Journal of Machine Learning Research () Submitted ;

Probabilistic Programming with Gaussian Process Memoization

Editor:

Abstract

This paper describes the *Gaussian process memoizer*, a probabilistic programming technique that uses Gaussian processes to provide a statistical alternative to memorization. Memoizing a target procedure results in a self-caching wrapper that remembers previously computed values. Gaussian process memoization additionally produces a statistical emulator based on a Gaussian process whose predictions automatically improve whenever a new value of the target procedure becomes available. This paper also introduces an efficient implementation, named `gpmem`, that can use kernels given by a broad class of probabilistic programs. The flexibility of `gpmem` is illustrated via three applications: (i) GP regression with hierarchical hyper-parameter learning, (ii) Bayesian structure learning via compositional kernels generated by a probabilistic grammar, and (iii) a bandit formulation of Bayesian optimization with automatic inference and action selection. All applications share a single 50-line Python library and require fewer than 20 lines of probabilistic code each.

1. Introduction

Probabilistic programming could be revolutionary for machine intelligence due to universal inference engines and the rapid prototyping for novel models (Ghahramani, 2015). This propels the design and testing of new models as well as the incorporation of complex prior knowledge which currently is a difficult and time consuming task. Probabilistic programming languages aim to provide a formal language to specify probabilistic models in the style of computer programming and can represent any computable probability distribution as a program. In this work, we will introduce new features of Venture, a recently developed probabilistic programming language. We consider Venture the most compelling of the probabilistic programming languages because it is the first probabilistic programming language suitable for general purpose use (Mansinghka et al., 2014). Venture comes with scalable performance on hard problems and with a general purpose inference engine. The inference engine deploys Markov Chain Monte Carlo (MCMC) methods (for an introduction, see Andrieu et al., 2003). MCMC lends itself to models with complex structures such as probabilistic programs or hierarchical Bayesian non-parametric models since they can provide a vehicle to express otherwise intractable integrals necessary for a fully Bayesian representation. MCMC is scalable, often distributable and also compositional. That is, one can arbitrarily chain MCMC kernels to infer over several hierarchically connected or nested models as they will emerge in probabilistic programming.

One powerful model rarely treated in probabilistic programming languages are Gaussian Processes (GPs). GPs are gaining increasing attention for representing unknown functions

by posterior probability distributions in various fields such as machine learning, signal processing, computer vision and bio-medical data analysis. Making GPs available in probabilistic programming is crucial to allow a language to solve a wide range of problems. Hard problems include but are not limited to hierarchical prior construction (Neal, 1997), Bayesian Optimization (Snoek et al., 2012) and systems for inductive learning of symbolic expressions such as the one introduced in the Automated Statistician project (Duvenaud et al., 2013; Lloyd et al., 2014). Learning such symbolic expressions is a hard problem that requires careful design of approximation techniques since standard inference method do not apply.

In the following, we will present GP memoization (`gpmem`) as a novel probabilistic programming technique that solves such hard problems. `gpmem` introduces a statistical alternative to standard memoization. Our contribution is threefold:

- we introduce an efficient implementation of `gpmem` in form of a self-caching wrapper that remembers previously computed values;
- we illustrate the statistical emulator that `gpmem` produces and how it improves with every data-point that becomes available; and
- we show how one can solve hard problems of state-of-the-art machine learning related to GP using `gpmem` in a Bayesian fashion and with only a few lines of Venture code.

We evaluate the contribution on problems posed by the GP community using real world and synthetic data by assessing quality in terms of posterior distributions of symbolic outcome and in terms of the residuals produced by our probabilistic programs. The paper is structured as follows, we will first provide some background on memoization. We will explain programming in Venture and provide a brief introduction to GPs. We introduce `gpmem` and its use in probabilistic programming and Bayesian modeling. Finally, we will show how we can apply `gpmem` on problems of causally structured hierarchical priors for hyper-parameter inference, structure discovery for Gaussian Processes and Bayesian Optimization including experiments with real world and synthetic data.

1.1 Background on Probabilistic Programming

A *probabilistic programming language* is a language for defining probabilistic generative models and inference on those models. For us, a *model program* means an executable description of a generative model. An *inference program* is an executable description of an inference algorithm, which may also make use of randomness. In this paper, our examples will use the Venture probabilistic programming language (Mansinghka et al., 2014), but the concepts illustrated apply to probabilistic programming in general. Notably, in Venture, the sample space for all random variables occurring in a model is the set of all possible execution traces of the model program (see (Mansinghka et al., 2014, §4) and Section 1.2). In fact, we take this as a definition: for a probabilistic program written in Venture, we define the *model program* to consist of all instructions (synonymously, *directives*) which enter the execution trace, and the *inference program* to consist of all directives, including untraced directives which manipulate the existing trace.

The range of models describable as probabilistic programs is very broad, containing graphical models (including plate notation) as a special case. There is a precise sense in

which any computable generative model can be expressed as a probabilistic program (Ackerman et al., 2010). The power of the modelling language comes not from executability of models—in other words, not from the ability to perform *forward-sampling*—but from the rich additional layers of description which allow automatic reasoning (such as automatic dependency tracking) and semi-automatic reasoning (such as exploratory inference programming) about models.

The range of inference algorithms describable as probabilistic programs is also very broad. General-purpose versions of rejection sampling, Gibbs sampling, Metropolis–Hastings, mean-field, and slice sampling are built into Venture as primitives (Mansinghka et al., 2014). Custom inference algorithms can be written as programs that manipulate traces: below we implement Thompson sampling as a custom inference program.

Listing 1 shows a simple probabilistic program containing only modelling instructions:

Listing 1: A simple probabilistic program (written in Venture) containing only modelling instructions.

```
assume rain = bernoulli( 0.3)
assume sprinkler = if( rain) { false }
                   else      {bernoulli( 0.8) }
assume grass_wet = bernoulli( if( rain) { if( sprinkler) { 0.99 }
                                          else           { 0.95 } }
                              else {      if( sprinkler) { 0.6 }
                                          else         { 0.1 } } )
assume humidity = if( rain) { normal( 95 2)}
                  else { normal( 50 20) }
```

This program reads as follows: It is raining (R) with probability 0.3. If it is raining, then the sprinkler (S) is never on; if it is not raining, then the sprinkler is on with probability 0.8. The probability that the grass is wet (G) depends on both whether it is raining and whether the sprinkler is on, according to the table

	R	¬R
S	0.99	0.6
¬S	0.95	0.1

Finally, the humidity is a continuous random variable whose parameters depend on whether it is raining: $\mathcal{N}(95, 2)$ (normal with mean 95 and variance 2) if it is raining, and $\mathcal{N}(50, 20)$ if it is not raining.

The program in Listing 1 induces the following directed graphical model:

Not all model programs can be described by a graphical model: due to branching and recursion, the set of random variables may vary from execution to execution, as may the dependency structure of even a fixed set of variables. In fact, the class of computable distributions that can be described by a graphical model is precisely the class of models

that can be written as a probabilistic program with fixed control flow. Note that the program in Listing 1, while it does not have fixed control flow as written, has a finite number of possible control paths and thus can be rewritten with fixed control flow by table lookup into an enumeration of all possible paths. There are other cases in which the class of models expressible in an existing framework can be seen as those arising from a restricted class of probabilistic programs; one such case is plate notation in graphical models, which corresponds to probabilistic programs with fixed control flow which employ memoization via `mem` (see (Goodman et al., 2008, §2.1)) and treat the value of the memoized procedure at each input as a random variable. However, the full class of models expressible as probabilistic programs does not correspond to an existing notational framework. This is not surprising: we would expect that a computationally universal class of models requires a computational language to describe.[1]

Suppose the value of a random variable is observed—say, $\mathrm{H} = 85$—and we wish to infer values of other random variables. Rejection sampling of full executions of the model program, which in this context we call *global rejection sampling*, produces joint samples from the posterior distribution $P(\mathrm{R}, \mathrm{S}, \mathrm{G} \mid \mathrm{H} = 85)$; it is achieved in Venture by the directive

```
infer rejection( default, all, 1)
```

Global rejection sampling is essentially the only inference program that can be employed on a black-box model program with no accompanying source code or inspection infrastructure. Our model, however, is not a black box; due to the richness of the description language, a wide range of inference programs are readily implementable. Rejection sampling can be performed on one random variable at a time, conditioned on the current values of the other variables, resulting in a Markov chain Monte Carlo (MCMC) inference routine whose state is a random tuple $(\mathrm{R}, \mathrm{S}, \mathrm{G})$ whose distribution, after sufficiently many transitions, is approximately the posterior. Note that inference on one variable at a time allows the programmer to choose the order in which variables are inferred, or to choose not to infer the values of variables on which the variable of interest does not depend. This need not be done manually: Venture dynamically tracks dependencies between random variables, which, for example, allows one to run Metropolis–Hastings inference on a single random variable X in which proposals only modify the values of those variables that depend on X:

```
infer mh( default, one, 1)
```

(Here the keywords `default one` mean that X is chosen randomly from the set of random variables in the model, but it is also possible to specify an explicit choice of X by supplying different arguments to `mh`.) While standard inference routines such as rejection sampling and Metropolis–Hastings are built into Venture, inference in general can be (and in Venture, is) fully programmable, allowing for composite inference strategies which use different inference subroutines on different parts of the model at different times, as well as dynamic choices

1. The language of probabilistic Turing machines is also "universal" in the sense that it could describe a program for forward-sampling at the machine level. But such a description lacks many of the useful characteristics of a probabilistic program: semantics for what the random variables are and what they depend on, and abstraction and modularity to allow for local changes to the model program.

of inference strategy which depend on state accrued during the execution of the model program and inference thereupon.

1.2 Traces and Interactivity

The output Y of a generative model can be thought of as the projection of a random variable J to just the non-latent parts. That is,

$$Y : \omega \in \Omega \overset{J}{\mapsto} (x, y) \overset{\text{proj}}{\longmapsto} y,$$

where Ω is the sample space; x is referred to as the "latent" data for this sample. If our generative model J is computable, and is embodied by the model program `J`, then as we mentioned in Section 1.1, without loss of generality we can take Ω to simply be the set of all possible executions of `J`, as is done in Venture.

Just as a generative model J can be extended by making additional random choices $J' \sim P(J'|J)$ conditioned on the value of J, an execution trace can be extended by executing more model directives. Consequently, the state of a model program can be inspected interactively, without making a commitment as to whether there is more model code to be executed later. In Venture, the current trace can be extended using the `predict` directive (or `assume`, which does the same thing and then assigns the resulting value to a named variable; see **?** and **?** for a detailed treatment of all the Venture directives). The hypothetical result of evaluating an expression `e` inside the model can be queried effectlessly using `sample`, which is equivalent to `predict` followed by `forget`: an evaluation of `e` is added to the current trace and then immediately deleted, and the value is returned. `predict` and `sample` are not interchangeable; that is to say, the meaning of a program (and the distribution of its possible traces) can depend on which evaluations are added to the trace and which are forgotten. We will see this below in Section **??**.

1.3 Memoization

Memoization is the practice of storing previously computed values of a function so that future calls with the same inputs can be evaluated by lookup rather than recomputation. Research on the Church language (Goodman et al., 2008) pointed out that although memoization does not change the semantics of a determinstic program, it does change that of a stochastic program. The authors provide an intuitive example: let f be a function that flips a coin and return "head" or "tails". The probability that two calls of f are equivalent is 0.5. However, if the function call is memoized, it is 1.

In fact, there is an infinite range of possible caching policies (specifications of when to use a stored value and when to recompute), each potentially having a different semantics. Any particular caching policy can be understood by random world semantics (Poole, 1993; Sato, 1995) over the stochastic program: each possible world corresponds to a mapping from function input sequence to function output sequence (McAllester et al., 2008). In Venture, these possible worlds are first-class objects, known as *traces* (Mansinghka et al., 2014).

