# Bayesian Synthesis of Probabilistic Programs for Automatic Data Modeling

FERAS A. SAAD, Massachusetts Institute of Technology, USA
MARCO F. CUSUMANO-TOWNER, Massachusetts Institute of Technology, USA
ULRICH SCHAECHTLE, Massachusetts Institute of Technology, USA
MARTIN C. RINARD, Massachusetts Institute of Technology, USA
VIKASH K. MANSINGHKA, Massachusetts Institute of Technology, USA

We present new techniques for automatically constructing probabilistic programs for data analysis, interpretation, and prediction. These techniques work with probabilistic domain-specific data modeling languages that capture key properties of a broad class of data generating processes, using Bayesian inference to synthesize probabilistic programs in these modeling languages given observed data. We provide a precise formulation of Bayesian synthesis for automatic data modeling that identifies sufficient conditions for the resulting synthesis procedure to be sound. We also derive a general class of synthesis algorithms for domain-specific languages specified by probabilistic context-free grammars and establish the soundness of our approach for these languages. We apply the techniques to automatically synthesize probabilistic programs for time series data and multivariate tabular data. We show how to analyze the structure of the synthesized programs to compute, for key qualitative properties of interest, the probability that the underlying data generating process exhibits each of these properties. Second, we translate probabilistic programs in the domain-specific language into probabilistic programs in Venture, a general-purpose probabilistic programming system. The translated Venture programs are then executed to obtain predictions of new time series data and new multivariate data records. Experimental results show that our techniques can accurately infer qualitative structure in multiple real-world data sets and outperform standard data analysis methods in forecasting and predicting new data.

CCS Concepts: • **Mathematics of computing** → *Probability and statistics*; *Bayesian computation*; *Markov-chain Monte Carlo methods*; *Time series analysis*; *Multivariate statistics*; • **Software and its engineering** → *Domain specific languages*; • **Theory of computation** → *Probabilistic computation*; *Bayesian analysis*; *Data modeling*; *Denotational semantics*; • **Computing methodologies** → *Machine learning*;

Additional Key Words and Phrases: probabilistic programming, Bayesian inference, synthesis, model discovery

**ACM Reference Format:**
Feras A. Saad, Marco F. Cusumano-Towner, Ulrich Schaechtle, Martin C. Rinard, and Vikash K. Mansinghka. 2019. Bayesian Synthesis of Probabilistic Programs for Automatic Data Modeling. *Proc. ACM Program. Lang.* 3, POPL, Article 37 (January 2019), 32 pages. https://doi.org/10.1145/3290350

## 1 INTRODUCTION

Data analysis is an important and longstanding activity in many areas of natural science, social science, and engineering [Tukey 1977; Gelman and Hill 2007]. Within this field, probabilistic approaches that enable users to more accurately analyze, interpret, and predict underlying phenomena behind their data are rising in importance and prominence [Murphy 2012].

A primary goal of modern data modeling techniques is to obtain an artifact that explains the data. With current practice, this artifact takes the form of either a set of parameters for a fixed model

2475-1421/2019/1-ART37
https://doi.org/10.1145/3290350

[Nie 1975; Spiegelhalter et al. 1996; Plummer 2003; Cody and Smith 2005; Hyndman and Khandakar 2008; Rasmussen and Nickisch 2010; Seabold and Perktold 2010; Pedregosa et al. 2011; James et al. 2013] or a set of parameters for a fixed probabilistic program structure [Pfeffer 2001; Milch et al. 2005; Goodman et al. 2008; McCallum et al. 2009; Wood et al. 2014; Goodman and Stuhlmüller 2014; Carpenter et al. 2017; Pfeffer 2016; Salvatier et al. 2016; Tran et al. 2017; Ge et al. 2018]. With this approach, users manually iterate over multiple increasingly refined models before obtaining satisfactory results. Drawbacks of this approach include the need for users to manually select the model or program structure, the need for significant modeling expertise, limited modeling capacity, and the potential for missing important aspects of the data if users do not explore a wide enough range of model or program structures.

In contrast to this current practice, we model the data with an ensemble of probabilistic programs sampled from a joint space of program structures and parameters. This approach eliminates the need for the user to select a specific model structure, extends the range of data that can be easily and productively analyzed, and (as our experimental results show) delivers artifacts that more accurately model the observed data. A challenge is that this approach requires substantially more sophisticated and effective probabilistic modeling and inference techniques.

We meet this challenge by combining techniques from machine learning, program synthesis, and programming language design. From machine learning we import the Bayesian modeling framework and the Bayesian inference algorithms required to express and sample from rich probabilistic structures. From program synthesis we import the concept of representing knowledge as programs and searching program spaces to derive solutions to technical problems. From programming language design we import the idea of using domain-specific languages to precisely capture families of computations. The entire framework rests on the foundation of a precise formal semantics which enables a clear formulation of the synthesis problem along with proofs that precisely characterize the soundness guarantees that our inference algorithms deliver.

### 1.1 Automated Data Modeling via Bayesian Synthesis of Probabilistic Programs in Domain-Specific Languages

We use Bayesian inference to synthesize ensembles of probabilistic programs sampled from domain-specific languages given observed data. Each language is designed to capture key properties of a broad class of data generating processes. Probabilistic programs in the domain-specific language (DSL) provide concise representations of probabilistic models that summarize the qualitative and quantitative structure in the underlying data generating process. The synthesized ensembles are then used for data analysis, interpretation, and prediction.

We precisely formalize the problem of Bayesian synthesis of probabilistic programs for automatic data modeling. To each expression in the domain-specific language we assign two denotational semantics: one that corresponds to the prior probability distribution over expressions in the DSL and another that corresponds to the probability distribution that each DSL expression assigns to new data. We provide sufficient conditions on these semantics needed for Bayesian synthesis to be well-defined. We outline a template for a broad class of synthesis algorithms and prove that algorithms that conform to this template and satisfy certain preconditions are sound, i.e., they converge asymptotically to the Bayesian posterior distribution on programs given the data.

Our approach provides automation that is lacking in both statistical programming environments and in probabilistic programming languages. Statistical programming environments such as SAS [Cody and Smith 2005], SPSS [Nie 1975] and BUGS [Spiegelhalter et al. 1996] require users to first choose a model family and then to write code that estimates model parameters. Probabilistic programming languages such as Stan [Carpenter et al. 2017], Figaro [Pfeffer 2016], and Edward [Tran et al. 2017] provide constructs that make it easier to estimate model parameters given data, but still require users to write probabilistic code that explicitly specifies the underlying model structure. Our approach, in contrast, automates model selection within the domain-specific languages. Instead of delivering a single model or probabilistic program, it delivers ensembles of synthesized probabilistic programs that, together, more accurately model the observed data.

## 1.2 Inferring Qualitative Structure by Processing Synthesized Programs

In this paper we work with probabilistic programs for model families in two domains: (i) analysis of univariate time series data via Gaussian processes [Rasmussen and Williams 2006]; and (ii) analysis of multivariate tabular data using nonparametric mixture models [Mansinghka et al. 2016]. The synthesized programs in the domain-specific language provide a compact model of the data that make qualitative properties apparent in the surface syntax of the program. We exploit this fact to develop simple program processing routines that automatically extract these properties and present them to users, along with a characterization of the uncertainty with which these properties are actually present in the observed data.

For our analysis of time series data, we focus on the presence or absence of basic temporal features such as linear trends, periodicity, and change points in the time series. For our analysis of multivariate tabular data, we focus on detecting the presence or absence of predictive relationships, which may be characterized by nonlinear or multi-modal patterns.

## 1.3 Predicting New Data by Translating Synthesized Programs to Venture

In addition to capturing qualitative properties, we obtain executable versions of the synthesized probabilistic programs specified in the DSL by translating them into probabilistic programs specified in Venture [Mansinghka et al. 2014]. This translation step produces executable Venture programs that define a probability distribution over new, hypothetical data from a generative process learned from the observed data and that can deliver predictions for new data. Using Venture as the underlying prediction platform allows us to leverage the full expressiveness of Venture's general-purpose probabilistic inference machinery to obtain accurate predictions, as opposed to writing custom prediction engines on a per-DSL basis.

For time series data analyzed with Gaussian processes, we use the Venture programs to forecast the time series into the future. For multivariate tabular data analyzed with mixture models, we use the programs to simulate new data with similar characteristics to the observed data.

## 1.4 Experimental Results

We deploy the method to synthesize probabilistic programs and perform data analysis tasks on several real-world datasets that contain rich probabilistic structure.

For time series applications, we process the synthesized Gaussian process programs to infer the probable existence or absence of temporal structures in multiple econometric time series with widely-varying patterns. The synthesized programs accurately report the existence of structures when they truly exist in the data and the absence of structures when they do not. We also demonstrate improved forecasting accuracy as compared to multiple widely-used statistical and baselines, indicating that they capture patterns in the underlying data generating processes.

For applications to multivariate tabular data, we show that synthesized programs report the existence of predictive relationships with nonlinear and multi-modal characteristics that are missed by standard pairwise correlation metrics. We also show that the predictive distribution over new data from the probabilistic programs faithfully represents the observed data as compared to simulations from generalized linear models and produce probability density estimates that are orders of magnitude more accurate than standard kernel density estimation techniques.

### 1.5 Contributions

This paper makes the following contributions:

- **Bayesian synthesis of probabilistic programs.** It introduces and precisely formalizes the problem of Bayesian synthesis of probabilistic programs for automatic data modeling in domain-specific data modeling languages. It also provides sufficient conditions for Bayesian synthesis to be well-defined. These conditions leverage the fact that expressions in the DSL correspond to data models and are given two denotational semantics: one that corresponds to the prior probability distribution over expressions in the DSL and another that corresponds to the probability distribution that each DSL expression assigns to new data. It also defines a template for a broad class of synthesis algorithms based on Markov chains and proves that algorithms which conform to this template and satisfy certain preconditions are sound, i.e., they converge to the Bayesian posterior distribution on programs given the data.
- **Languages defined by probabilistic context-free grammars.** It defines a class of domain-specific data modeling languages defined by probabilistic context-free grammars and identifies when these languages satisfy the sufficient conditions for sound Bayesian synthesis. We also provide an algorithm for Bayesian synthesis for this class of DSLs that conforms to the above template and prove that it satisfies the soundness preconditions.
- **Example domain-specific languages for modeling time series and multivariate data.** It introduces two domain-specific languages, each based on a state-of-the-art model discovery technique from the literature on Bayesian statistics and machine learning. These languages are suitable for modeling broad classes of real-world datasets.
- **Applications to inferring qualitative structure and making quantitative predictions.** It shows how to use the synthesized programs to (i) infer the probability that qualitative structure of interest is present in the data, by processing collections of programs produced by Bayesian synthesis, and (ii) make quantitative predictions for new data.
- **Empirical results in multiple real-world domains.** It presents empirical results for both inferring qualitative structure and predicting new data. The results show that the qualitative structures match the real-world data, for both domain specific languages, and can be more accurate than those identified by standard techniques from statistics. The results also show that the quantitative predictions are more accurate than multiple widely-used baselines.

## 2 EXAMPLE

We next present an example that illustrates our synthesis technique and applications to data interpretation and prediction. Figure 1a presents a time series that plots world-wide airline passenger volume from 1948 to 1960 [Box and Jenkins 1976]. The data contains the year and passenger volume data points. As is apparent from the time series plot, the data contain several temporal patterns. Passenger volume linearly increases over time, with periodic fluctuations correlated with the time of year — volume peaks in summer and falls in winter, with a small peak around winter vacation.

### 2.1 Gaussian Process Models

Gaussian processes (GPs) define a family of nonparametric regression models that are widely used to model a variety of data [Rasmussen and Williams 2006]. It is possible to use GPs to discover complex temporal patterns in univariate time series [Duvenaud et al. 2013]. We first briefly review the Gaussian process, which places a prior distribution over functions $f : \mathcal{X} \to \mathbb{R}$. In our airline passenger example, $x \in \mathcal{X}$ is a time point and $f(x) \in \mathbb{R}$ is passenger volume at time $x$. The GP prior can express both simple parametric forms, such as polynomial functions, as well as more complex relationships dictated by periodicity, smoothness, and time-varying functionals. Following

```
// ** PRIOR OVER DSL SOURCE CODE **
assume get_hyper ~ mem((node) ~> {
  -log_logistic(log_odds_uniform() #hypers:node)
});
assume choose_primitive =  mem((node) ~> {
  base_kernel = uniform_discrete(0, 5) #structure:node;
  cond(
    (base_kernel == 0)(["WN",  get_hyper(pair("WN", node))]),
    (base_kernel == 1)(["C",   get_hyper(pair("C", node))]),
    (base_kernel == 2)(["LIN", get_hyper(pair("LIN", node))]),
    (base_kernel == 3)(["SE",  get_hyper(pair("SE", node))]),
    (base_kernel == 4)(["PER", get_hyper(pair("PER_l", node)),
                               get_hyper(pair("PER_t", node))]))
});
assume choose_operator = mem((node) ~> {
  operator_symbol ~ categorical(
    simplex(0.45, 0.45, 0.1), ["+", "*", "CP"])
    #structure:pair("operator", node);
  if (operator_symbol == "CP") {
    [operator_symbol, hyperprior(pair("CP", node)), .1]
  } else { operator_symbol }
});
assume generate_random_dsl_code = mem((node) ~> {
  cov = if (flip(.3) #structure:pair("branch", node)) {
    operator ~ choose_operator(node);
    [operator,
      generate_random_dsl_code(2 * node),
      generate_random_dsl_code((2 * node + 1))]
  } else { choose_primitive(node) };
  ["+", cov, ["WN", 0.01]]
});
assume dsl_source ~ generate_random_dsl_code(node:1);
```

```
// ** TRANSLATING DSL CODE INTO VENTURE **
assume ppl_source ~ generate_venturescript_code(dsl_source);
assume gp_executable = venture_eval(ppl_source);
```

```
// ** DATA OBSERVATION PROGRAM **
define xs = get_data_xs("./data.csv");
define ys = get_data_ys("./data.csv");
observe gp_executable(${xs}) = ys;
```

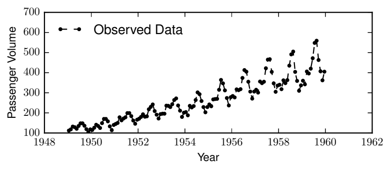
```
// ** BAYESIAN SYNTHESIS PROGRAM **
resample(60);
for_each(arange(T), (_) -> {
  resimulate([|structure|], one, steps:100);
  resimulate([|hypers|], one, steps:100)})
```

```
// ** PROCESSING SYNTHESIZED DSL CODE **
define count_kernels = (dsl, kernel) -> {
  if contains(["*", "+", "CP"], dsl[0]) {
    count_kernels(dsl[1], kernel) + count_kernels(dsl[2], kernel)
  } else { if (kernel == dsl[0]) {1} else {0} }
};
define count_operators = (dsl, operator) -> {
  if contains(["*", "+", "CP"], dsl[0]) {
    (if (operator == dsl[0]) {1} else {0})
      + count_operators(dsl[1], operator)
      + count_operators(dsl[2], operator)
  } else { 0 }
};
```

```
// ** SAMPLING FROM VENTURE EXECUTABLE FOR PREDICTION **
define xs_test = get_data_xs("./data.test.csv");
define ys_test = get_data_ys("./data.test.csv");
define ys_pred = sample_all(gp_executable(${xs_test}));
```

(a) Observed Dataset

```
>> plot(xs, ys, "Year", "Passenger Volume")
```



(b) Synthesized DSL and PPL Programs

```
>> sample_all(dsl_source)[0]
['+',
  ['*',
    ['+', ['WN',49.5], ['C',250.9]],
    ['+', ['PER', 13.2, 8.6],
      ['+',
        ['LIN', 1.2],
        ['LIN', 4.9]]]],
  ['WN', 0.1]]
>> sample_all(ppl_source)[0]
make_gp(gp_mean_const(0.),
  ((x1,x2) -> {((x1,x2) -> {((x1,x2) ->
  {((x1,x2) ->
  if (x1==x2) {49.5} else {0})(x1,x2)
    + ((x1,x2) -> {250.9})(x1,x2)})(x1,x2)
    * ((x1,x2) -> {((x1,x2) -> {
    -2/174.2400*sin(2*pi/8.6*
      abs(x1-x2))**2})(x1,x2)
    + ((x1,x2) -> {((x1,x2) ->
    {(x1-1.2)*(x2-1.2)})(x1,x2)
    + ((x1,x2) ->
      {(x1-4.9)*(x2-4.9)})(x1,x2)})
    (x1,x2)})(x1,x2)})(x1,x2)
    + ((x1,x2) -> if (x1==x2) {0.1}
    else {0})(x1,x2)}))
```

(c) Processing Synthesized DSL Program

```
// estimate probability that data has
// change point or periodic structure
>> mean(mapv((prog) -> {
    count_kernels(prog, "PER")
    + count_operators(prog, "CP") > 0
  }, sample_all(dsl_source)))
0.97
```

(d) Predictions from PPL Program

```
>> plot(xs, ys, "Year", "Passenger Volume",
    xs_test, ys_test, ys_pred)
```

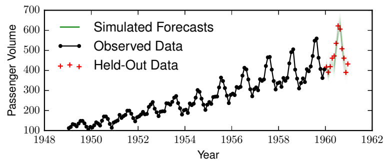


Fig. 1. Overview of Bayesian synthesis and execution of probabilistic programs in the Gaussian process DSL.

notation of Rasmussen and Williams [2006], we formalize a GP $f \sim \mathrm{GP}(m, k)$ with mean function $m : \mathcal{X} \to \mathcal{Y}$ and covariance function $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ as follows: $f$ is a collection of random variables $\{f(x) : x \in \mathcal{X}\}$. Give time points $\{x_1, \ldots, x_n\}$ the vector of random variables $[f(x_1), \ldots, f(x_n)]$ is jointly Gaussian with mean vector $[m(x_1), \ldots, m(x_n)]$ and covariance matrix $[k(x_i, x_j)]_{1 \le i,j \le n}$.

