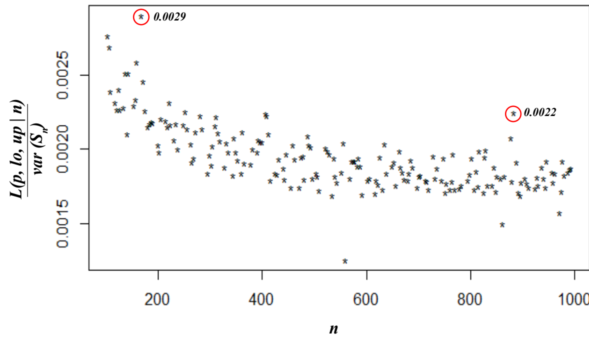

Figure 13: Values of sample size against measure of the ratio of likelihood to variance from Equation 1. The value of n chosen by DePP is 178 (indicated by the left red circle). The two red circles compare two particular values of n to show that the ratio doesn't change significantly even for n close to 900 (in fact, it was slightly lower for the latter).

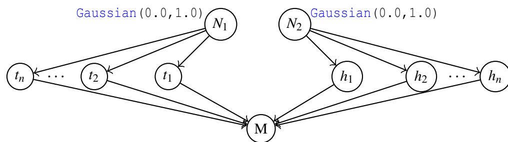

Figure 14: Bayesian network showing early inference.

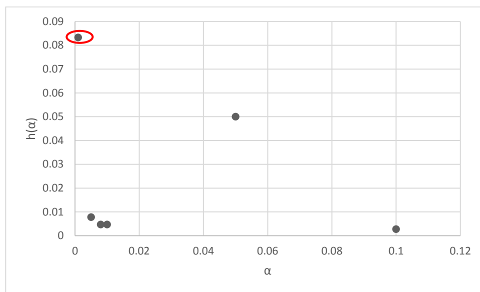

Figure 15: Values of α vs $h(\alpha)$. Since $h(\alpha)$ is maximum at ~ 0.001, DePP would suggest this value of α to be used for learning.

6.2.3 Top-k

We implemented a search engine using the Lucene [3] library which provides an API for scoring and indexing documents. We ran the search engine on a real dataset of 50,000 queries on a conference database [1]. Lucene takes a query q, pre-processes it, finds matches, scores them, and returns the top-k documents that match the query. Although top k is a hyper-parameter, many search engines, including Lucene, pick a single value for k and do not statistically tune this value as their system evolves. In our implementation DePP automatically infers k for each query as explained in section 5.2.3. Table 2 shows the final values of top-k inferred by DePP for five queries: `algorithm`, `artificial`, `machine`, `inference`, and `statistical`.

Table 3 shows the time taken to run the above three case studies with and without meta-inference (i.e., with hard-coded values of the hyper-parameters). For measuring the time for linear regression, we hard-coded the same value of α that DePP picked. For the search engine, we used $k = 3$ as the default and computed the sum of the execution times for all 5 queries. For sample size n, we used a hard-coded value of 1000. As the table shows, there is no significant effect in the performance of the benchmarks after integrating meta-inference.

7. Related Work

A variety of probabilistic programming models [7, 21, 27, 31] introduce efficient sampling techniques [5, 18] and tools for verification and synthesis of probabilistic programs [6, 9, 23–25, 33]. This section describes the relevant previous work in these areas.

Probabilistic programming languages The statistics, machine learning, and programming languages communities have recently reignited interest in *probabilistic programming languages*. These languages let domain experts specify statistical models in the familiar environment of a traditional programming language and supply generic statistical inference algorithms to estimate the model's parameters. Inference can either use *exact* methods for restricted categories of programs [14, 34] or *sampling* strategies such as MCMC, which work for general programs but introduce estimation error [5, 11, 28]. Recently, Hur et al. [18] found accuracy problems in existing MCMC-based sampling algorithms for probabilistic programming languages.

The key difference with this work is that the probabilistic programming languages literature focuses on small programs where statistical inference occurs only once. Typically, the programmer specifies a single statistical model from which a single result is produced by sampling the entire program. Sampling-based systems report their margin of error to the user but do not consider the impact of estimation error when a program uses the results of statistical inference for further computations. This one-shot inference pattern describes popular probabilistic programming languages such as Church [17], Infer.NET [22], R2 [11]; MAYHAP [33], which can only verify a single `passert` per program; and Uncertain⟨T⟩ [7], which does not track dependence past a statistical test.