1.4 Venture

Venture is a probabilistic programming language that allows both, the design of custom inference algorithms and arbitraly nested and hierarchical models. It comes with two different front end syntaxes, one Scheme-like and one JavaScript-like.

Venture provides a powerful and concise way to represent probability distributions, including distributions with a dynamically determined and unbounded set of random variables. In this paperwe will use only the four basic Venture directives: `assume`, `observe`, `sample` and `infer` (Mansinghka et al., 2014).

- `assume` induces a hypothesis space for (probabilistic) models including random variables by binding the result of a supplied expression to a supplied symbol.
- `sample` samples the value of the supplied expression within the current model program. Such an expression is evaluated within a trace of the model program. Thus, the value of an expression within a model program is a random variable, whose randomness comes from the distribution on possible execution traces of the program. The
- `observe` constrains the supplied expression to have the supplied value. In other words, all samples taken after an `observe` are conditioned on the observed data.
- `infer` uses the supplied inference program changes the modeled probability distribution. This will result approximate sampling from the true posterior, conditioned on the model and constraints introduced by `assume` and `observe`. The posterior on any random variable can then be approximately sampled by calling `sample` to extract values from the modeled distribution.

`infer` is commonly done using the Metropolis–Hastings algorithm (MH) (Metropolis et al., 1953). Many of the most popular MCMC algorithms can be interpreted as special cases of MH (Andrieu et al., 2003). We can outline the MH algorithm as follows. The following two-step process is repeated as long as desired (say, for T iterations): First we sample x^* from a proposal distribution q:

$$x^* \sim q(x^* \mid x^t); \tag{1}$$

then we accept this proposal ($x^{t+1} \leftarrow x^*$) with probability

$$\alpha = \min\left\{1, \frac{p(x^*)q(x^t \mid x^*)}{p(x^t)q(x^* \mid x^t)}\right\}; \tag{2}$$

if the proposal is not accepted then we take $x^{t+1} \leftarrow x^t$.

Venture includes a built-in generic MH inference program which performs the above steps on any specified set of random variables in the model program. In that inference program, partial execution traces play the role of x above.

2. Gaussian Processes Memoization

We now introduce GP related theory and notations. We work exclusively with two-variable regression problems. Let the data be pairs of real-valued scalars $\{(x_i, y_i)\}_{i=1}^n$ (complete

data will be denoted by column vectors $\mathbf{x}$, $\mathbf{y}$). In regression, one tries to learn a functional relationship $y_i = f(x_i)$, where the function f is to be learned. GPs present a non-parametric way to express prior knowledge on the space of possible functions f. Formally, a GP is an infinite-dimensional extension of the multivariate Gaussian distribution. For any finite set of inputs $\mathbf{x}$, the marginal prior on $f(\mathbf{x})$ is the multivariate Gaussian

$$f(\mathbf{x}) \sim \mathcal{N}(m(\mathbf{x}), k(\mathbf{x}, \mathbf{x})),$$

where $m(\mathbf{x}) = \mathbb{E}_f[f(\mathbf{x})]$ is the mean function and $k(\mathbf{x}, \mathbf{x}') = \mathrm{Cov}_f(f(\mathbf{x}), f(\mathbf{x}'))$ is the covariance function, a.k.a. kernel.[2] Together m and k characterize the distribution of f; we write

$$f \sim \mathcal{GP}(m, k).$$

In all examples below, our prior mean function m is identically zero; this is the most common choice. The marginal likelihood can be expressed as:

$$p(f(\mathbf{x}) = \mathbf{y} \mid \mathbf{x}) = \int p(f(\mathbf{x}) = \mathbf{y} \mid f, \mathbf{x})\; p(f|\mathbf{x})\, df \tag{3}$$

where here $p(f|\mathbf{x}) = p(f) \sim \mathcal{GP}(m, k)$ since we assume no dependence of f on $\mathbf{x}$. We can sample a vector of unseen data $\mathbf{y}^* = f(\mathbf{x}^*)$ from the predictive posterior with

$$\mathbf{y}^* \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma}), \tag{4}$$

a multivariate normal with mean vector

$$\boldsymbol{\mu} = k(\mathbf{x}^*, \mathbf{x})\, k(\mathbf{x}, \mathbf{x})^{-1}\, \mathbf{y} \tag{5}$$

and covariance matrix

$$\boldsymbol{\Sigma} = k(\mathbf{x}^*, \mathbf{x}^*) - k(\mathbf{x}^*, \mathbf{x}) k(\mathbf{x}, \mathbf{x})^{-1} k(\mathbf{x}, \mathbf{x}^*). \tag{6}$$

Often one assumes the values $\mathbf{y}$ are noisily measured, that is, one only sees the values of $\mathbf{y}_{\text{noisy}} = \mathbf{y} + \mathbf{w}$ where $\mathbf{w}$ is Gaussian white noise with variance σ^2_{noise}. In that case, the log-likelihood is

$$\log p(\mathbf{y}_{\text{noisy}} \mid \mathbf{x}) = -\frac{1}{2}\mathbf{y}^\top (\boldsymbol{\Sigma} + \sigma^2_{\text{noise}}\mathbf{I})^{-1}\mathbf{y} - \frac{1}{2}\log\left|\boldsymbol{\Sigma} + \sigma^2_{\text{noise}}\mathbf{I}\right| - \frac{n}{2}\log 2\pi \tag{7}$$

where n is the number of data points. Both log-likelihood and predictive posterior can be computed efficiently in a Venture SP with an algorithm that resorts to Cholesky factorization(Rasmussen and Williams, 2006, chap. 2) resulting in a computational complexity of $\mathcal{O}(n^3)$ in the number of data points.

The covariance function governs high-level properties of the observed data such as linearity, periodicity and smoothness. The most widely used form of covariance function is the squared exponential:

$$k(x, x') = \sigma^2 \exp\left(-\frac{(x - x')^2}{2\ell^2}\right), \tag{8}$$

2. Note that $m(\mathbf{x}) = (m(x_i))_{i=1}^n$ and $k(\mathbf{x}, \mathbf{x}') = \left(k(x_i, x'_{i'})\right)_{1 \le i' \le n'}^{1 \le i \le n}$, where n' is the number of entries in $\mathbf{x}'$.

where σ and ℓ are hyperparameters: σ is a scaling factor and ℓ is the typical length-scale.

Adjusting hyperparameters results in a new covariance function with the same qualitative human-interpretation; more drastically different covariance functions are achieved by changing the structure of the covariance function. Note that covariance function structures are compositional: adding or multiplying two valid covariance functions results in another valid covariance function.

It has been suggested that adding covariance structures k_1, k_2 together,

$$k_3(x, x') = k_1(x, x') + k_2(x, x'), \tag{9}$$

corresponds to combining global structures, while multiplying covariance functions,

$$k_4(x, x') = k_1(x, x') \times k_2(x, x'), \tag{10}$$

corresponds to combining local structures (Duvenaud et al., 2013). Note that both k_3 and k_4 are valid covariance function structures.

Venture includes the primitive `make_gp`, which takes as arguments a unary function `mean` and a binary (symmetric, positive-semidefinite) function `cov` and produces a function `g` distributed as a Gaussian process with the supplied mean and covariance. For example, a function $\mathtt{g} \sim \mathcal{GP}(0, \mathrm{SE})$, where SE is a squared-exponential covariance

$$\mathrm{SE}(x, x') = \sigma^2 \exp\left(\frac{(x - x')^2}{2\ell}\right)$$

with $\sigma = 1$ and $\ell = 1$, can be instantiated as follows:

```
assume zero = make_const_func( 0.0)
assume se = make_squaredexp( 1.0, 1.0)
assume g  = make_gp( zero, se)
```

There are two ways two view `g` as a "random function." In the first view, the `assume` directive that instantiates `g` does not use any randomness—only the subsequent calls to `g` do—and coherence constraints are upheld by the interpreter by keeping track of which evaluations of `g` exist in the current trace. Namely, if the current trace contains evaluations of `g` at the points $x_1, \ldots, x_N$ with return values $y_1, \ldots, y_N$, then the next evaluation of `g` (say, jointly at the points $x_{N+1}, \ldots, x_{N+n}$) will be distributed according to the joint conditional distribution

$$P\left((\mathtt{g}\ x_{N+1}), \ldots, (\mathtt{g}\ x_{N+n}) \;\middle|\; (\mathtt{g}\ x_i) = y_i \text{ for } i = 1, \ldots, N\right).$$

In the second view, `g` is a randomly chosen deterministic function, chosen from the space of all deterministic real-valued functions; in this view, the `assume` directive contains *all* the randomness, and subsequent invocations of `g` are deterministic. The first view is procedural and is faithful to the computation that occurs behind the scenes in Venture. The second view is declarative and is faithful to notations like "$g \sim P(g)$" which are often used in mathematical treatments. Because a model program could make arbitrarily many calls to `g`, and the joint distribution on the return values of the calls could have arbitrarily high entropy,

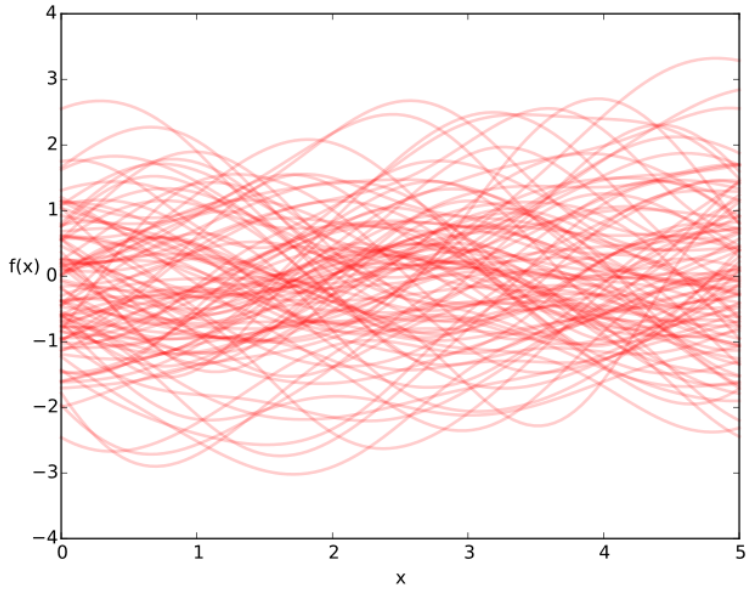
it is not computationally possible in finite time to choose the entire function g all at once as in the second view. Thus, it stands to reason that any computationally implementable notion of "nonparametric random functions" must involve incremental random choices in one way or another, and Gaussian processes in Venture are no exception.

