
Probabilistic Programming with Gaussian Process Memoization

Anonymous Author(s)

Affiliation

Address

email

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 very powerful model yet unseen 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

054 to hierarchical prior construction (Neal, 1997), Bayesian Optimization Snoek et al. (2012) and sys-
055 tems for inductive learning of symbolic expressions such as the one introduced in the Automated
056 Statistician project Duvenaud et al. (2013); Lloyd et al. (2014). Learning such symbolic expressions
057 is a hard problem that requires careful design of approximation techniques since standard inference
058 method do not apply.

059 In the following, we will present `gpmem` as a novel probabilistic programming technique that solves
060 such hard problems. `gpmem` introduces a statistical alternative to standard memoization. Our con-
061 tribution is threefold:

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

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

078 2 Background

079 2.1 Memoization

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

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

094 2.2 Venture

097 Venture is a compositional language for custom inference strategies that comes with a Scheme-
098 like and a JavaScript-like front-end syntaxes. Its implementation is based on on three concepts:
099 (i) *stochastic procedures* that specify and encapsulate random variables, analogously to conditional
100 probability tables in a Bayesian network; (ii) *execution traces* that represent (partial) execution his-
101 tories and track the conditional dependencies of the random variables occurring therein; and (iii)
102 *scaffolds* that partition execution histories and factor global inference problems into sub-problems.
103 These building blocks provide a powerful and concise way to represent probability distributions,
104 including distributions with a dynamically determined and unbounded set of random variables. In
105 this paperwe will use only the four basic Venture directives: ASSUME, OBSERVE, SAMPLE and
INFER.

- 107 • ASSUME induces a hypothesis space for (probabilistic) models including random variables
by binding the result of a supplied expression to a supplied symbol.

- Whereas in Scheme an expression is evaluated within an environment, in Venture an expression is evaluated within a (partial) 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 SAMPLE directive samples the value of the supplied expression within the current model program.
- 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 to mutate the execution trace. For a correct inference program, this will result approximate sampling from the true posterior on execution traces, 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 trace.

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 :

$$x^* \sim q(x^* | x^t); \quad (1)$$

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

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

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.

2.3 Gaussian Processes

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 $k(\mathbf{x}, \mathbf{x}') = \text{Cov}_f (f(\mathbf{x}), f(\mathbf{x}'))$ is the covariance function, a.k.a. kernel.¹ 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:

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

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

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

a multivariate normal with mean vector

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

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

162 and covariance matrix

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

165 Often one assumes the values \mathbf{y} are noisily measured, that is, one only sees the values of $\mathbf{y}_{\text{noisy}} =$
 166 $\mathbf{y} + \mathbf{w}$ where \mathbf{w} is Gaussian white noise with variance σ_{noise}^2 . In that case, the log-likelihood is

$$\log p(\mathbf{y}_{\text{noisy}} | \mathbf{x}) = -\frac{1}{2}\mathbf{y}^\top(\Sigma + \sigma_{\text{noise}}^2 \mathbf{I})^{-1}\mathbf{y} - \frac{1}{2}\log|\Sigma + \sigma_{\text{noise}}^2 \mathbf{I}| - \frac{n}{2}\log 2\pi \quad (7)$$

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

173 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:

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

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

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

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

$$k_3(x, x') = k_1(x, x') + k_2(x, x'), \quad (9)$$

186 corresponds to combining global structures, while multiplying covariance functions,

$$k_4(x, x') = k_1(x, x') \times k_2(x, x'), \quad (10)$$

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

191 3 Venture GPs

193 The Venture procedure `make-gp` takes as input a mean function and a covariance function, and
 194 outputs a procedure for sampling from a Gaussian process. In effect, each call to this procedure
 195 samples from (4) conditioned on the return values of all previous samples. `make-gp` allows us to
 196 perform GP inference in Venture with only a few lines of code. We can concisely express a wide
 197 variety of GPs: simple smoothing with fixed hyper-parameters, or a prior on hyper-parameters, or
 198 a custom covariance function. Inference on hyper-parameters can be performed using Venture's
 199 built-in MH operator or a custom inference strategy.

200 Venture code to create and sample from a GP with a smoothing kernel and hyperparameters is shown
 201 in Listing 3.

```

203 // HYPER-PARAMETERS
204 assume sf = tag(quote(hyper), 0, gamma(1,1))
205 assume l = tag(quote(hyper), 1, gamma(1,1))

206 // COVARIANCE FUNCTION
207 assume se = make_squaredexp(sf, l)

208 // MAKE GAUSSIAN PROCESS
209 assume gp = make_gp( 0, se)

210 // INCORPORATE OBSERVATIONS
211 observe gp(array x[1],...x[n])= array(y[1],...,y[n])

212 // INFER HYPER-PARAMETERS
213 infer mh(quote(hyper), one, 1)))
214
215

```

216 The first two lines declare the hyper-parameters. We tag both of them to belong to the “scope”
 217 quote(hyper). Scopes may be further subdivided into blocks, on which block proposals can be
 218 made. In the program above we assign two blocks, 0 and 1. These tags are supplied to the inference
 219 program (in this case, MH) to specify on which random variables inference should be done. In this
 220 paper, we use MH inference throughout and do not use block proposals; MH inference is done on
 221 one variable at a time.

