Data-Driven Synthesis of Full Probabilistic Programs

Sarah Chasins, Phitchaya Mangpo Phothilimthana

University of California, Berkeley, USA

Abstract. Probabilistic programming languages (PPLs) provide users a clean syntax for concisely representing probabilistic processes and easy access to sophisticated built-in inference algorithms. Unfortunately, writing a PPL program by hand can be difficult for non-experts, requiring extensive knowledge of statistics and deep insights into the data. To make the modeling process easier, we have created a tool that synthesizes PPL programs from relational datasets. Our synthesizer leverages the input data to generate a program sketch, then applies simulated annealing to complete the sketch. We introduce a data-guided approach to the program mutation stage of simulated annealing; this innovation allows our tool to scale to synthesizing complete probabilistic programs from scratch. We find that our synthesizer produces accurate programs from 10,000-row datasets in 21 seconds on average.

1 Introduction

Probabilistic programming languages (PPLs) enable users who are not experts in statistics to cleanly and concisely express probabilistic models [10, 20, 11]. They offer users simple abstractions and easy access to sophisticated statistical inference algorithms [16, 22, 4, 18] for analyzing their models.

However, writing a PPL model by hand is still challenging for non-statisticians and non-programmers. First, understanding data is difficult. Reviewing large amounts of data to develop a mental model is time-consuming, and humans are prone to misinterpretations and biases. Second, translating insights to a precise statistical model of the data is difficult. To write probabilistic models that reflect their insights, users must first learn some probability theory, understand the subtleties of various probability distributions, and express the details of how different variables in a model should depend on each other.

For these reasons, we believe PPL models should be synthesized from datasets automatically. PPL models offer an interesting point in the modeling design space. Expressing models in PPLs does not make them more expressive or more accurate, but it does give users access to powerful abstractions. They can easily ask how likely an event is in a model, performing complicated inference tasks with a single line of code. They can turn a generative model into a classifier or a predictor in under a minute. They can hypothesize alternative worlds or insert interventions and observe how those edits change outcomes. The PPL synthesis in this paper is not aimed at producing models that exceed the accuracy of state-of-the-art ML on important problems, but we believe a PL-centric approach does put usable, powerful models in the hands of non-experts.

To date, we know of one tool, PSketch[23], that synthesizes PPL programs. PSketch takes as input a PPL sketch and a dataset. A sketch, in this case, is a nearly complete PPL program, with some holes. Once a user expresses which variables may affect each hole, PSketch synthesizes expressions to fill the holes. While synthesizing partial PPL programs is already a tremendous step forward, the sketch writing process still requires users to carefully inspect the data, write most of the program structure, and specify causal dependencies. Ultimately, the user still writes a piece of code that is quite close to a complete model.

We introduce DaPPer (Data-Guided Probabilistic Program Synthesizer), a tool that synthesizes full PPL models from relational datasets. Our system decomposes the PPL synthesis problem into three stages. In the first stage, we generate a graph of dependencies between variables using one of three techniques: including all possible dependencies, analyzing the correlation between variables, or applying network deconvolution [8]. We use the dependency graph to write a program sketch that restricts the program structure. Second, we fill the holes in our sketch using a data-guided stochastic synthesis approach built on top of simulated annealing. At each iteration of our search, we mutate the candidate program and use the input dataset to tune some program parameters. We follow PSketch in computing the candidate's score — its likelihood given the dataset — using Mixtures of Gaussian distributions. Finally, after we obtain an accurate program from the prior stage, we use a redundancy reduction algorithm to make the output program more readable while maintaining its accuracy.

We have evaluated our synthesizer on a suite of 14 benchmarks, a mix of existing PPL models and models designed to stress our tool. Each benchmark in the suite has 10,000 rows of training data and 10,000 rows of test data, both generated from the same probabilistic model; thus, each benchmark is also associated with a ground truth PPL program to which we can compare synthesized programs. In our experiments, our synthesizer produced accurate models in 21 seconds on average. To test whether our approach works on real data, we also used DaPPer to synthesize a model of airline delay data. Leveraging our target PPL's built-in inference functionality, we used this model to predict flight delays.

This paper makes the following contributions:

- We present a tool for synthesizing PPL models from data. To our knowledge, this is the first synthesizer that generates full PPL models.
- We introduce a data-guided stochastic technique for generating candidate programs. Data-guidance improves synthesis time by two orders of magnitude compared to a data-blind approach.
- We compare three techniques for generating dependency graphs from data.
- We present an algorithm for improving program readability after synthesis while maintaining accuracy. We can reduce the size of a synthesized program by up to 8x with less than a 1% penalty in accuracy.

2 Probabilistic Programs

Probabilistic programming languages are standard languages with two additional constructs: (i) random variables whose values are drawn from probability distributions, and (ii) mechanisms for conditioning variable values on the observed values of other variables [12]. Although these constructs form a common backbone,

```
random Boolean Burglary ~ BooleanDistrib(0.001);
random Boolean Earthquake ~ BooleanDistrib(0.002);
random Boolean Alarm ~
  if Burglary then
    if Earthquake then BooleanDistrib(0.95) else  BooleanDistrib(0.94)
  else
    if Earthquake then BooleanDistrib(0.29) else BooleanDistrib(0.001);
random Boolean JohnCalls ~ if Alarm then BooleanDistrib(0.9) else BooleanDistrib(0.05);
random Boolean MaryCalls ~ if Alarm then BooleanDistrib(0.7) else BooleanDistrib(0.01);
```

Fig. 1: The classic burglary model in BLOG (a PPL).

other language features vary greatly from PPL to PPL[10, 20, 11]. Probabilistic programs offer a natural way to represent probabilistic models. For example, the classic burglary model can be expressed with the PPL program in Figure 1.

The output of a probabilistic program is not a value but a probability distribution over all possible values. While a deterministic program produces the same value for a given variable during every execution, a probabilistic program may produce different values. The value of each variable is drawn from one or more probability distributions, as defined by the programmer. We can obtain an approximation of the distribution of a variable by running the program many times. For example, if we run the burglary program in Figure 1 many times, we observe that `Burglary` has the value true in approximately 0.001 of the executions. `Alarm` only becomes common if `Burglary` or `Earthquake` is true, but both are rare, so running the program also reveals that `Alarm` is often false.

Programmers can add observations in PPLs to obtain posterior probability distributions conditioned on the observations. For example, if we run our sample program with an observation statement **`obs`** `JohnCalls = true`, the program rejects executions in which `JohnCalls` is false, and we observe that in many runs `Burglary` is also true. For a thorough introduction to PPLs, we recommend [12].

3 System Overview

We will explain DaPPer with a working example, a model of how a student's tiredness and skill level affect performance on a test. The inputs to our tool are a relational dataset (e.g. Table 1) and a hypothesis about the direction of causal links between variables. Each column in a dataset is treated as a variable in the output program, and each row represents an independent run of the program. A causation hypothesis is an ordering of the dataset column identifiers; for our running example, the hypothesis might be `tired` $\rightarrow$ `skillLevel` $\rightarrow$ `testPerformance`. The order specifies the direction of dependencies but does not restrict which variables are connected. From the given order, we can conclude that if `tired` and `testPerformance` are related, `tired` affects `testPerformance`, rather than `testPerformance` affecting `tired`. While this means that our tool still demands some insights from users, they only have to use their knowledge of the world to guess in which direction causality may flow; that is, our tool does not ask 'is there a relationship between tiredness and test performance?' but it asks 'if there is a relationship between the two, does tiredness affect test performance, or does test performance affect tiredness?' In Figure 2, we show the ground truth program that produced the data in Table 1.

tired	skillLevel	testPerformance
true	10.591	27.437
false	12.862	67.976
false	8.727	70.787
true	10.333	31.113
true	11.440	31.592
...	...	..

Table 1: Dataset with the tiredness, skill level, and test performance of several students.

```
random Boolean tired ~ BooleanDistrib(.5);
random Real skillLevel ~ Gaussian(10, 7);
random Real testPerformance ~
  if skillLevel > 13.0 then
    if tired then Gaussian(70, 15) else Gaussian(95, 5)
  else
    if tired then Gaussian(30, 15) else Gaussian(70, 5);
```

Fig. 2: Running example, a program that expresses a model of how a student's tiredness and skill level affect test performance.