Behind the scenes, Venture attaches a state $D = (\mathbf{x}_{\text{past}}, \mathbf{y}_{\text{past}})$ to each procedure created by `make_gp`, where $\mathbf{x}_{\text{past}}$ is a vector of all past inputs to g in the current trace, and $\mathbf{y}_{\text{past}}$ is the corresponding vector of outputs. (The state D is updated each time an invocation of g is added to or removed from the current trace.) Thus, the incremental random choices made by a GP g in Venture are simply samples from the conditional distribution $P(\mathrm{g} \mid D)$. That is, if g was initialized as `make_gp mean cov)` and has accrued state $D = (\mathbf{x}_{\text{past}}, \mathbf{y}_{\text{past}})$, then

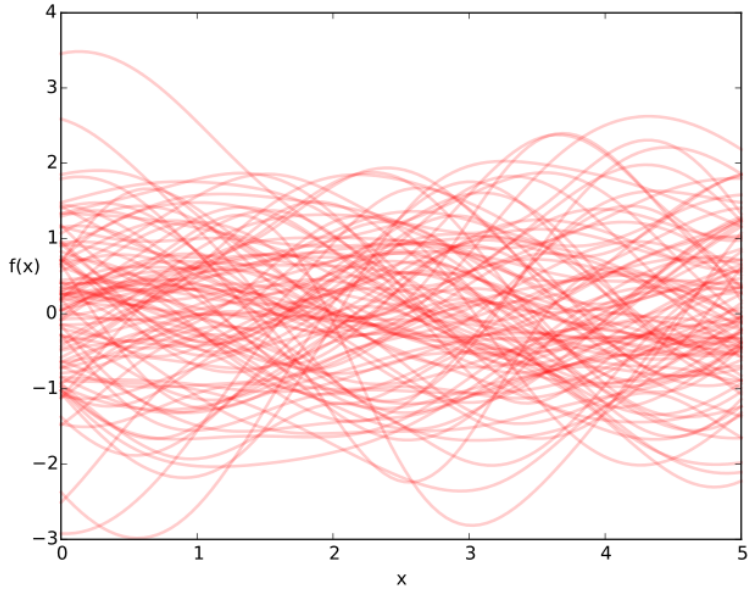
$$\texttt{(g x)} \sim P\left(\texttt{(}\mathrm{g}_0\ \mathbf{x}\texttt{)} \mid \texttt{(}\mathrm{g}_0\ \mathbf{x}_{\text{past}}\texttt{)} = \mathbf{y}_{\text{past}}\right), \qquad \text{where } \mathrm{g}_0 \sim \texttt{make\_gp mean cov)}.$$

Statefully represented random functions highlight the difference between `predict` and `sample` (see Section 1.2). To illustrate, consider the following program:

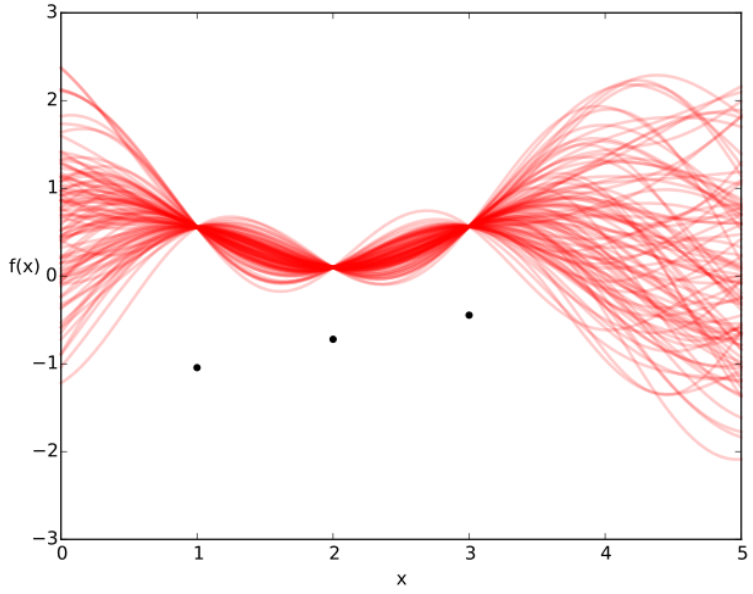
```
assume zero = make_const_func( 0.0)
assume se =  make_squaredexp( 1.0, 1.0)
assume g = make_gp( zero, se)

call_back( draw_gp_curves( g( ...)))
```




```
sample( g( array(1 2 3)))
\\ [-1.04331626 -0.72016875 -0.44525212]
call_back( draw_gp_curves( g( ...)))
```




```
predict( g( array(1 2 3)))
\\ [ 0.55736593  0.09784888  0.56553757]
call_back( draw_gp_curves( g( ...)))
```



(The callback `draw_gp_curves` draws a cloud of curves sampled from `g`.) The `sample` directive does not affect the state of `g`, so the distribution of `g` after the `sample` is the same as the prior. The `predict`, however, does affect the state; the distribution of `g` after the `predict` directive is conditioned on the values returned by `g` inside the directive. This explains why, in the final plot, all curves go through the points returned by the previous `predict` directive, and do not go through the points returned by the `sample` directive (which are marked with black dots for reference).

2.1 A Bayesian interpretation

We illustrate and compare Bayesian and frequentist view points on GP with a simple example (Fig. 1). We show how in a simple model, two outliers can bias a maximum a posteriori inference. The data where generated with:

$$y = 2x + 15 \tag{11}$$

and outliers are generated with a parallel line:

$$\hat{y} = 2x + 40. \tag{12}$$

We add some small amount of white noise. We generate eight data points with (11) and two with (12). Since we suspect the underlying data generating mechanism to be linear, we fit a linear kernel with a constant covariance as intercept and some white noise:

$$\mathbf{K} = \mathrm{LIN} + \mathrm{C} + \mathrm{WN}. \tag{13}$$

where we upper case matrix notation denotes the covariance matrix of the complete training data. Omitting the scaling parameter for the linear kernel, there are two hyperparameters to learn, that is the noise variance and the hyper-parameter for the constant function. Maximum a posteriori (MAP) inference fits the single one best line and accounts for the outliers with a large noise scaling parameter. MH does better. It assigns a small amount of probability mass to a different scaling parameter and a larger constant. The resulting prediction (indicating with the predictive mean in figure 1) is closer to the true underlying function.

2.2 Gaussian Process Memoization

We now introduce a language construct for probabilistic programming called a *statistical memoizer*. In addition to providing an elegant linguistic framework for function learning–related tasks such as Bayesian optimization, the statistical memoizer highlights shortcomings of the notations currently used in statistics, and how language constructs from programming allow the expression of models which would be cumbersome (prohibitively so, in some cases) to express in statistics notation.

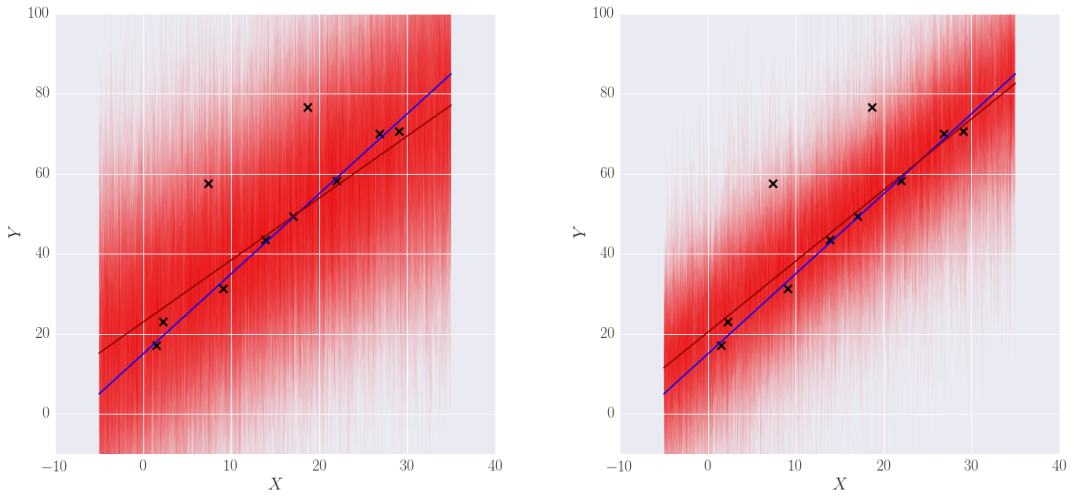
Suppose we have a function `f` which can be evaluated, but as in Section **??**, we wish to learn about the behavior of `f` using as few evaluations as possible. The statistical memoizer, which here we give the name `mem&em`, was motivated by this purpose. It produces two outputs:

$$\mathtt{f} \xrightarrow{\mathtt{mem\&em}} (\mathtt{f}_{\mathrm{probe}}, \mathtt{f}_{\mathrm{emu}}).$$

The function $\mathtt{f}_{\mathrm{probe}}$ calls `f` and stores the output in a memo table, just as traditional memoization does. The function $\mathtt{f}_{\mathrm{emu}}$ is an online statistical emulator which uses the memo table as its training data. A fully Bayesian emulator, modelling the true function `f` as a random function $f \sim P(f)$, would satisfy

$$(\mathtt{f}_{\mathrm{emu}}\ \mathtt{x}_1\ \ldots\ \mathtt{x}_k) \sim P\left(f(\mathtt{x}_1), \ldots, f(\mathtt{x}_k) \mid f(\mathtt{x}) = (\mathtt{f}\ \mathtt{x}) \text{ for each } \mathtt{x} \text{ in memo table}\right).$$

Non-Bayesian emulators are possible as well, but here we will focus on the Bayesian case. Different implementations of the statistical memoizer can have different prior distributions



(a) MAP (b) MH

Figure 1: (a) depicts MAP inference on the data, (b) depicts MH for hyperparameter inference. The blue line is the actual data generating function. Red are samples drawn from the posterior. The dark red line is the posterior predictive mean. We see that the MH shifts the posterior closer to the ground truth than MAP.

$P(f)$; our main example is a Gaussian process prior (implemented as `gpmem` below). Note that we require the ability to sample $\mathtt{f}_{\text{emu}}$ jointly at multiple inputs because the values of $f(\mathtt{x}_1), \ldots, f(\mathtt{x}_k)$ will in general be dependent.

To illustrate the linguistic power of `mem&em`, consider a simple model program that uses `mem&em`, procedure abstraction, and a loop:

```
1  define choose_next_point = proc(em) { gridsearch_argmax( em) }
2
3   // Here stats( em) is the memo table {(x_i, y_i)}.
4   // Take the (x,y) pair with the largest y.
5  define extract_answer = proc(em) { first(max( stats( em, second)))}
6
7  assume theta =  tag( quote( params) ⟨parameters for the statistical emulator⟩))
8  assume (f_probe f_emu) =  mem&em(f, theta))
9
10 for t...T:
11   f_probe( choose_next_point( f_emu))
12 extract_answer( f_emu)
```

This program is written in a Venture-flavored pseudocode language that we call "pseudo-Venture." Most of the code we present, including Figure **??** and Listings **??**, **??**, and **??**, will be in pseudo-Venture in order to simplify the presentation. The runnable Venture

code is essentially the same, except for some rough edges of syntax and other inessential complications; see Appendix **??** for more information.

An equivalent model in typical statistics notation, written in a way that attempts to be as linguistically faithful as possible, is

$$\begin{aligned} f^{(0)} &\sim P(f) \\ \mathtt{x}^{(t)} &= \arg\max_x f^{(t)}(x) \\ f^{(t+1)} &\sim P\left(f^{(t)} \;\middle|\; f^{(t)}(\mathtt{x}^{(t)}) = \mathtt{(f\ x^{(t)})}\right). \end{aligned}$$

We highlight two points:

1. The linguistic constructs for abstraction are not present in statistics notation. The equation $\mathtt{x}^{(t)} = \arg\max_x f^{(t)}(x)$ has to be inlined in statistics notation: while something like
$$\mathtt{x}^{(t)} \sim F(P^{(t)})$$
(where $P^{(t)}$ is a probability measure, and F here plays the role of `choose_next_point`) is mathematically coherent, it is not what statisticians would ordinarily write. This lack of abstraction then makes modularity, or the easy digestion of large but finely decomposable models, more difficult in statistics notation.

2. Adding a single line to the program, such as

```
infer mh( quote( params), one, 50)
```

after line 11 to infer the parameters of the emulator[3], would require a significant refactoring and elaboration of the statistics notation. One basic reason is that probabilistic programs are written procedurally, whereas statistical notation is declarative; reasoning declaratively about the dynamics of a fundamentally procedural inference algorithm is often unwieldy due to the absence of programming constructs such as loops and mutable state.

We implement this by memoizing a target procedure in a wrapper that remembers previously computed values. The technique gives rise to a number of use-cases of GPs in a fully Bayesian setting which are otherwise hard to implement.