The prior mean is typically set to zero (as it can be absorbed by the covariance). The functional form of the covariance $k$ defines essential features of the unknown function $f$, allowing the GP to (i) fit structural patterns in observed data, and (ii) make time series forecasts. GP covariance functions (also called kernels) can be created by composing a set of simple base kernels through sum, product, and change point operators [Rasmussen and Williams 2006, Section 4.2]. We therefore define the following domain-specific language for expressing the covariance function of a specific GP:

$$\begin{aligned} K \in \mathsf{Kernel} ::= {} & (\mathtt{C}\ v) \mid (\mathtt{WN}\ v) \mid (\mathtt{SE}\ v) \mid (\mathtt{LIN}\ v) \mid (\mathtt{PER}\ v_1\ v_2) && \text{[BaseKernels]} \\ & \mid (+\ K_1\ K_2) \mid (\times\ K_1\ K_1) \mid (\mathtt{CP}\ v\ K_1\ K_2) && \text{[CompositeKernels]} \end{aligned}$$

The base kernels are constant (C), white noise (WN), squared exponential (SE), linear (LIN), and periodic (PER). Each base kernel has one or more numeric parameters $v$. For example, LIN has an x-intercept and PER has a length scale and period. The composition operators are sum (+), product (×), and change point (CP, which smoothly transitions between two kernels at some x-location). Given a covariance function specified in this language, the predictive distribution over airline passenger volumes $[f(x_1), \ldots, f(x_n)]$ at time points $(x_1, \ldots, x_n)$ is a standard multivariate normal.

## 2.2 Synthesized Gaussian Process Model Programs

The first code box in Figure 1 (lines 148-171) presents Venture code that defines a prior distribution over programs in our domain-specific Gaussian process language. This code implements a probabilistic context-free grammar that samples a specific program from the DSL. In the code box, we have two representations of the program: (i) a program in the domain-specific language (`dsl_source`, line 169); and (ii) the translation of this program into executable code in the Venture probabilistic programming language (`ppl_source`, line 171). When evaluated using `venture_eval`, the translated Venture code generates a stochastic procedure that implements the Gaussian process model (`gp_executable`, line 172). Given the observed time points (`xs`, line 173 of the second code box in Figure 1), the stochastic procedure `gp_executable` defines the probability of observing the corresponding passenger volume data (`ys`, line 174) according to the GP. This probability imposes a posterior distribution on the random choices made by `generate_random_dsl_code` (line 163) which sampled the `dsl_source`. The next step is to perform Bayesian inference over these random choices to obtain a collection of programs (approximately) sampled from this posterior distribution. Programs sampled from the posterior tend to deliver a good fit between the observed data and the Gaussian process. In our example we configure Venture to sample 60 programs from the (approximately inferred) posterior (line 176). The third code box in Figure 1 actually performs the inference, specifically by using a Venture custom inference strategy that alternates between performing a Metropolis-Hastings step over the program structure (base kernels and operators) and a Metropolis-Hastings step over the program parameters (lines 177-179).

## 2.3 Inferring Qualitative Structure by Processing Synthesized Programs

Figure 1b presents one of the programs sampled from the posterior. The syntactic structure of this synthesized program reflects temporal structure present in the airline passenger volume data. Specifically, this program models the data as the sum of linear and periodic components (base kernels) along with white noise. Of course, this is only one of many programs (in our example, 60 programs) sampled from the posterior. To analyze the structure that these programs expose in the data, we check for the existence of different base kernels and operators in each program (fourth

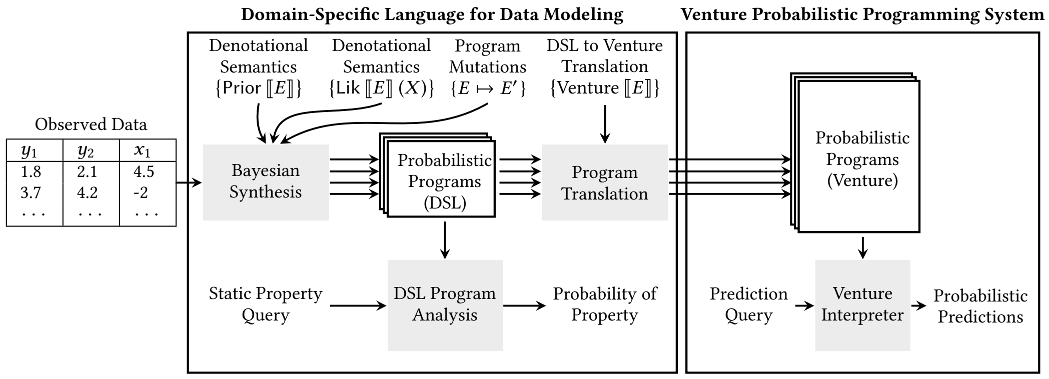

Domain-Specific Language for Data Modeling
Venture Probabilistic Programming System
Denotational Semantics {Prior ⟦E⟧}
Denotational Semantics {Lik ⟦E⟧ (X)}
Program Mutations {E ↦ E′}
DSL to Venture Translation {Venture ⟦E⟧}
Observed Data
y1 y2 x1
1.8 2.1 4.5
3.7 4.2 -2
Bayesian Synthesis
Probabilistic Programs (DSL)
Program Translation
Probabilistic Programs (Venture)
Static Property Query
DSL Program Analysis
Probability of Property
Prediction Query
Venture Interpreter
Probabilistic Predictions

Fig. 2. Components of Bayesian synthesis of probabilistic programs for automatic data modeling.

code box in Figure 1 and Figure 1c). By averging across the ensemble of synthesized program we estimate the probable presence of each structure in the data. In our example, all of the programs have white noise, 95% exhibit a linear trend, 95% exhibit periodicity, and only 22% contain a change point. These results indicate the likely presence of both linear and periodic structure.

### 2.4 Predicting New Data via Probabilistic Inference

In addition to supporting structure discovery by processing the synthesized domain-specific program (`dsl_source`), it is also possible to sample new data directly from the Venture executable (`gp_executable`) obtained from the translated program (`ppl_source`). In our example, this capability makes it possible to forecast future passenger volumes. The final code box in Figure 1 presents one such forecast, which we obtain by sampling data from the collection of 60 synthesized stochastic procedures (the green line in Figure 1d overlays all the sampled predictions). The forecasts closely match the actual held-out data, which indicates that the synthesized programs effectively capture important aspects of the underlying temporal structure in the actual data.

## 3 BAYESIAN SYNTHESIS IN DOMAIN-SPECIFIC DATA MODELING LANGUAGES

We next present a general framework for Bayesian synthesis in probabilistic domain-specific languages. We formalize the probabilistic DSL $\mathcal{L}$ as a countable set of strings, where an expression $E \in \mathcal{L}$ represents the structure and parameters of a family of statistical models specified by the DSL (such as the space of all Gaussian process models from Section 2). We associate $\mathcal{L}$ with a pair of denotational semantics Prior and Lik that describe the meaning of expressions $E$ in the language:

- The "prior" semantic function $\text{Prior} : \mathcal{L} \to (0, 1]$, where $\text{Prior}\,[\![E]\!]$ is a positive real number describing the probability that a given random process generated $E$.
- The "likelihood" semantic function $\text{Lik} : \mathcal{L} \to (\mathcal{X} \to \mathbb{R}_{\geq 0})$, where $\text{Lik}\,[\![E]\!]$ is a probability function over a data space $\mathcal{X}$, where each item $X \in \mathcal{X}$ has a relative probability $\text{Lik}\,[\![E]\!]\,(X)$.

These two semantic functions are required to satisfy three technical conditions.

*Condition 3.1 (Normalized Prior).* Prior defines a valid probability distribution over $\mathcal{L}$:

$$\sum_{E \in \mathcal{L}} \text{Prior}\,[\![E]\!] = 1. \tag{1}$$

If $\mathcal{L}$ is finite then this condition can be verified directly by summing the values. Otherwise if $\mathcal{L}$ is countably infinite then a more careful analysis is required (see Section 4.2).

*Condition 3.2 (Normalized Likelihood).* For each $E \in \mathcal{L}$, $\mathsf{Lik}\,[\![E]\!]$ is a probability distribution (for countable data) or density (for continuous data) over $\mathcal{X}$:

$$\forall E \in \mathcal{L}. \begin{cases} \sum_{X \in \mathcal{X}} \mathsf{Lik}\,[\![E]\!]\,(X) = 1 & \text{(if } \mathcal{X} \text{ is a countable space)} \\ \int_{X \in \mathcal{X}} \mathsf{Lik}\,[\![E]\!]\,(X)\mu(dX) = 1 & \text{(if } \mathcal{X} \text{ is a general space with base measure } \mu\text{).} \end{cases} \tag{2}$$

*Condition 3.3 (Bounded Likelihood).* Lik is bounded, and is non-zero for some expression $E$:

$$\forall X \in \mathcal{X}.\ 0 < c_X^{\max} ::= \sup\left\{\mathsf{Lik}\,[\![E]\!]\,(X) \mid E \in \mathcal{L}\right\} < \infty. \tag{3}$$

For each $X \in \mathcal{X}$, define $c_X ::= \sum_{E \in \mathcal{L}} \mathsf{Lik}\,[\![E]\!]\,(X) \cdot \mathsf{Prior}\,[\![E]\!]$ to be the marginal probability of $X$ (which is finite by Conditions 3.1 and 3.3). When the semantic functions satisfy the above conditions, they induce a new semantic function $\mathsf{Post} : \mathcal{L} \to \mathcal{X} \to \mathbb{R}_{\geq 0}$, called the posterior distribution:

$$\mathsf{Post}\,[\![E]\!]\,(X) ::= (\mathsf{Lik}\,[\![E]\!]\,(X)\ \mathsf{Prior}\,[\![E]\!])/c_X. \tag{4}$$

Lemma 3.4. *Let $\mathcal{L}$ be a language whose denotational semantics* Prior *and* Lik *satisfy Conditions 3.1, 3.2, and 3.3. For each $X \in \mathcal{X}$, the function $\lambda E.\mathsf{Post}\,[\![E]\!]\,(X)$ is a probability distribution over $\mathcal{L}$.*

Proof. Fix $X$. Then $\sum_{E \in \mathcal{L}} \mathsf{Post}\,[\![E]\!]\,(X) = \sum_{E \in \mathcal{L}}(\mathsf{Lik}\,[\![E]\!]\,(X)\ \mathsf{Prior}\,[\![E]\!])/c_X = c_X/c_X = 1$. □

The objective of Bayesian synthesis can now be stated.

*Objective 3.5 (Bayesian Synthesis).* Let $\mathcal{L}$ be a language whose denotational semantics Prior and Lik satisfy Conditions 3.1, 3.2, and 3.3. Given a dataset $X \in \mathcal{X}$, generate expressions $E$ with probability $\mathsf{Post}\,[\![E]\!]\,(X)$.

Figure 2 shows the main components of our approach. Assuming we can achieve the objective of Bayesian synthesis, we now outline how to use synthesized probabilistic programs in the DSL for (i) inferring qualitative statistical structure, by using simple program analyses on the program text; and (ii) predicting new data by translating them into executable Venture probabilistic programs.

## 3.1 Inferring Probabilities of Qualitative Properties in Synthesized DSL Programs

After synthesizing an ensemble of $n$ DSL expressions $\{E_1, \dots, E_n\}$ according to the posterior probability distribution given $X$, we can obtain insight into the learned models by processing the synthesized DSL expressions. In this paper, a typical query has the signature $\mathsf{HasProperty} : \mathcal{L} \to \{0, 1\}$ that checks whether a particular property holds in a given DSL program. In other words, $\mathsf{HasProperty}\,[\![E]\!]$ is 1 if the $E$ reflects the property, and 0 otherwise. We can use the synthesized programs to form an unbiased estimate of the posterior probability of a property:

$$\Pr\left\{\mathsf{HasProperty}\,[\![E]\!] \mid X\right\} \approx \frac{1}{n}\sum_{i=1}^{n} \mathsf{HasProperty}\,[\![E_i]\!]\,. \tag{5}$$

## 3.2 Forming Predictions by Translating DSL Programs into Venture Programs

In our formalism, each DSL expression $E \in \mathcal{L}$ is a probabilistic program that represents the structure and parameters of a family of statistical models specified by the domain-specific language. In this paper, we translate probabilistic programs $E$ in the DSL into new probabilistic programs $\mathsf{Venture}\,[\![E]\!]$ specified in Venture [Mansinghka et al. 2014]. A main advantage of translating DSL programs into Venture programs is that we can reuse general-purpose inference machinery in Venture [Mansinghka et al. 2018] to obtain accurate predictions, as opposed to writing custom interpreters for data prediction on a per-DSL basis. Sections 5 and 6 show two concrete examples of how DSL programs can be translated into Venture programs and how to use inference programming in Venture to obtain predictions on new data.

## 3.3 Markov Chain Monte Carlo Algorithms for Bayesian Synthesis

We next describe Markov chain Monte Carlo (MCMC) techniques to randomly sample expressions $E \in \mathcal{L}$ with probability that approximates $\mathsf{Post}\,[\![E]\!]\,(X)$ to achieve Objective 3.5. In particular, we employ a general class of MCMC sampling algorithms (Algorithm 1) that first obtains an expression $E_0 \in \mathcal{L}$ such that $\mathsf{Lik}\,[\![E]\!]\,(X) > 0$ and then iteratively generates a sequence of expressions $E_1, \dots, E_n \in \mathcal{L}$. The algorithm iteratively generates $E_i$ from $E_{i-1}$ using a DSL-specific *transition operator* $\mathcal{T}$, which takes as input expression $E \in \mathcal{L}$ and a data set $X$, and stochastically samples an expression $E'$ with probability denoted $\mathcal{T}(X, E \to E')$, where $\sum_{E' \in \mathcal{L}} \mathcal{T}(X, E \to E') = 1$ for all $E \in \mathcal{L}$. The algorithm returns the final expression $E_n$. Implementing Bayesian synthesis for a data-modeling DSL $\mathcal{L}$ requires implementing three procedures:

(1) GENERATE-EXPRESSION-FROM-PRIOR, which generates expression $E$ with probability $\mathsf{Prior}\,[\![E]\!]$.
(2) EVALUATE-LIKELIHOOD$(X, E)$, which evaluates $\mathsf{Lik}\,[\![E]\!]\,(X)$.
(3) GENERATE-NEW-EXPRESSION$(X, E)$, which generates expression $E'$ from expression $E$ with probability $\mathcal{T}(X, E \to E')$.

**Algorithm 1** Template of Markov chain Monte Carlo algorithm for Bayesian synthesis.

```
procedure BAYESIAN-SYNTHESIS($X, \mathcal{T}, n$)
    do
        $E_0 \sim$ GENERATE-EXPRESSION-FROM-PRIOR()
    while $\mathsf{Lik}\,[\![E_0]\!]\,(X) = 0$
    for $i = 1 \dots n$ do
        $E_i \sim$ GENERATE-NEW-EXPRESSION($X, E_{i-1}$))
    return $E_n$
```

Using MCMC for synthesis allows us to rigorously characterize soundness conditions. We now describe three conditions on the operator $\mathcal{T}$ that are sufficient to show that Algorithm 1 generates expressions from arbitrarily close approximation to the Bayesian posterior distribution on expressions $\mathsf{Post}\,[\![E]\!]\,(X)$. In Section 4.3 we show how to construct such an operator for a class of DSLs generated by probabilistic context-free grammars and prove that this operator satisfies the conditions.

*Condition 3.6 (Posterior invariance).* If an expression $E$ is sampled from the posterior distribution and a new expression $E' \in \mathcal{L}$ is sampled with probability $\mathcal{T}(X, E \to E')$, then $E'$ is also a sample from the posterior distribution:

$$\sum_{E \in \mathcal{L}} \mathsf{Post}\,[\![E]\!]\,(X) \cdot \mathcal{T}(X, E \to E') = \mathsf{Post}\,[\![E']\!]\,(X).$$

*Condition 3.7 (Posterior irreducibility).* Every expression $E'$ with non-zero likelihood is reachable from every expression $E \in \mathcal{L}$ in a finite number of steps. That is, for all pairs of expressions $E \in \mathcal{L}$ and $E' \in \{\mathcal{L} : \mathsf{Lik}\,[\![E']\!]\,(X) > 0\}$ there exists an integer $n \geq 1$ and a sequence of expressions $E_1, E_2, \dots, E_n$ where $E_1 = E$ and $E_n = E'$ such that $\mathcal{T}(X, E_{i-1} \to E_i) > 0$ for all $i \in \{2, \dots n\}$.

*Condition 3.8 (Aperiodicity).* There exists some expression $E \in \mathcal{L}$ such that the transition operator has a non-zero probability of returning to the same expression, i.e. $\mathcal{T}(X, E \to E) > 0$.

We now show that Algorithm 1 gives asymptotically correct results provided these three conditions hold. First, we show that it is possible to obtain an expression $E_0 \in \mathcal{L}$ where $\mathsf{Lik}\,[\![E_0]\!]\,(X) > 0$ using a finite number of invocations of GENERATE-EXPRESSION-FROM-PRIOR.

Lemma 3.9. *The do-while loop of Algorithm 1 will terminate with probability 1, and the expected number of iterations in the do-while loop is at most $c_X^{\max}/c_X$.*

Proof. The number of iterations of the loop is geometrically distributed with mean $p$, given by:

$$\begin{aligned} p &= \sum_{E \in \mathcal{L}} \mathsf{Prior}\, [\![E]\!] \cdot \mathbb{I}[\mathsf{Lik}\, [\![E]\!]\, (X) > 0] = \frac{1}{c_X^{\max}} \sum_{E \in \mathcal{L}} \mathsf{Prior}\, [\![E]\!] \cdot c_X^{\max} \cdot \mathbb{I}[\mathsf{Lik}\, [\![E]\!]\, (X) > 0] \\ &\geq \frac{1}{c_X^{\max}} \sum_{E \in \mathcal{L}} \mathsf{Prior}\, [\![E]\!] \cdot \mathsf{Lik}\, [\![E]\!]\, (X) = \frac{c_X}{c_X^{\max}}. \end{aligned}$$

Therefore, the expected number of iterations of the do-while loop is at most $1/p = c_X^{\max}/c_X < \infty$. □

We denote the probability that Algorithm 1 returns expression $E$, given that it started with expression $E_0$ by $\mathsf{ApproxPost}^1_{E_0}\, [\![E]\!]\, (X)$, and define the $n$-step probability inductively:

$$\begin{aligned} \mathsf{ApproxPost}^1_{E_0}\, [\![E]\!]\, (X) &::= \mathcal{T}(X, E_0 \to E) \\ \mathsf{ApproxPost}^n_{E_0}\, [\![E]\!]\, (X) &::= \sum_{E' \in \mathcal{L}} \mathcal{T}(X, E' \to E) \cdot \mathsf{ApproxPost}^{n-1}_{E_0}\, [\![E']\!]\, (X) \qquad (n > 1). \end{aligned}$$

We can now state a key convergence theorem, due to Tierney [1994].