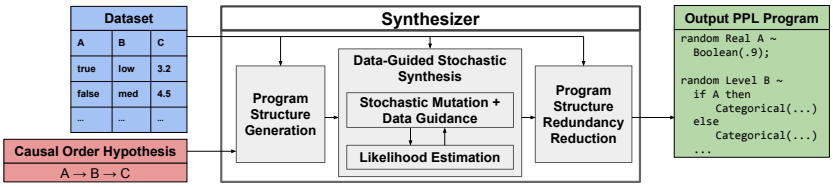


Fig. 3: A system diagram illustrating the workflow of the synthesizer.

DaPPer generates PPL programs by decomposing the synthesis task into three subtasks: (i) dependency graph generation, (ii) data-guided stochastic synthesis, and (iii) redundancy reduction. In this section, we briefly discuss their roles, and how they interoperate, as illustrated in Figure 3.

3.1 Dependency Graph Generation

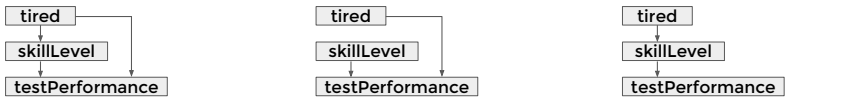
The synthesizer's first task is to determine whether any given random variable — any given dataset column — depends on any other random variables. This problem corresponds to the model selection problem in Bayesian networks. In our context, this is the *dependency graph generation problem* because a directed graph of which variables affect which other variables defines a program structure.

We explore three techniques for generating dependency graphs. First, a Complete approach produces the largest possible dependency graph that the user's causation hypothesis permits. Second, a Simple Correlation approach adds edges greedily in order of correlation, from highest to lowest, excluding edges for which there already exists a path. Third, we use an existing Network Deconvolution algorithm [8]. Given a dataset like the one in Table 1 and the causation hypothesis represented by the order of the columns (tired → skillLevel → testPerformance), the three techniques produce the dependency graphs depicted in Figure 4.

Given a dependency graph, we generate a sketch. For each variable x depending on $\{x_1, x_2, ...\}$, we define x as a nested conditional expression with holes:

```
x ~ if (x1 ?? ??)
      then if (x2 ?? ??) ...
      else if (x2 ?? ??) ...
```

For each condition `(xi ?? ??)`, the first `??` is a comparison operator, and the second `??` is an expression. Figure 5 shows the sketches generated from the dependency graphs in Figure 4.



(a) Complete (b) Simple correlation (c) Network deconvolution

Fig. 4: The dependency graphs generated for the dataset sampled in Table 1.

```
random Boolean tired ~
  BooleanDistrib(??);
random Real skillLevel ~
  if tired then ?? else ??;
random Real testPerformance ~
  if tired then
    if (skillLevel ?? ??)
      then ?? else ??
  else
    if (skillLevel ?? ??)
      then ?? else ??;
```

(a) Complete

```
random Boolean tired ~
  BooleanDistrib(??);
random Real skillLevel ~ ??;
random Real testPerformance ~
  if tired then
    if (skillLevel ?? ??)
      then ?? else ??
  else
    if (skillLevel ?? ??)
      then ?? else ??;
.
```

(b) Simple correlation

```
random Boolean tired ~
  BooleanDistrib(??);
random Real skillLevel ~
  if tired then ?? else ??;
random Real testPerformance ~
  if (skillLevel ?? ??)
    then ?? else ??;
.
```

(c) Network deconvolution

Fig. 5: The program skeletons generated from the dependency graphs in Figure 4. We use **??** to represent a part of the program that we have not yet synthesized.

3.2 Data-Guided Stochastic Synthesis

The second task is to complete the holes **??** in the program sketch generated by the previous step. We use simulated annealing (SA) to complete this task.

In each SA iteration, DaPPer creates a new program candidate from a current candidate by (i) mutating an expression, then (ii) deterministically updating all parameters in the program associated with the mutated expression, using knowledge about the data. Consider the candidate program in Figure 6(a), a completion of the Figure 5(c) program sketch. To generate a new candidate, our tool randomly mutates one condition, changing the RHS of a comparison from 15 to 16; this produces the sketch in Figure 6(b). With the condition changed, the parameters for the distributions are out of date. DaPPer identifies all rows of the input data in which `skillLevel` is less than 16, then uses those rows to select parameters for the distributions in the true branch, and the remaining rows for the false branch, producing the new program in Figure 6(c).

To evaluate programs, DaPPer uses a custom likelihood estimation approach.

3.3 Redundancy Reduction

Sometimes the most accurate model that results from the synthesis step is a model that distinguishes between many cases, even more than the ground truth. Such models can be very large, often unreadable. Because our goal is to produce readable programs from which users can extract high-level insights, this is undesirable for our purposes.

To improve the readability of our outputs, we developed a redundancy reduction algorithm that collapses similar branches. Because this is applied as a final processing stage to an already synthesized program, the reduction process is very fast, and users can easily and quickly tune the amount of reduction to their needs, based on the output at hand.

```
random Boolean tired ~
  BooleanDistrib(.500);
random Real skillLevel ~
  if tired
    then Gaussian(9.987, 7.000)
    else Gaussian(10.000, 7.003);
random Real testPerformance ~
  if (skillLevel < 15)
    then Gaussian(44.875, 7.729)
    else UniformReal(25.300, 120.276);
```

(a) Current candidate

```
random Boolean tired ~
  BooleanDistrib(.500);
random Real skillLevel ~
  if tired
    then Gaussian(9.987, 7.000)
    else Gaussian(10.000, 7.003);
random Real testPerformance ~
  if (skillLevel < 16)
    then Gaussian(??, ??)
    else UniformReal(?? , ??);
```

(b) Stochastic mutation

```
random Boolean tired ~
  BooleanDistrib(.500);
random Real skillLevel ~
  if tired
    then Gaussian(9.987, 7.000)
    else Gaussian(10.000, 7.003);
random Real testPerformance ~
  if (skillLevel < 16)
    then Gaussian(46.227, 7.663)
    else UniformReal(25.300, 125.113);
```

(c) Data-guided completion

Fig. 6: One mutation of the program in Figure 5(c). **Pink** highlights the changes.

prog	::=	**statement prog** | ϵ
statement	::=	'random' **type ident** '~' **expr** ';'
type	::=	'Boolean' | 'Real' | **categoricalTypeIdent**
expr	::=	**distrib** | **condExpr**
condExpr	::=	'if' **cond** 'then' **expr** 'else' **expr**
distrib	::=	'BooleanDistrib(' **real** ')' | 'Categorical({' **categoricalMap** '})'
		| 'Gaussian(' **real** ',' **real** ')' | 'UniformReal(' **real** ',' **real** ')'
		| 'Gamma(' **real** ',' **real** ')' | 'Beta(' **real** ',' **real** ')'
categoricalMap	::=	**categoricalValue** '->' **real** | **categoricalMap** ',' **categoricalMap**
cond	::=	**ident** | **ident cmpOp cmpExpr** | **cond** '|' **cond**
cmpOp	::=	'==' | '>' | '<'
cmpExpr	::=	**ident** | **boolean** | **categoricalValue** | **real**
		| **cmpExpr numOp cmpExpr**
numOp	::=	'+' | '-' | '*'

Fig. 7: The subset of the BLOG language used by our synthesizer.

4 Language

DaPPer generates programs from the grammar shown in Figure 7. This grammar represents a subset of the BLOG language [20], centered on the features necessary for declaring random variables.[1] While BLOG has many interesting features that set it apart from other PPLs, such as open-universe semantics, our synthesized programs do not make use of these. DaPPer needs only the features that allow it to introduce random variables drawn from distributions and describe how they depend on other random variables.

Many PPLs can express such programs, so many PPLs would be reasonable target languages. We chose to synthesize programs in the BLOG language, but with small changes to our code generator, DaPPer could easily target others.

5 Generating Dependency Graphs

We use dependency graphs to generate a skeleton program structure, a program with variable declarations and partial conditional expressions in their definitions, but without the conditions or bodies. To generate these dependency graphs, we have tested three approaches, which we describe here.

[1] We leave out details of type declarations, which BLOG requires and our tool synthesizes, but which present no interesting technical challenges.

5.1 Complete

The complete approach constructs a dependency graph based on the assumption that each variable depends on all other variables that precede it in the user's causation hypothesis. Note that this approach does not use the input dataset. If the user provides the hypothesis $A \to B \to C \to D$, the complete approach produces a graph in which A depends on no other variables, B depends on $\{A\}$, C depends on $\{A, B\}$, and D depends on $\{A, B, C\}$.

If the causation hypothesis is correct, this approach is always sufficient to express the ground truth. It breaks the outcomes into the largest number of cases and thus theoretically allows the greatest customization of the program to the input data. However, it may also introduce redundancy, distinguishing between cases even when the distinction does not affect the outcome.

