Efficient Synthesis of Probabilistic Programs

Aditya V. Nori
Microsoft Research, India
adityan@microsoft.com

Sherjil Ozair
IIT Delhi, India
sherjilozair@gmail.com

Sriram K. Rajamani
Microsoft Research, India
sriram@microsoft.com

Deepak Vijaykeerthy
Microsoft Research, India
t-dvijay@microsoft.com

Abstract

We show how to automatically synthesize probabilistic programs from real-world datasets. Such a synthesis is feasible due to a combination of two techniques: (1) We borrow the idea of "sketching" from synthesis of deterministic programs, and allow the programmer to write a skeleton program with "holes". Sketches enable the programmer to communicate domain-specific intuition about the structure of the desired program and prune the search space, and (2) we design an efficient Markov Chain Monte Carlo (MCMC) based synthesis algorithm to instantiate the holes in the sketch with program fragments. Our algorithm aims to efficiently synthesize a probabilistic program that is most consistent with the data.

A core difficulty in synthesizing probabilistic programs is computing the likelihood $\mathcal{L}(P \mid D)$ of a candidate program P generating data D. We propose an approximate method to compute likelihoods using mixtures of Gaussian distributions, thereby avoiding expensive computation of integrals. The use of such approximations enables us to speed up evaluation of the likelihood of candidate programs by a factor of 1000, and makes Markov Chain Monte Carlo based search feasible. We have implemented our algorithm in a tool called PSKETCH, and our results are encouraging—PSKETCH is able to automatically synthesize 16 non-trivial real-world probabilistic programs.

Categories and Subject Descriptors D.1.2 [*Programming Techniques*]: Automatic Programming—Program Synthesis

General Terms Design, Experimentation, Performance

Keywords Probabilistic Programming, Program Synthesis, Markov Chain Monte Carlo

PLDI'15 June 13–17, 2015, Portland, OR, USA
Copyright © 2015 ACM 978-1-4503-3468-6/15/06…$15.00
DOI: http://dx.doi.org/10.1145/2737924.2737982

1. Introduction

Probabilistic models with causal dependencies can be succinctly written as probabilistic programs [6, 8, 15, 16, 23–25]. One of the main advantages of writing probabilistic programs is that they allow *non* machine learning experts to focus on the specification of the probabilistic model, and not worry about complex issues such as how to implement inference over the model. Indeed, existing probabilistic programming tools are able to automatically generate inference code from specifications written as probabilistic programs, thus reducing the degree of expertise required to implement a machine learning or AI task.

Another advantage of probabilistic programs is that they use programming notation familiar to developers. Just like programs can be thought of as transformers of states, probabilistic programs can be thought of as transformers of distributions, and this allows programmers to generalize intuitions they have about deterministic programs to intuitions about probabilistic programs. However, developers of probabilistic programs need to have sufficient expertise in probability and statistics—they need to be able to choose the right distributions for variables, compose primitive distributions to build complex distributions, and estimate parameters of these distributions. Such tasks are still difficult to perform for several programmers. To alleviate this problem, we study the problem of automatically synthesizing probabilistic programs from available data. Intuitively, we would like to automatically search over the space of all probabilistic programs, and find the program that best matches the available data. However, such a search is infeasible since the space of probabilistic programs is huge.

In order to make the problem tractable, we use the idea of *sketching* [28], which is a very popular technique used in synthesis of deterministic programs, and ask the programmer to provide a skeleton or "sketch" of the probabilistic program with some "holes" that are unfilled. The task of the synthesizer is to find the right instantiation of the holes to get a complete program. Sketches enable programmers to specify domain knowledge they may have about the problem at hand, and also dramatically reduces search complexity from the space of all probabilistic programs to the space of all possible instantiations of the holes.

Even though sketches prune the search space, efficiently synthesizing probabilistic programs is still computationally challenging due to reasons we describe next. Let us denote a program sketch P with a hole as $P[\cdot]$, and the instantiation of the hole with a program fragment H as $P[H]$. For each possible instantiation H, the synthesizer needs to evaluate a score indicating how the completed probabilistic program $P[H]$ matches with the available data D. Computing such a score involves estimating the likelihood of data D being generated by a generative model represented by the probabilistic program $P[H]$, and this estimation is also computationally intensive. For instance, if x_1 and x_2 are Gaussian random variables with density functions $\Delta_1(x, \mu_1, \sigma_1)$ and $\Delta_2(x, \mu_2, \sigma_2)$ respectively (where μ_1, μ_2 and σ_1, σ_2 are the respective means and standard deviations for these distributions), the likelihood of the program "$y = x_1 + x_2$" generating the data "$y = k$" for some constant k requires computing the integral $\int \Delta_1(x, \mu_1, \sigma_1) \cdot \Delta_2(k - x, \mu_2, \sigma_2)\mathrm{d}x$. These integrals are difficult to evaluate—computing such indefinite integrals symbolically for arbitrary distributions is unknown, and computing definite integrals using numerical techniques is expensive.

Let $\mathcal{L}(P[H] \mid D)$ denote the likelihood of program $P[H]$ with respect to the data D [20]. In other words, this is the probability or density (if D is a continuous random variable) of the data D given the parameter $P[H]$. From the discussion above, we know that computing likelihoods involves evaluating integrals. Even though techniques for automatically computing $\mathcal{L}(P[H] \mid D)$ have been studied before (see for example [2]), it is expensive to evaluate such likelihoods for a large number of candidate completions of the hole in $P[\cdot]$. Our key insight is that we can approximate the computation of likelihoods using mixture of Gaussians without performing any integration. Further, such an approximation can be computed efficiently, and using this approximation we can speed up computing the likelihood $\mathcal{L}(P[H] \mid D)$ for a candidate H instantiating the hole $P[\cdot]$ significantly. Consequently, we are able to build a Markov Chain Monte Carlo (MCMC) search to navigate large search spaces for instantiating holes, and improve the number of candidate programs evaluated by the MCMC algorithm by a factor of 1000 (See Figure 8). We also demonstrate (empirically) that this approximate computation of likelihood does not affect the quality of the probabilistic program synthesized by the MCMC search, and we are able to synthesize programs that have likelihoods that are close to manually written programs for the same task.

We have implemented our ideas in a tool called PSKETCH, and have used PSKETCH to automatically synthesize 16 non-trivial probabilistic programs thus showing the effectiveness of our approach.

The rest of the paper is organized as follows: we introduce probabilistic programs in Section 2. In Section 3, we informally describe our approach with the help of an example. Section 4 formalizes the problem as well as describe our synthesis algorithm. Section 5 describes our empirical evaluation. Section 6 places our work in the context of existing work, and finally, Section 7 concludes the paper.

2. Probabilistic Programs

We consider probabilistic programs that are imperative programs with two added constructs: (1) the ability to draw values at random from distributions via probabilistic assignments, and (2) the ability to condition values of variables in a program via observe statements. We refer the reader to [9] for a gentle introduction to probabilistic programming.

Consider the program in Figure 1. This program represents a simplified version of a Bayesian model called TrueSkill [12] for estimating the skills of players based on the games played by them. Informally, based on prior estimates of the players' skills and the outcomes of a set of games, TrueSkill computes the revised estimates for players' skills. Such a rating can be used to give points, or match players having comparable skills for an improved gaming experience.

In particular, the TrueSkill program `TS` takes as input an array of games called `games` (each element has type `struct game`), where each game has a unique identifier `id`, and is played between players `p1` and `p2` with a result that is either 1 or 0 which is the ob-

```
struct game {
  int id;
  int p1;
  int p2;
  int result;
};

TS(struct game[] games, int count)
1:  double[] skills;
2:  int[] r;
3:  for i=0 to count-1
4:    skills[i] := Gaussian(100, 10);
5:  for g in games
6:    perf1 := Gaussian(skills[g.p1], 15);
7:    perf2 := Gaussian(skills[g.p2], 15);
8:    r[g.id] = perf1 > perf2;
9:  for g in games
10:   observe(g.result == r[g.id]);
11: return skills;
```

Figure 1: TrueSkill [12].

served outcome of the game, and `count` which is the total number of players. Lines 3–4 contain probabilistic assignments that define the prior for the skill of each player, which is a Gaussian distribution with mean 100 and standard deviation 10. Lines 5–8 define the generative model for the variables—the performance of the players in each game (which depends on the players' respective skills) and the result of each game (which depends on the performance of the players in the game). Finally, lines 9–10 condition the model based on the evidence (and this is done via the observe statement in line 10). The semantics of the `observe` statement classifies runs which satisfy the boolean expression `g.result` == `r[g.id]` as *valid* runs. Runs that do not satisfy this condition are classified as *invalid* runs. The meaning of a probabilistic program is the distribution of the expression returned by it (conditioned on valid runs of the program). Note that the observe statement constrains performances in a game, and implicitly the skills of the players, since performances depend on skills. Finally, in line 11, the `skills` are returned, and the meaning of the probabilistic program is the distribution over elements of `skills`.