From the standpoint of computation, a data set of the form $\{(x_i, y_i)\}$ can be thought of as a function $y = f_{\text{restr}}(x)$, where f_{restr} is restricted to only allow evaluation at a specific set of inputs x (Alg. 1).

Modelling the data set with a GP then amounts to trying to learn a smooth function f_{emu} ("emu" stands for "emulator") which extends f to its full domain. Indeed, if f_{restr} is a foreign procedure made available as a black-box to Venture, whose secret underlying pseudo code is in Alg. 1, then the `OBSERVE` code can be rewritten using `gpmem` as follows (where here the data set `D` has keys `x[1]`,...,`x[n]`):

3. The definition of these parameters depends on the particular implementation of `mem&em`, and could for example be the hyperparameters of the covariance function of a GP.

Algorithm 1 Restricted Function

1: **function** $f_{\text{RESTR}}(x)$
2: **if** $x \in \mathcal{D}$ **then**
3: **return** $\mathcal{D}[x]$
4: **else**
5: raise Exception('Illegal input')
6: **end if**
7: **end function**

Listing 2: Observation with `gpmem`

```
assume (f_probe f_emu) =  gpmem( f_restr)
for i=1 to n:
  predict f_compute( x[i])
  infer mh(quote(hyper-parameters), one, 100)
sample (f_emu( array( 1, 2, 3))
```

This rewriting has at least two benefits: (i) readability (in some cases), and (ii) amenability to active learning. As to (i), the statistical code of creating a Gaussian process is replaced with a memoization-like idiom, which will be more familiar to programmers. As to (ii), when using `gpmem`, it is quite easy to decide incrementally which data point to sample next: for example, the loop from `x[1]` to `x[n]` could be replaced by a loop in which the next index `i` is chosen by a supplied decision rule. In this way, we could use `gpmem` to perform online learning using only a subset of the available data.

3. Example Applications

3.1 Hyperparameter estimation with structured hyper-prior

The probability of the hyper-parameters of a GP as above defined above and given covariance function structure $\mathbf{K}$ can be described as:

$$P(\boldsymbol{\theta} \mid \mathbf{D}, \mathbf{K}) = \frac{P(\mathbf{D} \mid \boldsymbol{\theta}, \mathbf{K}) P(\boldsymbol{\theta} \mid \mathbf{K})}{P(\mathbf{D} \mid \mathbf{K})}. \tag{14}$$

Let the $\mathbf{K}$ be the sum of a smoothing and a white noise (WN) kernel. For this case, Neal suggested the problem of outliers in data as a use-case for a hierarchical Bayesian treatment of Gaussian processes (1997)[4]. The work suggests a hierarchical system of hyper-parameterization. Here, we draw hyper-parameters from a Γ distributions:

$$\ell^{(t)} \sim \Gamma(\alpha_1, \beta_1),\ \sigma^{(t)} \sim \Gamma(\alpha_2, \beta_2) \tag{15}$$

and in turn sample the α and β from Γ distributions as well:

$$\alpha_1^{(t)} \sim \Gamma(\alpha_\alpha^1, \beta_\alpha^1),\ \alpha_2^{(t)} \sim \Gamma(\alpha_\alpha^2, \beta_\alpha^2), \cdots \tag{16}$$

4. In (Neal, 1997) the sum of an SE plus a constant kernel is used. We keep the WN kernel for illustrative purposes.

Assuming the covariance structure is additive comprised of a smoothing and a white noise kernel, one can represent this kind of model using gpmem with only a few lines of code:

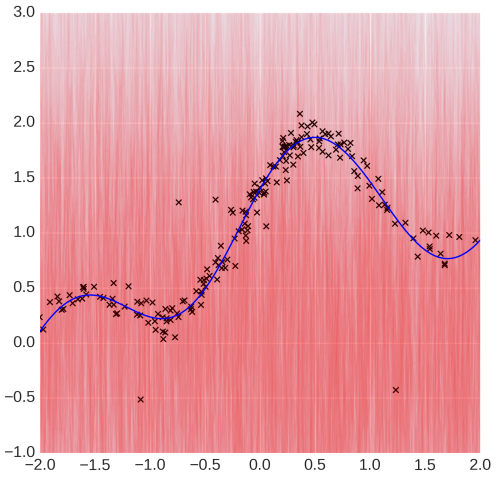
Listing 3: Hyper-Prior Learning

```
/// SETTING UP THE MODEL
assume alpha_sf = tag(quote(hyperhyper), 0, gamma(7, 1))
assume beta_sf = tag(quote(hyperhyper), 1, gamma(7, 1))
assume alpha_l = tag(quote(hyperhyper), 2, gamma(7, 1))
assume beta_l = tag(quote(hyperhyper), 3, gamma(7, 1))

// Parameters of the covariance function
assume sf = tag(quote(hyper), 0, gamma(alpha_sf, beta_sf)))
assume l = tag(quote(hyper), 1, gamma(alpha_l, beta_l)))
assume sigma = tag(quote(hyper), 2, uniform_continuous(0, 2))

// The covariance function
assume se = make_squaredexp(sf, l)
assume wn = make_whitenoise(sigma)
assume composite_covariance = add_funcs(se, wn)

/// PERFORMING INFERENCE
// Create a prober and emulator using gpmem
assume f_restr = get_neal_blackbox()
assume (f_compute, f_emu) = gpmem(f_restr, composite_covariance)

// Probe all data points
for n ... N
    predict f_compute(get_neal_data_xs(n))

// Infer hypers and hyperhypers
infer repeat(100, do(
    mh(quote(hyperhyper), one, 2),
    mh(quote(hyper), one, 1)))
```

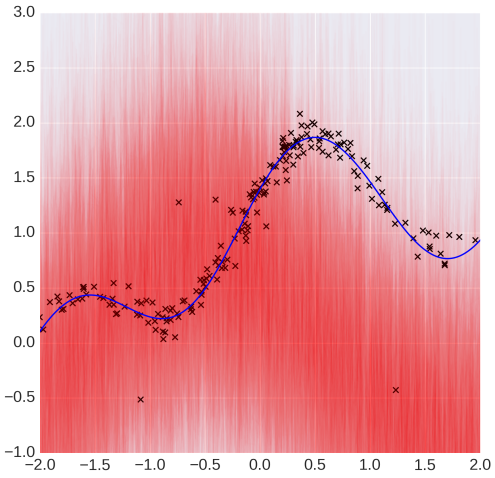
Neal provides a custom inference algorithm setting and evaluates it using the following synthetic data problem. Let f be the underlying function that generates the data:

$$f(x) = 0.3 + 0.4x + 0.5\sin(2.7x) + \frac{1.1}{(1+x^2)} + \eta \quad with \;\; \eta \sim \mathcal{N}(0, \sigma) \tag{17}$$

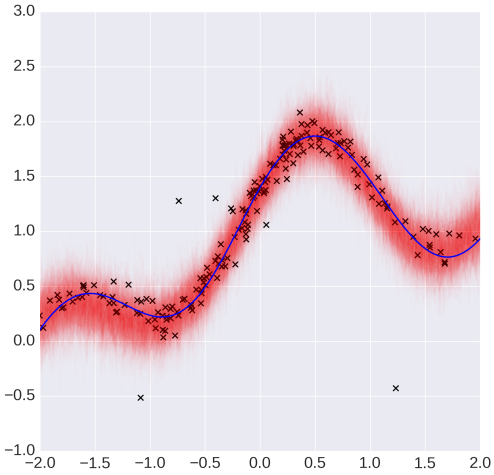
We synthetically generate outliers by setting $\sigma = 0.1$ in 95% of the cases and to $\sigma = 1$ in the remaining cases. gpmem can capture the true underlying function within only 100 MH steps on the hyper-parameters to get a good approximation for their posterior (see Fig. 2). Note that Neal devises an additional noise model and performs a large number of Hybrid-Monte Carlo and Gibbs steps. We illustrate the hyper-parameter by showing the shift of the distribution on the noise parameter σ (Fig. 3). We see that gpmem learns the



(a) Prior Inference

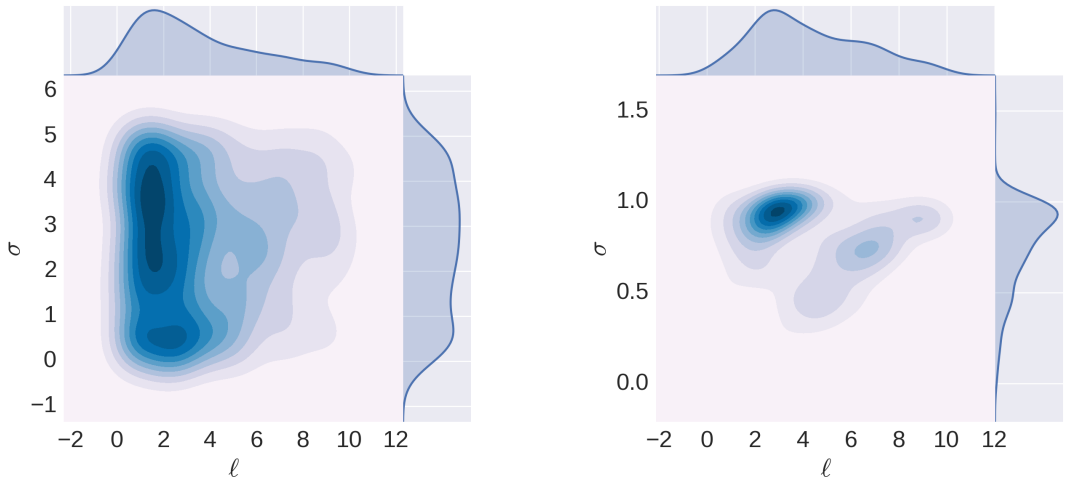


(b) Observed



(c) Inferred

Figure 2: (a)-(c) shows `gpmem` on Neal's example. We see that prior renders functions all over the place (a). After `gpmem` observes some data-points, an arbitrary smooth trend with a high level of noise is sampled. After running inference on the hierarchical system of hyper-parameters we see that the posterior reflects the actual curve well. Outliers are treated as such and do not confound the GP.



(a) Before (b) After

Figure 3: Hyper-parameter inference on the parameter of the noise kernel. We show a 100 samples drawn from the distribution on σ. One can clearly recognise the shift from the uniform prior $\mathcal{U}(0, 5)$ to a double peak distribution around the two modes - normal and outlier.

posterior distribution well, the posterior even exhibits a bimodal histogram when sampling σ 100 times reflecting the two modes of data generation, that is normal noise and outliers[5].

3.2 Discovering symbolic models for time series

The space of possible kernel composition is infinite. Combining inference over this space with the problem of finding a good parameterization that could potentially explain the observed data best poses a hard problem. The natural language interpretation of the meaning of a kernel and its composition renders this a problem of symbolic computation. Duvenaud and colleagues note that a sum of kernels can be interpreted as logical OR operations and kernel multiplication as logical AND (2013). This is due to the kernel rendering two points similar if k_1 OR k_2 outputs a high value in the case of a sum. Respectively, multiplication of two kernels results in high values only if k_1 AND k_2 have high values (see Fig. 4 exemplifies how to interpret global vs. local aspects and its symbolic analog respectively). In the following, we will refer to covariance functions that are not composite as base covariance functions.