5.2 Correlation Heuristic

The correlation heuristic approach uses information from the input dataset as well as the causation hypothesis. It calculates the correlation for every pair of columns in the dataset. The pairs are sorted according to the effect size of the correlation. We iterate through the sorted list of pairs, checking for each pair (A, B) whether there is already a path in the dependency graph between A and B. If yes, we do nothing. If no, we add an edge between A and B; the direction of the edge is determined by the positions of A and B in the causation hypothesis ordering. If we reach a point in the list of pairs where the correlation effect size or statistical significance is very low, we stop adding edges.

See Appendix 13 for a summary of how we produce correlation measures for columns with incompatible types.

5.3 Network Deconvolution

Our final approach uses the network deconvolution algorithm, developed by Feizi et al [8], a method for inferring which nodes in a network directly affect each other, based on an observed correlation matrix that reflects both direct and indirect effects. To build a PPL model, we can observe the correlation between each pair of columns, but should only condition a variable on the variables that directly affect it. Thus, a direct link from x to y in network deconvolution corresponds to an immediate dependence of y on x in a PPL model.

Network deconvolution takes as input a *similarity* matrix. We use a symmetric square matrix of correlations, reflecting the association of each column with each other column in the dataset. Each entry in the network deconvolution output matrix represents the likelihood that a column pair is connected by a direct edge. For each entry in the output matrix, we add an edge to the dependency graph if it is above a low threshold.

6 Data-Guided Stochastic Synthesis

DaPPer applies simulated annealing (SA) to synthesize PPL programs. We use an exponential cooling schedule and the standard Kirkpatrick acceptance probability function. To apply SA, we must generate new candidate programs that

are 'adjacent' to existing programs. Our synthesizer decomposes the process of creating a new candidate program into two stages: make a random mutation of the current candidate program (Section 6.1), then tune all parameters in the affected subtree of the AST to best match the input dataset (Section 6.2). Once a new candidate has been produced, SA scores the candidate to decide whether to accept or reject it (Section 6.3).

6.1 Mutations

The first step in creating a new adjacent program is to randomly mutate the current program. We allow three classes of mutation, described below.

Conditions Our synthesizer is permitted to synthesize conditions of a restricted form (see **cond** in Figure 7). They must have a single identifier (fixed based on the dependency graph) on the LHS, and an expression on the RHS. Because we deterministically generate fixed RHSs for conditions with Boolean and categorical variables in the LHS (e.g., `boolVar == true`, `boolVar == false`), the mutation process may not manipulate those conditions. Instead, its primary role is to generate new RHSs for conditions associated with real-valued variables. To alter a RHS, the mutator may: (i) replace any constant or use of a real-valued variable with a new constant or real-valued variable, (ii) slightly adjust a current constant, (iii) add, remove, or change a **numOp**, (iv) change a **cmpOp**.

Branches Less commonly, the mutator may add or remove a condition associated with a real-valued variable. The mutator may not alter the structure of the dependency graph, but it may add a branch to an existing case split or remove a branch if it has more than two.

Distribution Selection Finally, the mutator may alter what type of distribution appears in the body of a conditional, for definitions of real-valued variables. For instance, it may change a **`Gaussian`** to a **`UniformReal`** distribution.

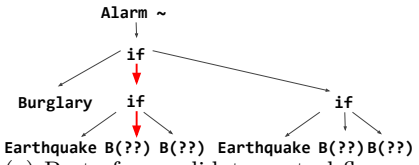
6.2 Data Guidance

Once we obtain the control flow for a new candidate program from the mutator, we tune the distribution parameters to fit the dataset. For each distribution node in the abstract syntax tree, we identify the path condition associated with the node. We convert the path condition into a filter on the input dataset. For instance, consider the control flow in Figure 8(a). To produce the parameter for the first Boolean distribution node, we would produce the path condition $Burglary \wedge Earthquake$. Using this as a filter over the input dataset would produce the rows highlighted in Figure 8(b). Once we have identified the subset of the dataset consistent with a distribution's path condition, we use the subset to calculate appropriate distribution parameters. For instance, if the distribution is a Gaussian, we calculate the mean and variance.

Once DaPPer completes this process for all distributions whose path conditions are affected by the mutation, the candidate generation stage is complete.

6.3 Likelihood Estimation

To evaluate how well a program models a dataset, we must compute the likelihood $\mathcal{L}(P|D)$ of a candidate program P given the input data D. Computing



(a) Part of a candidate control flow.

Burglary	Earthquake	Alarm
T	F	T
F	F	F
T	T	T
F	T	F
T	T	F
...	...	...

(b) Red rows satisfy (a)'s red path condition.

Fig. 8: Filtering a dataset for a path condition in the classic burglary model.

the exact likelihood [6] requires expensive integral computations, which makes scoring slow. Thus, we adopt the method used by PSketch [23] for approximating likelihood using Mixtures of Gaussian (MoG) distributions, which they show is three orders of magnitude faster than precise likelihood calculation [23].

The approach is to symbolically approximate every variable in a candidate program with MoG or Bernoulli distributions. We approximate real-valued expressions using a MoG, whose probability density function (PDF) is:

$$MoG(x; n, \boldsymbol{w}, \boldsymbol{\mu}, \boldsymbol{\sigma}) = \sum_{i=1}^{n} w_i \cdot g(x; \mu_i, \sigma_i)$$

where $\boldsymbol{w}, \boldsymbol{\mu}$, and $\boldsymbol{\sigma}$ are vectors of size n, representing the weight, mean, and standard deviation of each Gaussian distribution in the mixture. The function g is the PDF of a univariate Gaussian distribution. A Boolean expression is simply modeled as a Bernoulli distribution, $Brn(x; p)$. Therefore, we approximate each variable v in a program as a MoG with the PDF $MoG_v(x) = MoG(x; n_v, \boldsymbol{w_v}, \boldsymbol{\mu_v}, \boldsymbol{\sigma_v})$ or a Bernoulli with the PDF $Brn(x; p_v)$.

The approximation of $\mathcal{L}(P|D)$ is the product of the likelihood of all possible values of all variables from D given the program P:

$$\mathcal{L}(P|D) = \prod_{v \in P_{RV}} \prod_{x \in D[v]} MoG_v(x) \times \prod_{v \in P_{BV}} \prod_{x \in D[v]} Brn(x; p_v)$$

where P_{RV} is a set of real variables and P_{BV} is a set of Boolean variables in the program P that appear in data D. $D[v]$ is a set of values of the variable v in D.

In addition to the distributions supported by PSketch, we add support for categorical distributions and uniform distributions, which we describe here.

Categorical Distribution A categorical distribution specifies probabilities for each value in a finite discrete set. The PDF of a categorical distribution is:

$$Ctg(x; \{(x_i \to p_i) | i \in \{1, ..., k\}\}) = p_i \text{when } x = x_i$$

We introduce reduction rules to symbolically evaluate expressions with categorical distributions, shown in Figure 9. The first rule evaluates an `if` expression, which may contain categorical distributions. The second rule evaluates a `case` expression. Although a `case` expression can be desugared to a

$$
\begin{array}{rl}
[\![ite(Brn(x;p), Y_1, Y_2)]\!] := & [\![(p \otimes Y_1) \oplus ((1-p) \otimes Y_2)]\!] \\
[\![case([(Ctg_v(x) == x_1, Y_1), (Ctg_v(x) == x_2, Y_2), ..., & \\
\quad (Ctg_v(x) == x_k, Y_k)])]\!] := & [\![(p_1 \otimes Y_1) \oplus (p_2 \otimes Y_2) \oplus ... \oplus (p_k \otimes Y_k)]\!] \\
\text{where } Ctg_v(x) = Ctg(x; \{(x_i \to p_i) | i \in \{1, ..., k\}\}) & \text{where } Y_i \text{ is either Bernoulli, categorical, or MoG}
\end{array}
$$

$$
\begin{array}{rl}
[\![Brn(x;p_1) \oplus Brn(x;p_1)]\!] := & Brn(x; p_1 + p_2) \\
[\![Ctg(x; \{(x_i \to p_i^1) | i \in \{1, ..., k\}\}) \oplus & \\
\quad Ctg(x; \{(x_i \to p_i^2) | i \in \{1, ..., k\}\})]\!] := & Ctg(x; \{(x_i \to p_i^1 + p_i^2) | i \in \{1, ..., k\}\}) \\
[\![MoG(x; n_1, \boldsymbol{w_1}, \boldsymbol{\mu_1}, \boldsymbol{\sigma_1}) \oplus MoG(x; n_2, \boldsymbol{w_2}, \boldsymbol{\mu_2}, \boldsymbol{\sigma_2})]\!] := & MoG(x; n_1 + n_2, \boldsymbol{w_1} \| \boldsymbol{w_2}, \boldsymbol{\mu_1} \| \boldsymbol{\mu_2}, \boldsymbol{\sigma_1} \| \boldsymbol{\sigma_2}) \\
& \text{where } \| \text{ represents vector concatenation}
\end{array}
$$

$$
\begin{array}{rl}
[\![c \otimes Brn(x;p)]\!] := & Brn(x; c \times p) \\
[\![c \otimes Ctg(x; \{(x_i \to p_i) | i \in \{1, ..., k\}\})]\!] := & Ctg(x; \{(x_i \to c \times p_i) | i \in \{1, ..., k\}\}) \\
[\![c \otimes MoG(x; n, \boldsymbol{w}, \boldsymbol{\mu}, \boldsymbol{\sigma})]\!] := & MoG(x; n, c \times \boldsymbol{w}, \boldsymbol{\mu}, \boldsymbol{\sigma})
\end{array}
$$