222 The ASSUME directives describe the GP model: sf and l (corresponding to σ and ℓ) are drawn
 223 from independent $\Gamma(1, 3)$ distributions. The squared exponential covariance function can be defined
 224 outside the Venture code in a conventional programming language (e.g. Python) and imported as a
 225 foreign SP. In that way, the user can define custom covariance functions using his or her language
 226 and libraries of choice, without having to port existing code into Venture’s modelling language. In
 227 the above, the factory function make-se, which produces a squared exponential function with the
 228 supplied hyperparameters, is imported from Python (we have omitted the Python code). In the next
 229 line make-se is used to produce a covariance function SE, whose (random) hyperparameters are
 230 l and sf. Finally, we declare GP to be a Gaussian process with mean zero and covariance function
 231 SE.

232 3.1 Gaussian process memoization: gpmem

233 gpmem is an extension of previous approaches to stochastic memoization. Previous approaches
 234 such as the one introduced for Dirichlet Processes in the Church language (Goodman et al., 2008)
 235 deal with the discrete-valued domain. We introduce memoization for the continuous valued domain.
 236 This is important for problems of regression and optimization where target functions should not be
 237 artificially discretized. We present a probabilistic programming technique that uses a GP to provide
 238 a statistical variant of memoization.

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

242 3.2 A Bayesian interpretation

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

$$y = 2x + 15 \quad (11)$$

246 and outliers are generated with a parallel line:

$$\hat{y} = 2x + 40. \quad (12)$$

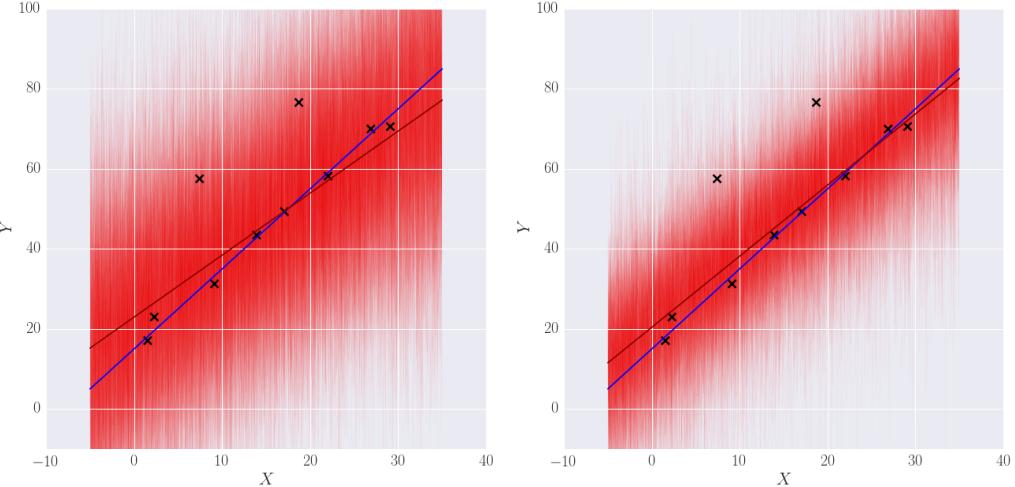
247 We add some small amount of white noise. We generate eight data points with (11) and two with
 248 (12). Since we suspect the underlying data generating mechanism to be linear, we fit a linear kernel
 249 with a constant covariance as intercept and some white noise.

$$\mathbf{K} = \text{LIN} + \mathbf{C} + \text{WN} \quad (13)$$

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

257 3.2.1 Data modelling as a special case of gpmem

258 From the standpoint of computation, a data set of the form $\{(x_i, y_i)\}$ can be thought of as a function
 259 $y = f_{\text{restr}}(x)$, where f_{restr} is restricted to only allow evaluation at a specific set of inputs x . Modelling
 260 the data set with a GP then amounts to trying to learn a smooth function f_{emu} (“emu” stands for
 261 “emulator”) which extends f to its full domain. Indeed, if f_{restr} is a foreign procedure made available
 262 as a black-box to Venture, whose secret underlying source code is:



(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.

```

294
295 def f_restr(x):
296     if x in D:
297         return D[x]
298     else:
299         raise Exception('Illegal input')
300 
```

Then the OBSERVE code in Listing 3 can be rewritten using `gpmem` as follows (where here the data set `D` has keys $x[1], \dots, x[n]$):

```

303 [ASSUME (list f_compute f_emu) (gpmem f_restr) ]
304 for i=1 to n:
305     [PREDICT (f_compute x[i]) ]
306     [INFER (MH {hyper-parameters} one 100) ]
307     [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.

3.2.2 The efficacy of learning hyperparameters

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

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

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)². The work suggests a hierarchical system of hyper-parameterization. Here, we draw hyper-parameters from a Γ distributions:

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

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

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

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

```

/// 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
predict mapv(f_compute, get_neal_data_xs())

// 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:

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

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 devices 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 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 outliers³.

²In (Neal, 1997) the sum of an SE plus a constant kernel is used. We stick to the WN kernel for illustrative purposes.

³For this pedagogical example we have increased the probability for outliers in the data generation slightly from 0.05 to 0.2.