Inference Inference can either use *exact* methods for restricted categories of programs [14, 34] or *sampling* strategies such as MCMC, which work for general programs but introduce estimation error [5, 11, 28]. Recently, Hur et al. [18] found accuracy problems in existing MCMC-based sampling algorithms for probabilistic programming languages. In WebPPL [16] and Figaro [29], programs make explicit calls to inference similar to our model. This choice

makes the programming model much more flexible and gives the programmers freedom to compose inferences in many more ways compared to other models. Additionally, DePP also ensures that multiple inference calls compose correctly.

Hyper-parameters Domingos points out how machine learning exhibits errors and some methods for avoiding them [13]. R2 [5] is an efficient MCMC sampler that relies on program analysis and is much faster compared to other samplers. MAYHAP [33] and Uncertain⟨T⟩ [7] automatically compute the sample sizes based on known statistical results such as Chernoff bound [12] and sequential probability ratio tests [35] respectively. DePP goes further by automating the inference of other hyper-parameters such as top-k and learning rate in addition to sample size.

Verification, debugging, and synthesis A recent framework called PrivInfer [6] can prove differential privacy of programs that implement Bayesian machine learning algorithms by using relational refinement types. Type system based approaches [9, 32] check correctness by introducing types to represent probabilistic or approximate data. The Rely [10] programming model uses program analysis to check whether an approximate program satisfies the reliability specifications. Stochastic contracts [24] have been used for real time systems to dynamically check bounds for worst case execution times. Probabilistic programming is not the only domain where debugging is complicated by non-determinism—debugging concurrent programs for example is extremely challenging due to non-determinism in thread scheduling [26]. Fortunately, in the probabilistic programming domain, we can control the non-determinism to some extent by controlling the seeds of the random number generators.

8. Conclusions

We presented a formalism based on Decorated Bayesian Networks (DBN) to represent statistical inference in probabilistic programs. Based on DBN, we implemented a debugger, DePP, that automatically detects errors specific to probabilistic programs and demonstrated it on example programs. DePP is the first debugging tool dedicated to probabilistic programs. It uses a combination of techniques covering two domains—programming languages and statistics. For instance, lifting ordinary computations to monadic bind operators and analyzing the structure of the resulting Bayesian network to track dependence is a core programming language technique while tuning hyper-parameters for inference algorithms is a statistics technique. On the other hand, analyzing a program's Bayesian network to identify the random variables at the leaves and finding a correlation matrix among them is a combination of both. For probabilistic programs, knowing the exact cause of an error is difficult because on top of usual programming errors, they may also have errors due to approximation, dependence, incorrect models and poor data. Consequently, the fact that DePP combines both statistical analysis and program analysis is valuable since prior to finding a bug, one cannot know which field's techniques are applicable.

Acknowledgements

We thank Archibald Samuel Elliott for insightful conversations regarding the semantics of FLEXI. We also thank various members of the UW PLSE lab for their feedback on earlier versions of the paper. We are very grateful to the anonymous reviewers for their helpful comments.

References

[1] Microsoft Academic Graph. `https://academicgraph.blob.core.windows.net/graph-2016-02-05/index.html`.

[2] LIBSVM Data: Classification (Binary Class). `https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary.html`.

[3] LUCENE.net seach engine library. `https://lucenenet.apache.org/`.

[4] UCI Machine Learning Repository. `https://archive.ics.uci.edu/ml/datasets/SML2010`.

[5] S. R. Aditya Nori, Chung-Kil Hur. R2: An efficient MCMC sampler for probabilistic programs. In *AAAI Conference on Artificial Intelligence (AAAI)*. AAAI, 2014.

[6] G. Barthe, G. P. Farina, M. Gaboardi, E. J. G. Arias, A. Gordon, J. Hsu, and P.-Y. Strub. Differentially private bayesian programming. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 68–79, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4139-4. doi: 10.1145/2976749.2978371. URL `http://doi.acm.org/10.1145/2976749.2978371`.

[7] 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.

[8] S. Boslaugh and P. Watters. *Statistics in a Nutshell: A Desktop Quick Reference*. In a Nutshell (O'Reilly). O'Reilly Media, 2008. ISBN 9781449397814. URL `https://books.google.com/books?id=ZnhgO65Pyl4C`.

[9] B. Boston, A. Sampson, D. Grossman, and L. Ceze. Probability type inference for flexible approximate programming. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2015.

[10] M. Carbin, S. Misailovic, and M. Rinard. Verifying quantitative reliability of programs that execute on unreliable hardware. In *OOPSLA*, 2013.

[11] A. T. Chaganty, A. V. Nori, and S. K. Rajamani. Efficiently sampling probabilistic programs via program analysis. In *AISTATS*, 2013.

[12] H. Chernoff. A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. *The Annals of Mathematical Statistics*, pages 493–507, 1952.