Theorem 3.10 (Convergence of MCMC [Tierney 1994]). *If Condition 3.6 and Condition 3.7 and Condition 3.8 hold for some language $\mathcal{L}$, transition operator $\mathcal{T}$, and data $X \in \mathcal{X}$, then*

$$\forall E_0, E \in \mathcal{L}.\ \mathsf{Lik}\, [\![E_0]\!]\, (X) > 0 \implies \lim_{n \to \infty} \mathsf{ApproxPost}^n_{E_0}\, [\![E]\!]\, (X) \to \mathsf{Post}\, [\![E]\!]\, (X).$$

## 4 BAYESIAN SYNTHESIS FOR DOMAIN-SPECIFIC LANGUAGES DEFINED BY PROBABILISTIC CONTEXT-FREE GRAMMARS

Having described the general framework for Bayesian synthesis in general domain-specific languages, we now focus on the class of domain-specific languages which are generated by a probabilistic context-free grammar (PCFG) over symbolic expressions (s-expressions). We begin by describing the formal model of the grammar and the languages it can produce. We then outline a default denotational semantics Prior, derive a general MCMC algorithm for all languages in this class, and prove that the synthesis technique converges to the posterior distribution over expressions.

### 4.1 Tagged Probabilistic Context-Free Grammars with Random Symbols

Probabilistic context-free grammars are commonly used models for constructing languages [Jelinek et al. 1992]. We describe a special type of PCFG, called a tagged PCFG with random symbols, by extending the standard definition in two ways: (i) we require the grammar to produce s-expressions containing a unique phrase tag for each production rule, which guarantees that the grammar is unambiguous; and (ii) we allow each non-terminal to specify a probability distribution over symbols in the alphabet. The model is formally described below.

*Definition 4.1 (Tagged probabilistic context-free grammar with random symbols).* A tagged probabilistic context-free grammar with random symbols is a tuple $G = (\Sigma, N, R, T, P, Q, S)$ where

- $\Sigma$ is a finite set of terminal symbols.
- $N ::= \{N_1, \dots, N_m\}$ is a finite set of non-terminal symbols.
- $R ::= \{R_{ik} \mid i = 1, \dots, m; k = 1, \dots, r_i\}$ is a set of production rules, where $R_{ik}$ is the kth production rule of non-terminal $N_i$. Each production rule $R_{ik}$ is a tuple of the form

$$R_{ik} ::= (N_i, T_{ik}, \tilde{N}_1 \cdots \tilde{N}_{h_{ik}}), \tag{6}$$

where $h_{ik} \geq 0$ is the number of non-terminals on the right-hand side of $R_{ik}$ and each $\tilde{N}_j \in N$ ($j = 1, 2, \dots, h_{ik}$) is a non-terminal symbol. If $h_{ik} = 0$ then $\tilde{N}_1 \cdots \tilde{N}_{h_{ik}} = \epsilon$ is the empty string and we call $R_{ik}$ a *non-recursive* production rule. Otherwise it is a *recursive* rule.

- $T ::= \{T_{ik} \mid i = 1, \dots, m; k = 1, \dots, r_i\}$ is a set of phrase tag symbols, disjoint from $N$, where $T_{ik}$ is a unique symbol identifying the production rule $R_{ik}$.
- $P : T \to (0, 1]$ is a map from phrase tag symbols to their probabilities, where $P(T_{ik})$ is the probability that non-terminal $N_i$ selects its $k^{\text{th}}$ production rule $R_{ik}$. For each non-terminal $N_i$, the probabilities over its production rules sum to unity $\sum_{k=1}^{r_i} P(T_{ik}) = 1$.
- $Q : T \times \Sigma \to [0, 1]$ is a map from phrase tags and terminal symbols to probabilities, where $Q(T_{ik}, s)$ is the probability that production rule $R_{ik}$ of non-terminal $N_i$ draws the terminal symbol $s \in \Sigma$. For each tag $T_{ik}$, the probabilities over symbols sum to unity $\sum_{s \in \Sigma} Q(T_{ik}, s) = 1$.
- $S \in N$ is a designated start symbol.

We additionally assume that grammar $G$ is *proper*: production rules must be cycle-free, and there are no useless symbols in $\Sigma \cup N \cup T$ [Nijholt 1980].

The production rules from Eq (6) define the rewriting rules that describe how to syntactically replace a non-terminal $N_i$ with a tagged s-expression. We now describe how an evaluator uses these production rules to generate tagged s-expressions according to the probabilities $P$ and $Q$. The big-step sampling semantics for this evaluator are shown below, where the notation $N_i \Downarrow_G^p E$ means that starting from non-terminal $N_i$, the evaluator yielded expression $E$ with $p$ being the total probability of all the phrase tags and terminal symbols in the generated s-expression.

[Sampling: Non-Recursive Production Rule]

$$\frac{(N_i, T_{ik}, \epsilon) \in R,\ s \in \Sigma,\ Q(T_{ik}, s) > 0}{N_i \Downarrow_G^{P(T_{ik})Q(T_{ik},s)} (T_{ik}\ s)}$$

[Sampling: Recursive Production Rule]

$$\frac{(N_i, T_{ik}, \tilde{N}_1 \cdots \tilde{N}_{h_{ik}}) \in R,\ \tilde{N}_1 \Downarrow_G^{p_1} E_1,\ \dots,\ \tilde{N}_{h_{ik}} \Downarrow_G^{p_{h_{ik}}} E_j}{N_i \Downarrow_G^{P(T_{ik}) \prod_{z=1}^{h_{ik}} p_z} (T_{ik}\ E_1\ \dots\ E_{h_{ik}})}$$

In words, when the evaluator encounters a non-terminal symbol $N_i$, it chooses a production rule $R_{ik}$ with probability $P(T_{ik})$. If the selected production rule is non-recursive, the evaluator randomly samples a symbol $s \in \Sigma$ with probability $Q(T_{ik}, s)$, and returns an s-expression starting with $T_{ik}$ followed by the random symbol. Otherwise, if the selected production rule is recursive, the evaluator recursively evaluates all the constituent non-terminals and returns an s-expression starting with $T_{ik}$ followed by all the evaluated sub-expressions. Note that each evaluation step yields an s-expression where the first element is a phrase tag $T_{ik}$ that unambiguously identifies the production rule $R_{ik}$ that was selected to produce the expression. Hence, every expression maps uniquely to its corresponding parse tree simply by reading the phrase tags [Turbak et al. 2008]. As a result, the probability of any expression under its sampling semantics is unambiguous which is essential for the soundness properties established in Section 4.3.

Finally, we let $\mathcal{L}(G, N_i)$ denote the set of all strings that can be yielded starting from non-terminal $N_i$ ($i = 1, \dots, m$), according to the sampling semantics $\Downarrow_G$. The *language* $\mathcal{L}(G)$ generated by $G$ is the set of all strings derivable from the start symbol $S$, so that $\mathcal{L}(G) ::= \mathcal{L}(G, S)$. Conceptually, for a probabilistic domain-specific language $\mathcal{L}$ specified by tagged PCFGs, terminal symbols $s \in \Sigma$ are used to represent the program parameters and tag symbols $t \in T$ represent the program structure.

## 4.2 A Default Prior Semantics with the Normalization Property

Having described the big-step sampling semantics for expressions $E \in \mathcal{L}$ generated by a tagged probabilistic context-free grammar $G$, we next describe the "prior" denotational semantics of an expression, given by the semantic function Prior : $\mathcal{L}(G) \to (0, 1]$. To aid with the construction, we first introduce some additional notation. Define the semantic function Expand : $\mathcal{L}(G) \to N \to [0, 1]$

which takes an expression and a non-terminal symbol, and returns the probability that the non-terminal evaluates to the given expression:

$$\begin{aligned}\text{Expand}\,[\![(T_{ik}\ s)]\!]\,(N_i) &::= P(T_{ik}) \cdot Q(T_{ik}, s)\\ \text{Expand}\,[\![(T_{ik}\ E_1\ \cdots\ E_{h_{ik}})]\!]\,(N_i) &::= P(T_{ik}) \cdot \textstyle\prod_{z=1}^{h_{ik}} \text{Expand}\,[\![E_z]\!]\,(\tilde{N}_z)\\ &\qquad \text{where } R_{ik} = (T_{ik}, N_i, \tilde{N}_1, \ldots, \tilde{N}_{h_{ik}}),\end{aligned}$$

for $i = 1, \ldots, n$ and $k = 1, \ldots, r_i$.

Lemma 4.2. *For each non-terminal $N_i$ and for all expressions $E \in \mathcal{L}(G, N_i)$, we have*

$$\text{Expand}\,[\![E]\!]\,(N_i) = p \textit{ if and only if } N_i \Downarrow_G^p E.$$

Proof. By structural induction on the s-expression $E$. □

Recalling that $S$ is the start symbol of $G$, we define the prior semantics

$$\text{Prior}\,[\![E]\!] ::= \text{Expand}\,[\![E]\!]\,(S). \tag{7}$$

Having established the correspondence between the sampling semantics and denotational semantics, we next provide necessary and sufficient conditions on $G$ that are needed for Prior to be properly normalized as required by Condition 3.1. We begin with the following definition:

*Definition 4.3 (Consistency [Booth and Thompson 1973]).* A probabilistic context-free grammar $G$ is consistent if the probabilities assigned to all words derivable from $G$ sum to 1.

The following result of Booth and Thompson [1973] plays a key role in our construction.

Theorem 4.4 (Sufficient Condition for Consistency [Booth and Thompson 1973]). *A proper probabilistic context-free grammar $G$ is consistent if the largest eigenvalue (in modulus) of the expectation matrix of $G$ is less than 1.*

The expectation matrix of $G$ can be computed explicitly using the transition rule probabilities $P$ for non-terminal symbols [Gecse and Kovács 2010, Equation 2]. If $G$ is non-recursive then it is necessarily consistent since $\mathcal{L}(G)$ consists of a finite number of finite length words. Otherwise, if $G$ is recursive, consistency is equivalently described by having the expected number of steps in the stochastic rewriting-process $\Downarrow_G$ be finite Gecse and Kovács [2010]. To ensure that the Prior semantic function of $G$ is correctly normalized, we construct the expectation matrix and confirm that the modulus of the largest eigenvalue is less than 1. (Note that the probabilities $Q$ of random symbols in the tagged PCFG are immaterial in the analysis, since they appear only in non-recursive production rules and do not influence the production rule probabilities $P$.) We henceforth require every tagged probabilistic context-free grammar to satisfy the stated conditions for consistency.

## 4.3 Bayesian Synthesis Using Markov Chain Monte Carlo

We now derive a Bayesian synthesis algorithm specialized to domain-specific languages which are generated by a tagged context-free grammar, and prove that the algorithm satisfies the conditions for convergence in Theorem 3.10. Recall that implementing Algorithm 1 requires an implementation of three procedures:

(1) generate-expression-from-prior, which generates expression $E$ with probability Prior $[\![E]\!]$ (subject to Condition 3.1). Letting $E \sim \text{Expand}\,[\![\cdot]\!]\,(N_i)$ mean that $E$ is sampled randomly with probability Expand $[\![E]\!]\,(N_i)$ $(N_i \in N)$, we implement this procedure by sampling $E \sim \text{Expand}\,[\![\cdot]\!]\,(S)$ as described in Section 4.2.

(2) evaluate-likelihood$(X, E)$, which evaluates Lik $[\![E]\!]\,(X)$ (subject to Conditions 3.2 and 3.3). This procedure is DSL specific; two concrete examples are given in Sections 5 and 6.

(3) GENERATE-NEW-EXPRESSION$(X, E)$, which generates a new expression $E'$ from the starting expression $E$ with probability $\mathcal{T}(X, E \to E')$ (subject to Conditions 3.7, 3.6, and 3.8), which we describe below.

We construct a class of transition operators $\mathcal{T}(X, E \to E')$ that (i) stochastically replaces a random sub-expression in $E$; then (ii) stochastically accepts or rejects the mutation depending on how much it increases (or decreases) the value of $\text{Lik}\,[\![E]\!]\,(X)$. Our construction applies to any data-modeling DSL generated by a tagged PCFG and is shown in Algorithm 2. In this section we establish that the transition operator in Algorithm 2 satisfies Conditions 3.7, 3.6, and 3.8 for Bayesian synthesis with Markov chain Monte Carlo given in Section 3.3.

**Algorithm 2** Transition operator $\mathcal{T}$ for a context-free data-modeling language.

```
1: procedure GENERATE-NEW-EXPRESSION($E, X$)    ▷ Input expression $E$ and data set $X$
2:     $a \sim \text{SelectRandomElementUniformly}(A_E)$    ▷ Randomly select a node in parse tree
3:     $(N_i, E_{\text{hole}}) \leftarrow \text{Sever}_a\,[\![E]\!]$    ▷ Sever the parse tree and return the non-terminal symbol at the sever point
4:     $E_{\text{sub}} \sim \text{Expand}\,[\![\cdot]\!]\,(N_i)$    ▷ Generate random $E_{\text{sub}}$ with probability $\text{Expand}\,[\![E_{\text{sub}}]\!]\,(N_i)$
5:     $E' \leftarrow E_{\text{hole}}[E_{\text{sub}}]$    ▷ Fill hole in $E_{\text{hole}}$ with expression $E_{\text{sub}}$
6:     $L \leftarrow \text{Lik}\,[\![E]\!]\,(X)$    ▷ Evaluate likelihood for expression $E$ and data set $X$
7:     $L' \leftarrow \text{Lik}\,[\![E']\!]\,(X)$    ▷ Evaluate likelihood for expression $E'$ and data set $X$
8:     $p_{\text{accept}} \leftarrow \min\{1, (|A_E|/|A_{E'}|) \cdot (L'/L)\}$    ▷ Compute the probability of accepting the mutation
9:     $r \sim \text{UniformRandomNumber}([0,1])$    ▷ Draw a random number from the unit interval
10:    if $r < p_{\text{accept}}$ then    ▷ If-branch has probability $p_{\text{accept}}$
11:        return $E'$    ▷ Accept and return the mutated expression
12:    else    ▷ Else-branch has probability $1 - p_{\text{accept}}$
13:        return $E$    ▷ Reject the mutated expression, and return the input expression
```

Let $G$ be a grammar from Definition 4.1 with language $\mathcal{L} ::= \mathcal{L}(G)$. We first describe a scheme for uniquely identifying syntactic locations in the parse tree of each expression $E \in \mathcal{L}$. Define the set $A ::= \{(a_1, a_2, \dots, a_l) \mid a_i \in \{1, 2, \dots, h_{\max}\}, l \in \{0, 1, 2, \dots\}\}$ to be a countably infinite set that indexes nodes in the parse tree of $E$, where $h_{\max}$ denotes the maximum number of symbols that appear on the right of any given production rule of the form Eq (6) in the grammar. Each element $a \in A$ is a sequence of sub-expression positions on the path from the root node of a parse tree to another node. For example, if $E = (t_0\ E_1\ E_2)$ where $E_1 = (t_1\ E_3\ E_4)$ and $E_2 = (t_2\ E_5\ E_6)$, then the root node of the parse tree has index $a_{\text{root}} ::= ()$; the node corresponding to $E_1$ has index $(1)$, the node corresponding to $E_2$ has index $(2)$; the nodes corresponding to $E_3$ and $E_4$ have indices $(1,1)$ and $(1,2)$ respectively; and the nodes corresponding to $E_5$ and $E_6$ have indices $(2,1)$ and $(2,2)$. For an expression $E$, let $A_E \subset A$ denote the finite subset of nodes that exist in the parse tree of $E$.