Knowledge about the composite nature of covariance functions is not new, however, until recently, the choice and the composition of covariance functions were done ad-hoc. The Automatic Statistician Project[6]. came up with an approximate search over the possible space of kernel structures (Duvenaud et al., 2013; Lloyd et al., 2014). However, a fully Bayesian treatment was not available before. To the best of our knowledge, we are the

5. For this pedagogical example we have increased the probability for outliers in the data generation slightly from 0.05 to 0.2
6. `http://www.automaticstatistician.com/`

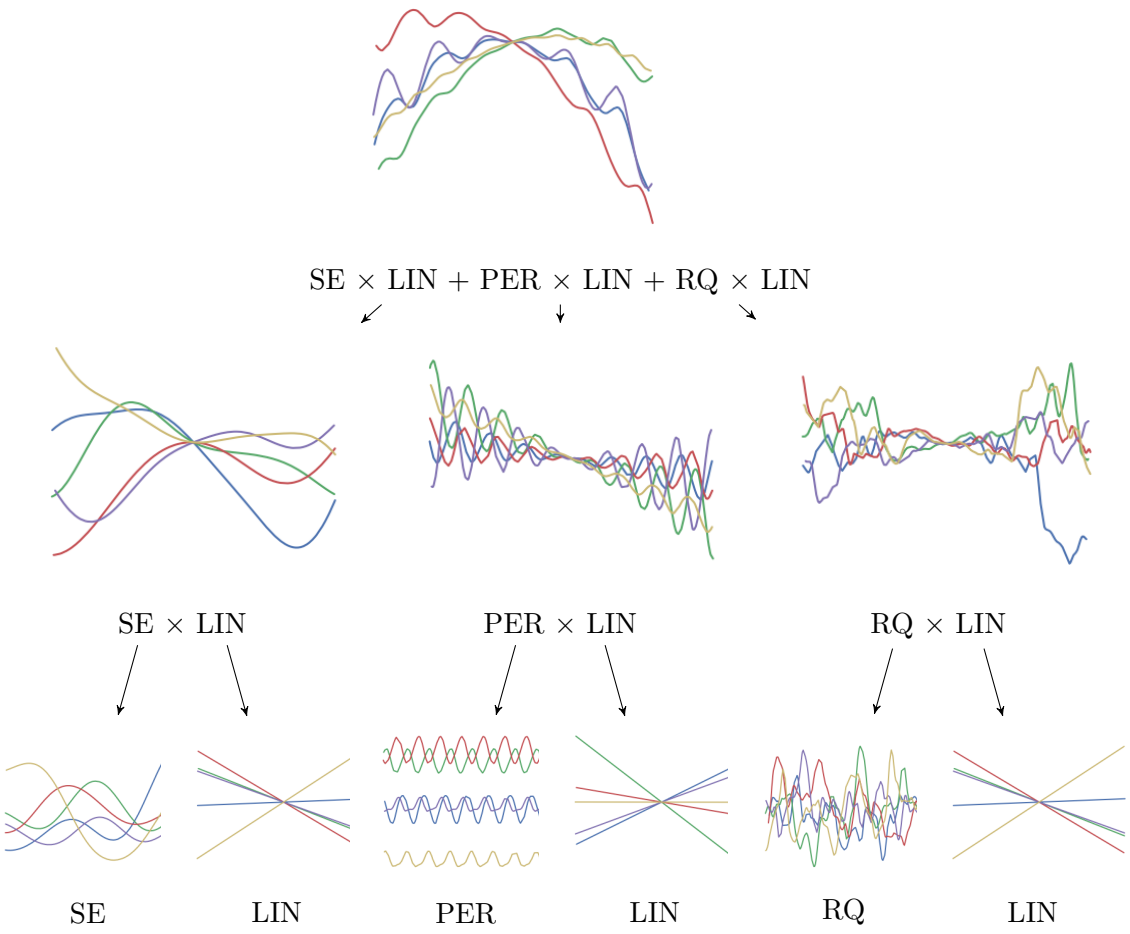


Figure 4: Composition of covariance function parsed using the example SE × LIN + PER × LIN + RQ × LIN. We deploy kernel summation (+) and kernel multiplication (×).

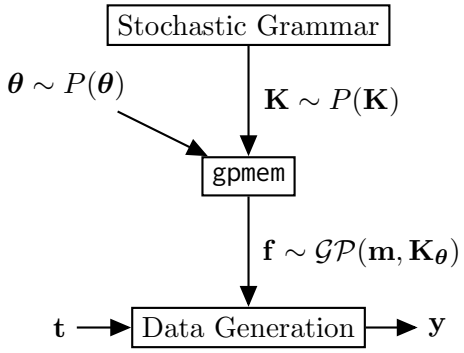


Figure 5: Graphical description of Bayesian GP structure learning.

first one to present a fully Bayesian solution to this. We deploy a probabilistic context free grammar for our prior on structures(see Fig. 5)

Our probabilistic programming based MCMC framework approximates the following intractable integrals of the expectation for the prediction:

$$\mathbb{E}[y^* \mid x^*, \mathbf{D}, \mathbf{K}] = \iint f(x^*, \boldsymbol{\theta}, \mathbf{K})\, P(\boldsymbol{\theta} \mid \mathbf{D}, \mathbf{K})\, P(\mathbf{K}|\boldsymbol{\Omega}, s, n)\, \mathbf{d}\boldsymbol{\theta}\mathbf{d}\mathbf{K}. \tag{18}$$

This is done by sampling from the posterior probability distribution of the hyper-parameters and the possible kernel:

$$y^* \approx \frac{1}{T} \sum_{t=1}^{T} f(x^*|\boldsymbol{\theta}^{(t)}, \mathbf{K}^{(t)}). \tag{19}$$

In order to provide the sampling of the kernel, we introduce a stochastic process that simulates the grammar for algebraic expressions of covariance function algebra:

$$\mathbf{K}^{(t)} \sim P(\mathbf{K} \mid \boldsymbol{\Omega}, s, n) \tag{20}$$

Here, we start with a set of possible kernels and draw a random subset. For this subset of size n, we sample a set of possible operators that operate on the base kernels.

The marginal probability of a kernel structure which allows us to sample is characterized by the probability of a uniformly chosen subset of the set of n possible covariance functions times the probability of sampling a global or a local structure which is given by a binomial distribution:

$$P(\mathbf{K} \mid \boldsymbol{\Omega}, s, n) = P(\boldsymbol{\Omega} \mid s, n) \times P(s \mid n) \times P(n), \tag{21}$$

with

$$P(\boldsymbol{\Omega} \mid s, n) = \binom{n}{r} p_{+\times}^{k} (1 - p_{+\times})^{n-k} \tag{22}$$

and

$$P(s \mid n) = \frac{n!}{\mid s \mid!} \tag{23}$$

where $P(n)$ is a prior on the number of base kernels used which can sample from a discrete uniform distribution. This will strongly prefer simple covariance structures with few base

kernels since individual base kernels are more likely to be sampled in this case due to (23). Alternatively, we can approximate a uniform prior over structures by weighting $P(n)$ towards higher numbers. It is possible to also assign a prior for the probability to sample global or local structures, however, we have assigned complete uncertainty to this with the probability of a flip $p = 0.5$.

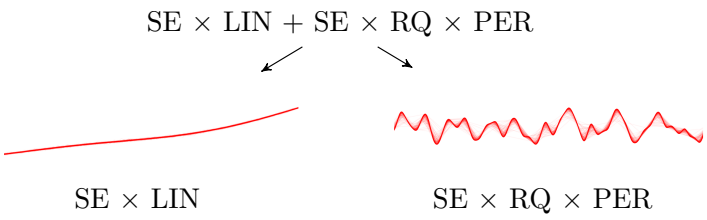
Many equivalent covariance structures can be sampled due to covariance function algebra and equivalent representations with different parameterization (Lloyd et al., 2014). Certain covariance functions can differ in terms of the hyper-parameterization but can be absorbed into a single covariance function with a different parameterization. To inspect the posterior of these equivalent structures we convert each kernel expression into a sum of products and subsequently simplify. Rules for this simplification can be found in appendix B.

The Automatic Statistician Project is implemented using deterministic optimization and search. We reproduce results from this Project in a Bayesian fashion using `gpmem` (Listing 3.2). We first define a prior on the hypothesis space. Note that, as in the implementation of the Automatic Statistician, we upper-bound the complexity of the space of covariance functions we want to explore. We also put vague priors on hyper-parameters.

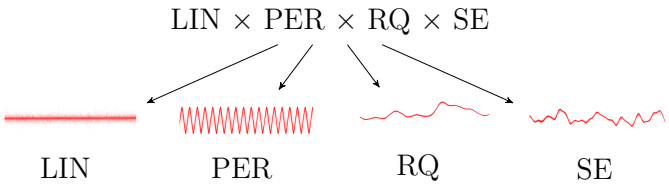
```
// GRAMMAR FOR KERNEL STRUCTURE
assume kernels = list(se, wn, lin, per, rq) // defined as above

// prior on the number of kernels
assume p_number_k = uniform_structure(n)
assume sub_s = tag(quote(grammar), 0,
                           subset(kernels, p_number_k))

assume grammar = proc(l) {
  // kernel composition
  if (size(l) <= 1)
       { first(l) }
  else { if (bernoulli())
                  { add_funcs(first(l), grammar(rest(l))) }
              else { mult_funcs(first(l), grammar(rest(l))) }
       }
  }

assume K = tag(quote(grammar), 1, grammar(sub_s))

assume (f_compute f_emu) = gpmem(f_restr, K)

// Probe all data points
for n ... N
    predict f_compute(get_data_xs(n))

// PERFORMING INFERENCE
infer  repeat(2000, do(
                        mh(quote(grammar), one, 1),
                        for kernel ∈ K
                            mh(quote(hyper_kernel), one, 1)))
```

We defined the space of covariance structures in a way allowing us to reproduce results for covariance function structure learning as in the Automatic Statistician. This lead to coherent results, for example for the airline data set describing monthly totals of international airline passengers (Box et al., 1997, according to Duvenaud et al., 2013). The sample is identical with the highest scoring result reported in previous work using a search-and-score method (Duvenaud et al., 2013) for the CO_2 data set (see Rasmussen and Williams, 2006 for a description) and the predictive capability is comparable. However, the components factor in a different way due to different parameterization of the individual base kernels. We see that the most probable alternatives for a structural description both recover the data dynamics (Fig. 6 for the airline data set).

We further investigated the quality of our stochastic processes by running a leave one out cross-validation computing residuals to gain confidence on the posterior. Confident about our results, we can now query the data for certain structures being present. We illustrate this using the Mauna Loa data used in previous work on automated kernel discovery (Duvenaud et al., 2013). We assume a relatively simple hypothesis space consisting of only four kernels,



(a)



(b)

Figure 6: We see two possible hypotheses for the composite structure of $\mathbf{K}$. (a) Most frequent sample drawn from the posterior on structure. We have found two global components. First, a smooth trend (LIN × SE) with a non-linear increasing slope. Second, a periodic component with increasing variation and noise. (b) Second most frequent sample drawn from the posterior on structure. We found one global component. It is comprised of local changes that are periodic and with changing variation.

a linear, a smoothing, a periodic and a white noise kernel. In this experiment, we resort to the white noise kernel instead RQ (similar to (Lloyd et al., 2014)). We can now run the algorithm, compute a posterior of structures. We can also query this posterior distribution for the marginal of certain simple structures to occur. We demonstrate this in Fig. 7