[13] P. Domingos. A few useful things to know about machine learning. *CACM*, 55(10):78–87, 2012.

[14] T. Gehr, S. Misailovic, and M. T. Vechev. PSI: exact symbolic inference for probabilistic programs. In *Computer Aided Verification (CAV)*, 2016.

[15] W. R. Gilks and P. Wild. Adaptive rejection sampling for Gibbs sampling. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 41(2):337–348, 1992. ISSN 00359254, 14679876. URL `http://www.jstor.org/stable/2347565`.

[16] N. D. Goodman and A. Stuhlmüller. The Design and Implementation of Probabilistic Programming Languages. `http://dippl.org`, 2014. Accessed: 2016-10-10.

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

[18] C.-K. Hur, A. V. Nori, S. K. Rajamani, and S. Samuel. A provably correct sampler for probabilistic programs. In *Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, 2015.

[19] A. Kak. ML, MAP, and Bayesian The Holy Trinity of Parameter Estimation and Data Prediction. `https://engineering.purdue.edu/kak/Trinity.pdf`, 2014. Accessed: 2016-11-1.

[20] W. M. Koolen, T. v. Erven, and P. D. Grünwald. Learning the learning rate for prediction with expert advice. In *Proceedings of the 27th International Conference on Neural Information Processing Systems*, NIPS'14, pages 2294–2302, Cambridge, MA, USA, 2014. MIT Press. URL `http://dl.acm.org/citation.cfm?id=2969033.2969083`.

[21] D. Kozen. Semantics of probabilistic programs. In *Symposium on Foundations of Computer Science*, pages 101–114, Oct 1979.

[22] T. Minka, J. Winn, J. Guiver, and D. Knowles. Infer.NET 2.5, 2012. Microsoft Research Cambridge. `http://research.microsoft.com/infernet`.

[23] S. Misailovic, M. Carbin, S. Achour, Z. Qi, and M. C. Rinard. Chisel: Reliability- and accuracy-aware optimization of approximate computational kernels. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2014.

[24] C. Nandi, A. Monot, and M. Oriol. Stochastic contracts for runtime checking of component-based real-time systems. In *Proceedings of the 18th International ACM SIGSOFT Symposium on Component-Based Software Engineering*, CBSE, 2015. ISBN 978-1-4503-3471-6. URL `http://doi.acm.org/10.1145/2737166.2737173`.

[25] A. V. Nori, S. Ozair, S. K. Rajamani, and D. Vijaykeerthy. Efficient synthesis of probabilistic programs. *SIGPLAN Not.*, 50(6):208–217, June 2015. ISSN 0362-1340. doi: 10.1145/2813885.2737982. URL `http://doi.acm.org/10.1145/2813885.2737982`.

[26] C.-S. Park and K. Sen. Concurrent breakpoints. *SIGPLAN Not.*, 47(8): 331–332, Feb. 2012. ISSN 0362-1340. doi: 10.1145/2370036.2145880. URL `http://doi.acm.org/10.1145/2370036.2145880`.

[27] S. Park, F. Pfenning, and S. Thrun. A probabilistic language based upon sampling functions. In *POPL*, 2005.

[28] A. Pfeffer. A general importance sampling algorithm for probabilistic programs. Technical Report TR-12-07, Harvard University, 2007. `ftp://ftp.deas.harvard.edu/techreports/tr-12-07.pdf`.

[29] A. Pfeffer. Figaro: An object-oriented probabilistic programming language. Technical report, Charles River Analytics, 2009.

[30] D. Prokhorov. Ijcnn neural network competition. 2001. `http://www.geocities.ws/ijcnn/nnc_ijcnn01.pdf`.

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

[32] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. EnerJ: Approximate data types for safe and general low-power computation. In *PLDI*, 2011.

[33] 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.

[34] S. Sankaranarayanan, A. Chakarov, and S. Gulwani. Static analysis for probabilistic programs: Inferring whole program properties from finitely many paths. In *PLDI*, 2013.

[35] A. Wald. *Sequential Tests of Statistical Hypotheses*, pages 256–298. Springer New York, New York, NY, 1992. ISBN 978-1-4612-0919-5. doi: 10.1007/978-1-4612-0919-5_18. URL `http://dx.doi.org/10.1007/978-1-4612-0919-5_18`.

[36] F. Zamora-Martnez, P. Romeu, P. Botella-Rocamora, and J. Pardo. On-line learning of indoor temperature forecasting models towards energy efficiency. *Energy and Buildings*, 83:162 – 172, 2014. ISSN 0378-7788. doi: http://dx.doi.org/10.1016/j.enbuild.2014.04.034. URL `http://www.sciencedirect.com/science/article/pii/S0378778814003569`.