Semantics. Unlike a deterministic program which produces the same output every time it is run (with the same input), a probabilistic program produces different outputs each time it is run. The semantics of a probabilistic program is the probability distribution over the possible values of the outputs produced by the program. Informally, the distribution can be produced by running the program a large number of times and producing histograms (or other statistical measures) over the outputs produced from the runs. There is prior work on precisely defining formal denotational and operational semantics for probabilistic programs, and we refer the interested reader to [10, 17].

Inference. Calculating the distribution specified by a probabilistic program is called *inference*. The inferred probability distribution is called *posterior probability* distribution, and the initial guess made by the program is called the *prior probability* distribution. One way to perform inference is to compile the probabilistic program to a probabilistic model [16, 23] over which inference is performed using message passing algorithms such as belief propagation and its variants [26]. Alternatively, one can execute the program several times using sampling to execute probabilistic statements, and observe the values of the desired variables in valid runs [8, 13, 24], and compute statistics on the data to infer an approximation to the desired distribution.

```
/* Sketch */
struct game {
  int id;
  int p1;
  int p2;
  int result;
};

TSSketch(struct game[] games, int count)
1:  double[] skills;
2:  int[] r;
3:  for i=0 to count-1
4:    skills[i] := ??;
5:  foreach g in games
6:    r[g.id] = ??(skills[g.p1], skills[g.p2]);
7:  foreach g in games
8:   observe(g.result == r[g.id]);
9: return skills;


/* Data */
```

m.p1	m.p2	result
0	1	1
1	2	1
0	2	1

id	skill
0	105
1	95
2	90

Figure 2: Sketch and data for TrueSkill.

3. Overview

Writing good probabilistic programs is non-trivial. For instance, an inexperienced programmer may replace lines 6–8 in the TrueSkill program in Figure 1, with the simpler assignment,

$$\mathtt{r}[\mathtt{g.id}] = \mathtt{skills}[\mathtt{g.p1}] > \mathtt{skills}[\mathtt{g.p2}]$$

However, such a program constrains the skills of players more than what we would like to. Just because the player g.p1 won against g.p2 in one game, it is premature to conclude that the skill of g.p1 is more than that of g.p2. Player g.p2 may actually have a higher skill than player g.p1 but may have lost the game because of other extraneous reasons (such as headache or poor sleep during the previous night). The program in Figure 1 accounts for such possibilities using the variables perf1 and perf2, and is a better fit for the data that is available (that is, the likelihood of the program with respect to the data is higher).

Even though probabilistic programs free programmers from thinking about inference, the programmer still needs training to use idioms such as the use of latent variables (such as perf1 and perf2) in the above example. The goal of our paper is to automatically search over the space of probabilistic programs and automatically synthesize a probabilistic program that best fits the given the output data we expect.

Sketching. Since the space of probabilistic programs is very large, searching through this space and finding a program that maximizes likelihood of that program with respect to the data is infeasible. We borrow the idea of *sketching* from the synthesis of deterministic programs [28] and allow the programmer to write a skeleton program with holes to communicate her design intuitions, and allow the synthesis algorithm to search over a more constrained space. As a concrete example, in the case of TrueSkill, if the programmer can specify that r[g.id] depends on the values of skills[g.p1] and skills[g.p2], but not specify the exact nature of the dependence, and the synthesis tool can automatically generate the code in lines 6–8 which calculate r[g.id], we can greatly assist programmers.

We consider the problem of automatically synthesizing a probabilistic program from data that contains information about the dependences between variables in the program, and a sketch that contains the skeletal dependency structure between the variables of the probabilistic programs. Unlike the synthesis problem for deterministic programs where holes can be instantiated by constants from a finite set, or by program fragments from a finite set of possibilities specified by a template, we would like the holes to be instantiated from an unbounded space of (probabilistic) program fragments. Consequently, we cannot use standard synthesis techniques based on constraint solvers [28] for our purpose.

Figure 2 shows the sketch and associated data for the TrueSkill example. As shown in the figure, the data indicates how the skills of the players affect the outcome of a game. In this example, as seen from the data, player 0 beats player 1, player 1 beats player 2, and player 0 beats player 2. The skills as a consequence of these games are also a part of the data as shown in Figure 2. In this data set, the user of the synthesis system has picked values 105, 95 and 90 for player 0, player 1 and player 2 respectively. These values were picked to be consistent with the user's expectation that "*if player 1 beats player 2, then player 1 has higher skill than player 2*". In the sketch, the $??$ represents a *hole* which is the part that the user is not sure about. This is exactly the code that our system PSKETCH is expected to automatically synthesize. The syntax $??$ in line 4 indicates a hole that does not depend on other variables, whereas the syntax ??(...) with arguments (line 6) denotes a hole that depends on its arguments. The goal of the synthesis system is to replace the holes with program fragments that are consistent with the chosen data for the skills, so that when the program is fed a new data set with different sets of players and different set of game results, a suitable distribution over skills is generated, that is consistent with the skill values for the example data given by the user. Using the sketch and data shown in Figure 2, PSKETCH is able to automatically synthesize the program TS in Figure 1. Informally, PSKETCH searches for code snippets to replace the holes such that the snippets are most consistent with the data.

Metropolis Hastings Search. Our synthesis algorithm searches through the space of possible completions of holes using the Metropolis Hastings (MH) algorithm [3] (which is a specific kind of Markov Chain Monte Carlo algorithm). The goal of MH is to construct a random walk through a Markov Chain whose steady state probability distribution is a desired distribution $T(x)$. MH works by starting at some arbitrary state x_0 and choosing each state x_n from the previous state x_{n-1} using a mechanism to propose the new state and a mechanism to accept or reject the proposed state. Given a current state x_{n-1}, the next state x_n is chosen by sampling from a proposal distribution $Q(x_n, x_{n-1})$, and the generated state x_n is accepted with probability:

$$\mathtt{accept}(x_n \mid x_{n-1}) = \min\left\{1, \frac{T(x_n) \times Q(x_{n-1}, x_n)}{T(x_{n-1}) \times Q(x_n, x_{n-1})}\right\}$$

In our instantiation of MH, we build a Markov Chain over the space of all possible completions of the holes in the program. Recall that we denote a sketch with a hole as $P[\cdot]$, and the program in which the hole is completed by a program fragment H as $P[H]$. Our sketches allow multiple holes in the program, and in such cases the completion H is really a tuple of completions for all the holes. For simplicity, we continue to use the notation $P[H]$ in the multiple-hole case as well. Given a current program $P[H]$, we mutate the completion H of holes to a new completion H', and accept or reject H' with probability given by the MH acceptance ratio. The new completion H' is obtained from H by randomly applying mutation operations that mutate variables, constants and operators on the abstract syntax tree of H. See Section 4.1 for details.

Efficiently approximating likelihoods. In the above formulation of MH, at each point x_n during the random walk, we need to evaluate the probability $T(x_n)$ from the desired distribution T. Since our random walk is over the possible completions of holes, in our case, the probability distribution $T(\cdot)$ is the probability of the current completion $P[H]$ of the program, given data D. Using Bayes rule, this can be shown to be proportional to the probability that the completion $P[H]$ generates the data D, denoted by $\mathsf{Pr}(D \mid P[H])$ assuming uniform priors over all the completions (see Section 4). Computing this likelihood $\mathsf{Pr}(D \mid P[H])$ requires evaluating integrals and is expensive. Using existing techniques it takes on the order of 100 milliseconds to compute the likelihood for a given program, and this greatly limits the number of completions we can explore using MH search within a reasonable amount of time. We use a novel approximation technique to speed up the computation of likelihoods. Our first observation is that the *Mixture of Gaussians* (MoG) distribution is known to have a universal approximation property [22]. As a result, we can approximate any continuous distribution using a MoG distribution. Moreover, MoG distributions are closed under usual operations such as addition, subtraction and conditionals. Thus, we can symbolically calculate the likelihoods of expression trees written using MoG distributions at compile time, and plug in the desired data to evaluate the likelihood $\mathsf{Pr}(D \mid P[H])$ in linear time. As a result of this approximation, we are able to evaluate approximate likelihoods in the order of one tenth of a millisecond, allowing to increase the efficiency of the MH search by a factor of 1000. See section 4.3 for a precise description of the MoG approximation of likelihood computation.

In practice, we find that in all the examples we have tried, it suffices for the programmer to specify only the deterministic part of the program, and leave out all the code that deals with probabilistic reasoning using holes. Our algorithm is able to synthesize programs that match the best hand-written program we know of in terms of the likelihood value (see Table 1 in Section 5).