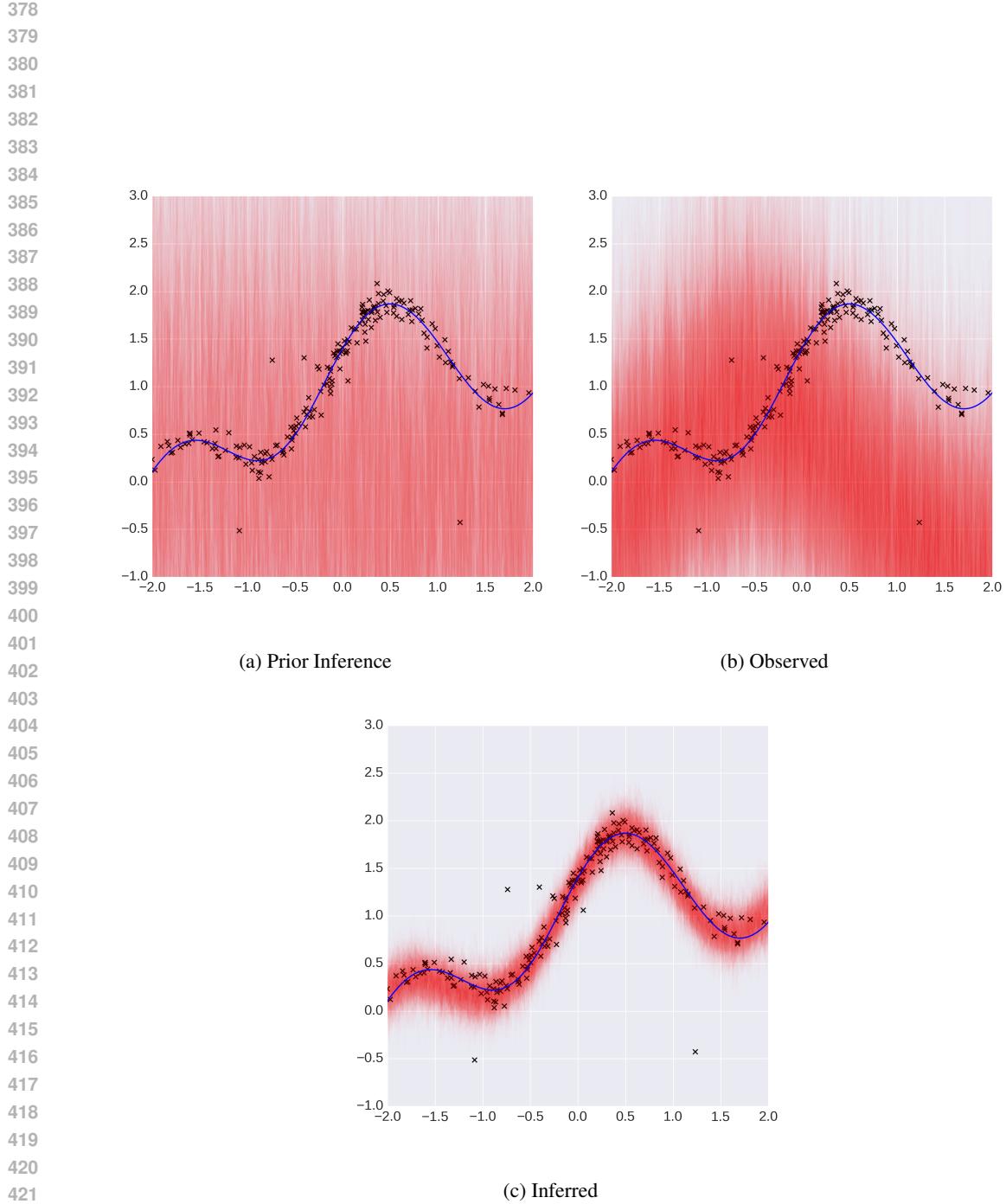


Figure 2: (a)-(c) shows `gpmem` on Neal’s example. We see that prior renders functions all over the place (a). After `gpmem` observes a some data-points an arbitrary smooth trend with a high level of noise is sampled. After running inference on the hierarchical system of hyper-parameters we see that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