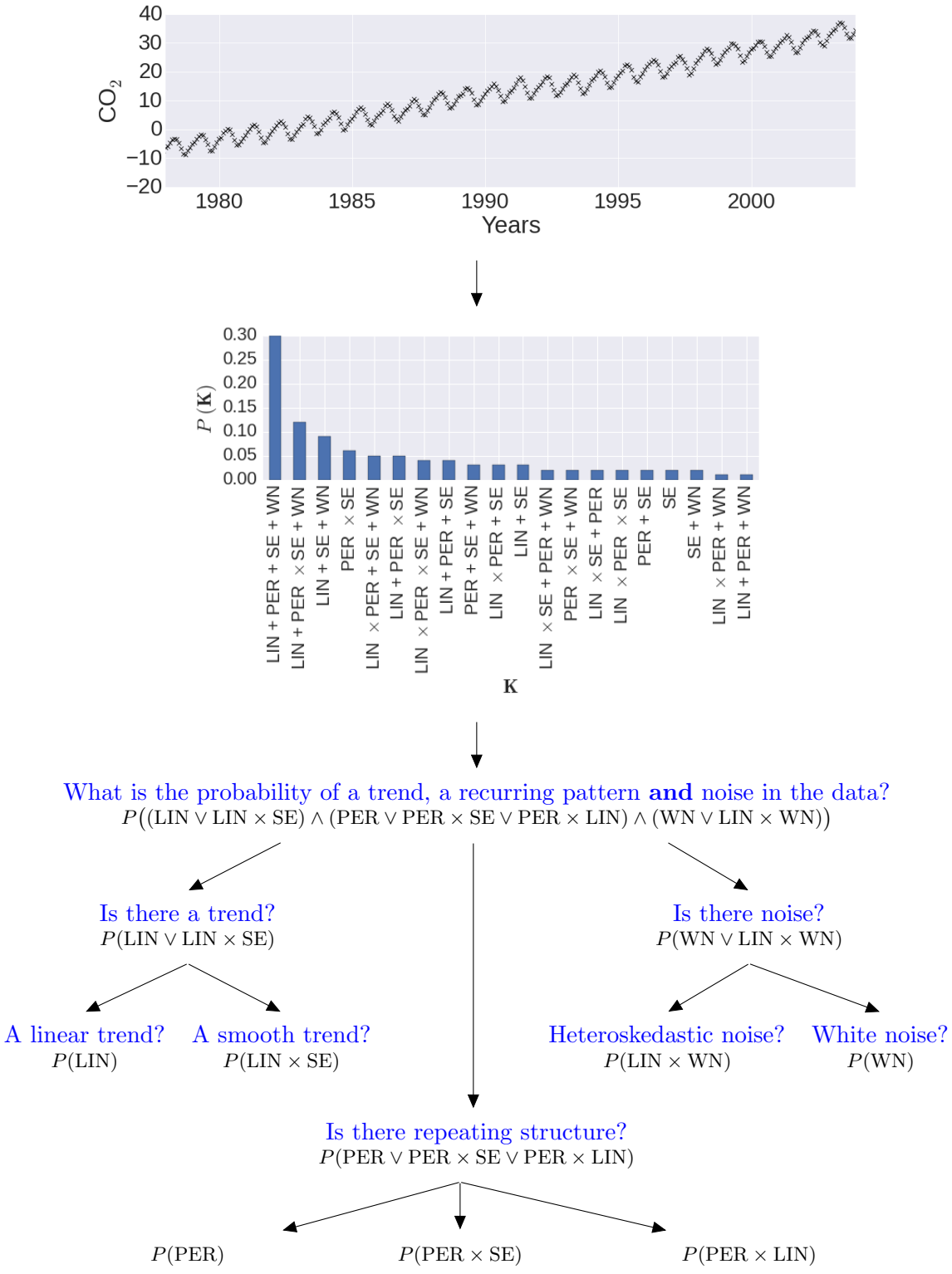


Figure 7: Like in symbolic reasoning we can query the data if some logical statements are probable to be true. Here, we are interested in three qualitative aspects of the data, which we formulate in terms of the following: 1. Is it true that there is a trend? Is it true that there are recurring patterns and is it true that that there is white noise or heteroskedastic noise in the data?

3.3 Bayesian optimization via Thompson sampling

In this section, we introduce Thompson sampling, the basic solution strategy underlying the Bayesian optimization implementations in Section **??**. First, we cast our Bayesian optimization task as a Markov decision process (MDP) and give a high-level algorithmic description of how Thompson sampling can be used to choose actions in an approximately optimal manner for this MDP. We note that probabilistic programming has the expressive power to support exploration of much richer context spaces than are typically used in computational Thompson sampling. Next, we present a code template for expressing Thompson sampling as a probabilistic program using a statistical memoizer. Finally, we give a full mathematical specification of one particular application of Thompson sampling, which (among others) will be implemented in Section **??**.

Framework

Thompson sampling Thompson (1933) is a widely used Bayesian framework for addressing the trade-off between exploration and exploitation in the so-called multi-armed (or continuum-armed) bandit problem. In this subsection, we cast the multi-armed bandit problem as a one-state Markov decision process, and describe how Thompson sampling can be used to choose actions for that MDP.

The setup is as follows: An agent is to take a sequence of actions $a_1, a_2, \ldots$ from a (possibly infinite) set of possible actions $\mathcal{A}$. After each action, a reward $r \in \mathbb{R}$ is received, according to an unknown conditional distribution $P_{\text{true}}(r \mid a)$. The agent's goal is to maximize the total reward received for all actions in an online manner. In Thompson sampling, the agent accomplishes this by placing a prior distribution $P(\vartheta)$ on the possible "contexts" $\vartheta \in \Theta$. Here a context is a believed model of the conditional distributions $\{P(r \mid a)\}_{a \in \mathcal{A}}$, or at least, a believed statistic of these conditional distributions which is sufficient for deciding an action a. If actions are chosen so as to maximize expected reward, then one such sufficient statistic is the believed conditional mean $V(a \mid \vartheta) = \mathbb{E}[r \mid a; \vartheta]$, which can be viewed as a believed value function. For consistency with what follows, we will assume our context ϑ takes the form $(\theta, \mathbf{a}_{\text{past}}, \mathbf{r}_{\text{past}})$ where $\mathbf{a}_{\text{past}}$ is the vector of past actions, $\mathbf{r}_{\text{past}}$ is the vector of their rewards, and θ (the "semicontext") contains any other information that is included in the context.

In this setup, Thompson sampling has the following steps:

Algorithm 2 Thompson sampling.

Repeat as long as desired:

1. **Sample.** Sample a semicontext $\theta \sim P(\theta)$.
2. **Search (and act).** Choose an action $a \in \mathcal{A}$ which (approximately) maximizes $V(a \mid \vartheta) = \mathbb{E}[r \mid a; \vartheta] = \mathbb{E}[r \mid a;\, \theta, \mathbf{a}_{\text{past}}, \mathbf{r}_{\text{past}}]$.
3. **Update.** Let r_{true} be the reward received for action a. Update the believed distribution on θ, i.e., $P(\theta) \leftarrow P_{\text{new}}(\theta)$ where $P_{\text{new}}(\theta) = P(\theta \mid a \mapsto r_{\text{true}})$.

Note that when $\mathbb{E}[r|a;\vartheta]$ (under the sampled value of θ for some points a) is far from the true value $\mathbb{E}_{P_{\text{true}}}[r \mid a]$, the chosen action a may be far from optimal, but the information gained by probing action a will improve the belief ϑ. This amounts to "exploration." When $\mathbb{E}[r \mid a;\vartheta]$ is close to the true value except at points a for which $\mathbb{E}[r \mid a;\vartheta]$ is low, exploration will be less likely to occur, but the chosen actions a will tend to receive high rewards. This amounts to "exploitation." The trade-off between exploration and exploitation is illustrated in Figure 8. Roughly speaking, exploration will happen until the context ϑ is reasonably sure that the unexplored actions are probably not optimal, at which time the Thompson sampler will exploit by choosing actions in regions it knows to have high value.

Typically, when Thompson sampling is implemented, the search over contexts $\vartheta \in \Theta$ is limited by the choice of representation. In traditional programming environments, θ often consists of a few numerical parameters for a family of distributions of a fixed functional form. With work, a mixture of a few functional forms is possible; but without probabilistic programming machinery, implementing a rich context space Θ would be an unworkably large technical burden. In a probabilstic programing language, however, the representation of heterogeneously structured or infinite-dimensional context spaces is quite natural. Any computable model of the conditional distributions $\{P(r \mid a)\}_{a\in\mathcal{A}}$ can be represented as a stochastic procedure $(\lambda(a)\ldots)$. Thus, for computational Thompson sampling, the most general context space $\widehat{\Theta}$ is the space of program texts. Any other context space Θ has a natural embedding as a subset of $\widehat{\Theta}$.

Thompson Sampling with a Statistical Memoizer

Thompson sampling as described above can be expressed compactly in terms of a statistical memoizer, as in Listing 4 (see also Figure 9). Here and in what follows, to simplify the treatment, we assume the true reward function is deterministic, i.e., for each fixed a, the conditional distribution $P_{\text{true}}(r \mid a)$ is a delta distribution. We thus treat the notations $a, r, \mathbf{a}_{\text{past}}, \mathbf{r}_{\text{past}}$ as synonyms for $x, y, \mathbf{x}_{\text{past}}, \mathbf{y}_{\text{past}}$, respectively, differing only in that they carry the connotations of "action" and "reward."

Listing 4: Code template for Thompson sampling in pseudo-Venture using a statistical memoizer. The choice of statistical memoizer (`mem&em`), prior on semicontexts (`prior_on_semicontexts`), and approximation strategy for maximizing the emulator (`argmax`) are not included.

```
assume theta = tag(quote( theta ), 0,  prior_on_semicontexts())
assume (r_probe r_emulate) = mem&em( do_action, theta)

infer repeat(15, do(pass,
     a = mc_argmax(r_emulate, quote(_)),
     r_probe(a)
     mh(quote(theta), one, 50)))
```

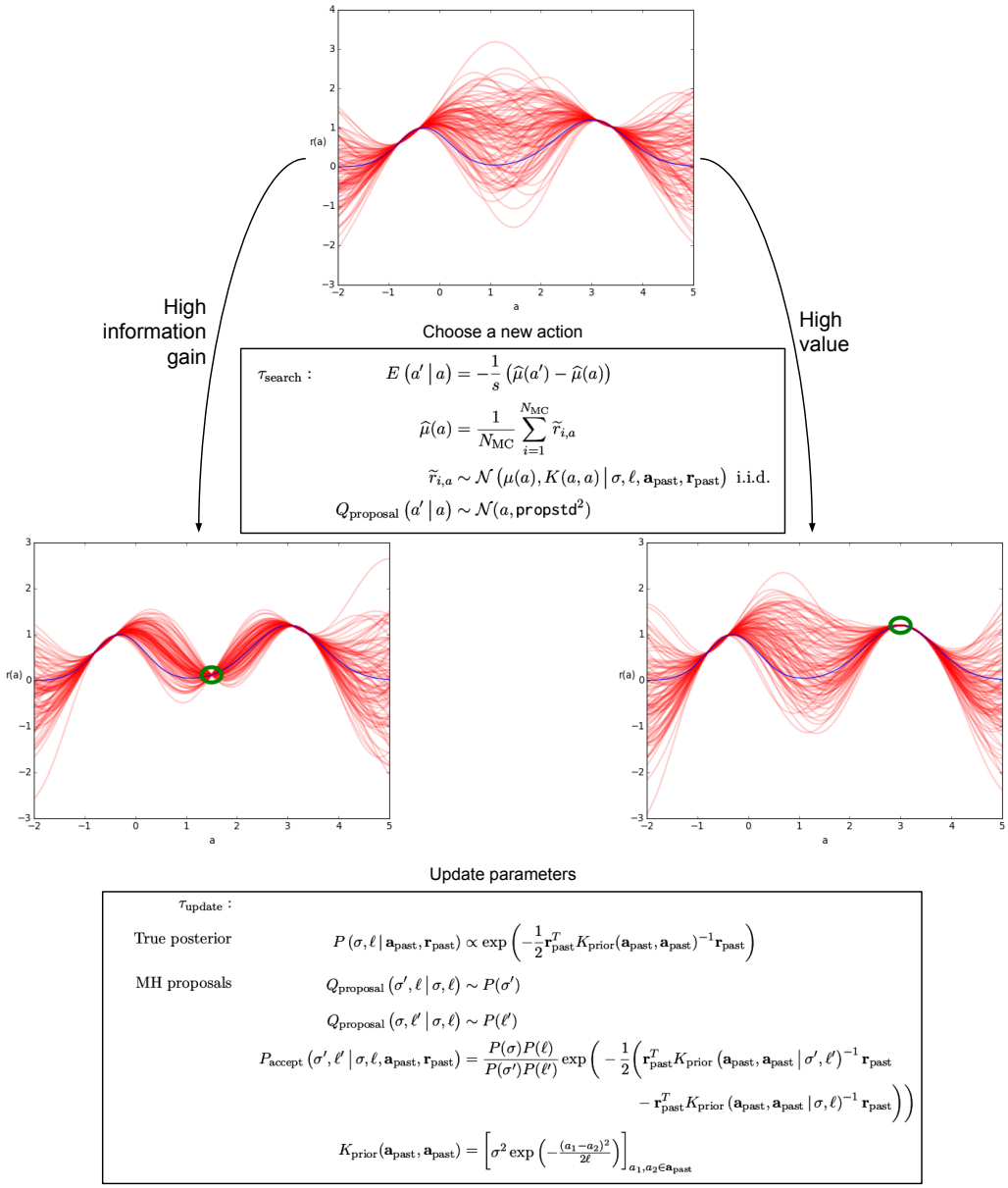


Figure 8: Two possible actions (in green) for an iteration of Thompson sampling. The believed distribution on the value function V is depicted in red. In this example, the true reward function is deterministic, and is drawn in blue. The action on the right receives a high reward, while the action on the left receives a low reward but greatly improves the accuracy of the believed distribution on V. The transition operators τ_{search} and τ_{update} are described in Section 3.3.