4. The Synthesis Algorithm

We consider a language that is an imperative probabilistic programming language with facilities for specifying sketches. The syntax of the language is shown in Figure 3. In particular, the sketching language has an additional construct called a *hole*. Holes can be used anywhere where an expression is expected, and serves as a placeholder for unknown code. There are two types of holes:

- *Independent hole* (denoted $??$). This type of hole is a placeholder for code that does not make use of any variables defined in the environment provided by the sketch.
- *Hole with dependences* (denoted $??(\mathcal{E}_1, \cdots, \mathcal{E}_n)$). This type of hole is a placeholder for code which makes use of the parameters passed to it, i.e., $\mathcal{E}_1, \cdots, \mathcal{E}_n$.

Given a sketch $P[\cdot]$ (a program in the sketching language defined in Figure 3), we are interested in synthesizing a program $P[H]$ that is "most" consistent with the data D (where H is an instantiation of the hole). If the sketch has multiple holes, we abuse notation and use "$\cdot$" to denote a tuple of holes, and H to denote a tuple of completions to these holes. Specifically, given a sketch $P[\cdot]$, we want to synthesize a probabilistic program $P[H]$ that maximizes the probability of the data D. In other words, we are interested in the probabilistic program $P[H^*]$ such that:

$$H^* = \arg\max_H \ \mathsf{Pr}(P[H] \mid D) \tag{1}$$

where H^* is the desired instantiation of the hole, and the search is performed over the set of all program fragments H that are valid completions of the sketch $P[\cdot]$. Using Bayes' rule, Equation 1 is equivalent to:

$$H^* = \arg\max_H \ \mathsf{Pr}(P[H])\mathsf{Pr}(D \mid P[H]) \tag{2}$$

Assuming a uniform prior $\mathsf{Pr}(P[H])$ (i.e., all completions H are equally likely), we have:

$$H^* = \arg\max_H \mathsf{Pr}(D \mid P[H]) \tag{3}$$

where $\mathsf{Pr}(D \mid P[H])$ is the likelihood function for the program $P[H]$. This optimization problem can be looked upon as a maximum likelihood (ML) estimation [20] over the higher order search space of programs. Exact ML estimation is usually very expensive except in the case of simple probabilistic programs. We tackle this problem by approximating the ML estimate via standard MCMC sampling [20]. The synthesis algorithm MCMC-SYN (shown in Algorithm 1) takes a dataset D and a sketch $P[\cdot]$ as input, and returns a probabilistic program $P[H^*]$, where H^* is an approximate solution to Equation 2.

First, MCMC-SYN chooses a probabilistic program fragment H from the set of valid completions $\Sigma_{P[\cdot]}$ of the sketch $P[\cdot]$ (line

r	$\in$	Real	
x	$\in$	Vars	
uop	::=	$\{!\}$	unary operators
bop	::=	$\{+, -, \times, \&\&, \|\|, >\}$	binary operators
top	::=	$\{\texttt{ite}\}$	ternary operators
$\mathcal{D}$	::=	$\cdots$	declarations
$\mathcal{E}$	::=		expressions
		$\mid x$	variable
		$\mid c$	constant
		$\mid \mathcal{E}_1$ **bop** $\mathcal{E}_2$	binary operation
		$\mid$ **uop** $\mathcal{E}$	unary operation
		$\mid$ **top** $\mathcal{E}_1\mathcal{E}_2\mathcal{E}_3$	ternary operation
		$\mid ??$	hole
		$\mid ??(\mathcal{E}_1, \cdots, \mathcal{E}_n)$	hole with dependences
$\mathcal{S}$	::=		statements
		$\mid$ `skip`	skip
		$\mid x = \mathcal{E}$	deterministic assignment
		$\mid x \sim \texttt{Dist}(\bar{\theta})$	probabilistic assignment
		$\mid$ `observe` (φ)	observe
		$\mid \mathcal{S}_1; \mathcal{S}_2$	sequential composition
		$\mid$ `if` $\mathcal{E}$ `then` $\mathcal{S}_1$ `else` $\mathcal{S}_2$	conditional composition
		$\mid$ `for` `i` $:= 1$ `to` n `do` $\mathcal{S}$	for loop
$\mathcal{P}$	::=	$\mathcal{D}\ \mathcal{S}$	programs

Figure 3: Syntax of the sketching language.

Algorithm 1 MCMC-SYN

Input: Data D, and sketch $P[\cdot]$.
Output: Program $P[H^*]$.

```
1:  $\mathcal{S} := \emptyset$
2:  $H \sim \Sigma_{P[\cdot]}$
3:  for $i$ to $N$ do
4:      $H' := \texttt{mutate}(H)$
5:      if $\texttt{accept}(P[H'] \mid P[H])$ then
6:          $H := H'$
7:      end if
8:      $\mathcal{S} := \mathcal{S} \cup \{H\}$
9:  end for
10: $H^* := \texttt{max}_{H \in \mathcal{S}} \mathsf{Pr}(D \mid P[H])$
11: return $P[H^*]$
```

2). Then, it iteratively (for a maximum of `N` iterations) performs the following steps (lines 3–9):

1. First, in line 4, a probabilistic program fragment H' is obtained by mutating the program fragment H using a proposal distribution. The procedure `mutate` is described in Section 4.1).
2. Next, in lines 5–9, if H' is accepted then, we add H' to a set $\mathcal{S}$ of accepted program fragment samples. Otherwise, the previous sample H is added to $\mathcal{S}$. The acceptance criterion is described in Section 4.2.

In line 10, we select the program fragment H^* in the set of samples $\mathcal{S}$ with the maximum value of $\mathsf{Pr}(D \mid P[H^*])$. Finally, in line 11, MCMC-SYN returns the desired synthesized program $P[H^*]$.

4.1 Program Mutation

The `mutate` function in line 4 of Algorithm 1 takes a probabilistic program H as input, and returns a probabilistic program H' sampled from a proposal distribution $\mathsf{Pr}(H' \mid H)$. To sample a program H' from $\mathsf{Pr}(H' \mid H)$, we first pick a number n from a geometric distribution (this is to ensure that the probability of picking n decreases with increasing n). Next, starting with the program H, n mutation operations are sequentially applied to obtain H' (essentially, this corresponds to the process of mutating the program until we get a "good" program). Each mutation operation is defined over the abstract syntax tree (AST) of the program H, and proceeds as follows: First, a node from the AST of H is chosen uniformly at random. Then, one of the following mutation operations is performed, depending on the type of the node:

- **Operation-1**. If the node is a variable which is a parameter to the hole, then the variable is replaced with one of the other parameters of the hole chosen uniformly at random.
- **Operation-2**. If the node is a constant with a real value c, then it is replaced by a sample from a normal distribution with mean c and a predefined standard deviation.
- **Operation-3**. If the node is an operator, then the operator is replaced by a new operator which is chosen uniformly at random from the set of operators with equivalent type.
- **Operation-4**. This operation is applicable for all node types. It first replaces the node with a non-terminal (from Figure 3) corresponding to the subtree rooted at that node. Then, a entire subtree for the non-terminal is produced using the syntax rules in Figure 3 chosen with a bias to replace all non-terminals with terminals.

Depending on the node type only some of the mutation operations may be applicable, and one of these applicable operations is chosen uniformly at random. For example, for a variable node, Operation-1 and Operation-4 are applicable, and one of these two operations is chosen uniformly at random. We illustrate the mutation process with the help of an example. Consider the program snippet H_1 shown below:

```
Gaussian(x, 11.3) + Gaussian(y, 11.9)
```

We illustrate how H_1 is transformed to the program snippet H_2 shown below

```
Gaussian(x, 11.7) - Gaussian(z, 14.3+y)
```

upon applying a sequence of 4 mutation operations. As described earlier, program mutation first chooses the number of mutation operations to be performed. In our example, this number $n = 4$. First, the constant node "11.7" is chosen and mutated to "11.3" using Operation-2. Second, the "+" node is chosen and replaced with "-" using Operation-3. Third, the "y" node is chosen and replaced with "z" using Operation-1. Finally, the node "11.9" is chosen, and replaced with the nonterminal $\mathcal{E}$ using Operation-4, and then the expression tree "14.3 + y" is derived from $\mathcal{E}$ using the production rules from Figure 3. Once the mutated program is generated, we perform a quick syntactic check and reject programs which are nonsensical (perform use before definition, produce ill-typed expressions, etc.). Even though some undesirable programs escape our check, these programs get low likelihood scores (using the equations in Section 4.3), and hence the MCMC search never selects undesirable programs. We also ensure that parameters of distributions are only variables (and not general expressions) while generating the programs.