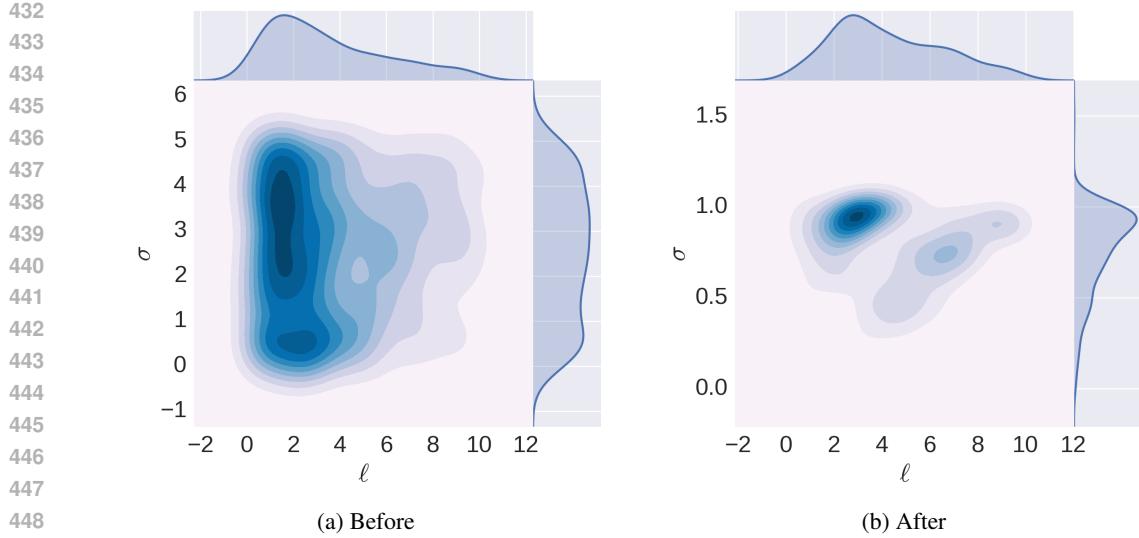


Figure 3: Hyper-parameter inference on the parameter of the noise kernel. We show 100 samples drawn from the distribution on σ . One can clearly recognise the shift from the uniform prior $\mathcal{U}(0, 5)$ to a double peak distribution around the two modes - normal and outlier.

3.2.3 Broader applicability of gpmem

More generally, gpmem is relevant not just when a data set is available, but also whenever we have at hand a function f_{restr} which is expensive or impractical to evaluate many times. gpmem allows us to model f_{restr} with a GP-based emulator f_{emu} , and also to use f_{emu} during the learning process to choose, in an online manner, an effective set of probe points $\{x_i\}$ on which to use our few evaluations of f_{restr} . This idea is illustrated in detail in Section 4. Before doing this, we will illustrate another benefit of having a probabilistic programming apparatus for GP modelling: the linguistically unified treatment of inference over structure and inference over parameters. This unification makes interleaved joint inference over structure and parameters very natural, and allows us to give a short, elegant description of what it means to ‘learn the covariance function,’ both in prose and in code. Furthermore, the example in Section 3.3 below recovers the performance of current state-of-the-art GP-based models.

3.3 Structure Learning

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 Automated Statistician Project 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 of this was not done before. To the best of our knowledge, we are the 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)

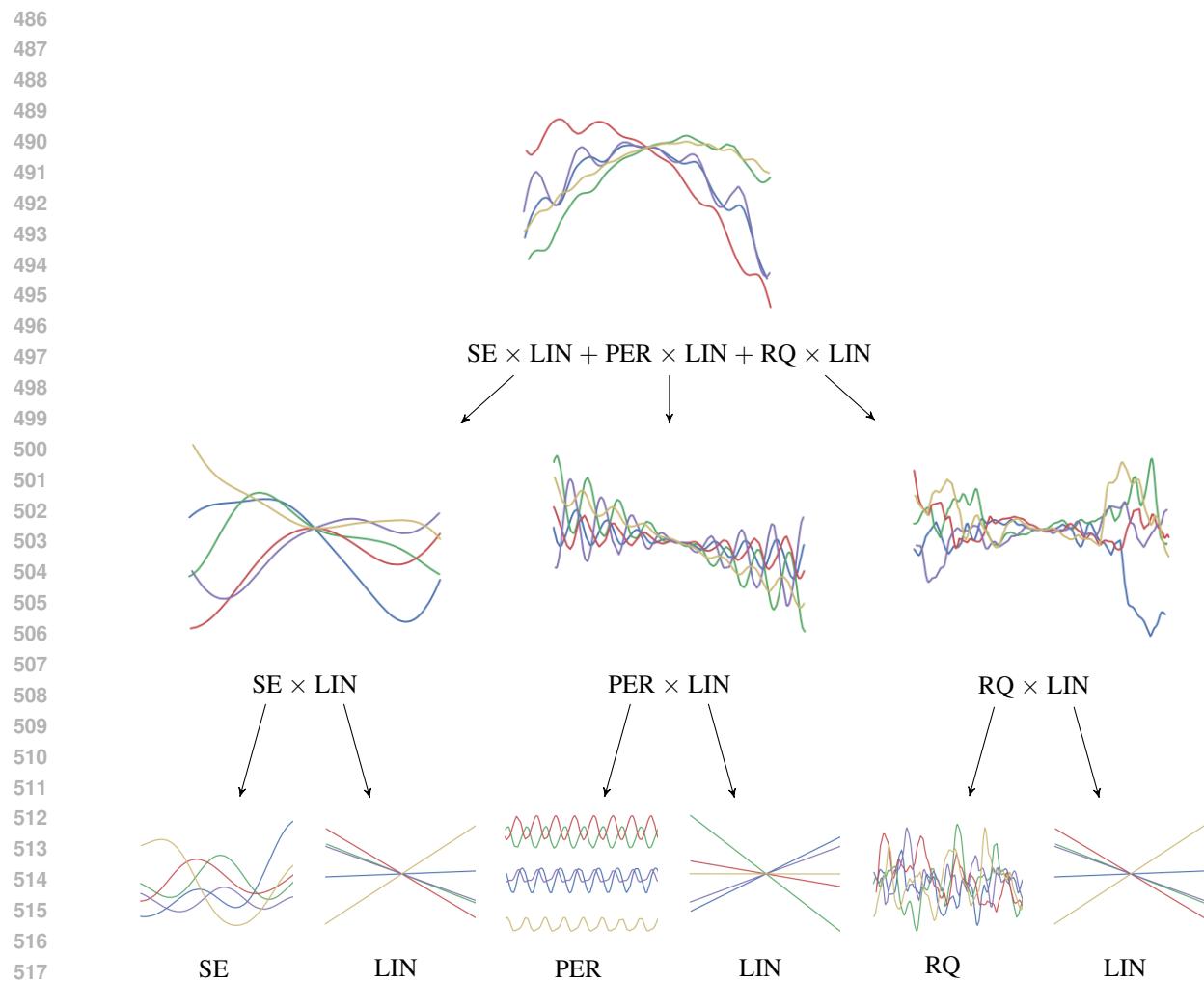


Figure 4: Composition of covariance function parsed using the example $\text{SE} \times \text{LIN} + \text{PER} \times \text{LIN} + \text{RQ} \times \text{LIN}$