Fig. 9: Reduction rules to symbolically execute expressions that use categorical distributions.

nested `if` expression, the PDF that results is often less precise. In particular, $case([(Ctg_v(x) == x_1, Y_1), (Ctg_v(x) == x_2, Y_2), ...])$ can be desugared to $ite(Ctg_v(x) == x_1, Y_1, ite(Ctg_v(x) == x_2, Y_2, ...))$, where $Ctg_v(x) = Ctg(x; \{(x_i \to p_i) | i \in \{1, ..., k_v\}\})$. When we evaluate the former expression, we expect the resulting distribution to be the summation of Y_1, Y_2, ..., Y_{k-1}, Y_k weighted by p_1, p_2, ..., p_{k-1}, p_k respectively. However, if we evaluate the latter expression using the first rule, we will obtain the summation of Y_1, Y_2, ..., Y_{k-1}, Y_k weighted by p_1, $(1-p_1)p_2$, ..., $(\prod_{i=1}^{k-2}(1-p_i))p_{k-1}$, $\prod_{i=1}^{k-1}(1-p_i)$ respectively. This is because the *ite* rule is designed for the scenario in which path conditions are independent. Therefore, we introduce the *case* rule to handle `case` expressions whose conditions are dependent and mutually exclusive in order to obtain better likelihood estimations. The remaining rules in Figure 9 define how to evaluate $\oplus$ and $\otimes$ used in the *ite* and *case* rules.

Uniform Real Distribution A uniform real distribution has constant probability for any real number between a lower bound a and an upper bound b. The PDF of a uniform real distribution is defined as:

$$
Uniform(x; a, b) = \begin{cases} \frac{1}{b-a} & \text{if } a \leq x \leq b \\ 0 & \text{otherwise} \end{cases}
$$

We approximate a uniform distribution as follows:

$$
\begin{aligned}
&[\![Uniform(x; a, b)]\!] := MoG(x; n, w, \boldsymbol{\mu}, \boldsymbol{\sigma}) \\
&\text{where } w_i = \frac{1}{n}, \mu_i = a + (i + \frac{1}{2}) \cdot \frac{b-a}{n}, \sigma_i = \frac{b-a}{n}
\end{aligned}
$$

We use $n = 32$. We can obtain better approximations by increasing n, but at the cost of slower evaluation.

7 Redundancy Reduction

To improve the readability of our outputs, we have developed a simple redundancy reduction algorithm that combines the similar branches of a given conditional expression. The key idea is to compare the parameters of branches' descendant distributions. For a pair of branches, we align the two sets of descendant distributions according to their associated path condition suffixes. These distribution pairs are the pairs that we combine if we collapse the branches into a single branch for executions that satisfy *either* of their path conditions. The decision to collapse them is based on both the differences in distribution pairs' parameters and on how much data DaPPer used to tune them.

For a concrete example, recall that in the classic Burglary model, `MaryCalls` does not depend on `JohnCalls`. After stochastic synthesis, we might see the following snippet within the `MaryCalls` definition: `if JohnCalls then Boolean(0.010) else Boolean(0.008)`. The parameters of the distributions in these two branches are very close. Further, we know both parameters were trained on a small number of rows. Given normal reduction parameters, this would lead our algorithm to collapse the two branches into something like `Boolean(0.009)` – essentially to conclude that `MaryCalls` does not depend on `JohnCalls`.

Our algorithm accepts two user-selected threshold parameters. By manipulating these parameters, users can explore different levels of readability without having to re-synthesize. Reduction is applied as a fast post-processing step, after the bulk of the synthesis is completed, so users can quickly and easily tune the amount of redundancy reduction to their needs. They can choose to edit and run analyzes on a readable version, knowing how much accuracy they have sacrificed, or they can temporarily adjust reduction parameters to extract high-level insights about program structure without permanently sacrificing accuracy.

See Appendix 14 for the full algorithm and details of the motivation.

8 Limitations

We see several limitations of the current synthesizer, all of which suggest interesting directions for future work.

Relational Input Data Our approach handles only relational datasets, specifically relational datasets that treat each row as an independent run of the program. While there are many such datasets and our tool already allows us to model many interesting processes, our current technique does not apply for datasets that cannot be transformed into this format. Thus, we cannot take advantage of some of the BLOG language's more interesting features (e.g., open-universe semantics, which allows programs in the language to represent uncertainty about whether variables exist, or how many variables there are of a given type). For our current synthesis model, we must know the number of variables.

Hidden Variables Our synthesis model assumes there are no hidden variables, that no additional columns of data about the world are necessary to produce a correct output. Our decision to exclude hidden variables is one of the crucial differences from the PSketch[23] approach. While PSketch is targeted at programmers looking to write functions that include randomness, we want

our tool to be accessible to scientists and social scientists modeling real world phenomena. For these purposes, we expect that hypothesizing the existence of a hidden variable — which may have no correspondence to any real world variable — would only confuse the user and make output models less intelligible and less useful to the target audience. We see the addition of *optional* hidden variable introduction as an interesting technical challenge and a good direction for future work. However, we would never make hidden variable hypotheses the default. Although we believe this is a improvement over PSketch from the perspective of our target user, this is a limitation from the perspective of a PSketch user.

Restricted Grammar For this first foray into synthesis of full PPL programs, we selected a fairly restricted grammar. We examined many existing BLOG models and designed a grammar that would express the sorts of models people already like to write. However, we find it easy to imagine models we would like to obtain that cannot be expressed in our chosen subset of BLOG. Although this may be the most serious limitation, it is also the most easily addressed, since we can cleanly extend our technique by simply expanding the grammar and the set of allowable mutations. Because each increase in the grammar size also expands the search space, this will probably be more than a trivial extension. We expect it may offer a good setting for exploring the use of a probabilistic language model and weighted search for a program synthesis application.

9 Evaluation

We evaluated DaPPer on a suite of 14 benchmarks. Each benchmark consists of 10,000 rows of training data, 10,000 rows of test data, and the ground truth BLOG program used to generate both datasets. Some benchmarks were taken directly from the sample programs packaged with the BLOG language. Since these were quite simple, we also wrote new programs to test whether our tool can synthesize more complex models. The DaPPer source and all benchmark programs and associated datasets are available at `github.com/schasins/PPL-synthesis`. DaPPer synthesized accurate programs for all benchmarks, taking less than 21 seconds on average. We also used DaPPer to generate a model of a real flight delay dataset.

9.1 Dependency Graph Generation

We start with an exploration of how dependency graphs affect synthesis time and accuracy. Recall that the Complete approach produces the largest possible dependency graph. The Correlation approach produces graphs smaller than Complete graphs but usually larger than Network Deconvolution (ND) graphs.