4.2 MH Acceptance Criterion

A mutation from program $P[H]$ to $P[H']$ is accepted with the following probability during MCMC sampling (line 5 in Algorithm 1):

$$\texttt{accept}(P[H'] \mid P[H]) = \min\left\{1, \frac{\mathsf{Pr}(D \mid P[H]')\mathsf{Pr}(P[H] \mid P[H'])}{\mathsf{Pr}(D \mid P[H])\mathsf{Pr}(P[H'] \mid P[H])}\right\}$$

This computation is often very expensive resulting in significant degradation in the performance of sampling. In order to overcome this issue, we propose an efficient approximation to the likelihood function $\mathsf{Pr}(D \mid P[H])$ which results in an order of magnitude speedup over the exact computation. We describe our approximation scheme to the likelihood computation next.

4.3 Efficient Likelihood Computation

The MCMC-SYN algorithm described in Algorithm 1 requires the computation of the likelihood function $\mathsf{Pr}(D \mid P[H])$ for each candidate program $P[H]$ and dataset D. The success of the algorithm depends crucially on very efficient computation of this likelihood for every sample program. Our initial implementation used the algorithm in [2] for computing the probability distribution for any variable in a probabilistic program. However, the algorithm makes use of integrals which are computationally expensive. Consequently, in our experiments, by using this algorithm, we were only able to compute the likelihoods for tens of sample programs per second which renders the MCMC search ineffective. In contrast, the likelihood computation approach described next allows us to compute likelihoods for ten thousands of sample programs per second (see Figure 8 in Section 5).

The main idea is to compute the likelihood expression of a probabilistic program symbolically, and represent it approximately using a family of simple distributions. We choose the family of Mixture of Gaussians (MoG) distribution for this purpose due to their universal approximation property [22]. A Gaussian mixture model is a weighted sum of $n_{\mathcal{E}}$ component Gaussian densities as defined by the following equation:

$$\texttt{MoG}(x; n_{\mathcal{E}}, \boldsymbol{w}, \boldsymbol{\mu}, \boldsymbol{\sigma}) = \sum_{i=1}^{n_{\mathcal{E}}} w_i g(x; \mu_i, \sigma_i)$$

where $\boldsymbol{w}$, $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ are arrays of size $n_{\mathcal{E}}$, and represent the mixing fraction, mean, and the standard deviation of each Gaussian density in the mixture. The probability density function $g(x; \mu_i, \sigma_i)$ for a simple univariate Gaussian distribution with mean μ_i and standard deviation σ_i is defined as:

$$g(x; \mu_i, \sigma_i) = \frac{1}{\sqrt{2\pi\sigma_i^2}} \exp(-\frac{(x-\mu_i)^2}{2\sigma_i^2})$$

Let $\mathcal{S}$ be the body of a probabilistic program $P[H]$. We first execute S symbolically using the operator $\texttt{LL}(\mathcal{S}, \nu_o, \rho_o)$ (defined in Figure 5), where ν_o is a initial environment that maps every variable to $\perp$, and ρ_o is the identity constraint (which evaluates to 1 on

```
TS(struct game[] games, int count)
  double[] skills;
  int[] r;
  double perf1, perf2;
  for i=0 to count-1
    skills[i] := Gaussian(100, 10);
  for g in games
    perf1 := Gaussian(skills[g.p1], 15);
    perf2 := Gaussian(skills[g.p2], 15);
    r[g.id] = perf1 > perf2;
  for g in games
    observe(g.result == r[g.id]);
  // $\rho = \rho \times \prod_{|g|}$ ite(g.result == r[g.id], 1, 0)
  return skills;
```

skill[0]	$\texttt{MoG}(\texttt{skill}[0]; 1, [1], [100], [10])$
skill[1]	$\texttt{MoG}(\texttt{skill}[1]; 1, [1], [100], [10])$
perf1	$\texttt{MoG}(\texttt{perf1}; 1, [1], [\texttt{skill}[0]], [15])$
perf2	$\texttt{MoG}(\texttt{perf2}; 1, [1], [\texttt{skill}[1]], [15])$
r[0]	$\texttt{Bernoulli}(\texttt{r}[0]; \frac{1}{2} + \frac{1}{2}\text{erf}(\frac{\texttt{skill}[0]-\texttt{skill}[1]}{30}))$

$$\Pr(D \mid P[H]) = \texttt{Bernoulli}(\texttt{r}[0]; \tfrac{1}{2} + \tfrac{1}{2}\text{erf}(\tfrac{\texttt{skill}[0]-\texttt{skill}[1]}{30})) \times \texttt{MoG}(\texttt{skill}[0]; 1, [1], [100], [10]) \times \texttt{MoG}(\texttt{skill}[1]; 1, [1], [100], [10]) \times \prod_{|g|} \texttt{ite}(\texttt{g.result} == \texttt{r[g.id]}, 1, 0)$$

Figure 4: A candidate completion of the sketch for TrueSkill (2 players & 1 game).

any state). Let (ν, ρ) the environment and constraint expression returned by $\texttt{LL}(\mathcal{S}, \nu_o, \rho_o)$. The likelihood $\Pr(D \mid P[H])$ is obtained by taking the product of the likelihood of each the values of D from the final environment ν, and taking the product with the constraints in ρ (ensuring that observe statements are respected).

Example. Figure 4 illustrates this procedure for an example. The table in the right side of Figure 4 shows the final environment obtained by symbolically executing the program using the $\texttt{LL}(\cdot)$ operator. Note that each variable maps to either a Bernoulli or mixture of Gaussians distribution (which is an approximation) using the rules from Figures 5 and 6, without any expensive computations (such as integration). The likelihood expression $\Pr(D \mid P[H])$ is obtained as a product of the final symbolic expressions of the variables r[0], skill[0] and skill[1] (for which we have data), and the product of the constraints obtained during evaluation of the observed statements by $\texttt{LL}(\cdot)$.

The $\texttt{LL}(\cdot)$ operator. We now explain the details of the $\texttt{LL}(\cdot)$ operator, which is one of the central contributions of this paper. The $\texttt{LL}(\cdot)$ operator takes a statement, an initial environment and an initial constraint as input, and symbolically executes the statement returning a final environment and a final set of constraints. The operator uses MoG distributions to perform approximations to continuous distributions. We assume that, in the case of an if-then-else statement, both branches update the same set of variables. We make a pre-pass to transform the program so that this assumption holds—as an example, we transform the program "$\texttt{if (e) then x = e1 else y = e2}$" to the program "$\texttt{if (e) then x = e1; y = y else x = x; y = e2}$" (that includes dummy identity assignments).

The rules for the $\texttt{LL}(\cdot)$ operator are shown in Figure 5 (with additional rules for expression evaluation in Figure 6). Given a statement $\mathcal{S}$, an environment ν mapping variables to either MoG or Bernoulli distributions, and a constraint ρ, we have that $\texttt{LL}(\mathcal{S}, \nu, \rho)$ returns a pair (ν', ρ'), where ν' is the updated environment and ρ' conjoins constraints from observe statements in $\mathcal{S}$ to ρ. The observe statement is the only primitive statement that affects the constraint ρ. All the other primitive statements update only the environment ν and leave the constraint ρ unaffected.

We have that $\texttt{LL}(\texttt{skip}, \nu, \rho)$ simply returns the inputs (ν, ρ). In the case of a deterministic assignment statement ($x = \mathcal{E}$), ρ remains unaffected and the corresponding entry to variable x in the environment is set to $[\![\mathcal{E}]\!]_\nu$ (using rules shown in Figure 6). For a probabilistic assignment statement ($x \sim \texttt{Dist}(\bar{\theta})$), ρ remains unaffected and the entry that corresponds to variable x in the environment is set to the symbolic expression of the approximate likelihood of the distribution in the statement. Due to the universal approximation property of MoG distributions, we can approximate all continuous distributions as MoG distributions. We show approximations for Beta, Gamma and Poisson distributions in Figure 5. The Gaussian distribution can be precisely represented with a single component mixture of Gaussians, and the Bernoulli distribution is represented "as is".

The observe statement ($\texttt{observe}(\varphi)$) adds an extra product to the constraint ρ. The resulting constraint evaluates to 0 if the expression φ evaluates to $\texttt{false}$ and helps reject program mutations that are inconsistent with observe statements during the MH search. The observe statement does not affect the environment ν.

In the case of sequential composition statement $(\mathcal{S}_1; \mathcal{S}_2)$, the function $\texttt{LL}(\cdot)$ is applied to the statements in the order in which they appear in the program. The environment updates are composed and the constraints are multiplied. For an if-then-else statement ($\texttt{if}\ \mathcal{E}\ \texttt{then}\ \mathcal{S}_1\ \texttt{else}\ \mathcal{S}_2$), we apply $\texttt{LL}(\cdot)$ to both the branches, and merge the resulting environments and constraints with the conditional expression. We use $UV(\mathcal{S})$ to denote the variables updated by statement $\mathcal{S}$. Recall that we have made sure that both branches of the conditional update the same set of variables using a pre-pass transformation. The for loop ($\texttt{for i} := 1\ \texttt{to}\ n\ \texttt{do}\ \mathcal{S}$) is handled by unrolling the loop, which is feasible since we assume bounded loops.