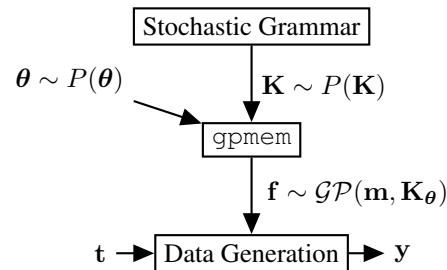


Figure 5: Graphical description of Bayesian GP structure learning.

540 Our probabilistic programming based MCMC framework approximates the following intractable
 541 integrals of the expectation for the prediction:
 542

$$546 \quad \mathbb{E}[y^* | x^*, \mathbf{D}, \mathbf{K}] = \iint f(x^*, \boldsymbol{\theta}, \mathbf{K}) P(\boldsymbol{\theta} | \mathbf{D}, \mathbf{K}) P(\mathbf{K} | \boldsymbol{\Omega}, s, n) d\boldsymbol{\theta} d\mathbf{K}. \quad (18)$$

551 This is done by sampling from the posterior probability distribution of the hyper-parameters and the
 552 possible kernel:
 553

$$554 \quad y^* \approx \frac{1}{T} \sum_{t=1}^T f(x^* | \boldsymbol{\theta}^{(t)}, \mathbf{K}^{(t)}). \quad (19)$$

556 In order to provide the sampling of the kernel, we introduce a stochastic process to the SP that
 557 simulates the grammar for algebraic expressions of covariance function algebra:
 558

$$561 \quad \mathbf{K}^{(t)} \sim P(\mathbf{K} | \boldsymbol{\Omega}, s, n) \quad (20)$$

566 Here, we start with a set of possible kernels and draw a random subset. For this subset of size n , we
 567 sample a set of possible operators that operate on the base kernels.
 568

569 The marginal probability of a kernel structure which allows us to sample is characterized by the
 570 probability of a uniformly chosen subset of the set of n possible covariance functions times the
 571 probability of sampling a global or a local structure which is given by a binomial distribution:
 572

$$573 \quad P(\mathbf{K} | \boldsymbol{\Omega}, s, n) = P(\boldsymbol{\Omega} | s, n) \times P(s | n) \times P(n), \quad (21)$$

574 with

$$575 \quad P(\boldsymbol{\Omega} | s, n) = \binom{n}{r} p_{+ \times}^k (1 - p_{+ \times})^{n-k} \quad (22)$$

576 and

$$577 \quad P(s | n) = \frac{n!}{|s|!} \quad (23)$$

579 where $P(n)$ is a prior on the number of base kernels used which can sample from a discrete uniform
 580 distribution. This will strongly prefer simple covariance structures with few base kernels since
 581 individual base kernels are more likely to be sampled in this case due to (23). Alternatively, we
 582 can approximate a uniform prior over structures by weighting $P(n)$ towards higher numbers. It is
 583 possible to also assign a prior for the probability to sample global or local structures, however, we
 584 have assigned complete uncertainty to this with the probability of a flip $p = 0.5$.
 585

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

592 For reproducing results from the Automated Statistician Project in a Bayesian fashion we first define
 593 a prior on the hypothesis space. Note that, as in the implementation of the Automated Statistician,
 594 we upper-bound the complexity of the space of covariance functions we want to explore. We also
 595 put vague priors on hyper-parameters.