Let $\square$ denote a hole in an expression. For notational convenience, we extend the definition of Expand to be a partial function with signature $\text{Expand} : \{\Sigma \cup T \cup \square\}^* \to N \to [0,1]$ and add the rule: $\text{Expand}\,[\![\square]\!]\,(N_i) ::= 1$ (for each $i = 1, \dots, m$). We define an operation Sever, parametrized by $a \in A$, that takes an expression $E$ and either returns a tuple $(N_i, E_{\text{hole}})$ where $E_{\text{hole}}$ is the expression with the sub-expression located at $a$ replaced with $\square$ and where $N_i$ is the non-terminal symbol from which the removed sub-expression is produced, or returns failure $(\varnothing)$ if the parse tree for $E$ has no node $a$:

$$\frac{a = (),\ \exists k.t \equiv T_{ik}}{(a, (t\ E_1 \cdots E_l)) \xrightarrow[\text{sever}]{} (N_i, \square)} \qquad \frac{((a_2, a_3, \dots), E_j) \xrightarrow[\text{sever}]{} (N_i, E_{\text{sev}}) \text{ and } a_1 = j}{(a, (t\ E_1\ \cdots E_l)) \xrightarrow[\text{sever}]{} (N_i, (t\ E_1\ \cdots\ E_{j-1}\ E_{\text{sev}}\ E_{j+1}\ \cdots\ E_l)}$$

$$\text{Sever}_a [\![(t\ E_1\ E_2\ \dots\ E_l)]\!] ::= \begin{cases} (N_i, E_{\text{hole}}) & \text{if } (a, (t\ E_1\ E_2\ \dots\ E_l)) \xrightarrow[\text{sever}]{} (N_i, E_{\text{hole}}) \\ \varnothing & \text{otherwise} \end{cases}$$

Note that for any expression $E \in \mathcal{L}$, setting $a = a_{\text{root}} \equiv ()$ gives $((), E) \xrightarrow[\text{sever}]{} (S, \square)$ where $S \in N$ is the start symbol. Also note that $E_{\text{hole}} \notin \mathcal{L}$ because $E_{\text{hole}}$ contains $\square$. For expression $E_{\text{hole}}$ that contains a single hole, where $(a, E) \xrightarrow[\text{sever}]{} (N_i, E_{\text{hole}})$ for some $a$ and $E$, let $E_{\text{hole}}[E_{\text{sub}}] \in \mathcal{L}$ denote the expression formed by replacing $\square$ with $E_{\text{sub}}$, where $E_{\text{sub}} \in \mathcal{L}(G, N_i)$. We further define an operation Subexpr, parametrized by $a$, that extracts the sub-expression corresponding to node $a$ in the parse tree:

$$\text{Subexpr}_a [\![(t\ E_1\ E_2\ \dots\ E_l)]\!] ::= \begin{cases} \varnothing & \text{if } a = () \text{ or } a_1 > k \\ E_j & \text{if } a = (j) \text{ for some } 1 \le j \le k \\ \text{Subexpr}_{(a_2, a_3, \dots)} [\![E_j]\!] & \text{if } i \ne (j) \text{ and } a_1 = j \text{ for some } 1 \le j \le k \end{cases}$$

Consider the probability that $\mathcal{T}$ takes an expression $E$ to another expression $E'$, which by total probability is an average over the uniformly chosen node index $a$:

$$\mathcal{T}(X, E \to E') = \frac{1}{|A_E|} \sum_{a \in A_E} \mathcal{T}(X, E \to E'; a) = \frac{1}{|A_E|} \sum_{a \in A_E \cap A_{E'}} \mathcal{T}(X, E \to E'; a),$$

where

$$\mathcal{T}(X, E \to E'; a) ::= \begin{cases} \text{Expand} [\![\text{Subexpr}_a [\![E']\!]]\!] (N_i) \cdot \alpha(E, E') + \mathbb{I}[E = E'](1 - \alpha(E, E')) \\ \qquad \text{if Sever}_a [\![E]\!] = \text{Sever}_a [\![E']\!] = (N_i, E_{\text{hole}}) \text{ for some } i \text{ and } E_{\text{hole}}, \\ 0 \qquad \text{otherwise,} \end{cases}$$

$$\alpha(E, E') ::= \min \left\{ 1, \frac{|A_E| \cdot \text{Lik} [\![E']\!] (X)}{|A_{E'}| \cdot \text{Lik} [\![E]\!] (X)} \right\}. \tag{8}$$

Note that we discard terms in the sum with $a \in A_E \setminus A_{E'}$ because for these terms $\text{Sever}_a [\![E']\!] = \varnothing$ and $\mathcal{T}(X, E \to E'; a) = 0$.

Lemma 4.5. *For $E \in \mathcal{L}$, if* $\text{Sever}_a [\![E]\!] = (N_i, E_{\text{hole}})$ *then* $\text{Expand} [\![\text{Subexpr}_a [\![E]\!]]\!] (N_i) > 0$.

Proof. If $\text{Sever}_a [\![E]\!] = (N_i, E_{\text{hole}})$ then $\text{Subexpr}_a [\![E]\!]$ is an expression with tag $t \in T_i$. Since the expression $E \in \mathcal{L}$, it follows that $\text{Expand} [\![\text{Subexpr}_a [\![E]\!]]\!] (N_i) > 0$ because otherwise $\text{Prior} [\![E]\!] = 0$ (a contradiction). □

Lemma 4.6. *For $E \in \mathcal{L}$, if* $\text{Sever}_a [\![E]\!] = (N_i, E_{\text{hole}})$ *then:*

$$\text{Expand} [\![E]\!] (S) = \text{Expand} [\![\text{Subexpr}_a [\![E]\!]]\!] (N_i) \cdot \text{Expand} [\![E_{\text{hole}}]\!] (S).$$

Proof. Each factor in $\text{Expand} [\![E]\!] (S)$ corresponds to a particular node $a'$ in the parse tree. Each factor corresponding to $a'$ appears in $\text{Expand} [\![\text{Subexpr}_a [\![E]\!]]\!] (N_i)$ if $a'$ is a descendant of $a$ in the parse tree, or in $\text{Expand} [\![E_{\text{hole}}]\!] (S)$ otherwise. □

Lemma 4.7. *For any $E, E' \in \mathcal{L}$ where* $\text{Sever}_a [\![E]\!] = \text{Sever}_a [\![E']\!] = (N_i, E_{\text{hole}})$*:*

$$\text{Prior} [\![E']\!] = \text{Prior} [\![E]\!] \cdot \frac{\text{Expand} [\![\text{Subexpr}_a [\![E']\!]]\!] (N_i)}{\text{Expand} [\![\text{Subexpr}_a [\![E]\!]]\!] (N_i)}.$$

Proof. Use $\text{Prior} [\![E]\!] = \text{Expand} [\![E]\!] (S)$ and $\text{Prior} [\![E']\!] = \text{Expand} [\![E']\!] (S)$, and expand using Lemma 4.6. □

Lemma 4.8. *The transition operator $\mathcal{T}$ of Algorithm 2 satisfies Condition 3.6 (posterior invariance).*

Proof. We show detailed balance for $\mathcal{T}$ with respect to the posterior:

$$\text{Post}\,[\![E]\!]\,(X) \cdot \mathcal{T}(X, E \to E') = \text{Post}\,[\![E']\!]\,(X) \cdot \mathcal{T}(X, E' \to E) \qquad (E, E' \in \mathcal{L}).$$

First, if $\text{Post}\,[\![E]\!]\,(X) \cdot \mathcal{T}(X, E \to E') = 0$ then $\text{Post}\,[\![E']\!]\,(X) \cdot \mathcal{T}(X, E' \to E) = 0$ as follows. Either $\text{Post}\,[\![E]\!]\,(X) = 0$ or $\mathcal{T}(X, E \to E') = 0$.

(1) If $\text{Post}\,[\![E]\!]\,(X) = 0$ we have $\text{Lik}\,[\![E]\!]\,(X) = 0$ which implies that $\alpha(E', E) = 0$ and therefore $\mathcal{T}(X, E' \to E) = 0$.
(2) If $\mathcal{T}(X, E \to E') = 0$ then $\mathcal{T}(X, E \to E'; a_{\text{root}}) = 0$ which implies either that the acceptance probability $\alpha(E, E') = 0$ or that $\text{Expand}\,[\![\text{Subexpr}_{a_{\text{root}}}\,[\![E']\!]]\!]\,(S) = 0$. If $\alpha(E, E') = 0$ then $\text{Lik}\,[\![E']\!]\,(X) = 0$ and $\text{Post}\,[\![E']\!]\,(X) = 0$. But $\text{Expand}\,[\![\text{Subexpr}_{E'}\,[\![a_{\text{root}}]\!]]\!]\,(S) = 0$ is a contradiction since $\text{Expand}\,[\![\text{Subexpr}_{E'}\,[\![a_{\text{root}}]\!]]\!]\,(S) = \text{Expand}\,[\![E']\!]\,(S) = \text{Prior}\,[\![E']\!] > 0$.

Next, consider $\text{Post}\,[\![E]\!]\,(X) \cdot \mathcal{T}(X, E \to E') > 0$ and $\text{Post}\,[\![E']\!]\,(X) \cdot \mathcal{T}(X, E' \to E) > 0$. It suffices to show that $\text{Post}\,[\![E]\!]\,(X) \cdot |A_{E'}| \cdot \mathcal{T}(X, E \to E'; a) = \text{Post}\,[\![E']\!]\,(X) \cdot |A_E| \cdot \mathcal{T}(X, E' \to E; a)$ for all $a \in A_E \cap A_{E'}$. If $E = E'$, then this is vacuously true. Otherwise $E \neq E'$, then consider two cases:

(1) If $\text{Sever}_a\,[\![E]\!] = \text{Sever}_a\,[\![E']\!] = (N_i, E_{\text{hole}})$ for some $i$ and $E_{\text{hole}}$, then it suffices to show that:

$$\begin{aligned}\text{Post}\,[\![E]\!]\,(X) \cdot |A_{E'}| \cdot \text{Expand}\,[\![\text{Subexpr}_a\,[\![E']\!]]\!]\,(N_i) \cdot \alpha(E, E') \\ = \text{Post}\,[\![E']\!]\,(X) \cdot |A_E| \cdot \text{Expand}\,[\![\text{Subexpr}_a\,[\![E]\!]]\!]\,(N_i) \cdot \alpha(E', E)\end{aligned}$$

Both sides are guaranteed to be non-zero because $\text{Post}\,[\![E]\!]\,(X) > 0$ implies $\text{Lik}\,[\![E]\!]\,(X) > 0$ implies $\alpha(E', E) > 0$, and by Lemma 4.5 (and similarly for $\text{Post}\,[\![E']\!]\,(X) > 0$). Therefore, it suffices to show that

$$\begin{aligned}\frac{\alpha(E, E')}{\alpha(E', E)} &= \frac{\text{Post}\,[\![E']\!]\,(X) \cdot |A_E| \cdot \text{Expand}\,[\![\text{Subexpr}_a\,[\![E]\!]]\!]\,(N_i)}{\text{Post}\,[\![E]\!]\,(X) \cdot |A'_E| \cdot \text{Expand}\,[\![\text{Subexpr}_a\,[\![E']\!]]\!]\,(N_i)} \\ &= \frac{\text{Prior}\,[\![E']\!] \cdot \text{Lik}\,[\![E']\!]\,(X) \cdot |A_E| \cdot \text{Expand}\,[\![\text{Subexpr}_a\,[\![E]\!]]\!]\,(N_i)}{\text{Prior}\,[\![E]\!] \cdot \text{Lik}\,[\![E]\!]\,(X) \cdot |A'_E| \cdot \text{Expand}\,[\![\text{Subexpr}_a\,[\![E']\!]]\!]\,(N_i)} \\ &= \frac{\text{Lik}\,[\![E']\!]\,(X) \cdot |A_E|}{\text{Lik}\,[\![E]\!]\,(X) \cdot |A'_E|},\end{aligned}$$

where for the last step we used Lemma 4.7 to expand $\text{Prior}\,[\![E']\!]$, followed by cancellation of factors. Note that if $\alpha(E, E') < 1$ then $\alpha(E', E) = 1$ and

$$\frac{\alpha(E, E')}{\alpha(E', E)} = \frac{\alpha(E, E')}{1} = \frac{\text{Lik}\,[\![E']\!]\,(X) \cdot |A_E|}{\text{Lik}\,[\![E]\!]\,(X) \cdot |A'_E|}.$$

If $\alpha(E', E) < 1$ then $\alpha(E, E') = 1$ and

$$\frac{\alpha(E, E')}{\alpha(E', E)} = \frac{1}{\alpha(E', E)} = \frac{\text{Lik}\,[\![E']\!]\,(X) \cdot |A_E|}{\text{Lik}\,[\![E]\!]\,(X) \cdot |A'_E|}.$$

If $\alpha(E, E') = 1 = \alpha(E', E)$ then $\alpha(E, E')/\alpha(E', E) = 1 = \alpha(E, E')$.

(2) If $\text{Sever}_a\,[\![E]\!] \neq \text{Sever}_a\,[\![E']\!]$, then $\mathcal{T}(X, E \to E'; a) = \mathcal{T}(X, E' \to E; a) = 0$.

If $E \neq E'$, then $\mathcal{T}(X, E \to E'; a) = \text{Prior}\,[\![\text{Subexpr}_a\,[\![E']\!]]\!] \cdot \alpha(E, E')$. Finally, posterior invariance follows from detailed balance:

$$\textstyle\sum_{E \in \mathcal{L}} \text{Post}\,[\![E]\!]\,(X) \mathcal{T}(X, E \to E') = \sum_{E \in \mathcal{L}} \text{Post}\,[\![E']\!]\,(X) \mathcal{T}(X, E' \to E) = \text{Post}\,[\![E']\!]\,(X).$$

□

Lemma 4.9. *The transition operator $\mathcal{T}$ of Algorithm 2 satisfies Condition 3.7 (posterior irreducibility).*

Proof. We show that for all expressions $E$ and $E' \in \{\mathcal{L} : \text{Post}\,[\![E']\!]\,(X) > 0\}$, $E'$ is reachable from $E$ in one step (that is, $\mathcal{T}(X, E \to E') > 0$). Note for all $E, E' \in \mathcal{L}$, $a_{\text{root}} \in A_E \cap A_{E'}$ and $\text{Sever}_{a_{\text{root}}}\,[\![E]\!] = \text{Sever}_{a_{\text{root}}}\,[\![E']\!] = (S, \Box)$. Since $\text{Post}\,[\![E']\!]\,(X) > 0$, we know that $\text{Prior}\,[\![E']\!] = \text{Expand}\,[\![E']\!]\,(S) > 0$, and $\text{Lik}\,[\![E']\!]\,(X) > 0$, which implies $\alpha(E, E') > 0$. Finally:

$$\mathcal{T}(X, E \to E') \geq \frac{1}{|A_E|} \cdot \mathcal{T}(X, E \to E'; a) \geq \text{Expand}\,[\![E']\!]\,(S) \cdot \alpha(E, E') > 0.$$

□

Lemma 4.10. *The transition operator $\mathcal{T}$ of Algorithm 2 satisfies Condition 3.8 (aperiodicity).*

Proof. For all $E \in \mathcal{L}$, we have $\mathcal{T}(X, E \to E) \geq (1/|A_E|) \cdot \text{Expand}\,[\![E]\!]\,(S) = (1/|A_E|) \cdot \text{Prior}\,[\![E]\!] > 0$. The inequality derives from choosing only $a = a_{\text{root}}$ in the sum over $a \in |A_E|$. □

We have thus established (by Lemmas 4.8, 4.9, and 4.10) that for a language $\mathcal{L}$ specified by a consistent probabilistic context-free grammar with well-defined Prior and Lik semantics (Conditions 3.1, 3.2, and 3.3) the Bayesian synthesis procedure in Algorithm 1 with the transition operator $\mathcal{T}$ from Algorithm 2 satisfies the preconditions for sound inference given in Theorem 3.10.

## 5 GAUSSIAN PROCESS DSL FOR UNIVARIATE TIME SERIES DATA

Section 2.1 introduced the mathematical formalism and domain-specific language for Gaussian process models. The DSL is generated by a tagged probabilistic context-free grammar as described in Definition 4.1, which allows us to reuse the general semantics and inference algorithms in Section 4. Figure 3 shows the core components of the domain-specific language: the syntax and production rules; the symbol and production rule probabilities; the likelihood semantics; and the translation procedure of DSL expressions into Venture. By using addition, multiplication, and change point operators to build composite Gaussian process kernels, it is possible to discover a wide range of time series structure from observed data [Lloyd et al. 2014]. The synthesis procedure uses Algorithm 1 along the generic transition operator for PCFGs in Algorithm 2, which is sound by Lemmas 4.8, 4.9, and 4.10. The supplementary material formally proves that the prior and likelihood semantics satisfy the preconditions for Bayesian synthesis.

This Gaussian process domain-specific language (whose implementation is shown in Figure 1) is suitable for both automatic model structure discovery of the covariance kernel as well as for estimating the parameters of each kernel. This contrasts with standard software packages for inference in GPs (such as python's scikit-learn [Pedregosa et al. 2011], or MATLAB's GPML [Rasmussen and Nickisch 2010]) which do not have the ability to synthesize different Gaussian process model structures and can only perform parameter estimation given a fixed, user-specified model structure which is based on either the user's domain-specific knowledge or (more often) an arbitrary choice.

### 5.1 Time Complexity of Bayesian Synthesis Algorithm

In Algorithm 2 used for synthesis, each transition step from an existing expression $K$ to a candidate new expression $K'$ requires the following key computations:

(1) Severing the parse tree at a randomly selected node using Sever (Algorithm 2, line 3). The cost is linear in the number $|K|$ of subexpressions in the overall s-expression $K$.
(2) Forward-sampling the probabilistic context-free grammar using Expand (Algorithm 2, line 4). The expected cost is linear in the average length of strings $l_G$ generated by the PCFG, which can be determined from the transition probabilities [Gecse and Kovács 2010, Eq. 3]).
(3) Assessing the likelihood under the existing and proposed expressions using $\text{Lik}\,[\![K]\!]\,((\mathbf{x}, \mathbf{y}))$ (Algorithm 2, line 6). For a Gaussian process with $n$ observed time points, the cost of building the covariance matrix $\text{Cov}\,[\![K]\!]$ is $O(|K|n^2)$ and the cost of obtaining its inverse is $O(n^3)$.

The overall time complexity of a transition from $K$ to $K'$ is thus $O(l_G + \max(|K|, |K'|)n^2 + n^3)$. This term is typically dominated by the $n^3$ term from assessing Lik. Several existing techniques use sparse approximations of Gaussian processes to reduce this complexity to $O(m^2 n)$ where $m \ll n$ is a parameter representing the size of a subsample of the full data to be used [Rasmussen and Williams 2006, Chapter 8]. These approximation techniques trade-off predictive accuracy with

**Syntax and Production Rules**

| | |
|---|---|
| $v \in$ Numeric | |
| $H \in$ Parameters ::= (gamma $v$) | [GammaParam] |
| $K \in$ Kernel ::= (const $H$) | [Constant] |
| \| (wn $H$) | [WhiteNoise] |
| \| (lin $H$) | [Linear] |
| \| (se $H$) | [SquaredExp] |
| \| (per $H_1$ $H_2$) | [Periodic] |
| \| (+ $K_1$ $K_2$) | [Sum] |
| \| (* $K_1$ $K_2$) | [Product] |
| \| (cp $H$ $K_1$ $K_2$) | [ChangePoint] |

**Production Rule Probabilities**

$$H : P(\text{gamma}) ::= 1.$$

$$K : P(\text{const}) = P(\text{wn}) = P(\text{lin}) = P(\text{se}) ::= 0.14,$$
$$P(+) = P(*) ::= 0.135,\ P(\text{cp}) ::= 0.03.$$

**Terminal Symbol Probabilities**

$$\text{gamma} : Q(\text{gamma}, v) ::= \frac{\beta^{\alpha} v^{\alpha-1} e^{-\beta v}}{\Gamma(\alpha)} \quad (\alpha = 1, \beta = 1).$$

**Prior Denotation**

Default prior for probabilistic context-free grammars (Section 4.2).