We describe the evaluation performed by $[\![]\!]_\nu$ for various types of expressions $\mathcal{E}$ in Figure 6. Given an arbitrary expression $\mathcal{E}$, we have that $[\![\mathcal{E}]\!]_\nu$ evaluates the expression in the environment ν and returns approximation in the form of either a Bernoulli or a MoG distribution. We exploit the property that MoG distributions are closed under the operations of addition, subtraction and conditionals. For instance, addition of two MoG distributions is always an MoG, and it is carried out by considering each pair of mixture components, and using the additive property of Gaussians on the two Gaussian components. The resulting distribution has a component corresponding to each pair of components of the operands, and the mixing fraction each component is the product of the mixing fractions of the pair of operand components. Other operations on pairs of MoG distributions are carried out in a similar way. The Gaussian distribution is not closed under the product operator, so we use an approximation to the likelihood in that case. This approximation is defined precisely in Figure 6.

In order to compare two MoG distributions with the $>$ operator, we use the error function (erf), which allows us to compute $\Pr(X > 0)$, and thus $\Pr(X > Y)$. In the case of $\texttt{ite}$, the number of components in the resulting distribution is the sum of the components in the two branches of the $\texttt{ite}$. The component means and

$$
\begin{aligned}
\texttt{LL}(\texttt{skip},\nu,\rho) &:= (\nu,\rho)\\
\texttt{LL}(x=\mathcal{E},\nu,\rho) &:= \textbf{let } \mathcal{E}' = [\![\mathcal{E}]\!]_\nu \textbf{ in}\\
&\quad\ (\nu[x \leftarrow \mathcal{E}'],\rho)\\
*\texttt{LL}(x \sim \texttt{Beta}(x;\alpha_1,\alpha_2),\nu,\rho) &:= \Big(\nu\Big[x \leftarrow \texttt{MoG}(x;1,[1],[\tfrac{\alpha_1}{\alpha_1+\alpha_2}],\\
&\qquad [\sqrt{\tfrac{\alpha_1\alpha_2}{(\alpha_1+\alpha_2)^2(\alpha_1+\alpha_2+1)}}])\Big],\rho\Big)\\
*\texttt{LL}(x \sim \texttt{Gamma}(\alpha_1,\alpha_2),\nu,\rho) &:= \Big(\nu\Big[x \leftarrow \texttt{MoG}(x;1,[1],[\alpha_1\alpha_2],\\
&\qquad [\sqrt{\alpha_1\alpha_2}])\Big],\rho\Big)\\
*\texttt{LL}(x \sim \texttt{Poisson}(\lambda),\nu,\rho) &:= (\nu[x \leftarrow \texttt{MoG}(x;1,[1],[\lambda],[\sqrt{\lambda}])],\rho)\\
\texttt{LL}(x \sim \texttt{Gaussian}(\mu,\sigma),\nu,\rho) &:= (\nu[x \leftarrow \texttt{MoG}(x;1,[1],[\mu],[\sigma])],\rho)\\
\texttt{LL}(x \sim \texttt{Bernoulli}(p),\nu,\rho) &:= (\nu[x \leftarrow \texttt{Bernoulli}(x;p)],\rho)\\
\texttt{LL}(\texttt{observe}(\varphi),\nu,\rho) &:= \textbf{let } \texttt{f} = \texttt{ite}(\varphi,1,0) \textbf{ in}\\
&\quad\ (\nu,\rho\times\texttt{f})\\
\texttt{LL}(\mathcal{S}_1;\mathcal{S}_2,\nu,\rho) &:= \textbf{let } (\nu',\rho') = \texttt{LL}(\mathcal{S}_1,\nu,\rho) \textbf{ and}\\
&\quad\ (\nu'',\rho'') = \texttt{LL}(\mathcal{S}_2,\nu',\rho') \textbf{ in}\\
&\quad\ (\nu'',\rho'')\\
\texttt{LL}(\texttt{if } \mathcal{E} \texttt{ then } \mathcal{S}_1 \texttt{ else } \mathcal{S}_2,\nu,\rho) &:= \textbf{let } \text{envmerge}(\nu,\nu_1,\nu_2,\mathcal{E},V) =\\
&\quad\ \nu_3 := \nu;\\
&\quad\ \textbf{foreach}(v \in V)\\
&\qquad \nu_3 :=\\
&\qquad\quad \nu_3[v \leftarrow [\![\texttt{ite}(\mathcal{E},\nu_1(v),\nu_2(v))]\!]_{\nu_3}];\\
&\quad\ \nu_3\\
&\quad\ \textbf{and}\\
&\quad\ (\nu_1,\rho_1) = \texttt{LL}(\mathcal{S}_1,\nu,1) \textbf{ and}\\
&\quad\ (\nu_2,\rho_2) = \texttt{LL}(\mathcal{S}_2,\nu,1) \textbf{ and}\\
&\quad\ \mathcal{E}' = [\![\mathcal{E}]\!]_\nu \textbf{ in}\\
&\quad\ (\text{envmerge}(\nu,\nu_1,\nu_2,\mathcal{E}',UV(\mathcal{S}_1)),\\
&\qquad \rho\times\texttt{ite}(\mathcal{E}',\rho_1,\rho_2))\\
\texttt{LL}(\texttt{for i} := 1 \texttt{ to } n \texttt{ do } \mathcal{S},\nu,\rho) &:= \textbf{let } \texttt{rec } \text{loopfunc}(\mathcal{S},n,\nu,\rho) =\\
&\quad\ \texttt{if}(n \geq 1) \texttt{ then}\\
&\quad\ \textbf{let}(\nu',\rho') = \texttt{LL}(\mathcal{S},\nu,\rho) \textbf{ in}\\
&\quad\ \text{loopfunc}(S,n-1,\nu',\rho')\\
&\quad\ \texttt{else } (\nu,\rho)\\
&\quad\ \textbf{in}\\
&\quad\ \text{loopfunc}(\mathcal{S},n,\nu,\rho)
\end{aligned}
$$

Figure 5: Given a statement $\mathcal{S}$, an environment ν, and a constraint ρ, the function $\texttt{LL}(\mathcal{S},\nu,\rho)$ returns (ν',ρ'), where ν' is the updated environment and ρ' is conjoins constraints from the observe statements in $\mathcal{S}$ to ρ. The expressions $\texttt{MoG}(x;n,\boldsymbol{w},\boldsymbol{\mu},\boldsymbol{\sigma})$ and $\texttt{Bernoulli}(x;p)$ are the likelihoods of mixture of Gaussians and Bernoulli distributions respectively. Rules which introduce approximations are marked by a *. Rules without a * are precise.

standard deviations remain the same. However, their mixing fractions get multiplied by the probability of the branch being taken, i.e., $\mathsf{Pr}(b)$ or $1-\mathsf{Pr}(b)$, where b is the event that the `if` branch is taken. Other operators such as `and` and `or` follow from the properties of mixture distributions and Gaussian distributions. When unsupported operators are present in an expression, $[\![]\!]_\nu$ returns the unit expression (which always evaluates to 1) as an approximation.

4.4 Convergence

Our synthesis algorithm MCMC-SYN (Algorithm 1) searches through the space of possible completions of holes using the Metropolis Hastings (MH) algorithm. The MH algorithm is always guaranteed to converge asymptotically (with the number of samples) to the target distribution [3]. In practice, with a reasonable number of samples obtained by running the algorithm for a few minutes, the MH algorithm converges to a reasonable approximation of the target distribution. We empirically demonstrate the convergence property of our approach by comparing the respective log-likelihoods of the target and the synthesized programs (see Table 1 in Section 5).

5. Evaluation

We have implemented our synthesis algorithm in a tool called PSKETCH, and evaluate its effectiveness on 16 probabilistic program benchmarks. All experiments were performed on a 2.00 GHz Intel 3rd Gen Core i7 processor system with 8 GB RAM running Microsoft Windows 8.

Goals. The goal of our empirical evaluation is to answer the following two questions:

1. Is PSKETCH able to synthesize interesting probabilistic programs? In order to answer this question, we collected various probabilistic programs, and replaced the interesting probabilistic computations in these programs with holes. Then we synthesized probabilistic programs from these sketches using PSKETCH and evaluated the quality of the generated programs (with respect to the original programs we started).
2. How much faster is the mixture of Gaussians approximation (described in Section 4.3) for computing likelihoods when compared with traditional methods for computing likelihoods? In other words, how much speedup does the approximation really buy us?