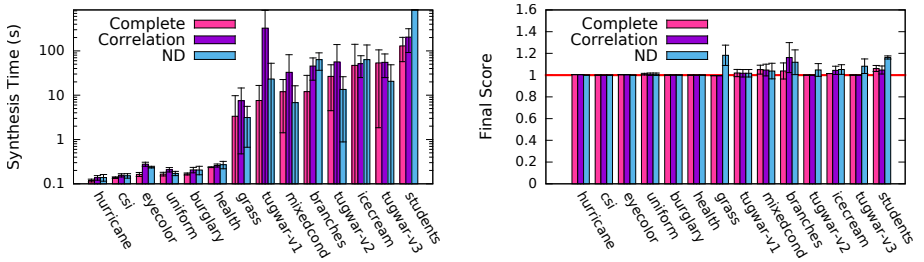
Figure 10(a) shows how the choice of dependency graph affects synthesis time. Each bar represents the average time DaPPer took to synthesize a program whose score on the test dataset is within 10% of the ground truth program's. Each synthesis task ran for a fixed number of SA iterations. The Figure 10(a) timing numbers represent the first point during those iterations at which the current candidate program achieves the target likelihood score on the test (hold-out) dataset. For all benchmarks except 'students,' all dependency graphs were

sufficient to reach the target accuracy within the allotted SA iterations. The average times to reach the target accuracy using the Complete and Correlation approaches were 20.90 and 55.77 seconds, respectively. If we exclude 'students,' we can compare all three; Complete averaged 12.54 seconds, Correlation 44.40, and ND 15.02.

We observe a number of trends playing out in the timing numbers. First, Complete gains a small early time advantage by generating a dependency graph without examining the input data, while Correlation and ND both face the overhead of calculating correlations. This head start gave Complete the win for the first six benchmarks. Second, using the dependency graph closest to that of the ground truth confers a time advantage. If the ground truth dependency graph is dense, Complete is typically closest, so Complete finishes first (e.g., 'tugwar-v1'). If the ground truth dependency graph is sparse, ND is usually closest, so ND finishes first (e.g., 'tugwar-v3'). Finally, if any approach eliminates a necessary dependency, synthesis may fail to reach the target accuracy. Recall from Figure 4 that ND dropped the direct dependence of `testPerformance` on `tired`. Thus, ND never reached the target accuracy for the 'students' benchmark.

Next, we evaluate whether DaPPer's performance is within the acceptable range. We cannot make a direct comparison to the most similar tool, PSketch, because the PSketch task is substantially different. Instead, we include some PSketch performance numbers to give a sense of the acceptable timescale. PSketch synthesized partial PPL programs using small datasets (100–400 rows) in 146 seconds on average. Note that large datasets are desirable because they result in more accurate programs, but they make synthesis slower because the likelihood estimator must use all the data to calculate a score at each iteration. To synthesize part of the 'burglary' model, PSketch took 89 seconds, while DaPPer synthesized a full 'burglary' model in 0.17 seconds. Again, since the tasks are very different, these numbers do not indicate that DaPPer outperforms PSketch. However, we are satisfied with DaPPer's synthesis times overall.

Figure 10(b) shows the likelihood scores of the final synthesized programs on the test datasets, normalized by the scores of the ground truth programs. Overall, the scores reached by the Complete, Correlation, and ND approaches, averaged



(a) Time (in log scale) to synthesize a program that achieves a target likelihood score on the test data. For each benchmark, the target is a score within 10% of the ground truth's score.

(b) The likelihood scores of the final synthesized programs on the test data, normalized by the score of the ground truth programs. Lower scores are better.

Fig. 10: Comparison of the three dependency graph generation approaches.

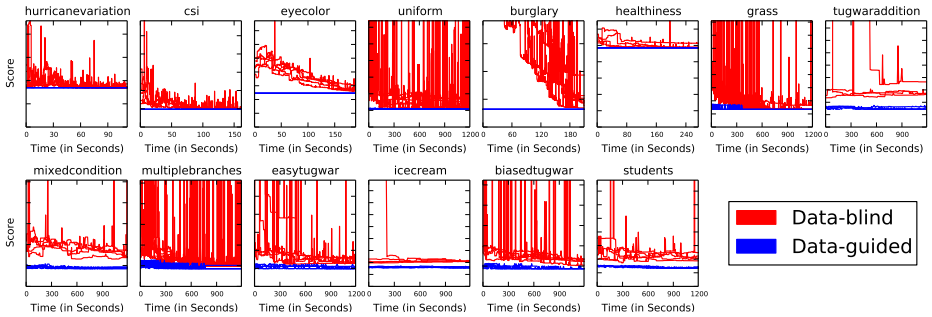


Fig. 11: Score over time for data-blind vs. data-guided synthesis. Lower is better.

across benchmarks, were 1.014, 1.024, and 1.051, respectively. As expected, larger dependency graphs typically allowed the synthesizer to reach better scores. Thus, Complete always produced likelihoods very close to those of the ground truth programs, with Correlation performing slightly worse, and ND worst of all. Still, even ND always produced likelihoods within 20% of the ground truth.

Given Complete's dominance in both synthesis time and accuracy, we conclude that Complete is the best dependency graph approach of the three we tried. If we were to select an approach on a case-by-case basis, we would only switch away from the Complete strategy when faced with a dataset for which we strongly suspect the dependency graph is sparse, and even then only if faster synthesis time is more critical than accuracy. The dominance of the Complete approach drove us to develop our redundancy reduction algorithm, which allows us to recover small, readable programs from large ones.

9.2 Data-Guided vs. Data-Blind Stochastic Synthesis

One of the primary innovations of our tool is its use of input data not only to score candidate programs but also to generate them. We evaluate DaPPer against DaPPer-blind. DaPPer-blind is a simple data-blind variation on our tool. It is identical to DaPPer in every way except: (i) in addition to DaPPer mutations, it may mutate distribution parameters and (ii) it does not run data-guided parameter adjustment after mutations.

Recall that after each mutation, DaPPer identifies affected distributions and tunes their parameters to reflect the input data that corresponds to the new path condition. Thus, the data-guided approach has the advantage of always producing programs tuned to the input data. However, filtering the data and calculating the appropriate parameters does impose a time penalty. For this reason, DaPPer-blind can complete more mutations per unit of time than DaPPer. Thus, it is not immediately clear which approach will perform better.

Our empirical evaluation reveals that the data-guided approach outperforms the data-blind approach. Figure 11 shows how the likelihood score changed over time for five runs of DaPPer and DaPPer-blind, for each benchmark. While 100% of the data-guided runs reached a likelihood within 10% of the ground truth's likelihood, only 36% of the data-blind runs reached that same target level. For the data-guided runs, the average time to achieve the target likelihood was 20.9

seconds. For the 36% of data-blind runs that did reach the target likelihood, the average time was 151.6 seconds. Thus, using a data-guided program generation approach offers at least a 7x speedup compared to a data-blind approach.

We can acquire a better speedup estimate by including the benchmarks for which the data-blind approach never reached the target accuracy. For each benchmark, we identified the best (lowest) score that all runs managed to reach (i.e. the best score of the worst run). Reaching these scores took data-blind synthesis an average of 347.63 seconds and took data-guided synthesis an average of 0.54 seconds, indicating that data-guidance provides a 648.9x speedup.

9.3 Redundancy Reduction

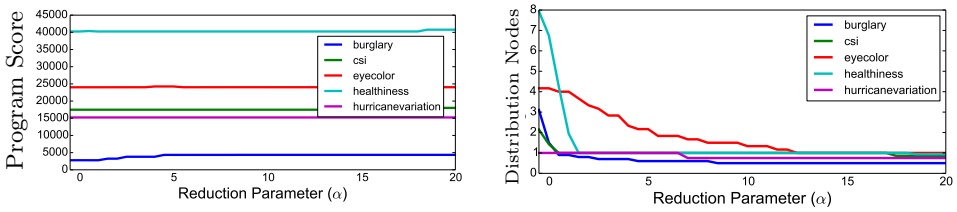
To explore the tradeoff between accuracy and readability, we evaluated how much accuracy we lose by applying our redundancy reduction algorithm to our synthesized programs. While Network Deconvolution offers small, readable programs by default, the other techniques do not. In this section, we explore the effects of redundancy reduction on programs synthesized using Complete dependency graphs on a subset of the 14 benchmarks[2]. We observe up to a 7.9x reduction in program size, with negligible decreases in accuracy.

Recall that we synthesize programs in which all AST leaf nodes are distribution nodes. For this reason, the number of distribution nodes is the most informative measure of program size and complexity.

As Figure 12(a) reveals, the reduction process does not make program alterations that substantially alter the likelihood score. However, it does significantly reduce the size of the program, producing output programs that are much more readable. Figure 12(b) shows the effects on program size, depicting the ratio of the number of distribution nodes in the output to the number of distribution nodes required to express the ground truth. We see that as the reduction parameter α increases, the synthesized programs ultimately converge to the ground truth range, but do so gradually enough that the user can explore a variety of different structures. As we see in Figure 12(a), even when users set α quite aggressively, the reduction algorithm does not tend to make merges that substantially alter the likelihood score. Most importantly, we think the benefits for readability and for extracting high-level insights are clear. For instance, in the case of the 'healthiness' model, reduction collapses a program with 127 distinct distribution nodes to a much more readable program with 16 distribution nodes.