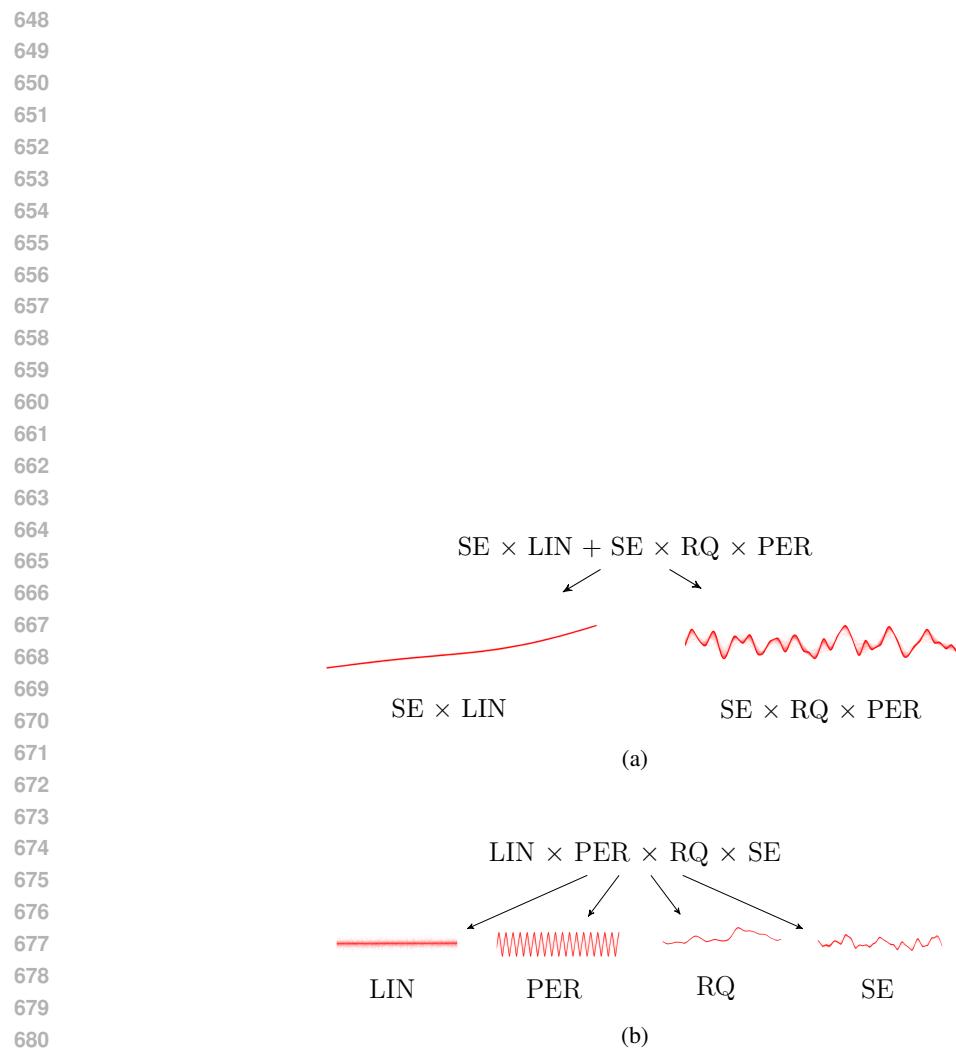
```

594 // GRAMMAR FOR KERNEL STRUCTURE
595 assume base_kernels = list(se, wn, lin, per, rq) // defined as above
596
597 // prior on the number of kernels
598 assume p_number_k = tag(quote(number_kernels), 0, uniform_structure(n))
599 assume s = tag(quote(choice_subset), 0, subset(base_kernels, p_number_k))
600
601 assume cov_compo = proc(l) {
602     // kernel composition
603     if (size(l) <= 1)
604         then { first(l) }
605     else { if (flip()) then { add_funcs(first(l), cov_compo(rest(l))) }
606             else { mult_funcs(first(l), cov_compo(rest(l))) }
607         }
608     }
609
610 assume K = tag(quote(composit), 0, cov_compo(s))
611
612 assume (f_compute f_emu) = gpmem(f_restr, K)
613 predict mapv(f_compute(get_data_xs)) // probe all data points
614
615 // PERFORMING INFERENCE
616 infer repeat(2000, do(
617     mh(quote(number_kernels), one, 1),
618     mh(quote(choice_subset), one, 1),
619     mh(quote(composit), one, 1),
620     for kernel ∈ K
621         mh(quote(hyperkernel), one, 10)))
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647

```

We defined the space of covariance structures in a way allowing us to reproduce results for covariance function structure learning as in the Automated 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 CO₂ 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 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 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



681
 682
 683
 684
 685
 686
 687
 688
 689
 690
 691
 692
 693
 694
 695
 696
 697
 698
 699
 700
 701

Figure 6: (a) most frequent sample drawn from the posterior on structure. We have found two global components. First, A smooth trend ($\text{LIN} \times \text{SE}$) 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.