Benchmarks. We picked 16 benchmark programs from existing literature on probabilistic programs. We describe the programs below:

- `Burglary` is Pearl's burglary example, first described in [14].
- `TrueSkill` [12] is a skill rating system which is a variant of the example described in Section 2.
- `Clinical` [23] models a lab setting where the effectiveness of a drug is inferred by observing its performance on control and placebo groups.
- The `Clickthrough` [23] benchmarks mainly consist of programs which model how the relevance and appeal of links in a web search result affect which links the user examines and clicks on. The programs `Clickthrough1` and `Clickthrough2` only model link examination. The programs `Clickthrough3` and `Clickthrough4` model both link examination and click, and thus are harder to synthesize.
- `Conference` [23] models the accept/reject decision of a paper based on the quality of the paper, the expertise of the reviewer, and the possible error made while reviewing the submission.
- `Grading` [1] models the answers given to questions by students based on the ability of the student, the difficulty and discriminative ability of the question, and the true answer. Using this model, it is possible to estimate the true answers to all questions in an exam, using data for only the responses students gave to the questions.
- `Handedness` [23] models the probability that a given person is left or right handed.
- `Gender Height` [23] models the height of male and female persons and their relative comparison.
- The `MoG` benchmarks consists of the Mixture of Gaussian model. To synthesize the `MoG1` benchmark, values for both the data and the latent variable are given as input to PSKETCH. For the MoG2 benchmark, values for only the data variable are given as input, and PSKETCH is expected to infer the existence

$$
\begin{aligned}
[\![c]\!]_\nu &:= \texttt{MoG}(c;1,[1.0],[c],[b]) \\
[\![\texttt{Bernoulli}(x;p)]\!]_\nu &:= \texttt{Bernoulli}(x;p) \\
[\![\texttt{MoG}(x;n,\boldsymbol{w},\boldsymbol{\mu},\boldsymbol{\sigma})]\!]_\nu &:= \texttt{MoG}(x;n,\boldsymbol{w},\boldsymbol{\mu},\boldsymbol{\sigma}) \\
[\![\textbf{uop } \mathcal{E}]\!]_\nu &:= \textbf{let } \mathcal{E}' = [\![\mathcal{E}]\!] \textbf{ in} \\
&\quad\; [\![\textbf{uop } \mathcal{E}']\!]_\nu \\
[\![\mathcal{E}_1 \textbf{ bop } \mathcal{E}_2]\!]_\nu &:= \textbf{let } \mathcal{E}_1' = [\![\mathcal{E}_1]\!]_\nu \textbf{ and} \\
&\quad\; \mathcal{E}_2' = [\![\mathcal{E}_2]\!]_\nu \textbf{ in} \\
&\quad\; [\![\mathcal{E}_1' \textbf{ bop } \mathcal{E}_2']\!]_\nu \\
[\![\textbf{top}(\mathcal{E}_1,\mathcal{E}_2,\mathcal{E}_3)]\!]_\nu &:= \textbf{let } \mathcal{E}_1' = [\![\mathcal{E}_1]\!]_\nu \textbf{ and} \\
&\quad\; \mathcal{E}_2' = [\![\mathcal{E}_2]\!]_\nu \textbf{ and} \\
&\quad\; \mathcal{E}_3' = [\![\mathcal{E}_3]\!]_\nu \textbf{ in} \\
&\quad\; [\![\textbf{top}(\mathcal{E}_1',\mathcal{E}_2',\mathcal{E}_3')]\!]_\nu
\end{aligned}
$$

$$
\begin{aligned}
*[\![\texttt{MoG}(x;n_1,\boldsymbol{w}_1,\boldsymbol{\mu}_1,\boldsymbol{\sigma}_1)+ \\
\texttt{MoG}(y;n_2,\boldsymbol{w}_2,\boldsymbol{\mu}_2,\boldsymbol{\sigma}_2)]\!]_\nu &:= \texttt{MoG}(z;n_3,\boldsymbol{w}_3,\boldsymbol{\mu}_3,\boldsymbol{\sigma}_3) \\
&\quad \text{where,} \\
&\quad z = x+y \\
&\quad n_3 = n_1 \times n_2 \\
&\quad w_3^{n_2 i+j} = w_1^i \times w_2^j \\
&\quad \mu_3^{n_2 i+j} = \mu_1^i + \mu_2^j \\
&\quad \sigma_3^{n_2 i+j} = \sqrt{(\sigma_1^i)^2 + (\sigma_2^j)^2}
\end{aligned}
$$

$$
\begin{aligned}
*[\![\texttt{MoG}(x;n_1,\boldsymbol{w}_1,\boldsymbol{\mu}_1,\boldsymbol{\sigma}_1)- \\
\texttt{MoG}(y;n_2,\boldsymbol{w}_2,\boldsymbol{\mu}_2,\boldsymbol{\sigma}_2)]\!]_\nu &:= \texttt{MoG}(z;n_3,\boldsymbol{w}_3,\boldsymbol{\mu}_3,\boldsymbol{\sigma}_3) \\
&\quad \text{where,} \\
&\quad z = x-y \\
&\quad n_3 = n_1 \times n_2 \\
&\quad w_3^{n_2 i+j} = w_1^i \times w_2^j \\
&\quad \mu_3^{n_2 i+j} = \mu_1^i - \mu_2^j \\
&\quad \sigma_3^{n_2 i+j} = \sqrt{(\sigma_1^i)^2 + (\sigma_2^j)^2}
\end{aligned}
$$

$$
\begin{aligned}
*[\![\texttt{MoG}(x;n_1,\boldsymbol{w}_1,\boldsymbol{\mu}_1,\boldsymbol{\sigma}_1)\times \\
\texttt{MoG}(y;n_2,\boldsymbol{w}_2,\boldsymbol{\mu}_2,\boldsymbol{\sigma}_2)]\!]_\nu &:= \texttt{MoG}(z;n_3,\boldsymbol{w}_3,\boldsymbol{\mu}_3,\boldsymbol{\sigma}_3) \\
&\quad \text{where,} \\
&\quad z = x * y \\
&\quad n_3 = n_1 \times n_2 \\
&\quad w_3^{n_2 i+j} = w_1^i \times w_2^j \\
&\quad \mu_3^{n_2 i+j} = \frac{\mu_1^i {\sigma_2^j}^2 + \mu_2^j {\sigma_1^i}^2}{{\sigma_1^i}^2 + {\sigma_2^j}^2} \\
&\quad \sigma_3^{n_2 i+j} = \frac{{\sigma_1^i}^2 \times {\sigma_2^j}^2}{{\sigma_1^i}^2 + {\sigma_2^j}^2}
\end{aligned}
$$

$$
\begin{aligned}
[\![\texttt{ite}(\texttt{Bernoulli}(x;p), \\
\texttt{MoG}(y;n_1,\boldsymbol{w}_1,\boldsymbol{\mu}_1,\boldsymbol{\sigma}_1), \\
\texttt{MoG}(y;n_2,\boldsymbol{w}_2,\boldsymbol{\mu}_2,\boldsymbol{\sigma}_2))]\!]_\nu &:= \texttt{MoG}(y;n_3,\boldsymbol{w}_3,\boldsymbol{\mu}_3,\boldsymbol{\sigma}_3) \\
&\quad \text{where,} \\
&\quad n_3 = n_1 + n_2 \\
&\quad w_3^i = w_1^i \times p \;\; \forall i \in \{0,\ldots,n_1-1\} \\
&\quad \mu_3^i = \mu_1^i \;\; \forall i \in \{0,\ldots,n_1-1\} \\
&\quad \sigma_3^i = \sigma_1^i \;\; \forall i \in \{0,\ldots,n_1-1\} \\
&\quad w_3^{n_1+j} = w_2^j \times (1-p) \\
&\quad \forall j \in \{0,\ldots,n_2-1\} \\
&\quad \mu_3^{n_1+j} = \mu_2^j \;\; \forall j \in \{0,\ldots,n_2-1\} \\
&\quad \sigma_3^{n_1+j} = \sigma_2^j \;\; \forall j \in \{0,\ldots,n_2-1\}
\end{aligned}
$$

$$
\begin{aligned}
*[\![\texttt{MoG}(z;1,[1],[\texttt{MoG}(x;n_1,\boldsymbol{w}_1,\boldsymbol{\mu}_1,\boldsymbol{\sigma}_1)], \\
[\texttt{MoG}(y;n_2,\boldsymbol{w}_2,\boldsymbol{\mu}_2,\boldsymbol{\sigma}_2)])]\!]_\nu &:= \texttt{MoG}(z;n_3,\boldsymbol{w}_3,\boldsymbol{\mu}_3,\boldsymbol{\sigma}_3) \\
&\quad \text{where,} \\
&\quad z = \mathcal{N}(x,y) \\
&\quad n_3 = n_1 \\
&\quad w_3^i = w_1^i \\
&\quad \mu_3^i = \mu_1^i \\
&\quad \sigma_3^i = \mu_2^i + (\sigma_1^i)^2 + \sigma_2^i
\end{aligned}
$$