**Likelihood Denotation**

$$\begin{aligned}
\text{Cov}\,[\![(\text{const (gamma } v))]\!]\,(x)(x') &::= v \\
\text{Cov}\,[\![(\text{wn (gamma } v))]\!]\,(x)(x') &::= v \cdot \mathbb{I}[x = x'] \\
\text{Cov}\,[\![(\text{lin (gamma } v))]\!]\,(x)(x') &::= (x - v)(x' - v) \\
\text{Cov}\,[\![(\text{se (gamma } v))]\!]\,(x)(x') &::= \exp(-(x - x')^2/v) \\
\text{Cov}\,[\![(\text{per (gamma } v_1) \text{ (gamma } v_2))]\!]\,(x)(x') &::= \exp(-2/v_1 \sin((2\pi/v_2)|x - x'|)^2) \\
\text{Cov}\,[\![(+\ K_1\ K_2)]\!]\,(x)(x') &::= \text{Cov}\,[\![K_1]\!]\,(x)(x') + \text{Cov}\,[\![K_2]\!]\,(x)(x') \\
\text{Cov}\,[\![(*\ K_1\ K_2)]\!]\,(x)(x') &::= \text{Cov}\,[\![K_1]\!]\,(x)(x') \times \text{Cov}\,[\![K_2]\!]\,(x)(x') \\
\text{Cov}\,[\![(\text{cp (gamma } v)\ K_1\ K_2)]\!]\,(x)(x') &::= \Delta(x, v)\Delta(x', v)(\text{Cov}\,[\![K_1]\!]\,(x)(x')) + (1 - \Delta(x, v))(1 - \Delta(x', v))(\text{Cov}\,[\![K_1]\!]\,(x)(x')) \\
&\quad \text{where } \Delta(x, v) ::= 0.5 \times (1 + \tanh(10(x - v))) \\
\text{Lik}\,[\![K]\!]\,((\mathbf{x}, \mathbf{y})) &::= \exp\Big( -1/2 \textstyle\sum_{i=1}^{n} y_i \left( \sum_{j=1}^{n} (\{[\text{Cov}\,[\![K]\!]\,(x_i)(x_j) + 0.01\delta(x_i, x_j)]_{i,j=1}^{n}\}_{ij}^{-1}) y_j \right) \\
&\qquad - 1/2 \log \left| [\text{Cov}\,[\![K]\!]\,(x_i)(x_j) + 0.01\delta(x_i, x_j)]_{i,j=1}^{n} \right| - (n/2) \log 2\pi \Big)
\end{aligned}$$

**DSL to Venture Translation**

$$\begin{aligned}
\text{VentureCov}\,[\![(\text{const (gamma } v))]\!] &::= \texttt{((x1, x2) -> \{}v\texttt{\})} \\
\text{VentureCov}\,[\![(\text{wn (gamma } v))]\!] &::= \texttt{((x1, x2) -> \{if (x1==x2) \{}v\texttt{\} else \{0\}\})} \\
\text{VentureCov}\,[\![(\text{lin (gamma } v))]\!] &::= \texttt{((x1, x2) -> \{(x1-}v\texttt{) * (x2-}v\texttt{)\})} \\
\text{VentureCov}\,[\![(\text{se (gamma } v))]\!] &::= \texttt{((x1, x2) -> \{exp((x1-x2)**2/}v\texttt{)\})} \\
\text{VentureCov}\,[\![(\text{per (gamma } v_1) \text{ (gamma } v_2))]\!]\,(x)(x') &::= \texttt{((x1, x2) -> \{-2/}v_1\texttt{ * sin(2*pi/}v_2\texttt{ * abs(x1-x2))**2\})} \\
\text{VentureCov}\,[\![(+\ K_1\ K_2)]\!] &::= \texttt{((x1, x2) -> \{}\text{VentureCov}\,[\![K_1]\!]\texttt{(x1, x2) + }\text{VentureCov}\,[\![K_2]\!]\texttt{(x1, x2)\})} \\
\text{VentureCov}\,[\![(*\ K_1\ K_2)]\!] &::= \texttt{((x1, x2) -> \{}\text{VentureCov}\,[\![K_1]\!]\texttt{(x1, x2) * }\text{VentureCov}\,[\![K_2]\!]\texttt{(x1, x2)\})} \\
\text{VentureCov}\,[\![((\text{cp (gamma } v)\ K_1\ K_2)]\!] &::= \texttt{((x1, x2) -> \{} \\
&\qquad \texttt{sig1 = sigmoid(x1, }v\texttt{, .1) * sigmoid(x2, }v\texttt{, .1);} \\
&\qquad \texttt{sig2 = (1-sigmoid(x1, }v\texttt{, .1)) * (1-sigmoid(x2, }v\texttt{, .1));} \\
&\qquad \texttt{sig1 * }\text{VentureCov}\,[\![K_1]\!]\texttt{(x1,x2) + sig2 * }\text{VentureCov}\,[\![K_2]\!]\texttt{(x1,x2)\})} \\
\text{VentureProg}\,[\![K]\!] &::= \texttt{assume gp = gaussian\_process(gp\_mean\_constant(0), }\text{VentureCov}\,[\![K]\!]\texttt{);}
\end{aligned}$$

Fig. 3. Components of domain-specific language for automatic data modeling of time series using Gaussian process models. This DSL belongs to the family of probabilistic context-free grammars from Section 4.

significant increase in scalability. Quiñonero-Candela and Rasmussen [2005] show that sparse Gaussian process approximation methods typically correspond to well-defined probabilistic models that have different likelihood functions than the standard Gaussian process, and so our synthesis framework can naturally incorporate sparse Gaussian processes by adapting the Lik semantics.

As for the quadratic factor $O(\max(|K|, |K'|)n^2)$, this scaling depends largely on the characteristics of the underlying time series data. For simple time series with only a few patterns we expect the program size $|K|$ to be small, whereas for time series with several complex temporal patterns then the synthesized covariance expressions $K$ are longer to obtain strong fit to the data.

**Syntax and Production Rules**

$x, y, w \in \text{Numeric}, a \in [m], s \in [n]$

| | |
|---|---|
| $P \in \text{Partition} ::= (\texttt{partition}\ B_1 \ldots B_k)$ | [Partition] |
| $B \in \text{Block} ::= (\texttt{block}\ (a_1 \ldots a_l)\ C_1 \ldots C_t)$ | [Block] |
| $C \in \text{Cluster} ::= (\texttt{cluster}\ s\ V_1\ V_2 \ldots V_l)$ | [Cluster] |
| $V \in \text{Variable} ::= (\texttt{var}\ a\ D)$ | [Variable] |
| $D \in \text{Dist} ::= (\texttt{normal}\ (x\ y))$ | [Gaussian] |
| $\mid (\texttt{poisson}\ y)$ | [Poisson] |
| $\mid (\texttt{categorical}\ w_1 \ldots w_q)$ | [Categorical] |

**Prior Denotation**

$$\text{Prior}\,[\![(\texttt{partition}\ B_1 \ldots B_k)]\!] ::= \frac{\prod_{i=1}^{k} \text{Prior}\,[\![B_k]\!]}{m!}$$

$$\text{Prior}\,[\![(\texttt{block}\ (a_1 \ldots a_l)\ C_1 \ldots C_t)]\!] ::= (l-1)!\,\frac{\prod_{i=1}^{t} \text{Prior}\,[\![C_i]\!]}{n!}$$

$$\text{Prior}\,[\![(\texttt{cluster}\ s\ V_1 \ldots V_l)]\!] ::= (s-1)!\,\prod_{i=1}^{l} \text{Prior}\,[\![V_i]\!]$$

$$\text{Prior}\,[\![(\texttt{var}\ a\ D)]\!] ::= \text{Prior}\,[\![D]\!]$$

$$\text{Prior}\,[\![(\texttt{normal}\ v\ y)]\!] ::= \sqrt{\frac{\lambda}{y^2 2\pi}}\,\frac{\beta^\alpha}{\Gamma(\alpha)}\left(\frac{1}{y^2}\right)^{\alpha+1} \exp\left(\frac{-(2\beta + \lambda(v-\eta)^2)}{2y^2}\right)$$

$$\text{Prior}\,[\![(\texttt{poisson}\ y)]\!] ::= \frac{\xi^\nu y^{\nu-1} e^{-\xi y}}{\Gamma(\nu)}$$

$$\text{Prior}\,[\![(\texttt{categorical}\ w_1 \ldots w_q)]\!] ::= \frac{\Gamma(\kappa)^q}{\Gamma(\kappa)}\prod_{i=1}^{q} w_i^\alpha$$

$$\alpha, \beta, \lambda, \eta, \xi, \nu, \kappa ::= \text{(statistical constants)}$$

**Likelihood Denotation**

$$\text{Lik}\,[\![(\texttt{partition}\ B_1 \ldots B_k)]\!]\,(\mathbf{X}) ::= \prod_{i=1}^{k} \text{Lik}\,[\![B_k]\!]\,(\mathbf{X})$$

$$\text{Lik}\,[\![(\texttt{block}\ (a_1 \ldots a_l)\ C_1 \ldots C_t)]\!]\,(\mathbf{X}) ::= \prod_{i=1}^{n}\sum_{j=1}^{t} \text{Lik}\,[\![C_j]\!]\,(\mathbf{X}_i)$$

$$\text{Lik}\,[\![(\texttt{cluster}\ s\ V_1 \ldots V_l)]\!]\,(\mathbf{x}) ::= \frac{s}{n}\prod_{i=1}^{l} \text{Lik}\,[\![V_i]\!]\,(\mathbf{x})$$

$$\text{Lik}\,[\![(\texttt{var}\ a\ D)]\!]\,(\mathbf{x}) ::= \text{Lik}\,[\![D]\!]\,(\mathbf{x}_a)$$

$$\text{Lik}\,[\![(\texttt{normal}\ v\ y)]\!]\,(x) ::= \frac{1}{\sqrt{2\pi y^2}}\,e^{-\left(\frac{x-v}{\sqrt{2}y}\right)^2}$$

$$\text{Lik}\,[\![(\texttt{poisson}\ y)]\!]\,(x) ::= y^x e^{-y}/x!$$

$$\text{Lik}\,[\![(\texttt{categorical}\ w_1 \ldots w_q)]\!]\,(x) ::= w_x$$

**DSL to Venture Translation**

Probabilistic Program in DSL

```
(partition
 (block (1)
  (cluster 6 (var 1 (normal 0.6 2.1)))
  (cluster 4 (var 1 (normal 0.3 1.7))))
 (block (2 3)
  (cluster 2 (var 2 (normal 7.6 1.9)
             (var 3 (poisson 12))))
  (cluster 3 (var 2 (normal 1.1 0.5)
             (var 3 (poisson 1))))
  (cluster 5 (var 2 (normal -0.6 2.9)
             (var 3 (poisson 4))))))
```

Probabilistic Program in Venture

```
assume block1_cluster =
  categorical(simplex([0.6, 0.4])) #block:1;

assume var1 = cond(
  (block1_cluster == 0) (normal(0.6, 2.1))
  (block1_cluster == 1) (normal(0.3, 1.7)));

assume block2_cluster =
  categorical(simplex([0.2, 0.3, 0.5])) #block:2;

assume var2 = cond(
  (block2_cluster == 0) (normal(7.6, 1.9))
  (block2_cluster == 1) (normal(1.1, 0.5))
  (block2_cluster == 2) (normal(-0.6, 2.9)));

assume var3 = cond(
  (block2_cluster == 0) (poisson(12))
  (block2_cluster == 1) (poisson(1))
  (block2_cluster == 2) (poisson(4)));
```

Prediction Query in Venture

```
observe var3 = 8;
infer repeat(100, {gibbs(quote(block), one)});
sample [var1, var2];
```

Fig. 4. Components of domain-specific language for automatic data modeling of multivariate tabular data using nonparametric mixture models. This DSL is context-sensitive and contains a custom prior denotation.

```
(partition
 (block (1)
  (cluster 6 (var 1 (normal 0.6 2.1)))
  (cluster 4 (var 1 (normal 0.3 1.7))))
 (block (2 3)
  (cluster 10 (var 2 (normal 7.6 1.9))
              (var 3 (normal -2.6 7.7)))))
```

mutation $\longrightarrow$

```
(partition
 (block (1 3)
  (cluster 6 (var 1 (normal 0.6 2.1))
             (var 3 (normal -0.3 0.9)))
  (cluster 4 (var 1 (normal 0.3 1.7)
             (var 3 (normal -6.1 4.8)))
 (block (2)
  (cluster 10 (var 2 (normal 7.6 1.9)))))
```

(a) Move a variable into an existing block, and choose new distribution(s)

.

```
(partition
 (block (1)
  (cluster 6 (var 1 (normal 0.6 2.1)))
  (cluster 4 (var 1 (normal 0.3 1.7))))
 (block (2 3)
  (cluster 10 (var 2 (normal 7.6 1.9))
              (var 3 (normal -2.6 7.7)))))
```

mutation $\longrightarrow$

```
(partition
 (block (1)
  (cluster 6 (var 1 (normal 0.6 2.1)))
  (cluster 4 (var 1 (normal 0.3 1.7))))
 (block (2)
  (cluster 10 (var 2 (normal 7.6 1.9))))
 (block (3)
  (cluster 10 (var 3 (normal 8.4 0.9)))))
```

(b) Move a variable into a new block of its own, and choose new distribution(s).

```
(partition
 (block (1)
  (cluster 6 (var 1 (normal 0.6 2.1)))
  (cluster 4 (var 1 (normal 0.3 1.7))))
 (block (2 3)
  (cluster 10 (var 2 (normal 7.6 1.9))
              (var 3 (normal -2.6 7.7)))))
```

mutation $\longrightarrow$

```
(partition
 (block (1)
  (cluster 5 (var 1 (normal 0.6 2.1)))
  (cluster 4 (var 1 (normal 0.3 1.7)))
  (cluster 1 (var 1 (normal -5.0 2.3))))
 (block (2 3)
  (cluster 10 (var 2 (normal 7.6 1.9))
              (var 3 (normal -2.6 7.7)))))
```

(c) Within one block, create a new cluster with weight 1 and decrease the weight of an existing cluster by 1.

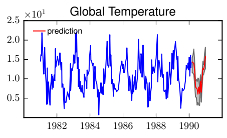
Fig. 5. Examples of mutation operators applied to a mixture modeling DSL program during Bayesian synthesis.

## 6 NONPARAMETRIC MIXTURE MODEL DSL FOR MULTIVARIATE TABULAR DATA

We now describe a second DSL for multivariate tabular data called MultiMixture. The setup starts with a data table containing $m$ columns and $n$ rows. Each column $c$ represents a distinct random variable $X_c$ and each row $r$ is a joint instantiation $\{x_{r1}, \dots, x_{rm}\}$ of all the $m$ random variables. Our DSL describes the data generating process for cells in the table based on the nonparametric "Cross-Categorization" mixture model introduced by Mansinghka et al. [2014], which we review.

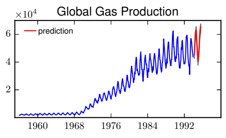
Figure 4 shows the components required for Bayesian synthesis in the MultiMixture DSL. Probabilistic programs in this DSL assert that the set of columns $[m] := \{1, \dots, m\}$ is partitioned into $k$ non-overlapping blocks $B_i$, which is specified by the `partition` production rule. Each `block` contains a set of unique columns $\{a_1, \dots, a_l\} \subset [m]$, where a particular column must appear in exactly one block of the `partition`. Each `block` has clusters $C_i$, where each cluster specifies a component in a mixture model. Each `cluster` has relative weight $s$, a set of `var` objects that specify a variable index $a$, and a primitive distribution $D$. The distributions are `normal` (with mean $x$ and variance $y$); `poisson` (with mean $y$); and `categorical` (with category weights $w_i$).