In Listing 4, `do_action` is a procedure which interfaces with the outside environment and returns the reward received. `do_action` is wrapped in the statistical memoizer `mem&em`, with parameters given by the semicontext `theta` (see Setion 3.3). `theta` is given a prior distribution, and as more actions `a` are probed, inference is performed on `theta`, and subsequent evaluations of the emulator `r_emulate` are affected accordingly. The action `a` chosen at each step is determined by the procedure `argmax`; for purposes of Thompson sampling, `argmax` (which takes a possibly stochastic procedure as its argument) should be implemented so that `(argmax func)` returns an approximation to $\arg\max_{\mathtt{a}} \mathbb{E}\left[\texttt{(func a)}\right]$.

A Mathematical Specification

Below, we give the mathematical specification of a particular application of Thompson sampling. This application has the following properties:

- The regression function has a Gaussian process prior.
- The actions $a_1, a_2, \ldots \in \mathcal{A}$ are chosen by a Metropolis-like search strategy with Gaussian drift proposals.
- The hyperparameters of the Gaussian process are inferred using Metropolis–Hastings sampling after each action.

In this version of Thompson sampling, the contexts ϑ are Gaussian processes over the action space $\mathcal{A} = [-20, 20] \subseteq \mathbb{R}$. That is,

$$V \sim \mathcal{GP}(\mu, K),$$

where the mean μ is a computable function $\mathcal{A} \to \mathbb{R}$ and the covariance K is a computable (symmetric, positive-semidefinite) function $\mathcal{A} \times \mathcal{A} \to \mathbb{R}$. This represents a Gaussian process $\{R_a\}_{a \in \mathcal{A}}$, where R_a represents the reward for action a. Computationally, we represent a context as a data structure

$$\vartheta = (\theta, \mathbf{a}_{\text{past}}, \mathbf{r}_{\text{past}}) = (\mu_{\text{prior}}, K_{\text{prior}}, \eta, \mathbf{a}_{\text{past}}, \mathbf{r}_{\text{past}}),$$

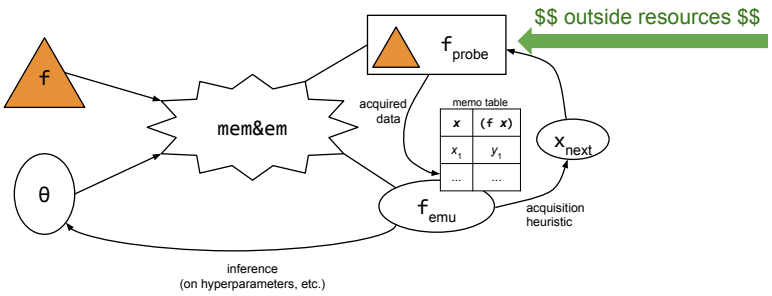
where:

- μ_{prior} is a procedure to be used as the prior mean function. To simplify the treatment, we take $\mu_{\text{prior}} \equiv 0$.
- K_{prior} is a procedure to be used as the prior covariance function. In this example, we use a fixed functional form[7] $K_{\text{prior}} = K_{\text{prior}}(\bullet \mid \sigma, \ell)$, the squared exponential

$$K_{\text{prior}}(a, a') = \sigma^2 \exp\left(-\frac{(a - a')^2}{2\ell}\right).$$

- η is a set of (hyper)parameters for the mean and covariance functions; in this case $\eta = \{\sigma, \ell\}$.

7. However, within this framework there is no reason why the functional form of K_{prior} cannot itself be random and be chosen by probabilistic inference. The support of $P(K_{\text{prior}})$ could, for example, be the search space of composite kernel structures explored in Duvenaud et al. **?**.



In general:

$$P\left((\mathtt{f}_{\text{emu}}\ \mathbf{x}) \,\middle|\, \text{memo table} = (\mathbf{x}_{\text{past}}, \mathbf{y}_{\text{past}})\right) \sim P\left(f(\mathbf{x}) \,\middle|\, f(\mathbf{x}_{\text{past}}) = \mathbf{y}_{\text{past}}\right),$$
$$\text{where } f \sim P\left(\mathtt{f}_{\text{emu}} \,\middle|\, \text{memo table} = (\emptyset, \emptyset)\right)$$

For $\mathcal{GP}(\mu_{\text{prior}}, K_{\text{prior}})$ prior on $\mathtt{f}_{\text{emu}}$:

$$P\left((\mathtt{f}_{\text{emu}}\ \mathbf{x}) \,|\, \mathbf{x}_{\text{past}}, \mathbf{y}_{\text{past}}\right) \sim \mathcal{N}\left(\mu(\mathbf{x}), K(\mathbf{x}, \mathbf{x})\right)$$
$$\mu(\mathbf{x}) = \mu_{\text{prior}}(\mathbf{x}) + K_{\text{prior}}(\mathbf{x}, \mathbf{x}_{\text{past}})\, K_{\text{prior}}(\mathbf{x}_{\text{past}}, \mathbf{x}_{\text{past}})^{-1} \left(\mathbf{y}_{\text{past}} - \mu_{\text{prior}}(\mathbf{x}_{\text{past}})\right)$$
$$K(\mathbf{x}, \mathbf{x}) = K_{\text{prior}}(\mathbf{x}, \mathbf{x}) - K_{\text{prior}}(\mathbf{x}, \mathbf{x}_{\text{past}})\, K_{\text{prior}}(\mathbf{x}_{\text{past}}, \mathbf{x}_{\text{past}})^{-1} K_{\text{prior}}(\mathbf{x}_{\text{past}}, \mathbf{x})$$

If f is (nearly) smooth, then as $|\mathbf{x}_{\text{past}}| \to \infty$, $\mathtt{f}_{\text{emu}} \approx \mathtt{f}$.

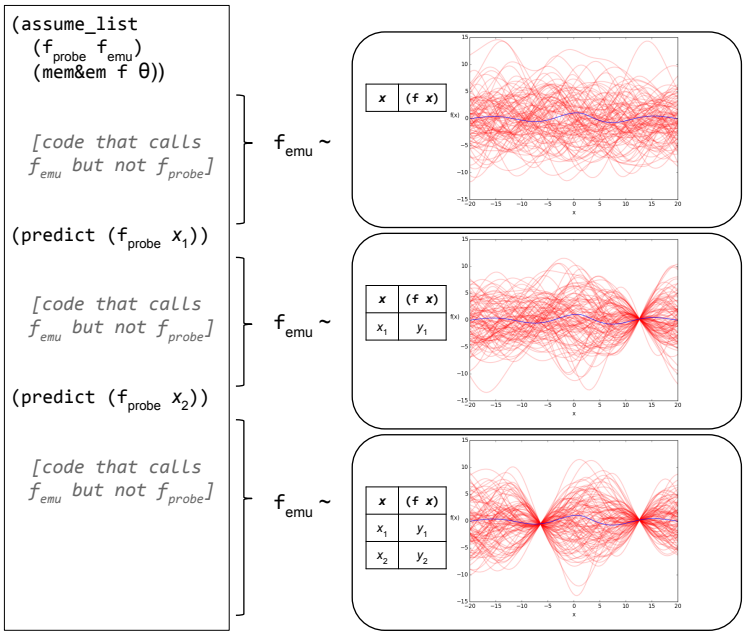


Figure 9: Top: Diagrammatic representation of Thompson sampling using the statistical memoizer `mem&em`. The emulator $\mathtt{f}_{\text{emu}}$ is used to choose a point $\mathtt{x}_{\text{next}}$ to acquire; the acquired data are incorporated into the memo table and the parameters θ are updated according to this new information. Middle: Mathematical specification of the statistical emulator $\mathtt{f}_{\text{emu}}$. Bottom: Depiction of the state of the emulator after zero, one, and two probes have been taken. Here the true value function V is drawn in blue; the believed distribution on V is drawn in red. Between probes, $\mathtt{f}_{\text{emu}}$ can be `sampled` but not `predicted`, as its state (see Section **??**) must not be changed. The state (i.e., the memo table) must only be changed when $\mathtt{f}_{\text{probe}}$ is called.

The posterior mean and covariance for such a context ϑ are gotten by the usual conditioning formulas (assuming, for ease of exposition as above, that the prior mean is zero):[8]

$$\begin{aligned}\mu(\mathbf{a}) &= \mu\left(\mathbf{a} \mid \mathbf{a}_{\text{past}}, \mathbf{r}_{\text{past}}\right) \\ &= K_{\text{prior}}(\mathbf{a}, \mathbf{a}_{\text{past}})\, K_{\text{prior}}(\mathbf{a}_{\text{past}}, \mathbf{a}_{\text{past}})^{-1}\, \mathbf{r}_{\text{past}} \\ K(\mathbf{a}, \mathbf{a}) &= K\left(\mathbf{a}, \mathbf{a} \mid \mathbf{a}_{\text{past}}, \mathbf{r}_{\text{past}}\right) \\ &= K_{\text{prior}}(\mathbf{a}, \mathbf{a}) - K_{\text{prior}}(\mathbf{a}, \mathbf{a}_{\text{past}})\, K_{\text{prior}}(\mathbf{a}_{\text{past}}, \mathbf{a}_{\text{past}})^{-1}\, K_{\text{prior}}(\mathbf{a}_{\text{past}}, \mathbf{a}).\end{aligned}$$

Note that the context space Θ is not a finite-dimensional parametric family, since the vectors $\mathbf{a}_{\text{past}}$ and $\mathbf{r}_{\text{past}}$ grow as more samples are taken. Θ is, however, quite easily representable as a computational procedure together with parameters and past samples, as we do in the representation $\vartheta = (\mu_{\text{prior}}, K_{\text{prior}}, \eta, \mathbf{a}_{\text{past}}, \mathbf{r}_{\text{past}})$.