$$
\begin{aligned}
[\![x]\!]_\nu &:= \nu(x) \\
[\![!(\texttt{Bernoulli}(x;p))]\!]_\nu &:= \texttt{Bernoulli}(y;1-p) \\
&\quad \text{where,} \\
&\quad y = \;!x
\end{aligned}
$$

$$
\begin{aligned}
*[\![\texttt{MoG}(x;n_1,\boldsymbol{w}_1,\boldsymbol{\mu}_1,\boldsymbol{\sigma}_1) > \\
\texttt{MoG}(y;n_2,\boldsymbol{w}_2,\boldsymbol{\mu}_2,\boldsymbol{\sigma}_2)]\!]_\nu &:= \texttt{Bernoulli}(z;p) \\
&\quad \text{where,} \\
&\quad z = \texttt{ite}((x>y),1,0) \\
&\quad p = \textstyle\sum_{i=0}^{n_1-1} \sum_{j=0}^{n_2-1} \frac{1}{2} \\
&\quad + \tfrac{1}{2} erf\Big(\frac{\mu_1^i - \mu_2^j}{\sqrt{2((\sigma_1^i)^2 + (\sigma_2^j)^2)}}\Big) \\
&\qquad \times (w_1^i \times w_2^j)
\end{aligned}
$$

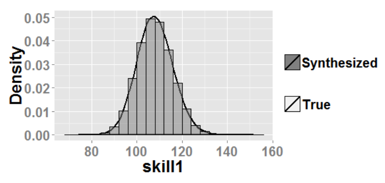
$$
\begin{aligned}
[\![\texttt{Bernoulli}(x;p_1)\&\& \\
\texttt{Bernoulli}(y;p_2)]\!]_\nu &:= \texttt{Bernoulli}(z;p_1 p_2) \\
&\quad \text{where,} \\
&\quad z = x\&\&y \\
[\![\texttt{Bernoulli}(x;p_1)|| \\
\texttt{Bernoulli}(y;p_2)]\!]_\nu &:= \texttt{Bernoulli}(z;1-(1-p_1)(1-p_2)) \\
&\quad \text{where,} \\
&\quad z = x||y \\
*[\![\texttt{ite}(\texttt{Bernoulli}(x;p_1), \\
\texttt{Bernoulli}(y;p_2), \\
\texttt{Bernoulli}(y;p_3))]\!]_\nu &:= \texttt{Bernoulli}(y;p_1 p_2 \\
&\qquad\qquad + (1-p_1)p_3)
\end{aligned}
$$

Figure 6: Given an arbitrary expression $\mathcal{E}$ and an environment ν, $[\![\mathcal{E}]\!]_\nu$ returns an approximation to $\mathcal{E}$ as either a Bernoulli or a mixture of Gaussians distribution. We use $\texttt{MoG}(x;n,\boldsymbol{w},\boldsymbol{\mu},\boldsymbol{\sigma})$ and $\texttt{Bernoulli}(x;p)$ to denote mixture of Gaussians and Bernoulli distribution respectively. Rules which introduce approximations are marked by a *. Rules without a * are precise. The function $erf(x)$ is called the error function: it equals twice the integral of a Gaussian distribution with mean 0 and standard deviation $1/\sqrt{2}$. The value of b is drawn from a $\texttt{Beta}(0.1,1)$ distribution. All rules assume that the variables x, y and z are distinct.

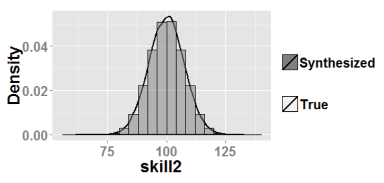
of a latent variable. For the `MoG3` benchmark, although the values for the hidden variable are not given, it is mentioned in the sketch.

- `RATS` [4] is a clinical study of the growth of a rat's weight based on the number of days in observation.
- The `Gaussian` benchmark is a simple Gaussian model for a single variable.

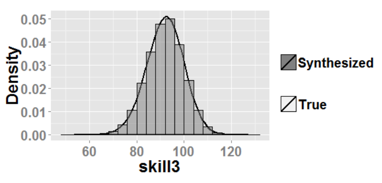
Results. For each probabilistic program benchmark, we generated data sets by running the program multiple times and collecting the outputs generated (recall that the output of a probabilistic program is a set of samples from the probability distribution represented by the program). We also wrote sketches for the benchmarks by replacing parts of the program that involved probabilistic reasoning with holes. As a specific example, we replaced the probabilistic computations in the TrueSkill example (as shown in Figure 1) with the sketch shown in Figure 2. We did similar replacement of probabilistic computations in all the 16 examples with holes. Then we ran PSKETCH on the sketches with the generated data sets to produce a solution. As seen in Table 1, PSKETCH takes time in minutes and often in seconds to synthesize a program from its sketch and data. The column "target program LL" is the data log-likelihood of the original program (which we call as "target



(a) Skill distributions of player 1.



(b) Skill distributions of player 2.



(c) Skill distributions of player 3.

Figure 7: Skill distributions of players 1, 2 and 3 w.r.t the actual and the synthesized `TrueSkill` program (3 players & 3 games).

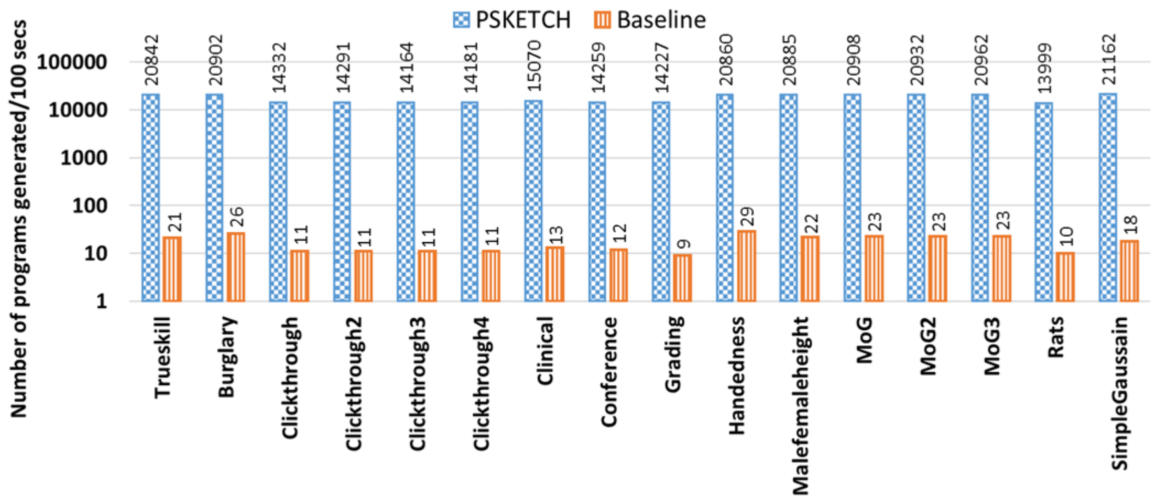


Figure 8: Number of sample programs generated per 100 secs with and without the approximation proposed in Section 4.3.

Table 1: Synthesis Results For PSKETCH.

Benchmark	time (sec)	target program LL	synthesized program LL	data set size
`Burglary`	89	-71.94	-71.37	100
`TrueSkill`	114	-718.33	-697.68	400
`Clinical`	149	-102.26	-98.09	100
`Clickthrough1`	117	-102.75	-103.91	400
`Clickthrough2`	37	-102.75	-102.34	400
`Clickthrough3`	120	-263.73	-263.82	400
`Clickthrough4`	312	-263.73	-263.12	400
`Conference`	113	-251.81	-195.33	400
`Grading`	353	-179.04	-181.82	400
`Handedness`	145	-90.71	-90.32	100
`Gender Height`	451	-780.02	-727.88	100
`MoG1`	113	-479.15	-472.59	100
`MoG2`	7	-405.27	-411.19	100
`MoG3`	2	-405.27	-405.43	100
`RATS`	215	-1140.68	-1047.54	400
`Gaussian`	10	-1483.67	-1479.2	400

program"), and column "synthesized program LL" is the data log-likelihood of the synthesized program. The column " data set size" is the number of rows in the data set used as the specification for synthesis.

Next, we evaluate how well the posterior distribution of the synthesized probabilistic program matches with the intended distribution. For this purpose, we computed posterior distributions on the skill variables of 3 particular players, and measured how they compared with the analytically computed skill values (which we computed manually from the original probabilistic program). Figure 7 compares the marginal distributions of the skill variables of the original program and the program synthesized by PSKETCH. It can be observed from the plots that the samples converge to the corresponding stationary distributions.