Since the MultiMixture DSL is not context-free, we specify custom Prior semantics that recursively describes the joint probability of all terms in an expression. These probabilities provide a hierarchical decomposition of the distribution in [Mansinghka et al. 2014, Section 2.2] based on the structure of the DSL program. Prior normalization (Condition 3.1) holds since the five production rules in the grammar are non-recursive. The Lik semantics break down the full probability of a data table of observations $\mathbf{X}$ into the cell-wise probabilities within each mixture component. The likelihood is bounded (Condition 3.3) since `poisson` and `categorical` likelihoods are bounded by one, and Prior $[\![$(`normal` $v\ y$)$]\!]$ is conjugate to Lik $[\![$(`normal` $v\ y$)$]\!]$ [Bernardo and Smith 1994]. The semantics



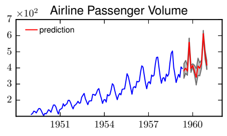
| Temporal Structure | $p^{\text{synth}}$ |
|---|---|
| ✓ White Noise | 100% |
| × Linear Trend | 16% |
| ✓ Periodicity | 92% |
| × Change Point | 4% |

(a)



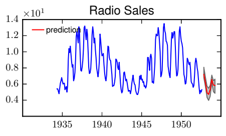
| Temporal Structure | $p^{\text{synth}}$ |
|---|---|
| ✓ White Noise | 100% |
| ✓ Linear Trend | 85% |
| ✓ Periodicity | 76% |
| ✓ Change Point | 76% |

(b)



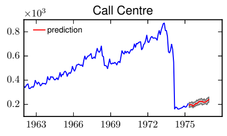
| Temporal Structure | $p^{\text{synth}}$ |
|---|---|
| ✓ White Noise | 100% |
| ✓ Linear Trend | 95% |
| ✓ Periodicity | 95% |
| × Change Point | 22% |

(c)



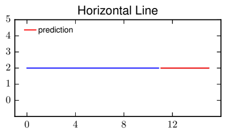
| Temporal Structure | $p^{\text{synth}}$ |
|---|---|
| ✓ White Noise | 100% |
| × Linear Trend | 6% |
| ✓ Periodicity | 93% |
| × Change Point | 23% |

(d)



| Temporal Structure | $p^{\text{synth}}$ |
|---|---|
| ✓ White Noise | 100% |
| ✓ Linear Trend | 93% |
| ✓ Periodicity | 97% |
| ✓ Change Point | 90% |

(e)



| Temporal Structure | $p^{\text{synth}}$ |
|---|---|
| × White Noise | 3% |
| × Linear Trend | 8% |
| × Periodicity | 1% |
| × Change Point | 2% |

(f)

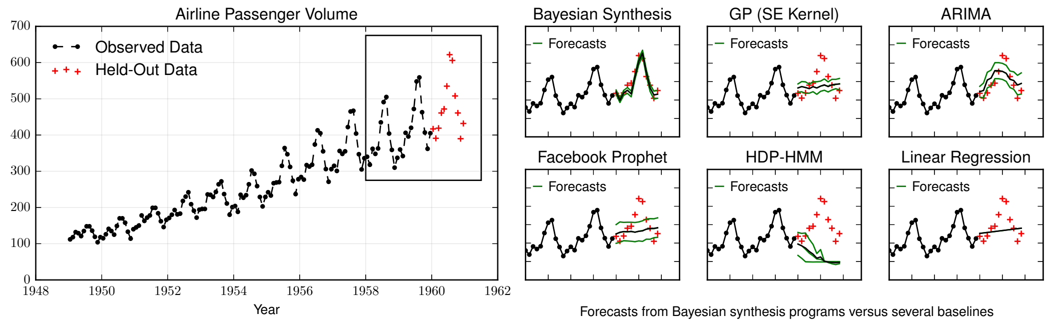
Fig. 6. Detecting probable temporal structures in multiple time series with varying characteristics. In each of the panels (a)–(f), the plot shows observed time series data in blue and the table identifies which temporal structures truly exist in the time series as well as the posterior probability $p^{\text{synth}}$ that each structure is present in a Gaussian process program from Bayesian synthesis given the data. As described in Eq (5) of Section 3.1, $p^{\text{synth}}$ is estimated by returning the fraction of programs in the ensemble that contain each structure. Programs from Bayesian synthesis accurately reflect the probable presence or absence of linear, periodic, and change point characteristics. The red lines show predictions from a randomly selected synthesized program, showing that they additionally capture compositions of temporal structures to faithfully model the data.

of MultiMixture are designed to improve on the model family from Mansinghka et al. [2014] by making it (i) easier to embed the model in a probabilistic language without enforcing complicated exchangeable coupling (i.e. the Venture programs for modeling and prediction in Figure 4); and (ii) possible to write a recursive, decomposable semantics for the prior and likelihood.

Since the MultiMixture DSL is not generated by a context-free grammar, we employ custom synthesis algorithms using the cycle of Gibbs kernels given in Mansinghka et al. [2016, Section 2.4]. Figure 5 shows several examples of program mutations that occur over the course of synthesis to sample from the approximate posterior distribution over DSL expressions. For a given expression $E \in \mathcal{L}$, a full sweep over all program mutations shown in Figure 5 has time complexity $O(mnkt)$, where $m$ is the number of columns, $n$ is the number of rows, $k$ is the number of `block` subexpressions, and $t$ is the maximum number of `cluster` subexpressions under any `block` subexpression. Full details of this algorithmic analysis can be found in Mansinghka et al. [2016, Section 2.4].

## 7 EXPERIMENTAL RESULTS

We have developed a set of benchmark problems in two domains: (i) time series data and (ii) multivariate tabular data. These benchmarks reflect a broad range of real-world data generating processes with varying qualitative structure. We evaluated these probabilistic programs in two ways. First, we qualitatively assessed the inferences about structure that were made by processing the text of the programs. Second, we quantitatively benchmarked the predictive accuracy of the synthesized programs. Probabilistic programs from Bayesian synthesis often provide improved accuracy over standard methods for data modeling.



Standardized Root Mean Squared Forecasting Error (RMSE) on Real-World Benchmark Problems

| | `temperature` | `airline` | `call` | `mauna` | `radio` | `solar` | `wheat` |
|---|---|---|---|---|---|---|---|
| Bayesian Synthesis | **1.0** | **1.0** | **1.0** | **1.0** | **1.0** | 1.47 | 1.50 |
| Gaussian Process (Squared Exponential Kernel) | 1.70 | 2.01 | 4.26 | 1.54 | 2.03 | 1.63 | 1.37 |
| Auto-Regressive Integrated Moving Average | 1.85 | 1.32 | 2.44 | 1.09 | 2.08 | **1.0** | 1.41 |
| Facebook Prophet | 2.00 | 1.83 | 5.61 | 1.23 | 3.09 | 1.73 | 1.29 |
| Hierarchical-DP Hidden Markov Model | 1.77 | 4.61 | 2.26 | 14.77 | 1.19 | 3.49 | 1.89 |
| Linear Regression | 1.30 | 1.79 | 6.23 | 2.19 | 2.73 | 1.57 | **1.0** |

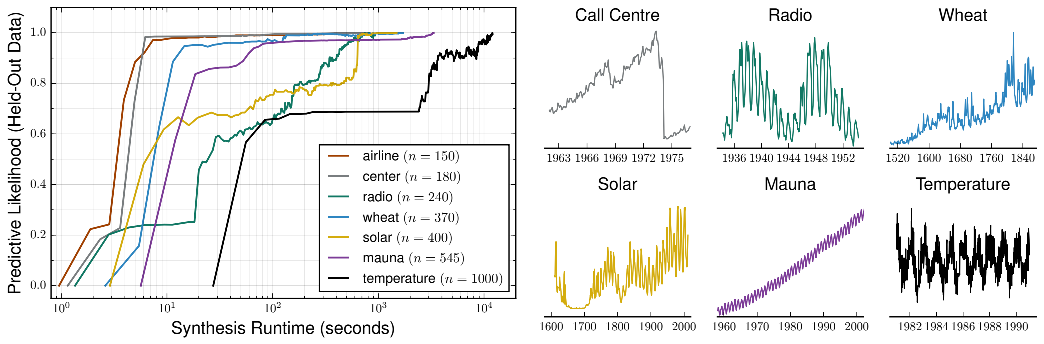
Fig. 7. Quantitative evaluation of forecasting using Gaussian process programs from Bayesian synthesis, as compared to five common baselines. The top panels show extrapolated time series by each method on the `airline` data. The table shows prediction errors achieved by each method on seven real-world time series.

## 7.1 Inferring Qualitative Structure from Time Series Data

This set of benchmark problems consists of five real-world time series datasets from Lloyd [2014]: (i) bi-weekly global temperature measurements from 1980 to 1990, (ii) monthly global gas production from 1956 to 1995, (iii) monthly airline passenger volume from 1948, (iv) monthly radio sales from 1935 to 1954, and (iv) monthly call center activity from 1962 to 1976. Each of the datasets has a different qualitative structure. For example, the global temperature dataset is comprised of yearly periodic structure overlaid with white noise, while the call center dataset has a linear trend until 1973 and a sharp drop afterwards. Figure 6 shows the five datasets, along with the inferences our technique made about the qualitative structure of the data generating process underlying each dataset. The results show that our technique accurately infers the presence or absence of each type of structure in each benchmark. Specifically, if a specific qualitative structure is deemed to be present if the posterior probability inferred by our technique is above 50%, and inferred to be absent otherwise, then the inferences from Bayesian synthesis match the real world structure in every case. To confirm that the synthesized programs report the absence of any meaningful structure when it is not present in the data, we also include a horizontal line shown in panel (f).

## 7.2 Quantitative Prediction Accuracy for Time Series Data

Figure 7 shows quantitative prediction accuracy for seven real world econometric time series. The top panel visually compares predictions from our method with predictions obtained by baseline methods on the airline passenger volume benchmark. We chose to compare against baselines that (i) have open source, re-usable implementations; (ii) are widely used and cited in the statistical literature; and (iii) are based on flexible model families that, like our Gaussian process technique, have default hyperparameter settings and do not require significant manual tuning to fit the data. Baseline methods include Facebook's Prophet algorithm [Taylor and Letham 2018]; standard econometric



(a) Runtime vs. Accuracy Profiles

(b) Time Series Datasets

Fig. 8. Synthesis runtime versus held-out predictive likelihood on the time series benchmarked in Figure 7. Panel (a) shows a log-linear plot of the median runtime and predictive likelihood values taken over 100 independent runs of synthesis. Higher values of predictive likelihood on the y-axis indicate better fit to the observed data. Panel (b) shows the time series datasets used to compute runtime and accuracy measurements.

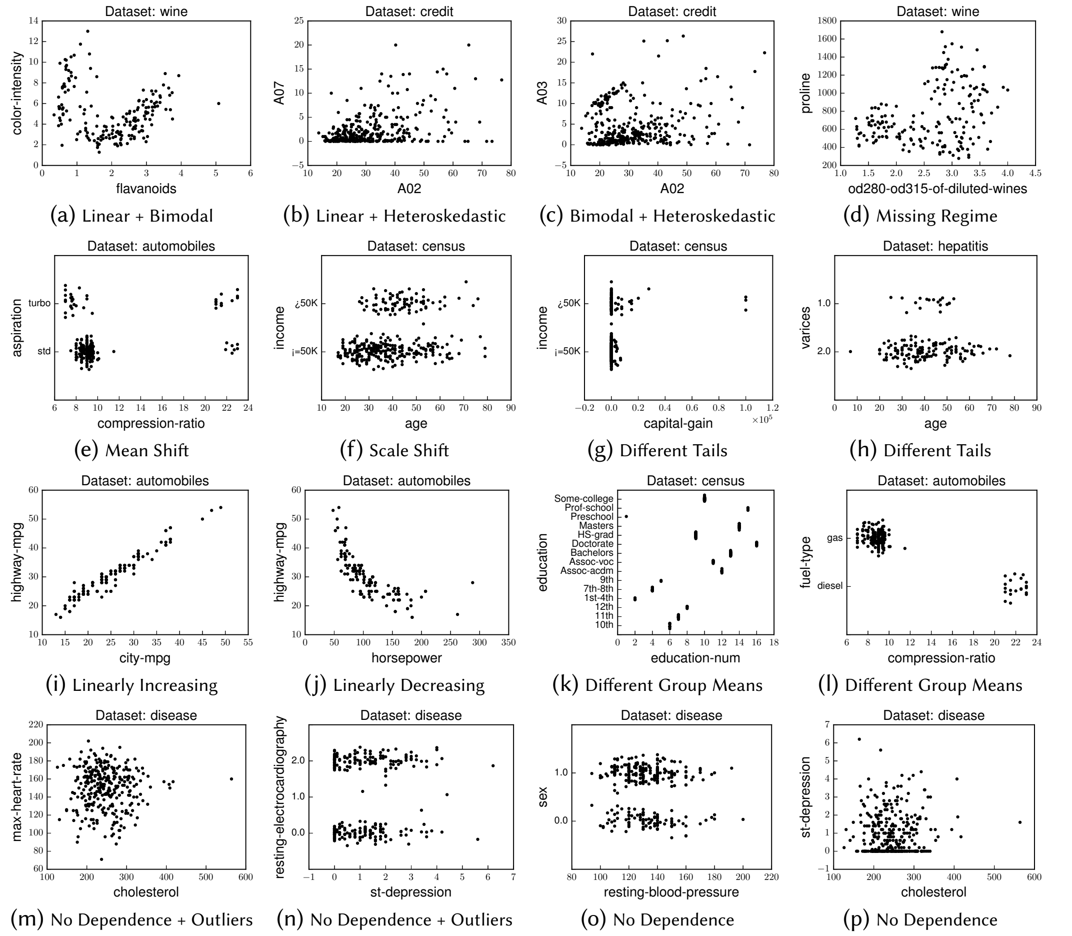
techniques, such as Auto-Regressive Integrated Moving Average (ARIMA) modeling [Hyndman and Khandakar 2008]; and advanced time series modeling techniques from nonparametric Bayesian statistics, such as the Hierarchical Dirichlet Process Hidden Markov Model [Johnson and Willsky 2013]. Bayesian synthesis is the only technique that accurately captures both quantitative and qualitative structure. The bottom panel of Figure 7 shows quantitative results comparing predictive accuracy on these datasets. Bayesian synthesis produces more accurate models for five of the seven benchmark problems and is competitive with other techniques on the other two problems.

## 7.3 Runtime Versus Prediction Accuracy for Time Series Data

Figure 8 shows a profile of the predictive likelihood on held-out data versus the synthesis runtime (in seconds) for each of the seven econometric time series. For each dataset, we ran 2000 steps of synthesis, taking runtime and predictive likelihood measurements at every 5 steps. The plotted runtime and predictive likelihood values represent the median values taken over 100 independent runs of Bayesian synthesis for each dataset. The predictive likelihood measurements on the y-axis are scaled between 0 and 1 so that these profiles can be compared across the seven datasets. We see that there is significant variance in the synthesis runtime versus prediction quality. For certain time series such as `airline` and `call` datasets (150 and 180 observations), prediction quality stabilizes after around 10 seconds; for `radio` and `solar` datasets (240 and 400 observations) the prediction quality takes around 1000 seconds to stabilize; and for the `temperature` dataset (1000 observations) the predictions continue to improve even after exceeding the maximum timeout.

Recall from Section 5.1 that each step of synthesis has a time complexity of $O(l_G + |E|n^2 + n^3)$. Scaling the synthesis to handle time series with more than few thousands data points will require the sparse Gaussian process techniques discussed in Section 5.1. It is also important to emphasize that the time cost of each iteration of synthesis is not only a function of the number of observations $n$ but also the size $|E|$ of the synthesized programs. Thus, while more observations typically require more time, time series with more complex temporal patterns typically result in longer synthesized programs. This trade-off can be seen by comparing the profiles of the mauna data (545 observations) with the radio data (240 observations). While mauna has more observations $n$, the radio data contains much more complex periodic patterns and which requires comparatively longer programs and thus longer synthesis times.

| | Variable 1 | Variable 2 | True Predictive Structure | | Predictive Relationship Detected By | | | |
|---|---|---|---|---|---|---|---|---|
| | | | | | Pearson Correlation | | Bayesian Synthesis | |
| (a) | flavanoids | color-intensity | linear + bimodal | ✓ | × | (0.03) | ✓ | (0.97) |
| (b) | A02 | A07 | linear + heteroskedastic | ✓ | × | (0.16) | ✓ | (0.89) |
| (c) | A02 | A03 | linear + bimodal + heteroskedastic | ✓ | × | (0.03) | × | (0.66) |
| (d) | proline | od280-of-wines | nonlinear + missing regime | ✓ | × | (0.09) | ✓ | (0.97) |
| (e) | compression-ratio | aspiration | mean shift | ✓ | × | (0.07) | ✓ | (0.98) |
| (f) | age | income | different group tails | ✓ | × | (0.06) | ✓ | (0.90) |
| (g) | age | varices | scale shift | ✓ | × | (0.00) | ✓ | (0.90) |
| (h) | capital-gain | income | different group tails | ✓ | × | (0.05) | × | (0.77) |
| (i) | city-mpg | highway-mpg | linearly increasing | ✓ | ✓ | (0.95) | ✓ | (1.00) |
| (j) | horsepower | highway-mpg | linearly decreasing | ✓ | ✓ | (0.65) | ✓ | (1.00) |
| (k) | education-years | education-level | different group means | ✓ | ✓ | (1.00) | ✓ | (1.00) |
| (l) | compression-ratio | fuel-type | different group means | ✓ | ✓ | (0.97) | ✓ | (0.98) |
| (m) | cholesterol | max-heart-rate | none (+ outliers) | × | × | (0.00) | × | (0.08) |
| (n) | cholesterol | st-depression | none (+ outliers) | × | × | (0.00) | × | (0.00) |
| (o) | blood-pressure | sex | none | × | × | (0.01) | × | (0.26) |
| (p) | st-depression | electrocardiography | none | × | × | (0.04) | × | (0.00) |



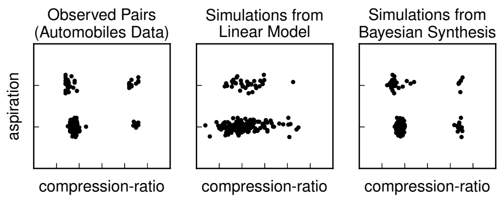
(a) Linear + Bimodal (b) Linear + Heteroskedastic (c) Bimodal + Heteroskedastic (d) Missing Regime

(e) Mean Shift (f) Scale Shift (g) Different Tails (h) Different Tails

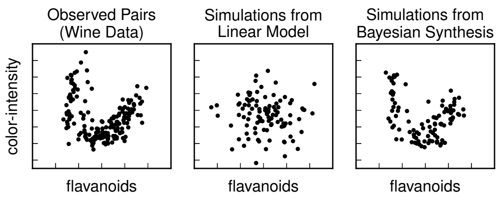
(i) Linearly Increasing (j) Linearly Decreasing (k) Different Group Means (l) Different Group Means

(m) No Dependence + Outliers (n) No Dependence + Outliers (o) No Dependence (p) No Dependence

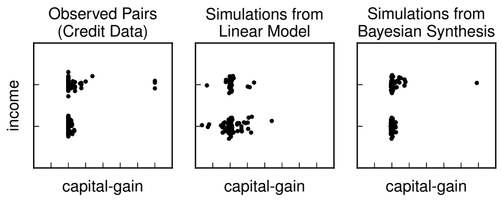
Fig. 9. Detecting probable predictive relationships between pairs of variables for widely varying dependence structures including nonlinearity, heteroskedasticity, mean and scale shifts, and missing regimes. The table in the top panel shows 16 pairs of variables, the true predictive structure between them, and indicators as to whether a predictive relationship, if any, was detected by Pearson correlation and Bayesian synthesis.