We combine the Update and Sample steps of Algorithm 2 by running a Metropolis–Hastings (MH) sampler whose stationary distribution is the posterior $P\left(\theta \mid \mathbf{a}_{\text{past}}, \mathbf{r}_{\text{past}}\right)$. The functional forms of μ_{prior} and K_{prior} are fixed in our case, so inference is only done over the parameters $\eta = \{\sigma, \ell\}$; hence we equivalently write $P\left(\sigma, \ell \mid \mathbf{a}_{\text{past}}, \mathbf{r}_{\text{past}}\right)$ for the stationary distribution. We make MH proposals to one variable at a time, using the prior as proposal distribution:

$$Q_{\text{proposal}}\left(\sigma', \ell \;\middle|\; \sigma, \ell\right) = P(\sigma')$$

and

$$Q_{\text{proposal}}\left(\sigma, \ell' \;\middle|\; \sigma, \ell\right) = P(\ell').$$

The MH acceptance probability for such a proposal is

$$P_{\text{accept}}\left(\sigma', \ell' \;\middle|\; \sigma, \ell\right) = \min\left\{1,\ \frac{Q_{\text{proposal}}\left(\sigma, \ell \mid \sigma', \ell'\right)}{Q_{\text{proposal}}\left(\sigma', \ell' \mid \sigma, \ell\right)} \cdot \frac{P\left(\mathbf{a}_{\text{past}}, \mathbf{r}_{\text{past}} \mid \sigma', \ell'\right)}{P\left(\mathbf{a}_{\text{past}}, \mathbf{r}_{\text{past}} \mid \sigma, \ell\right)}\right\}$$

Because the priors on σ and ℓ are uniform in our case, the term involving Q_{proposal} equals 1 and we have simply

$$\begin{aligned}P_{\text{accept}}\left(\sigma', \ell' \;\middle|\; \sigma, \ell\right) &= \min\left\{1,\ \frac{P\left(\mathbf{a}_{\text{past}}, \mathbf{r}_{\text{past}} \mid \sigma', \ell'\right)}{P\left(\mathbf{a}_{\text{past}}, \mathbf{r}_{\text{past}} \mid \sigma, \ell\right)}\right\} \\ &= \min\left\{1,\ \exp\left(-\frac{1}{2}\left(\mathbf{r}_{\text{past}}^T K_{\text{prior}}\left(\mathbf{a}_{\text{past}}, \mathbf{a}_{\text{past}} \;\middle|\; \sigma', \ell'\right)^{-1} \mathbf{r}_{\text{past}}\right.\right.\right. \\ &\qquad \left.\left.\left. - \mathbf{r}_{\text{past}}^T K_{\text{prior}}\left(\mathbf{a}_{\text{past}}, \mathbf{a}_{\text{past}} \mid \sigma, \ell\right)^{-1} \mathbf{r}_{\text{past}}\right)\right)\right\}.\end{aligned}$$

The proposal and acceptance/rejection process described above define a transition operator τ_{update} which is iterated a specified number of times; the resulting state of the MH Markov chain is taken as the sampled semicontext θ in Step 1 of Algorithm 2.

For Step 2 (Search) of Thompson sampling, we explore the action space using an MH-like transition operator τ_{search}. As in MH, each iteration of τ_{search} produces a proposal which is either accepted or rejected, and the state of this Markov chain after a specified number

8. Here, for vectors $\mathbf{a} = (a_i)_{i=1}^n$ and $\mathbf{a}' = (a'_i)_{i=1}^{n'}$, $\mu(\mathbf{a})$ denotes the vector $(\mu(a_i))_{i=1}^n$ and $K(\mathbf{a}, \mathbf{a}')$ denotes the matrix $\left[K(a_i, a'_j)\right]_{1 \le i \le n, 1 \le j \le n'}$.

of steps is the new action a. The Markov chain's initial state is the most recent action, and the proposal distribution is Gaussian drift:

$$Q_{\text{proposal}}\left(a' \mid a\right) \sim \mathcal{N}(a, \texttt{propstd}^2),$$

where the drift width `propstd` is specified ahead of time. The acceptance probability of such a proposal is

$$P_{\text{accept}}\left(a' \mid a\right) = \min\left\{1,\ \exp\left(-E\left(a' \mid a\right)\right)\right\},$$

where the energy function $E\left(\bullet \mid a\right)$ is given by a Monte Carlo estimate of the difference in value from the current action:

$$E\left(a' \mid a\right) = -\frac{1}{s}\left(\widehat{\mu}(a') - \widehat{\mu}(a)\right)$$

where

$$\widehat{\mu}(a) = \frac{1}{N_{\text{avg}}} \sum_{i=1}^{N_{\text{avg}}} \widetilde{r}_{i,a}$$

and

$$\widetilde{r}_{i,a} \sim \mathcal{N}(\mu(a), K(a,a))$$

and $\{\widetilde{r}_{i,a}\}_{i=1}^{N_{\text{avg}}}$ are i.i.d. for a fixed a. Here the temperature parameter $s \geq 0$ and the population size N_{avg} are specified ahead of time. Note the following properties of the above acceptance probability:

- Proposals of estimated value higher than that of the current action are always accepted, while proposals of estimated value lower than that of the current action are accepted with a probability that decays exponentially with respect to the difference in value.
- The rate of the decay is determined by the temperature parameter s, where high temperature corresponds to generous acceptance probabilities. For $s = 0$, all proposals of lower value are rejected; for $s = \infty$, all proposals are accepted.
- For points a at which the posterior mean $\mu(a)$ is low but the posterior variance $K(a,a)$ is high, it is possible (especially when N_{avg} is small) to draw a "wild" value of $\widehat{\mu}(a)$, resulting in a favorable acceptance probability.

Indeed, taking an action a with low estimated value but high uncertainty serves the useful function of improving the accuracy of the estimated value function at points near a (see Figure 8).[9,10] We see a comlete probabilistic program with `gpmem` implementing Bayesian optimization with Thompson Sampling below (Listing **??**).

9. At least, this is true when we use a smoothing prior covariance function such as the squared exponential.
10. For this reason, we consider the sensitivity of $\widehat{\mu}$ to uncertainty to be a desirable property; indeed, this is why we use $\widehat{\mu}$ rather than the exact posterior mean μ.

Listing 5: Bayesian optimization using gpmem

```
assume sf = tag(quote(hyper), 0, uniform_continuous(0, 10))
assume l = tag(quote(hyper), 1, uniform_continuous(0, 10))
assume se = make_squaredexp(sf, l)
assume blackbox_f = get_bayesopt_blackbox()
assume (f_compute, f_emulate) = gpmem(blackbox_f, se)

// A naive estimate of the argmax of the given function
define mc_argmax = proc(func) {
  candidate_xs = mapv(proc(i) {uniform_continuous(-20, 20)},
                      arange(20));
  candidate_ys = mapv(func, candidate_xs);
  lookup(candidate_xs, argmax_of_array(candidate_ys))
};

// Shortcut to sample the emulator at a single point without packing
// and unpacking arrays
define emulate_pointwise = proc(x) {
    run(sample(lookup(f_emulate(array(unquote(x))), 0)))
};

// Main inference loop
infer repeat(15, do(pass,
    // Probe V at the point mc_argmax(emulate_pointwise)
    predict(f_compute(unquote(mc_argmax(emulate_pointwise)))),
    // Infer hyperparameters
    mh(quote(hyper), one, 50)));
```

4. Discussion

We have shown Venture GPs. We have introduced novel stochastic processes for a probabilistic programming language. We showed how flexible non-parametric models can be treated in Venture in only a few lines of code. We evaluated our contribution on a range of hard problems for state-of-the-art Bayesian non-parametrics. Venture GPs showed competitive performance in all of them.

4.1 Contributions

Appendix

A Covariance Functions

SE and WN are defined in the text above, for completeness we will introduce the covariance:

$$k_{LIN}(x, x') = \theta(xx') \tag{24}$$

$$k_{PER}(x, x') = \theta \exp\left(\frac{2\sin^2(\pi(x - x')/p}{\ell^2}\right) \tag{25}$$

$$k_{RQ}(x, x') = \theta\left(1 + \frac{(x - x')^2}{2\alpha\ell^2}\right)^{-\alpha} \tag{26}$$

B Covariance Simplification

SE × SE	→	SE
{SE,PER,C,WN} × WN	→	WN
LIN + LIN	→	LIN
{SE,PER,C,WN,LIN} × C	→	{SE,PER,C,WN,LIN}

Rule 1 is derived as follows:

$$\begin{aligned}
\sigma_c^2 \exp(-\frac{(x - x')^2}{2\ell_c^2}) &= \sigma_a^2 \exp(-\frac{(x - x')^2}{2\ell_a^2}) \times \sigma_b^2 \exp(-\frac{(x - x')^2}{2\ell_b^2}) \\
&= \sigma_c^2 \exp(-\frac{(x - x')^2}{2\ell_a^2}) \times \exp(-\frac{(x - x')^2}{2\ell_b^2}) \\
&= \sigma_c^2 \exp\left(-\frac{(x - x')^2}{2\ell_a^2} - \frac{(x - x')^2}{2\ell_b^2}\right) \\
&= \sigma_c^2 \exp\left(-\frac{(x - x')^2}{2\ell_c^2}\right)
\end{aligned} \tag{27}$$

Rule 3 is derived as follows:

$$\theta_c(x \times x') = \theta_a(x \times x') + \theta_b(x \times x') \tag{28}$$

For stationary kernels that only depend on the lag vector between x and x' it holds that multiplying such a kernel with a WN kernel we get another WN kernel. Take for example the SE kernel:

$$\sigma_a^2 \exp\left(-\frac{(x - x')^2}{2\ell_c^2}\right) \times \sigma_b \delta_{x,x'} = \sigma_a \sigma_b \delta_{x,x'} \tag{29}$$

Multiplying any kernel with a constant obviously changes only the scale parameter of a kernel.

References

N. L. Ackerman, C. E. Freer, and D. M. Roy. On the computability of conditional probability. *arXiv preprint arXiv:1005.3014*, 2010.

C. Andrieu, N. De Freitas, A. Doucet, and M. I. Jordan. An introduction to MCMC for machine learning. *Machine learning*, 50(1-2):5–43, 2003.

G. E. P. Box, G. M. Jenkins, and G. C. Reinsel. *Time series analysis: forecasting and control*. 1997.

D. Duvenaud, J. R. Lloyd, R. Grosse, J. Tenenbaum, and Z. Ghahramani. Structure discovery in nonparametric regression through compositional kernel search. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pages 1166–1174, 2013.

Z. Ghahramani. Probabilistic machine learning and artificial intelligence. *Nature*, 521(7553): 452–459, 2015.

V. K. Goodman, N. D .and Mansinghka, D. Roy, K. Bonawitz, and J.B. Tenenbaum. Church: A language for generative models. In *Proceedings of the 24th Conference on Uncertainty in Artificial Intelligence, UAI 2008*, pages 220–229, 2008.

J. R. Lloyd, D. Duvenaud, R. Grosse, J. Tenenbaum, and Z. Ghahramani. Automatic construction and natural-language description of nonparametric regression models. In *Twenty-Eighth AAAI Conference on Artificial Intelligence*, 2014.

V. K. Mansinghka, D. Selsam, and Y. Perov. Venture: a higher-order probabilistic programming platform with programmable inference. *arXiv preprint arXiv:1404.0099*, 2014.

D. McAllester, B. Milch, and N. D. Goodman. Random-world semantics and syntactic independence for expressive languages. Technical report, 2008.

N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller. Equation of state calculations by fast computing machines. *The Journal of Chemical Physics*, 21 (6):1087–1092, 1953.

R. M. Neal. Monte carlo implementation of gaussian process models for bayesian regression and classification. *arXiv preprint physics/9701026*, 1997.

D. Poole. Probabilistic horn abduction and bayesian networks. *Artificial Intelligence*, 64 (1):81–129, 1993.

C. E. Rasmussen and C. K. I. Williams. *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press, 2006.

T. Sato. A statistical learning method for logic programs with distribution semantics. In *In Proceedings of the 12th International Conference on Logic Programming*. Citeseer, 1995.

J. Snoek, H. Larochelle, and R. P. Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in Neural Information Processing Systems*, pages 2951–2959, 2012.

W. R. Thompson. On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika*, pages 285–294, 1933.