Finally, we evaluate the effectiveness of the approximations for computing likelihoods proposed in Section 4.3. Figure 8 compares the number of sample programs generated per 100 secs with and without the approximation (where the integrals are computed using the approach proposed by Bhat et al. [2]) for each of the benchmarks. It is evident from the plots that using the approximation improves the runtime of the MCMC algorithm by a factor of 1000. From the values of the log likelihood of the synthesized programs (in Table 1), we also see that the approximate computation of likelihood does not affect the quality of the synthesized programs.

Summary. To summarize, our experiments demonstrate that (1) PSKETCH is able to synthesize interesting probabilistic programs from holes in several examples, (2) the programs synthesized by PSKETCH have likelihood values and posterior distributions very close to the target programs we expect to synthesize, and (3) our techniques for approximate computation of likelihoods during MCMC improved the runtime of PSKETCH by a factor of 1000.

6. Related Work

Our work is related to program synthesis in the programming languages area and the rich literature in automatically learning probabilistic models in the machine learning area.

Program synthesis. There have been several approaches for synthesis of deterministic programs [11, 28, 29]. The use of sketches for synthesizing deterministic programs was pioneered by Solar-Lezama et al. [28]. Our tool PSKETCH draws inspiration from this work and applies the notion of sketching to probabilistic programs. Recent work [27] deals with the problem of loop-free binary superoptimization, where the objective is to synthesize an optimal code sequence for a straight-line sequence of instructions. A novel feature of this work is that the search for an optimal code fragment is modeled as a stochastic search problem over the space of deterministic programs. There has been increasing interest in the machine learning community to tackle the problem of program synthesis via generative models. [18] propose an approach to synthesize deterministic programs for multiple related tasks using an hierarchical Bayesian model. [21] propose an approach to synthesize natural code using a variant of probabilistic context free grammars (PCFGs).

In contrast to these approaches, the synthesis algorithm in PSKETCH is a stochastic search over the space of probabilistic programs where the search is guided by the data. The problem of evaluating the likelihood of a probabilistic program with respect to the given data is central to this problem. Our novel approach of using the mixture of Gaussians approximation to solve this problem is an essential ingredient which makes the approach feasible in practice.

Learning probabilistic models. [19] propose an algorithm for learning Bayesian networks. [7] learn high tree-width Markov networks where inference is still tractable. More recently, [5] describe an algorithm for learning a new class of deep probabilistic models called sum-product networks (SPNs). In comparison, PSKETCH employs the notion of a sketch for a probabilistic program to significantly reduce the search complexity. It also uses the structure and semantics of the probabilistic program for efficient likelihood computation that makes the MCMC based search feasible. We are unaware of any work for learning the network structure of such a general class of models (as rich as programs), when only the skeletal dependency structure between the variables is provided as a sketch.

7. Conclusion

Probabilistic programming is an exciting and emerging area of active research with several applications on the horizon. However, writing probabilistic programs requires a certain degree of expertise in machine learning and statistics. In this paper, we have introduced the problem of synthesizing probabilistic programs, and proposed a solution that is based on stochastic search that relies on a mixture of Gaussians approximations, and also exploits domain knowledge in the form of program sketches and datasets. Our tool PSKETCH is able to successfully synthesize a number of complex probabilistic programs. With these encouraging results, we believe that our solution is one step towards making probabilistic programming more accessible to a larger class of application programmers.

Acknowledgments

We thank Timon Gehr and Martin Vechev for their help with fixing errors and improving the presentation of this paper.

References

[1] Y. Bachrach, T. Graepel, T. Minka, and J. Guiver. How to grade a test without knowing the answers—a bayesian graphical model for adaptive crowdsourcing and aptitude testing. *arXiv preprint arXiv:1206.6386*, 2012.

[2] S. Bhat, J. Borgström, A. D. Gordon, and C. V. Russo. Deriving probability density functions from probabilistic functional programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 508–522, 2013.

[3] S. Chib and E. Greenberg. Understanding the Metropolis-Hastings algorithm. *American Statistician*, 49(4):327–335, 1995.

[4] A. Gelman, J. B. Carlin, H. S. Stern, D. B. Dunson, A. Vehtari, and D. B. Rubin. *Bayesian data analysis*. CRC press, 2013.

[5] R. Gens and P. Domingos. Learning the structure of sum-product networks. In *International Conference on Machine Learning (ICML)*, pages 873–880, 2013.

[6] W. R. Gilks, A. Thomas, and D. J. Spiegelhalter. A language and program for complex Bayesian modelling. *The Statistician*, 43(1):169–177, 1994.

[7] V. Gogate, W. A. Webb, and P. Domingos. Learning efficient markov networks. In *Neural Information Processing Systems (NIPS)*, pages 748–756, 2010.

[8] N. D. Goodman, V. K. Mansinghka, D. M. Roy, K. Bonawitz, and J. B. Tenenbaum. Church: a language for generative models. In *Uncertainty in Artificial Intelligence (UAI)*, pages 220–229, 2008.

[9] A. D. Gordon, T. A. Henzinger, A. V. Nori, and S. K. Rajamani. Probabilistic programming. In *Future of Software Engineering, FOSE 2014*, pages 167–181, 2014.

[10] A. D. Gordon, T. A. Henzinger, A. V. Nori, and S. K. Rajamani. Probabilistic programming. In *Future of Software Engineering (FOSE)*, pages 167–181, 2014.

[11] S. Gulwani. Dimensions in program synthesis. In *Principles and Practice of Declarative Programming (PPDP)*, 2010. http://research.microsoft.com/~sumitg/pubs/ppdp10-synthesis.pdf.

[12] R. Herbrich, T. Minka, and T. Graepel. TrueSkill: A Bayesian skill rating system. In *Neural Information Processing Systems (NIPS)*, pages 569–576, 2006.

[13] M. D. Hoffman and A. Gelman. The no-U-turn sampler: Adaptively setting path lengths in Hamiltonian Monte Carlo. *Journal of Machine Learning Research*, in press, 2013.

[14] J. H. Kim and J. Pearl. A computational model for causal and diagnostic reasoning in inference systems. In *IJCAI*, volume 83, pages 190–193. Citeseer, 1983.

[15] S. Kok, M. Sumner, M. Richardson, P. Singla, H. Poon, D. Lowd, and P. Domingos. The Alchemy system for Statistical Relational AI. Technical report, University of Washington, 2007.

[16] D. Koller, D. A. McAllester, and A. Pfeffer. Effective Bayesian inference for stochastic programs. In *National Conference on Artificial Intelligence (AAAI)*, pages 740–747, 1997.

[17] D. Kozen. Semantics of probabilistic programs. *Journal of Computer and System Science (JCSS)*, 22:328–350, 1981.

[18] P. Liang, M. I. Jordan, and D. Klein. Learning programs: A hierarchical bayesian approach. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10), June 21-24, 2010, Haifa, Israel*, pages 639–646, 2010.

[19] D. Lowd and P. Domingos. Learning arithmetic circuits. In *Uncertainty in Artificial Intelligence (UAI)*, pages 383–392, 2008.

[20] D. J. C. MacKay. *Information Theory, Inference & Learning Algorithms*. Cambridge University Press, New York, NY, USA, 2002.

[21] C. J. Maddison and D. Tarlow. Structured generative models of natural source code. In *International Conference on Machine Learning (ICML)*, pages 649–657, 2014.

[22] V. Maz'ya and G. Schmidt. On approximate approximations using gaussian kernels. *IMA Journal of Numerical Analysis*, 16:13–29, 1996.

[23] T. Minka, J. Winn, J. Guiver, and A. Kannan. Infer.NET 2.3, 2009.

[24] A. V. Nori, C.-K. Hur, S. K. Rajamani, and S. Samuel. R2: An efficient mcmc sampler for probabilistic programs. In *AAAI Conference on Artificial Intelligence*. AAAI Press, July 2014.

[25] A. Pfeffer. The design and implementation of IBAL: A general-purpose probabilistic language. In *Statistical Relational Learning*, pages 399–432, 2007.

[26] J. Pfeffer. *Probabilistic Reasoning in Intelligence Systems*. Morgan Kaufmann, 1996.

[27] E. Schkufza, R. Sharma, and A. Aiken. Stochastic superoptimization. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 305–316, 2013.

[28] A. Solar-Lezama, R. M. Rabbah, R. Bodík, and K. Ebcioglu. Programming by sketching for bit-streaming programs. In *Programming Language Design and Implementation (PLDI)*, pages 281–294, 2005.

[29] S. Srivastava, S. Gulwani, and J. Foster. From program verification to program synthesis. In *Principles of Programming Languages (POPL)*, pages 313–326, 2010.