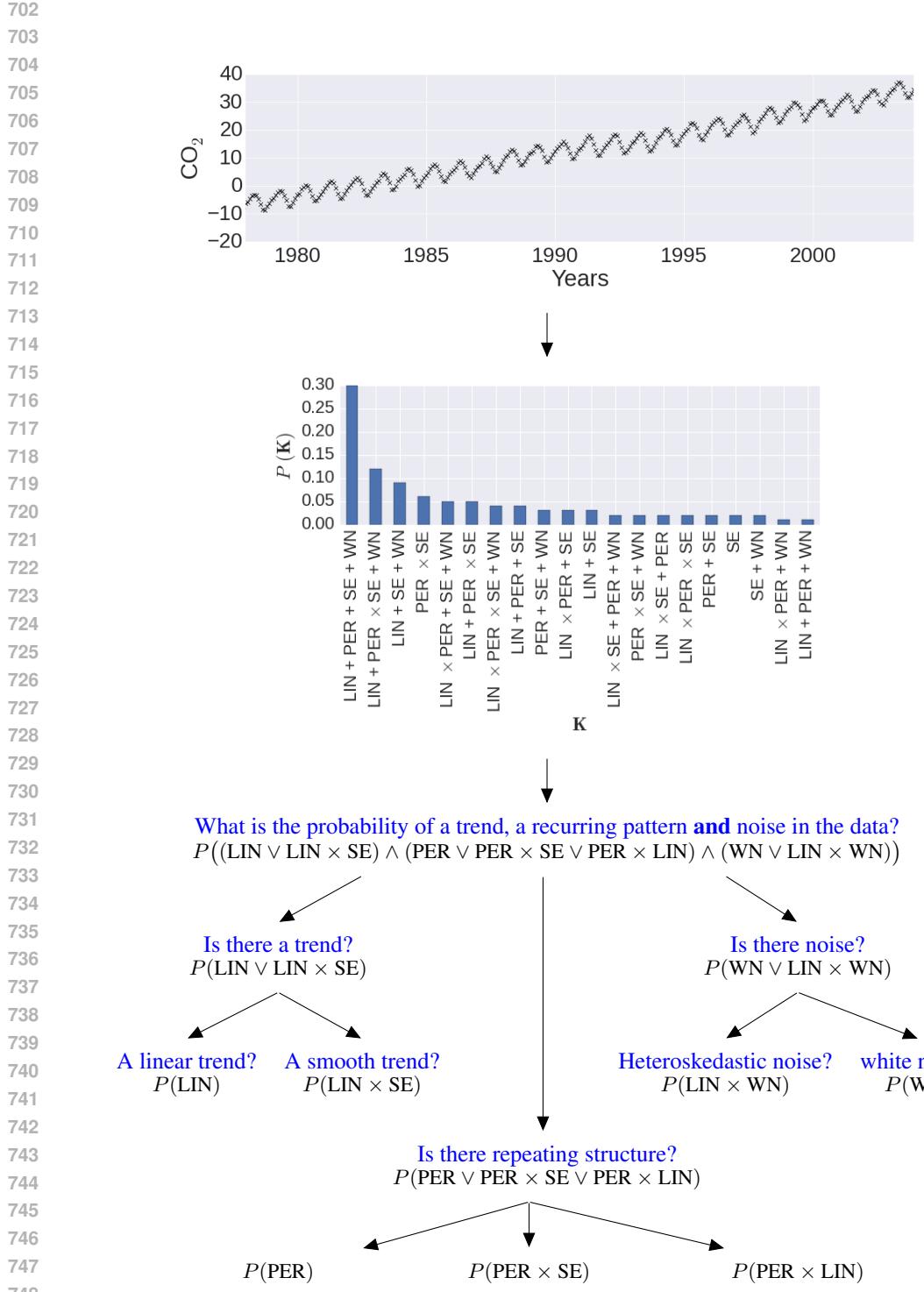


Figure 7: We can query the data, asking, similar to symbolic reasoning, if some statements are probable to be true. In this example, we are interested in three qualitative aspects of data, which we formulate in terms of the following: 1. Is it true that there is a trend? Is it true that there are recurring patterns and is it true that that there is noise in the data? Given the answers to these questions, we can then ask more specific questions.

756 **4 Bayesian Optimization**
757
758

759 Bayesian optimization casts the problem of finding the global maximum of an unknown function as
760 a hierarchical decision problem (Ghahramani, 2015). Evaluating the actual function may be very
761 expensive, either in computation time or in some other resource. For one example, when searching
762 for the best configuration for the learning algorithm of a large convolutional neural network, a large
763 amount of computational work is required to evaluate a candidate configuration, and the space of
764 possible configurations is high-dimensional. Another common example, alluded to in Section 3.2.3,
765 is data acquisition: for machine learning problems in which a large body of data is available, it is
766 often desirable to choose the right queries to produce a data set on which learning will be most
767 effective. In continuous settings, many Bayesian optimization methods employ GPs (e.g. Snoek
768 et al., 2012).

769 We have implemented a version of Thompson sampling using GPs in Venture. Thompson sam-
770 pling Thompson (1933) is a widely-used Bayesian framework for solving exploration-exploitation
771 problems. Our implementation has two notable features: (i) the ability to search over a broader
772 space of contexts than the parametric families that are typically used, and (ii) the parsimony of the
773 resulting probabilistic program.

774 **4.1 Thompson sampling framework**
775

776 We now lay out the setup of Thompson sampling for Markov decision processes (MDPs). An agent
777 is to take a sequence of actions a_1, a_2, \dots from a (possibly infinite) set of possible actions \mathcal{A} . After
778 each action, a reward $r \in \mathbb{R}$ is received, according to an unknown conditional distribution $P_{\text{true}}(r|a)$.
779 The agent's goal is to maximize the total reward received for all actions. In Thompson sampling,
780 the Bayesian agent accomplishes this by placing a prior distribution $P(\theta)$ on the possible "contexts"
781 $\theta \in \Theta$. Here a context is a believed model of the conditional distributions $\{P(r|a)\}_{a \in \mathcal{A}}$, or at least,
782 a believed statistic of these conditional distributions which is sufficient for deciding an action a .
783 One example of such a sufficient statistic is the conditional mean $V(a|\theta) = \mathbb{E}[r|a, \theta]$, which can be
784 thought of as a value function. Thompson sampling thus has the following steps, repeated as long
785 as desired:

- 786 1. Sample a context $\theta \sim P(\theta)$.
787
788 2. Choose an action $a \in \mathcal{A}$ which (approximately) maximizes $V(a|\theta) = \mathbb{E}[r|a, \theta]$.
789
790 3. Let r_{true} be the reward received for action a . Update the believed distribution on θ , i.e.,
791 $P(\theta) \leftarrow P_{\text{new}}(\theta)$ where $P_{\text{new}}(\theta) = P(\theta | a \mapsto r_{\text{true}})$.