Overall, we find that redundancy reduction allows us to benefit from the accuracy and fast synthesis times of the Complete approach without sacrificing readability and editability. For a more concrete illustration of the resultant readability, see Appendix 15's side-by-side comparison of a ground truth program and a synthesized program after redundancy reduction.

[2] We test on the subset of benchmarks for which we can align branches using exact path condition matches. We expect to expand the path condition matching scheme to align branches with close conditions in future work.



(a) Likelihood scores of the reduced programs. Lower is better. We vary α, a parameter of the reduction algorithm that users can adjust.

(b) Number of distribution nodes in the reduced programs, normalized by the ground truth programs' number of distribution nodes.

Fig. 12: Effects of redundancy reduction with varying α parameter values.

9.4 Case Study: Airline Delay Dataset

Although testing on data for which we have a ground truth model is the best way to investigate whether DaPPer produces correct programs, we also want to be sure that DaPPer functions well on real data. To that end, we completed a case study using our tool to produce a probabilistic model of a popular airline delay dataset from the U.S. Department of Transportation [1]. We selected this dataset because it has already been thoroughly studied, explored, and visualized. Thus, although we lacked a ground truth PPL model for this dataset, we knew from past work that we should expect delays to vary according to days of the week [15] and to increase over the course of the day [35].

We ran DaPPer on a dataset with 447,013 rows. The output program indicates that delays vary by day, reflecting the findings of Heike et al. [15]. It also indicates that delays rise as the departure time (time of day) rises, reflecting the findings of Wicklin et al. [35]. Taking advantage of BLOG's built-in inference algorithms, we used this model to predict flight delays on a holdout set of 10,000 dataset rows. On average, the model's predictions were off by less than 15 minutes. While 15 minutes is substantial, it is worth noting that delays in the dataset range from -82 to 1971. For comparison, a baseline predictor that always guessed the average flight delay had a root-mean-square-error (RMSE) of 39.4, while the DaPPer predictor had an RMSE of 24.1.

10 Related Work

The body of research that addresses learning programs from data is far greater than we can cover, encompassing the entire fields of machine learning and program synthesis. Since we are interested in generating models that are both readable and probabilistic, we will limit our discussion to approaches that offer at least one of those characteristics.

10.1 Readable and Probabilistic

Of the related work, our goals are most closely aligned with the goals of PSketch [23]. The primary difference between our tool and PSketch is the target user.

We want DaPPer to be accessible to a user who would not manually code even a partial PPL model. Naturally, this difference in target user comes with a number of technical differences. First and foremost, while PSketch requires the user to write a program sketch — including specifying which variables may affect each program hole — our tool requires no coding. This brings us to the second primary difference, which is that while PSketch can work over any dataset for which the user can write most of the model, our tool is targeted specifically at relational datasets in which each row represents an independent draw from the model. Third, our tool does not hypothesize the existence of hidden variables that do not appear in the input dataset. Fourth, PSketch is designed for small datasets (they tested on datasets up to size 400), while DaPPer is designed to handle datasets with hundreds of thousands of rows. To make this feasible, DaPPer uses data-guided mutations, while PSketch's mutations are data-blind.

10.2 Readable and Deterministic

There has been a massive body of work in deterministic program synthesis and program induction, some of which may be useful in future iterations of our tool. Many synthesizers use off-the-shelf constraint solvers to search for candidate programs and verify their correctness [32, 33, 13]. We cannot directly apply these techniques to our problem since we do not have precise correctness constraints. Some recent work uses constraint solving to synthesize programs that optimize a cost function, with no precise correctness constraint [7]; unfortunately, this approach is only applicable to cost functions without floating-point computations, which makes it incompatible with likelihood estimation.

On the other hand, stochastic synthesis is a good fit for our problem. Our synthesizer is among the many that apply simulated annealing. Other stochastic synthesizers perform MCMC sampling [29, 3]. Some use symbolic regression or other forms of genetic programming [26, 30, 36, 37]. In the future, we may investigate how varying the search technique affects DaPPer's performance.

Some tools use enumerative search. Previous work has shown that enumerative synthesizers outperform other synthesizers for some problems [3, 2, 34, 5, 25]. With custom pruning strategies, this may be another path to faster synthesis.

10.3 Unreadable and Probabilistic

The machine learning literature includes a rich body of work on learning Bayesian networks. Mainstream techniques fall into two categories: *constraint-based* and *search-and-score*. Constraint-based techniques focus on generating only the program structure. Search-and-score treats the problems of learning program structure and learning program parameters together.

Constraint-based techniques use statistical methods (e.g., chi-squared and mutual information) to identify relationships between variables in order to produce a network structure. In short, these techniques perform the same task as the first stage of our synthesizer. We have intentionally factored out the generation of the dependency graph from the rest of the synthesis process, which makes it easy to customize DaPPer with new structure learning approaches, including

constraint-based Bayesian learning approaches. This is a direction we hope to explore in the future.

Search-and-score techniques produce not just the network structure but a complete Bayesian network. Often these techniques produce outputs that can be translated directly to PPL programs. This makes them seem like a natural fit with our goals. Unfortunately, existing Bayesian learning techniques cannot produce readable models in the presence of continuous variables.

There are many search-and-score techniques for learning Bayesian networks that can be applied to discrete variables [14, 19]. To extend them to continuous variables, one approach is standard discretization [17, 38, 31]. Where our approach first synthesizes a program structure, then searches over the space of conditionals, search-and-score first fixes the set of conditionals (via discretization), then searches over the space of program structures. This approach leads discretization-based tools to produce dense ASTs with high branching factors. An alternative technique uses mixtures of truncated exponentials (MTEs) to do discretization more flexibly [21, 28]. Despite the attempt to reduce discretization, this approach still produces models that use large sets of massive switch statements with a different exponential distribution at each of hundreds of leaf nodes. In short, discretized search-and-score methods produce models that are difficult to read, understand, and adapt. The output models are accurate, but they do not succinctly express high-level insights into data.

Aside from Bayesian networks, a new class of models called sum-product networks (SPNs) [27, 9] is both probabilistic and learnable. SPNs are not suitable for our use case because they are much less readable and editable even than machine-written Bayesian networks.

Although modeling multiple interacting variables is a more common goal, some work learns probabilistic models of individual distributions. We know of one tool designed to generate a fast sampler with outputs that mimic the distribution of a set of input numbers [24]. If it is fed samples from a Gaussian distribution, rather than learning that the input can be modeled by a Gaussian, it learns a fast sampling procedure that produces Gaussian-like data. This tool does not meet our needs because it can only model a single random variable, not interacting variables, but also because its outputs are difficult to read and interpret.

11 Conclusion

This paper offers an alternative way for users without statistics expertise to create probabilistic models of their data. DaPPer synthesizes models quickly and produces human-readable PPL programs that users can explore, expand, and adapt. We introduce data-guided program mutation, which allows PPL synthesis to scale to generating full programs. We hope this extends the class of users willing to venture into using probabilistic models. We believe offering users full PPL programs without asking them to write even a single line of code is an important step towards making PPLs more accessible.

12 Acknowledgements

We thank Dawn Song and Rastislav Bodik for their thoughtful feedback. This work is supported in part by NSF Grants CCF–1139138, CCF–1337415, NSF ACI–1535191, and Graduate Research Fellowship DGE–1106400, a Microsoft Research PhD Fellowship, a grant from the U.S. Department of Energy, Office of Science, Office of Basic Energy Sciences Energy Frontier Research Centers program under Award Number FOA–0000619, and grants from DARPA FA8750–14–C–0011 and DARPA FA8750–16–2–0032, as well as gifts from Google, Intel, Mozilla, Nokia, and Qualcomm.