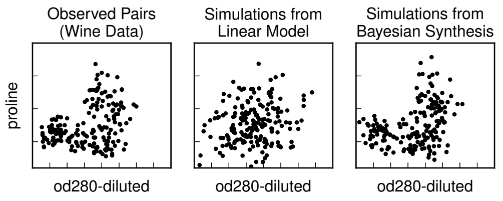
(a) Bimodal Dependence



(b) Bimodal Increasing Linear Dependence



(c) Heavy-Tailed Nonlinear Dependence



(d) Missing Regime Nonlinear Dependence

Fig. 10. Comparing the quality of predictive models discovered by Bayesian synthesis to commonly-used generalized linear models, for several real-world datasets and dependence patterns. In each of the panels (a)–(d), the first column shows a scatter plot of two variables in the observed dataset; the second column shows data simulated from a linear model trained on the observed data; and the third column shows data simulated from probabilistic programs obtained using Bayesian synthesis given the observed data. The synthesized programs are able to detect underlying patterns and emulate the true distribution, including nonlinear, multi-modal, and heteroskedastic relationships. Simulations from linear models have a poor fit.

## 7.4 Inferring Qualitative Structure from Multivariate Tabular Data

A central goal of data analysis for multivariate tabular data is to identify predictive relationships between pairs of variables [Ezekiel 1941; Draper and Smith 1966]. Recall from Section 6 that synthesized programs from the multivariate tabular data DSL place related variables in the same variable block expression. Using Eq (5) from Section 3.1, we report a relationship between a pair of variables if 80% of the synthesized programs place these variables in the same block. We compare our method to the widely-used Pearson correlation baseline [Abbott 2016] by checking whether the correlation value exceeds 0.20 (at the 5% significance level).

We evaluate the ability of our method to correctly detect related variables in six real-world datasets from the UCI machine learning repository [Dheeru and Karra Taniskidou 2017]: automobile data (26 variables), wine data (14 variables), hepatitis data (20 variables), heart disease data (15 variables), and census data (15 variables). Figure 9 shows results for sixteen selected pairs of variables in these datasets (scatter plots shown in 9a–9p). Together, these pairs exhibit a broad class of relationships, including linear, nonlinear, heteroskedastic, and multi-modal patterns. Four of the benchmarks, shown in the final row, have no predictive relationship.

The table in Figure 9 shows that our method correctly detects the presence or absence of a predictive relationship in 14 out of 16 benchmarks, where the final column shows the posterior probability of a relationship in each case. In contrast, Pearson correlation, shown in the second-to-last column, only obtains the correct answer in 8 of 16 benchmarks. This baseline yields incorrect answers for all pairs of variables with nonlinear, bimodal, and/or heteroskedastic relationships. These results show that Bayesian synthesis provides a practical method for detecting complex predictive relationships from real-world data that are missed by standard baselines.

Moreover, Figure 10 shows that probabilistic programs from Bayesian synthesis are able to generate entirely new datasets that reflect the pattern of the predictive relationship in the underlying data generating process more accurately than generalized linear statistical models.

## 7.5 Quantitative Prediction Accuracy for Multivariate Tabular Data

Another central goal of data analysis for multivariate tabular data is to learn probabilistic models that can accurately predict the probability of new data records, a task known as density estimation [Silverman 1986; Scott 2009]. Accurately predicting the probability of new data enables several important tasks such as data cleaning, anomaly detection, and imputing missing data. We obtain the probability of new data records according to the synthesized probabilistic programs by first translating the programs into Venture (right column of Figure 4) and then executing the translated programs in the Venture inference environment. Given a new data record, these programs immediately return the probability of the new data record according to the probabilistic model specified by the Venture probabilistic program. We compare our method to the widely-used multivariate kernel density estimation (KDE) baseline with mixed data types from Racine and Li [2004].

We evaluate our method's ability to accurately perform density estimation using 13 benchmark datasets adapted from Chasins and Phothilimthana [2017]. Each dataset was generated by a "ground-truth" probabilistic program written in BLOG [Milch et al. 2005]. Because the ground-truth program is available, we can compare the actual probability of each new data record (obtained from the ground-truth BLOG program) with its predicted probability (obtained from the synthesized Venture programs and from KDE). The results in the table in Figure 11 show that our method is more accurate than KDE, often by several orders of magnitude. A key advantage of our approach is that we synthesize an ensemble of probabilistic programs for each dataset. This ensemble enables us to approximate the full posterior distribution and then use this distribution to compute error bars for predicted probability values. The scatter plots in Figure 11 show that the error bars are well-calibrated, i.e. error bars are wider for data records where our method gives less accurate estimates. In contrast, KDE only gives point estimates with no error bars and no measure of uncertainty.

Median Error in Predictive Log-Likelihood of Held-Out Data

| Benchmark | Kernel Density Estimation | Bayesian Synthesis |
|---|---|---|
| biasedtugwar | $-1.58 \times 10^{-1}$ | $-3.13 \times 10^{-2}$ |
| burglary | $-4.25 \times 10^{-1}$ | $-1.45 \times 10^{-3}$ |
| csi | $-1.00 \times 10^{-1}$ | $-4.35 \times 10^{-4}$ |
| easytugwar | $-2.96 \times 10^{-1}$ | $-9.96 \times 10^{-2}$ |
| eyecolor | $-4.69 \times 10^{-2}$ | $-5.31 \times 10^{-3}$ |
| grass | $-3.91 \times 10^{-1}$ | $-4.49 \times 10^{-2}$ |
| healthiness | $-1.35 \times 10^{-1}$ | $-3.00 \times 10^{-3}$ |
| hurricane | $-1.30 \times 10^{-1}$ | $-2.11 \times 10^{-4}$ |
| icecream | $-1.51 \times 10^{-1}$ | $-7.07 \times 10^{-2}$ |
| mixedCondition | $-1.06 \times 10^{-1}$ | $-1.43 \times 10^{-2}$ |
| multipleBranches | $-4.12 \times 10^{-2}$ | $-1.22 \times 10^{-2}$ |
| students | $-1.74 \times 10^{-1}$ | $-5.47 \times 10^{-2}$ |
| tugwarAddition | $-2.60 \times 10^{-1}$ | $-1.38 \times 10^{-1}$ |
| uniform | $-2.72 \times 10^{-1}$ | $-1.26 \times 10^{-1}$ |

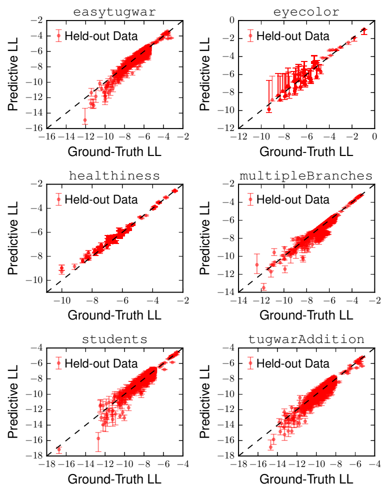


Fig. 11. Comparing the error in the predictive probabilities of held-out data according to Bayesian synthesis and KDE. For each of the 13 benchmark problems, a training set of 10,000 data records was used to synthesize probabilistic programs in our tabular data DSL and to fit KDE models. 10,000 new data records were then used to assess held-out predictive probabilities, which measures how well the synthesized probabilistic programs are able to model the true distribution. The scatter plots on the right show the full distribution of the true (log) probabilities (according to the ground-truth program) and the predictive probabilities (according to the synthesized programs) for six of the benchmarks (each red dot is a held-out data record). Estimates are most accurate and certain in the bulk of the distribution and are less accurate and less certain in the tails.

## 8 RELATED WORK

We discuss related work in five related fields: Bayesian synthesis of probabilistic programs (where this work is the first), non-Bayesian synthesis of probabilistic programs, probabilistic synthesis of non-probabilistic programs, non-probabilistic synthesis of non-probabilistic programs, and model discovery in probabilistic machine learning.

**Bayesian synthesis of probabilistic programs.** This paper presents the first Bayesian synthesis of probabilistic programs. It presents the first formalization of Bayesian synthesis for probabilistic programs and the first soundness proofs of Bayesian synthesis for probabilistic programs generated by probabilistic context-free grammars. It also presents the first empirical demonstration of accurate modeling of multiple real-world domains and tasks via this approach.

This paper is also the first to: (i) identify sufficient conditions to obtain a well-defined posterior distribution; (ii) identify sufficient conditions to obtain a sound Bayesian synthesis algorithm; (iii) define a general family of domain-specific languages with associated semantics that ensure that the required prior and posterior distributions are well-defined; (iv) present a sound synthesis algorithm that applies to any language in this class of domain-specific languages; and (v) present a specific domain-specific language (the Gaussian process language for modeling time series data) that satisfies these sufficient conditions.

Nori et al. [2015] introduce a system (PSKETCH) designed to complete partial sketches of probabilistic programs for modeling tabular data. The technique is based on applying sequences of program mutations to a base program written in a probabilistic sketching language. As we detail further below, the paper uses the vocabulary of Bayesian synthesis to describe the technique but contains multiple fundamental technical errors that effectively nullify its key claims. Specifically, Nori et al. [2015] proposes to use Metropolis-Hastings sampling to approximate the *maximum a posteriori* solution to the sketching problem. However, the paper does not establish that the prior or posterior distributions on programs are well-defined. The paper attempts to use a uniform prior distribution over an unbounded space of programs. However, there is no valid uniform probability measure over this space. Because the posterior is defined using the prior and the marginal likelihood of all datasets is not shown to be finite, the posterior is also not well-defined. The paper presents no evidence or argument that the proposed prior or posterior distribution is well-defined. The paper also asserts that the synthesis algorithm converges because the MH algorithm always converges, but it does not establish that the presented framework satisfies the properties required for the MH algorithm to converge. And because the posterior is not well-defined, there is no probability distribution to which the MH algorithm can converge. We further note that while the paper claims to use the MH algorithm to determine whether a proposed program mutation is accepted or rejected, there is in fact no tractable algorithm that can be used to compute the reversal probability of going back from a proposed to an existing program, which means the MH algorithm cannot possibly be used with the program mutation proposals described in the paper.

The presented system, PSKETCH, is based on applying sequences of program mutations to a base program written in a probabilistic sketching language with constructs (such as if-then-else, for loops, variable assignment, and arbitrary arithmetic operations) drawn from general-purpose programming languages. We believe that, consistent with the experimental results presented in the paper, working only with these constructs is counterproductive in that it produces a search space that is far too unconstrained to yield practical solutions to real-world data modeling problems in the absence of other sources of information that can more effectively narrow the search. We therefore predict that the field will turn to focus on probabilistic DSLs (such as the ones presented in this paper) as a way to more effectively target the problems that arise in automatic data modeling. As an example, while the sketching language in Nori et al. [2015] contains general-purpose programming constructs, it does not have domain-specific constructs that concisely represent Gaussian processes, covariance functions, or rich nonparametric mixture models used in this paper.

Hwang et al. [2011] use beam search over arbitrary program text in a subset of the Church [Goodman et al. 2008] language to obtain a generative model over tree-like expressions. The resulting search space is so unconstrained that, as the authors note in the conclusion, this technique does not apply to any real-world problem whatsoever. This drawback highlights the benefit of controlling the search space via an appropriate domain-specific language. In addition we note that although Hwang et al. [2011] also use the vocabulary of Bayesian synthesis, the proposed program-length prior over an unbounded program space is not shown to be probabilistically well-formed, nor is it shown to lead to a valid Bayesian posterior distribution over programs, nor is the stochastic approximation of the program likelihood shown to result in a sound synthesis algorithm.

**Non-Bayesian synthesis of probabilistic programs.** Ellis et al. [2015] introduce a method for synthesizing probabilistic programs by using SMT solvers to optimize the likelihood of the observed data with a regularization term that penalizes the program length. The research works with a domain-specific language for morphological rule learning. Perov and Wood [2014] propose to synthesize code for simple random number generators using approximate Bayesian computation. These two techniques are fundamentally different from ours. They focus on different problems, specifically morphological rule learning, visual concept learning, and random number generators, as opposed to data modeling. Neither is based on Bayesian learning and neither presents soundness proofs (or even states a soundness property). Moreover, both attempt to find a single highest-scoring program as opposed to synthesizing ensembles of programs that approximate a Bayesian posterior distribution over the sampled probabilistic programs. As this previous research demonstrates, obtaining soundness proofs or characterizing the uncertainty of the synthesized programs against the data generating process is particularly challenging for non-Bayesian approaches because of the ad-hoc nature of the synthesis formulation.

Tong and Choi [2016] describe a method which uses an off-the-shelf model discovery system [Lloyd 2014, ABCD] to learn Gaussian process models and the code-generate the models to Stan [Carpenter et al. 2017] programs. However, Tong and Choi [2016] does not formalize the program synthesis problem, nor does it present any formal semantics, nor does it show how to extract the probability that qualitative properties hold in the data, nor does it apply to multiple model families in a single problem domain let alone multiple problem domains. Chasins and Phothilimthana [2017] present a technique for synthesizing probabilistic programs in tabular datasets using if-else statements. Unlike our method, their approach requires the user to manually specify a causal ordering between the variables. This information may be difficult or impossible to obtain. Second, the proposed technique is based on using linear correlation, which we have shown in our experiments (Figures 10 and 9) fails to adequately capture complex probabilistic relationships. Finally, the paper is based on simulated annealing, not Bayesian synthesis, has no probabilistic interpretation, and does not claim to provide a notion of soundness.

**Probabilistic synthesis of non-probabilistic programs.** Schkufza et al. [2013] describe a technique for synthesizing high-performance X86 binaries by using MCMC to stochastically search through a space of programs and to deliver a single X86 program which satisfies the hard constraint of correctness and the soft constraint of performance improvement. While both the Bayesian synthesis framework in this paper and the superoptimization technique from Schkufza et al. [2013] use MCMC algorithms to implement the synthesis, they use MCMC in a fundamentally different way. In Schkufza et al. [2013], MCMC is used as a strategy to search through a space of deterministic programs that seek to minimize a cost function that itself has no probabilistic semantics. In contrast, Bayesian synthesis uses MCMC as a strategy to approximately sample from a space of probabilistic programs whose semantics specify a well-defined posterior distribution programs.

Bayesian priors over structured program representations which seek to sample from a posterior distribution over programs have also been investigated. Liang et al. [2010] use adapter grammars [Johnson et al. 2007] to build a hierarchical nonparametric Bayesian prior over programs specified in combinatory logic [Schönfinkel 1924]. Ellis et al. [2016] describe a method to sample from a bounded program space with a uniform prior where the posterior probability of a program is equal to (i) zero if it does not satisfy an observed input-output constraint, or (ii) geometrically decreasing in its length otherwise. These methods are used to synthesize arithmetic operations, text editing, and list manipulation programs. Both Liang et al. [2010] and Ellis et al. [2015] specify Bayesian priors over programs similar to the prior in this paper. However, these two techniques are founded on the assumption that the synthesized programs have deterministic input-output behavior and cannot be easily extended to synthesize programs that have probabilistic input-output behavior. In contrast, the Bayesian synthesis framework presented in this paper can synthesize programs with deterministic input-output behavior by adding hard constraints to the Lik semantic function, although alternative synthesis techniques may be required to make the synthesis more effective.

Lee et al. [2018] present a technique for speeding up program synthesis of non-probabilistic programs by using A* search to enumerate programs that satisfy a set of input-output constraints in order of decreasing prior probability. This prior distribution over programs is itself learned using a probabilistic higher-order grammar with transfer learning over a large corpus of existing synthesis problems and solutions. The technique is used to synthesize programs in domain-specific languages for bit-vector, circuit, and string manipulation tasks. Similar to the Bayesian synthesis framework in this paper, Lee et al. [2018] use PCFG priors for specifying domain-specific languages. However the fundamental differences are that the synthesized programs in Lee et al. [2018] are non-probabilistic and the objective is to enumerate valid programs sorted by their prior probability, while in this paper the synthesized programs are probabilistic so enumeration is impossible and the objective is instead to sample programs according to their posterior probabilities.

**Non-probabilistic synthesis of non-probabilistic programs.** Over the last decade program synthesis has become a highly active area of research in programming languages. Key techniques include deductive logic with program transformations [Burstall and Darlington 1977; Manna and Waldinger 1979, 1980], genetic programming [Koza 1992; Koza et al. 1997], solver and constraint-based methods [Solar-Lezama et al. 2006; Jha et al. 2010; Gulwani et al. 2011; Gulwani 2011; Feser et al. 2015], syntax-guided synthesis [Alur et al. 2013], and neural networks [Graves et al. 2014; Reed and de Freitas 2016; Balog et al. 2017; Bošnak et al. 2018]. In general these approaches have tended to focus on areas where uncertainty is not essential or even relevant to the problem being solved. In particular, synthesis tasks in this field apply to programs that exhibit deterministic input-output behavior in discrete problem domains with combinatorial solutions. There typically exists an "optimal" solution and the synthesis goal is to obtain a good approximation to this solution.

In contrast, in Bayesian synthesis the problem domain is fundamentally about automatically learning models of non-deterministic data generating processes given a set of noisy observations. Given this key characteristic of the problem domain, we deliver solutions that capture uncertainty at two levels. First, our synthesis produces probabilistic programs that exhibit noisy, non-deterministic input-output behavior. Second, our technique captures inferential uncertainty over the structure and parameters of the synthesized programs themselves by producing an ensemble of probabilistic programs whose varying properties reflect this uncertainty.

**Model discovery in probabilistic machine learning.** Researchers have developed several probabilistic techniques for discovering statistical model structures from observed data. Prominent examples include Bayesian network structures [Mansinghka et al. 2006], matrix-composition models [Grosse et al. 2012], univariate time series [Duvenaud et al. 2013], multivariate time series [Saad and Mansinghka 2018], and multivariate data tables [Mansinghka et al. 2016].

Our paper introduces a general formalism that explains and extends these modeling techniques. First, it establishes general conditions for Bayesian synthesis to be well-defined and introduces sound Bayesian synthesis algorithms that apply to a broad class of domain-specific languages.