References

1. RITA | BTS | Transtats. http://www.transtats.bts.gov/DL_SelectFields.asp?Table_ID=236&DB_Short_Name=On-Time, accessed: 2016-02-05
2. Akiba, T., Imajo, K., Iwami, H., Iwata, Y., Kataoka, T., Takahashi, N., Moskal, M., Swamy, N.: Calibrating research in program synthesis using 72,000 hours of programmer time. Tech. rep., MSR (2013)
3. Alur, R., Bodik, R., Dallal, E., Fisman, D., Garg, P., Juniwal, G., Kress-Gazit, H., Madhusudan, P., Martin, M.M.K., Raghothaman, M., Saha, S., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. In: SyGus Competition (2014)
4. Arora, N.S., Russell, S.J., Sudderth, E.B.: Automatic inference in BLOG. In: Statistical Relational Artificial Intelligence. AAAI Workshops, vol. WS-10-06. AAAI (2010)
5. Barthe, G., Crespo, J.M., Gulwani, S., Kunz, C., Marron, M.: From relational verification to simd loop synthesis. In: PPoPP (2013)
6. Bhat, S., Borgström, J., Gordon, A.D., Russo, C.: Deriving probability density functions from probabilistic functional programs. In: TACAS (2013)
7. Bornholt, J., Torlak, E., Grossman, D., Ceze, L.: Optimizing synthesis with metasketches. In: POPL (2016)
8. Feizi, S., Marbach, D., Médard, M., Kellis, M.: Network deconvolution as a general method to distinguish direct dependencies in networks. Nature biotechnology 31(8), 726–733 (Aug 2013)
9. Gens, R., Domingos, P.M.: Learning the structure of sum-product networks. In: ICML (2013)
10. Gilks, W.R., Thomas, A., Spiegelhalter, D.J.: A language and program for complex bayesian modelling. Journal of the Royal Statistical Society. Series D (The Statistician) 43(1), 169–177 (1994)
11. Goodman, N.D., Mansinghka, V.K., Roy, D.M., Bonawitz, K., Tenenbaum, J.B.: Church: A language for generative models. In: In UAI. pp. 220–229 (2008)
12. Gordon, A.D., Henzinger, T.A., Nori, A.V., Rajamani, S.K.: Probabilistic programming. In: FOSE 2014 (2014)
13. Gulwani, S., Jha, S., Tiwari, A., Venkatesan, R.: Synthesis of loop-free programs. In: PLDI (2011)
14. Heckerman, D.: Learning in graphical models. chap. A Tutorial on Learning with Bayesian Networks, pp. 301–354. MIT Press, Cambridge, MA, USA (1999)
15. Hofmann, H., Cook, D., Kielion, C., Schloerke, B., Hobbs, J., Loy, A., Mosley, L., Rockoff, D., Huang, Y., Wrolstad, D., Yin, T.: Delayed, canceled, on time, boarding... flying in the USA. Journal of Computational and Graphical Statistics 20(2), 287–290 (2011)

16. Koller, D., McAllester, D., Pfeffer, A.: Effective bayesian inference for stochastic programs. In: AAAI/IAAI (1997)
17. Kozlov, A.V., Koller, D.: Nonuniform dynamic discretization in hybrid networks. In: UAI (1997)
18. Li, L., Wu, Y., Russell, S.J.: Swift: Compiled inference for probabilistic programs. Tech. Rep. UCB/EECS-2015-12, EECS Department, University of California, Berkeley (Mar 2015), http://www.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-12.html
19. Lowd, D., Domingos, P.M.: Learning arithmetic circuits. In: UAI (2008)
20. Milch, B., Marthi, B., Russell, S., Sontag, D., Ong, D.L., Kolobov, A.: Blog: Probabilistic models with unknown objects. In: In IJCAI. pp. 1352–1359 (2005)
21. Moral, S., Rumi, R., Salmerón, A.: Symbolic and Quantitative Approaches to Reasoning with Uncertainty: 6th European Conference, chap. Mixtures of Truncated Exponentials in Hybrid Bayesian Networks, pp. 156–167. Springer Berlin Heidelberg, Berlin, Heidelberg (2001)
22. Nori, A.V., Hur, C.K., Rajamani, S.K., Samuel, S.: R2: An efficient mcmc sampler for probabilistic programs. In: AAAI (July 2014)
23. Nori, A.V., Ozair, S., Rajamani, S.K., Vijaykeerthy, D.: Efficient synthesis of probabilistic programs. In: PLDI (2015)
24. Perov, Y.N., Wood, F.D.: Learning probabilistic programs. CoRR abs/1407.2646 (2014), http://arxiv.org/abs/1407.2646
25. Phothilimthana, P.M., Thakur, A., Bodik, R., Dhurjati, D.: Scaling up superoptimization. In: ASPLOS (2016)
26. Poli, R., Graff, M., McPhee, N.F.: Free lunches for function and program induction. In: FOGA (2009)
27. Poon, H., Domingos, P.: Sum-product networks: A new deep architecture. In: ICCV Workshops (2011)
28. Romero, V., Rumí, R., Salmerón, A.: Learning hybrid bayesian networks using mixtures of truncated exponentials. International Journal of Approximate Reasoning 42(1–2), 54 – 68 (2006)
29. Schkufza, E., Sharma, R., Aiken, A.: Stochastic superoptimization. In: ASPLOS (2013)
30. Schmidt, M., Lipson, H.: Distilling free-form natural laws from experimental data. Science 324 (2009)
31. Shah, A., Woolf, P.J.: Python environment for bayesian learning: Inferring the structure of bayesian networks from knowledge and data. Journal of Machine Learning Research 10, 159–162 (2009)
32. Solar-Lezama, A., Tancau, L., Bodik, R., Seshia, S., Saraswat, V.: Combinatorial sketching for finite programs. In: ASPLOS (2006)
33. Torlak, E., Bodik, R.: A lightweight symbolic virtual machine for solver-aided host languages. In: PLDI (2014)
34. Udupa, A., Raghavan, A., Deshmukh, J.V., Mador-Haim, S., Martin, M.M., Alur, R.: Transit: Specifying protocols with concolic snippets. In: PLDI (2013)
35. Wicklin, R.: An analysis of airline delays with SAS/IMLr Studio (2009)
36. Wong, M.L., Leung, K.S.: Evolutionary program induction directed by logic grammars. Evol. Comput. 5(2) (Jun 1997)
37. Woodward, J.R., Bai, R.: Why evolution is not a good paradigm for program induction: A critique of genetic programming. In: ACM/SIGEVO GEC. GEC '09 (2009)
38. Yoo, C., Thorsson, V., Cooper, G.F.: Discovery of causal relationships in a gene-regulation pathway from a mixture of experimental and observational dna microarray data. In: Proceedings of PSB. pp. 498–509 (2002)

13 Appendix: Correlation of Incompatible Types

For both the simple correlation and network deconvolution approaches to dependency graph generation, we need to be able to measure the correlation between every pair of columns in a dataset. Between columns with the same type, this is relatively straightforward, but since we must produce correlation measures for every pair, we must take a somewhat unusual approach.

As our correlation measure for the correlation technique and our similarity score for deconvolution, we use Spearman correlation, a measure of rank correlation. This measure is suitable for quantitative, ordinal, and dichotomous nominal data. Thus it can be applied to all numeric distributions we use, as well as to boolean distributions, which are nominal and dichotomous. In contrast, categorical random variables are nominal but may take on more than two values. Many categorical random variables are used to represent ordinal data (data with an implied ranking, such as a variable with values 'low', 'medium', and 'high'). On the other hands, they may also be used for nominal data (data with no implied ranking, such as a variable with values 'linen', 'silk', 'cotton'). Also, even when they are used for ordinal data, the mapping from values to ranks is not provided. In other circumstances, for comparing nominal datasets, one might use nominal-specific measures of association, such as Cramer's V. However, for our purposes (and especially for the network deconvolution technique), it is important that all measures in the similarity matrix are comparable. Since a single dataset may include both quantitative and nominal data, using different metrics for different variable types would be unacceptable.

We address this problem with the observation that any categorical variable can be replaced with a set of boolean variables (one for each value of the categorical variable) to produce an equivalent model. Then, any variable that has a direct dependence on any of the boolean variables in the altered model would have a direct dependence on the original categorical variable. Thus, for any categorical variable with m available values, we produce columns of data to represent m boolean variables, one for each value v, such that the boolean variable takes value true if and only if the categorical variable produces value v. To compare the categorical variable with another variable A, we then calculate Spearman's rank correlation for each boolean variable with A. Our tool takes the conservative approach of using the highest correlation produced by any of the boolean variables as the final correlation value.

14 Appendix: Reducing the Incidence of Redundant Branches in Synthesized Programs

In this appendix, we describe our approach to reducing redundancy in program structures.

14.1 Design

For a concrete illustration of redundancy reduction, consider our running example. The output program from the synthesis step, shown in Figure 13(a),

```
random Boolean tired ~
  BooleanDistrib(.500);
random Real skillLevel ~
  if tired
    then Gaussian(9.987, 7.000)
    else Gaussian(10.000, 7.003);
random Real testPerformance ~
  if (skillLevel < 16)
    then Gaussian(46.227, 7.663)
  else if (skillLevel > 18.2)
    then Gaussian(46.358, 7.549)
  else
    UniformReal(25.300, 125.113);
```

(a) Before reduction

```
random Boolean tired ~
  BooleanDistrib(.500);
random Real skillLevel ~
  if tired
    then Gaussian(9.987, 7.000)
    else Gaussian(10.000, 7.003);
random Real testPerformance ~
  if (skillLevel < 16 | skillLevel > 18.2)
    then Gaussian(46.296, 7.599)
  else
    UniformReal(25.300, 125.113);

.
```

(b) After reduction

Fig. 13: When two different branches have very similar bodies, our redundancy reduction algorithm can merge them to make the output program smaller and more readable. **Pink** highlights the reduction.

contains two similar branches. Our redundancy reduction approach aligns the distributions of the two branches (based on path conditions) and compares their parameters. Since both parameters are similar, our approach chooses to collapse the branches, producing the output program in Figure 13(b).

As detailed in Algorithm 1, for each pair of distribution parameters, our algorithm makes a decision on whether they match based on how close their parameters are, but also on how much data was used to tune the parameters. If one distribution is `Boolean(.987)` and the other is `Boolean(.986)`, we are probably willing to collapse them. In contrast, if one distribution is `Boolean(.987)` and the other is `Boolean(.345)`, and both were tuned with many rows, we probably should not collapse them. However, if one of the distributions was tuned with only two rows of data, we may believe the discrepancy comes only from random chance and be willing to collapse them despite the large difference in the parameters. In this regard, we believe our algorithm follows much the same approach as a human programmer attempting to simplify such a program, comparing parameters, considering how much a parameter is likely to have been affected by chance. When redundancy reduction collapses two branches, it next tunes the distribution parameters for the descendant distributions.

Our algorithm uses two threshold parameters, α and β. Low α values lead the algorithm to do little reduction, while high α values produce small, highly reduced programs. The β parameter gives users direct control over the difference between parameter values that should always result in a reduction. For instance, if users anticipate that they will not benefit from seeing separate branches for `Boolean(0.94)` and `Boolean(0.96)`, they should set β to 0.02 to indicate that parameters with differences no more than 0.02 should always be collapsed, regardless of the amount of data used to estimate them. Although α is the primary determinant of how aggressively the algorithm collapses branches, users may find manipulation of β convenient if they know some magnitude of difference is unimportant for their use cases.

```
for each pair (b1, b2) of branches do
    if not structureMatch(b1, b2) then
        continue
    end
    match = True
    for i in 0 to b1.distribs().length do
        distrib1 = b1.distribs()[i] ;          // the ith distribution in branch 1
        distrib2 = b2.distribs()[i]
        /* each distribution is associated with a rows value, the number of
           dataset rows used to tune its parameters                          */
        minNumRows = min(distrib1.rows, distrib2.rows)
        /* if one of these distributions has params based on very little data,
           expect it may reflect randomness rather than ground truth         */
        threshold = α/(minNumRows^0.7) - β
        for j in 0 to distrib1.length do
            param1 = distrib1[j] ;             // the jth param of distribution 1
            param2 = distrib2[j]
            match = match ∧ (| param1 - param2 | < threshold)
        end
    end
    if match then
        collapse(b1, b2)
    end
end
```

Algorithm 1: A redundancy reduction algorithm for making synthesized programs more human-readable. The α and β parameters can be adjusted by the user to control the size and readability of the output program.

Applying this algorithm after the synthesis process offers both advantages and disadvantages. The primary and obvious disadvantage is that by reducing redundancy after SA, we give up the opportunity to reduce the SA time by running on a smaller program structure. However, we believe the advantages may make up for the reduction in synthesis time. With this approach, we allow SA to use all the distinctions it can use to obtain high accuracy, and only eliminate distinctions from the learned program after the fact, when it is clear they have not offered significant advantages. At this point in the process, redundancy reduction has access to all the information that has been learned during the earlier synthesis stages, and can make very informed decisions about which conditions to combine. It receives more information than the dependency graph generation stage receives. Also, as discussed in Section 7, because this modification is applied as a post-processing step, the user can quickly and easily explore different readability levels, tuning the amount of redundancy reduction to his or her needs.

We also feel this approach may be a more natural way to reduce program sizes, compared to aggressive dependency graph approaches like Network Deconvolution. This is because the individual branch collapse actions are intuitive to human users and could even be presented to the programmer for approval.

14.2 Future Work for Redundancy Reduction

Although we are satisfied with the outputs of the current redundancy reduction technique, we are also interested in pursuing a more principled approach. The current algorithm is excellent for allowing users to explore quickly, since it is fast and offers simple tuning parameters. It is also a clean way to handle many different distribution types with a unified algorithm. However, we see redundancy reduction as a natural place to apply methods for identifying whether a difference is statistically significant. Rather than use our magnitude of difference vs. magnitude of data heuristics, why not use real statistical hypothesis testing?

We see one potential drawback to this approach, which is that our redundancy reduction approach is intended to increase readability rather than reduce overfitting. We want users to be able to remove detail even when it is *not* the result of random chance or overfitting. In short, users should be able to eliminate a distinction even if it is statistically significant. We intentionally placed the redundancy reduction stage at the end so that users can quickly explore various levels of program size and readability, tuning programs to their individual needs. If we transition to a more principled approach, we would want to find a way to maintain this flexibility and the current level of user control. In future, we expect to explore this direction.

15 Appendix: Examples of Synthesized Programs

To give a sense of how readable DaPPer's output programs are, we include programs DaPPer produces for the two running examples we use throughout the paper, 'burglary' (Figure 14) and 'students' (Figure 15).

```
random Boolean Burglary ~
  BooleanDistrib(0.001);

random Boolean Earthquake ~
  BooleanDistrib(0.002);

random Boolean Alarm ~
  if Burglary
  then
    if Earthquake
    then BooleanDistrib(0.95)
    else BooleanDistrib(0.94)
  else
    if Earthquake
    then BooleanDistrib(0.29)
    else BooleanDistrib(0.001);

random Boolean JohnCalls ~
  if Alarm
  then BooleanDistrib(0.9)
  else BooleanDistrib(0.05);

random Boolean MaryCalls ~
  if Alarm
  then BooleanDistrib(0.7)
  else BooleanDistrib(0.01);
```

(a) The ground truth program for our 'burglary' benchmark.

```
random Boolean Burglary ~
  BooleanDistrib(0.0008);

random Boolean Earthquake ~
  BooleanDistrib(0.0018);

random Boolean Alarm ~
  if Burglary
  then

    BooleanDistrib(1.0)

  else
    if Earthquake
    then BooleanDistrib(0.33333333)
    else BooleanDistrib(0.00110286);

random Boolean JohnCalls ~
    if Alarm
    then BooleanDistrib(0.96)
    else BooleanDistrib(0.049824561);

random Boolean MaryCalls ~
    if Alarm
    then BooleanDistrib(0.76)
    else BooleanDistrib(0.008621553);
```

(b) The program DaPPer synthesizes for the 'burglary' benchmark.

Fig. 14: A side-by-side comparison of the ground truth 'burglary' program and the program DaPPer synthesizes for the 'burglary' dataset. This program was synthesized with the Complete dependency graph, then processed with redundancy reduction.

```
random Boolean tired ~
  BooleanDistrib(.5);

random Real skillLevel ~
  Gaussian(10, 7);

random Real testPerformance ~
  if skillLevel > 13.0
  then
    if tired
    then Gaussian(70, 15)
    else Gaussian(95, 5)
  else
    if tired
    then Gaussian(30, 15)
    else Gaussian(70, 5);
```

(a) The ground truth program for our 'students' benchmark.

```
random Boolean tired ~
  BooleanDistrib(0.5009);

random Real skillLevel ~
  Gaussian(9.947325,6.981384);

random Real testPerformance ~
  if tired
  then
    if (skillLevel > 13.0266062)
    then Gaussian(70.079381,13.760027)
    else Gaussian(30.156086,20.582181)
  else
    if (skillLevel < 12.5535461)
    then Gaussian(70.005204,5.063899)
    else UniformReal(64.2285,104.1120);
```

(b) The program DaPPer synthesizes for the 'students' benchmark.

Fig. 15: A side-by-side comparison of the ground truth 'students' program and a program DaPPer synthesizes for the 'students' dataset. We use colors to highlight the bodies of corresponding branches. This program was synthesized with the Correlation dependency graph and did not require redundancy reduction.