Second, we show how to estimate the probability that qualitative structures are present or absent in the data. This capability rests on a distinctive aspect of our formalism, namely that soundness is defined in terms of sampling programs from a distribution that converges to the Bayesian posterior, not just finding a single "highest scoring" program. As a result, performing Bayesian synthesis multiple times yields a collection of programs that can be collectively analyzed to form Monte Carlo estimates of posterior and predictive probabilities.

Third, our framework shows one way to leverage probabilistic programming languages to simplify the implementation of model discovery techniques. Specifically, it shows how to translate programs in a domain-specific language into a general-purpose language such as Venture, so that the built-in language constructs for predictive inference can be applied (rather than requiring new custom prediction code for each model family). A key advantage of this approach is that representing probabilistic models as synthesized probabilistic programs enable a wide set of capabilities for data analysis tasks [Saad and Mansinghka 2016, 2017].

Fourth, we implemented two examples of domain-specific languages and empirically demonstrate accuracy improvements over multiple baselines.

## 9 CONCLUSION

We present a technique for Bayesian synthesis of probabilistic programs for automatic data modeling. This technique enables, for the first time, users to solve important data analysis problems without manually writing statistical programs or probabilistic programs. Our technique (i) produces an ensemble of probabilistic programs that soundly approximate the Bayesian posterior, allowing an accurate reflection of uncertainty in predictions made by the programs; (ii) is based on domain-specific modeling languages that are empirically shown to capture the qualitative structure of a broad class of data generating processes; (iii) processes synthesized programs to extract qualitative structures from the surface syntax of the learned programs; (iv) translates synthesized programs into probabilistic programs that provide the inference machinery needed to obtain accurate predictions; and (v) is empirically shown to outperform baseline statistical techniques for multiple qualitative and quantitative data analysis and prediction problems.

Designing good probabilistic domain-specific languages and priors is a key characteristic of our approach to automatic data modeling. In this paper we focus on broad non-parametric Bayesian priors that are flexible enough to capture a wide range of patterns and are also tractable enough for Bayesian synthesis. In effect, the choice of prior is embedded in the language design pattern, is done once when the language is designed, and can be reused across applications. The high-level nature of the DSL and carefully designed priors allow us to restrict the space of probabilistic programs to those that are well-suited for a particular data modeling task (such as time series and multivariate data tables). Once an appropriate DSL is developed, the synthesis algorithms can then be optimized on a per-DSL basis. Future directions of research include developing optimized synthesis algorithms to improve scalability; adding language support for user-specified qualitative constraints that must hold in the synthesized programs; and information-flow analyses on the synthesized programs to quantify relationships between variables in the underlying data generating process.

## ACKNOWLEDGMENTS

This research was supported by the DARPA SD2 program (contract FA8750-17-C-0239); grants from the MIT Media Lab, the Harvard Berkman Klein Center Ethics and Governance of AI Fund, and the MIT CSAIL Systems That Learn Initiative; as well as an anonymous philanthropic gift.

## REFERENCES

Martin L. Abbott. 2016. Correlation. In *Using Statistics in the Social and Health Sciences with SPSS and Excel.* Chapter 11, 329–370.

Rajeev Alur, Rastislav Alur, Garvit Juniwal, Milo M. K. Juniwal, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *Proceedings of the Thirteenth International Conference on Formal Methods in Computer-Aided Design (FMCAD).* IEEE, 1–8.

Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2017. DeepCoder: Learning to write programs. In *International Conference on Learning Representations (ICLR).*

José-Miguel Bernardo and Adrian Smith. 1994. *Bayesian Theory.* John Wiley & Sons, Inc.

Taylor L. Booth and Richard A. Thompson. 1973. Applying probability measures to abstract languages. *IEEE Trans. Comput.* 22, 5 (1973), 442–450.

Matko Bošnak, Pushmeet Kohli, Tejas Kulkrani, Sebastian Riedel, Tim Rocktäschel, Dawn Song, and Robert Zinkov (Eds.). 2018. *Neural Abstract Machines and Program Induction Workshop v2.*

George E. P. Box and Gwilym Jenkins. 1976. *Time Series Analysis: Forecasting and Control.* Holden-Day, Inc.

Richard M. Burstall and John Darlington. 1977. A transformation system for developing recursive programs. *J. ACM* 24, 1 (1977), 44–67.

Bob Carpenter, Andrew Gelman, Matthew Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. 2017. Stan: A probabilistic programming language. *Journal of Statistical Software* 76, 1 (2017), 1–32.

Sarah Chasins and Phitchaya M. Phothilimthana. 2017. Data-driven synthesis of full probabilistic programs. In *Proceedings of the Twenty-Ninth International Conference on Computer Aided Verification (CAV)*, Rupak Majumdar and Viktor Kunčak (Eds.), Vol. 10426. Springer, 279–304.

Ron P. Cody and Jeffrey K. Smith. 2005. *Applied Statistics and the SAS Programming Language* (5 ed.). Prentice-Hall.

Dua Dheeru and Efi Karra Taniskidou. 2017. UCI Machine Learning Repository. http://archive.ics.uci.edu/ml. (2017).

Norman R. Draper and Harry Smith. 1966. *Applied Regression Analysis.* John Wiley & Sons, Inc.

David Duvenaud, James Lloyd, Roger Grosse, Joshua B. Tenenbaum, and Zoubin Ghahramani. 2013. Structure discovery in nonparametric regression through compositional kernel search. In *Proceedings of the Thirtieth International Conference on Machine Learning (ICML)*, Sanjoy Dasgupta and David McAllester (Eds.), Vol. 28. PMLR, 1166–1174.

Kevin Ellis, Armando Solar-Lezama, and Josh Tenenbaum. 2015. Unsupervised learning by program synthesis. In *Advances in Neural Information Processing Systems 28 (NIPS)*, Corinna Cortes, Neil D. Lawrence, Daniel D. Lee, Masashi Sugiyama, and Roman Garnett (Eds.). Curran Associates, 973–981.

Kevin Ellis, Armando Solar-Lezama, and Josh Tenenbaum. 2016. Sampling for Bayesian program learning. In *Advances in Neural Information Processing Systems 29 (NIPS)*, Daniel D. Lee, Masashi Sugiyama, Ulrich von Luxburg, I. Guyon, and Roman Garnett (Eds.). Curran Associates, 1297–1305.

Mordecai Ezekiel. 1941. *Methods of Correlation Analysis.* John Wiley & Sons, Inc.

John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing data structure transformations from input-output examples. In *Proceedings of the Thirty-Sixth ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).* ACM, 229–239.

Hong Ge, Kai Xu, and Zoubin Ghahramani. 2018. Turing: A language for flexible probabilistic inference. In *Proceedings of the Twenty-First International Conference on Artificial Intelligence and Statistics (AISTATS)*, Amos Storkey and Fernando Perez-Cruz (Eds.), Vol. 84. PMLR, 1682–1690.

Roland Gecse and Attila Kovács. 2010. Consistency of stochastic context-free grammars. *Mathematical and Computer Modelling* 52, 3 (2010), 490–500.

Andrew Gelman and Jennifer Hill. 2007. *Data Analysis Using Regression and Multilevel/Hierarchical Models.* Cambridge University Press.

Noah Goodman, Vikash Mansinghka, Daniel Roy, Keith Bonawitz, and Joshua Tenenbaum. 2008. Church: A language for generative models. In *Proceedings of the Twenty-Fourth Conference Annual Conference on Uncertainty in Artificial Intelligence (UAI).* AUAI Press, 220–229.

Noah D. Goodman and Andreas Stuhlmüller. 2014. The Design and Implementation of Probabilistic Programming Languages. http://dippl.org. (2014).

Alex Graves, Greg Wayne, and Ivo Danihelka. 2014. Neural Turing machines. (2014). arXiv:1410.5401

Roger Grosse, Ruslan Salakhutdinov, William Freeman, and Joshua B. Tenenbaum. 2012. Exploiting compositionality to explore a large space of model structures. In *Proceedings of the Twenty-Eighth Annual Conference on Uncertainty in Artificial Intelligence (UAI).* AUAI Press, 306–31.

Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the Thirty-Eighth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL).* ACM, 317–330.

Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. 2011. Synthesis of loop-free programs. In *Proceedings of the Thirty-Second ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).* ACM, 62–73.

Irvin Hwang, Andreas Stuhlmüller, and Noah D. Goodman. 2011. Inducing probabilistic programs by Bayesian program merging. (2011). arXiv:1110.5667

Rob Hyndman and Yeasmin Khandakar. 2008. Automatic time series forecasting: The forecast package for R. *Journal of Statistical Software* 27, 3 (2008), 1–22.

Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. 2013. *An Introduction to Statistical Learning: with Applications in R.* Springer.

Frederick Jelinek, John D. Lafferty, and Robert L. Mercer. 1992. Basic methods of probabilistic context free grammars. In *Speech Recognition and Understanding (NATO ASI Series, Sub-Series F: Computer and Systems Sciences)*, Pietro Laface and Renato De Mori (Eds.), Vol. 75. Springer, 345–360.

Sumit Jha, Sumit Gulwani, Sanjit A. Jha, and Ashish Tiwari. 2010. Oracle-guided component-based program synthesis. In *Proceedings of the Thirty-Second ACM/IEEE International Conference on Software Engineering (ICSE)*, Vol. 1. ACM, 215–224.

Mark Johnson, Thomas L. Griffiths, and Sharon Goldwater. 2007. Adaptor grammars: A framework for specifying compositional nonparametric Bayesian models. In *Advances in Neural Information Processing Systems 19 (NIPS)*, Bernard Schölkopf, John C. Platt, and Thomas Hoffman (Eds.). Curran Associates, 641–648.

Matthew J. Johnson and Alan S. Willsky. 2013. Bayesian nonparametric hidden semi-Markov models. *Journal of Machine Learning Research* 14, Feb (2013), 673–701.

John R. Koza. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press.

John R. Koza, Forest H. Bennett III, Andre David, Martin A. Keane, and Frank Dunlap. 1997. Automated synthesis of analog electrical circuits by means of genetic programming. *IEEE Transactions on Evolutionary Computation* 1, 2 (1997), 109–128.

Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. 2018. Accelerating search-based program synthesis using learned probabilistic models. In *Proceedings of the Thirty-Ninth ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).* ACM, 436–449.

Percy Liang, Michael I. Jordan, and Dan Klein. 2010. Learning programs: A hierarchical Bayesian approach. In *Proceedings of the Twenty-Seventh International Conference on Machine Learning (ICML)*, Johannes Fürnkranz and Thorsten Joachims (Eds.). Omnipress, 639–646.

James R. Lloyd. 2014. Kernel structure discovery research code. https://github.com/jamesrobertlloyd/gpss-research/tree/master/data. (2014).

James R. Lloyd, David Duvenaud, Roger Grosse, Joshua B. Tenenbaum, and Zoubin Ghahramani. 2014. Automatic construction and natural-language description of nonparametric regression models. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence (AAAI).* AAAI Publications.

Zohar Manna and Richard Waldinger. 1979. Synthesis: Dreams to programs. *IEEE Transactions on Software Engineering* 5, 4 (1979), 294–328.

Zohar Manna and Richard Waldinger. 1980. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems* 2, 1 (1980), 90–121.

Vikash Mansinghka, Charles Kemp, Thomas Griffiths, and Joshua B. Tenenbaum. 2006. Structured priors for structure learning. In *Proceedings of the Twenty-Second Conference Annual Conference on Uncertainty in Artificial Intelligence (UAI).* AUAI Press, 324–331.

Vikash Mansinghka, Daniel Selsam, and Yura Perov. 2014. Venture: A higher-order probabilistic programming platform with programmable inference. (2014). arXiv:1404.0099

Vikash Mansinghka, Patrick Shafto, Eric Jonas, Cap Petschulat, Max Gasner, and Joshua B. Tenenbaum. 2016. CrossCat: A fully Bayesian nonparametric method for analyzing heterogeneous, high dimensional data. *Journal of Machine Learning Research* 17, 138 (2016), 1–49.

Vikash K. Mansinghka, Ulrich Schaechtle, Shivam Handa, Alexey Radul, Yutian Chen, and Martin Rinard. 2018. Probabilistic programming with programmable inference. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).* ACM, 603–616.

Andrew McCallum, Karl Schultz, and Sameer Singh. 2009. FACTORIE: Probabilistic programming via imperatively defined factor graphs. In *Advances in Neural Information Processing Systems 22 (NIPS)*, Yoshua Bengio, Dale Schuurmans, John D. Lafferty, Christopher K. Williams, and Aron Culotta (Eds.). Curran Associates, 1249–1257.

Brian Milch, Bhaskara Marthi, Stuart Russell, David Sontag, Daniel L. Ong, and Andrey Kolobov. 2005. BLOG: Probabilistic models with unknown objects. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI).* Morgan Kaufmann Publishers Inc., 1352–1359.

Kevin P. Murphy. 2012. *Machine Learning: A Probabilistic Perspective.* MIT Press.

Norman H. Nie. 1975. *SPSS: Statistical Package for the Social Sciences.* McGraw-Hill.

Antinus Nijholt. 1980. *Context-Free Grammars: Covers, Normal Forms, and Parsing.* Springer.

Aditya V. Nori, Sherjil Ozair, Sriram K. Rajamani, and Deepak Vijaykeerthy. 2015. Efficient synthesis of probabilistic programs. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 208–217.

Fabian Pedregosa, Gael Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Edouard Duchesnay. 2011. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research* 12, Oct (2011), 2825–2830.

Yura N. Perov and Frank D. Wood. 2014. Learning probabilistic programs. (2014). arXiv:1407.2646

Avi Pfeffer. 2001. IBAL: A probabilistic rational programming language. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI)*, Vol. 1. Morgan Kaufmann Publishers Inc., 733–740.

Avi Pfeffer. 2016. *Practical Probabilistic Programming* (1 ed.). Manning Publications Co.

Martyn Plummer. 2003. JAGS: A program for analysis of Bayesian graphical models using Gibbs sampling. In *Proceedings of the Third International Workshop on Distributed Statistical Computing (DSC)*, Kurt Hornik, Friedrich Leisch, and Achim Zeileis (Eds.). Austrian Association for Statistical Computing.

Joaquin Quiñonero-Candela and Carl E. Rasmussen. 2005. A unifying view of sparse approximate Gaussian process regression. *Journal of Machine Learning Research* 6, Dec (2005), 1939–1959.

Jeff Racine and Qi Li. 2004. Nonparametric estimation of regression functions with both categorical and continuous data. *Journal of Econometrics* 119, 1 (2004), 99–130.

Carl Rasmussen and Christopher Williams. 2006. *Gaussian Processes for Machine Learning*. The MIT Press.

Carl E. Rasmussen and Hannes Nickisch. 2010. Gaussian processes for machine learning (GPML) toolbox. *Journal of Machine Learning Research* 11, Nov (2010), 3011–3015.

Scott Reed and Nando de Freitas. 2016. Neural programmer-interpreters. In *International Conference on Learning Representations (ICLR)*.

Feras Saad and Vikash Mansinghka. 2016. Probabilistic data analysis with probabilistic programming. (2016). arXiv:1608.05347

Feras Saad and Vikash Mansinghka. 2017. Detecting dependencies in sparse, multivariate databases using probabilistic programming and non-parametric Bayes. In *Artificial Intelligence and Statistics (AISTATS)*, Aarti Singh and Jerry Zhu (Eds.), Vol. 54. PMLR, 632–641.

Feras A. Saad and Vikash K. Mansinghka. 2018. Temporally-reweighted Chinese restaurant process mixtures for clustering, imputing, and forecasting multivariate time series. In *Proceedings of the Twenty-First Conference on Artificial Intelligence and Statistics (AISTATS)*, Amos Storkey and Fernando Perez-Cruz (Eds.), Vol. 84. PMLR, 755–764.

John Salvatier, Thomas V. Wiecki, and Christopher Fonnesbeck. 2016. Probabilistic programming in python using PyMC3. *PeerJ Computer Science* 2 (2016), e55.

Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic superoptimization. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 305–316.

Moses Schönfinkel. 1924. Über die Bausteine der mathematischen Logik. *Math. Ann.* 92 (1924), 305–316.

David W. Scott. 2009. *Multivariate Density Estimation: Theory, Practice, and Visualization.* John Wiley & Sons, Inc.

Skipper Seabold and Josef Perktold. 2010. Statsmodels: Econometric and statistical modeling with python. In *Proceedings of the Ninth Python in Science Conference (SciPy)*, Stêfan van der Walt and Jarrod Millman (Eds.). 57–61.

Bernard W. Silverman. 1986. *Density Estimation for Statistics and Data Analysis.* CRC Press.

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial sketching for finite programs. In *Proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 404–415.

David Spiegelhalter, Andrew Thomas, Nicky Best, and Wally Gilks. 1996. *BUGS 0.5: Bayesian Inference Using Gibbs Sampling Manual (version ii).* MRC Biostatistics Unit, Institute of Public Health, Cambridge, United Kingdom.

Sean J. Taylor and Benjamin Letham. 2018. Forecasting at scale. *The American Statistician* 72, 1 (2018), 37–45.

Luke Tierney. 1994. Markov chains for exploring posterior distributions. *The Annals of Statistics* 22, 4 (1994), 1701–1728.

Anh Tong and Jaesik Choi. 2016. Automatic generation of probabilistic programming from time series data. (2016). arXiv:1607.00710

Dustin Tran, Matthew D. Hoffman, Rif A. Saurous, Eugene Brevdo, Kevin Murphy, and David M. Blei. 2017. Deep probabilistic programming. In *International Conference on Learning Representations (ICLR)*.

John W. Tukey. 1977. *Exploratory Data Analysis.* Addison-Wesley Publishing Company.

Franklyn Turbak, David Gifford, and Mark A. Sheldon. 2008. *Design Concepts in Programming Languages.* MIT Press.

Frank Wood, Jan Willem van de Meent, and Vikash Mansinghka. 2014. A new approach to probabilistic programming inference. In *Proceedings of the Seventeenth International Conference on Artificial Intelligence and Statistics (AISTATS)*, Vol. 33. PMLR, 1024–1032.