792 Note that when $\mathbb{E}[r|a, \theta]$ (under the sampled value of θ for some points a) is far from the true value
793 $\mathbb{E}_{P_{\text{true}}}[r|a]$, the chosen action a may be far from optimal, but the information gained by probing
794 action a will improve the belief θ . This amounts to "exploration." When $\mathbb{E}[r|a, \theta]$ is close to the
795 true value except at points a for which $\mathbb{E}[r|a, \theta]$ is low, exploration will be less likely to occur, but
796 the chosen actions a will tend to receive high rewards. This amounts to "exploitation." Roughly
797 speaking, exploration will happen until the context θ is reasonably sure that the unexplored actions
798 are probably not optimal, at which time the sampler will exploit by choosing actions in regions it
799 knows to have high value.

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

810 **4.2 Thompson sampling in Venture**
 811

812 Because Venture supports sampling and inference on (stochastic-)procedure-valued random vari-
 813 ables (and the generative models which produce those procedures), Venture can capture arbitrary
 814 context spaces as described above. To demonstrate, we have implemented Thompson sampling
 815 in Venture in which the contexts θ are Gaussian processes over the action space $\mathcal{A} = \mathbb{R}$. That
 816 is, $\theta = (\mu, K)$, where the mean μ is a computable function $\mathcal{A} \rightarrow \mathbb{R}$ and the covariance K is
 817 a computable (symmetric, positive-semidefinite) function $\mathcal{A} \times \mathcal{A} \rightarrow \mathbb{R}$. This represents a Gaus-
 818 sian process $\{R_a\}_{a \in \mathcal{A}}$, where R_a represents the reward for action a . Computationally, we rep-
 819 represent a context not as a pair of infinite lookup tables for μ and K , but as a finite data structure
 820 $\theta = (K_{\text{prior}}, \sigma, \ell, \mathbf{a}_{\text{past}}, \mathbf{r}_{\text{past}})$, where
 821

- 822 • $K_{\text{prior}} = K_{\text{prior}, \sigma, \ell}$ is a procedure, with parameters σ, ℓ , to be used as the prior covariance
 823 function: $K_{\text{prior}}(a, a') = \sigma^2 \exp\left(-\frac{(a-a')^2}{2\ell^2}\right)$
 824
- 825 • σ and ℓ are (hyper)parameters for K_{prior}
- 826 • $\mathbf{a}_{\text{past}} = (a_i)_{i=1}^n$ are the previously probed actions
- 827 • $\mathbf{r}_{\text{past}} = (r_i)_{i=1}^n$ are the corresponding rewards

834 To simplify the treatment, we take prior mean $\mu_{\text{prior}} \equiv 0$. The mean and covariance for θ are then
 835 gotten by the usual conditioning formula:

836
 837

$$\begin{aligned} 838 \quad \mu(\mathbf{a}) &= \mu(\mathbf{a} \mid \mathbf{a}_{\text{past}}, \mathbf{r}_{\text{past}}) \\ 839 &= K_{\text{prior}}(\mathbf{a}, \mathbf{a}_{\text{past}}) K_{\text{prior}}(\mathbf{a}_{\text{past}}, \mathbf{a}_{\text{past}})^{-1} \mathbf{r}_{\text{past}} \\ 840 \quad K(\mathbf{a}, \mathbf{a}) &= K(\mathbf{a}, \mathbf{a} \mid \mathbf{a}_{\text{past}}, \mathbf{r}_{\text{past}}) \\ 841 &= K_{\text{prior}}(\mathbf{a}, \mathbf{a}) - K_{\text{prior}}(\mathbf{a}, \mathbf{a}_{\text{past}}) K_{\text{prior}}(\mathbf{a}_{\text{past}}, \mathbf{a}_{\text{past}})^{-1} K_{\text{prior}}(\mathbf{a}_{\text{past}}, \mathbf{a}). \\ 842 \\ 843 \\ 844 \end{aligned}$$

845 Note that even in this simple example, the context space Θ is not a finite-dimensional parametric
 846 family, since the vectors \mathbf{a}_{past} and \mathbf{r}_{past} grow as more samples are taken. Θ is, however, quite easily
 847 representable as a computational procedure together with parameters and past samples, as we do in
 848 the representation $\theta = (K_{\text{prior}}, \sigma, \ell, \mathbf{a}_{\text{past}}, \mathbf{r}_{\text{past}})$.

849
 850 **4.3 Implementation with gpmem**
 851

852 As a demonstration, we use Thompson sampling to optimize an unknown function $V(x)$ (the value
 853 function) using gpmem. We assume V is made available to Venture as a black-box. The code for
 854 optimizing V is given in Listing 1. For step 3 of Thompson sampling, the Bayesian update, we not
 855 only condition on the new data (the chosen action a and the received reward r), but also perform
 856 inference on the hyperparameters σ, ℓ using a Metropolis–Hastings sampler. These two inference
 857 steps take 1 line of code: 0 lines to condition on the new data (as this is done automatically by
 858 gpmem), and 1 line to call Venture’s built-in MH operator. The results are shown in Figure 8. We can
 859 see from the figure that, roughly speaking, each successive probe point a is chosen either because
 860 the current model V_{emu} thinks it will have a high reward, or because the value of $V_{\text{emu}}(a)$ has high
 861 uncertainty. In the latter case, probing at a decreases this uncertainty and, due to the smoothing
 862 kernel, also decreases the uncertainty at points near a . We thus see that our Thompson sampler
 863 simultaneously learns the value function and optimizes it.

864

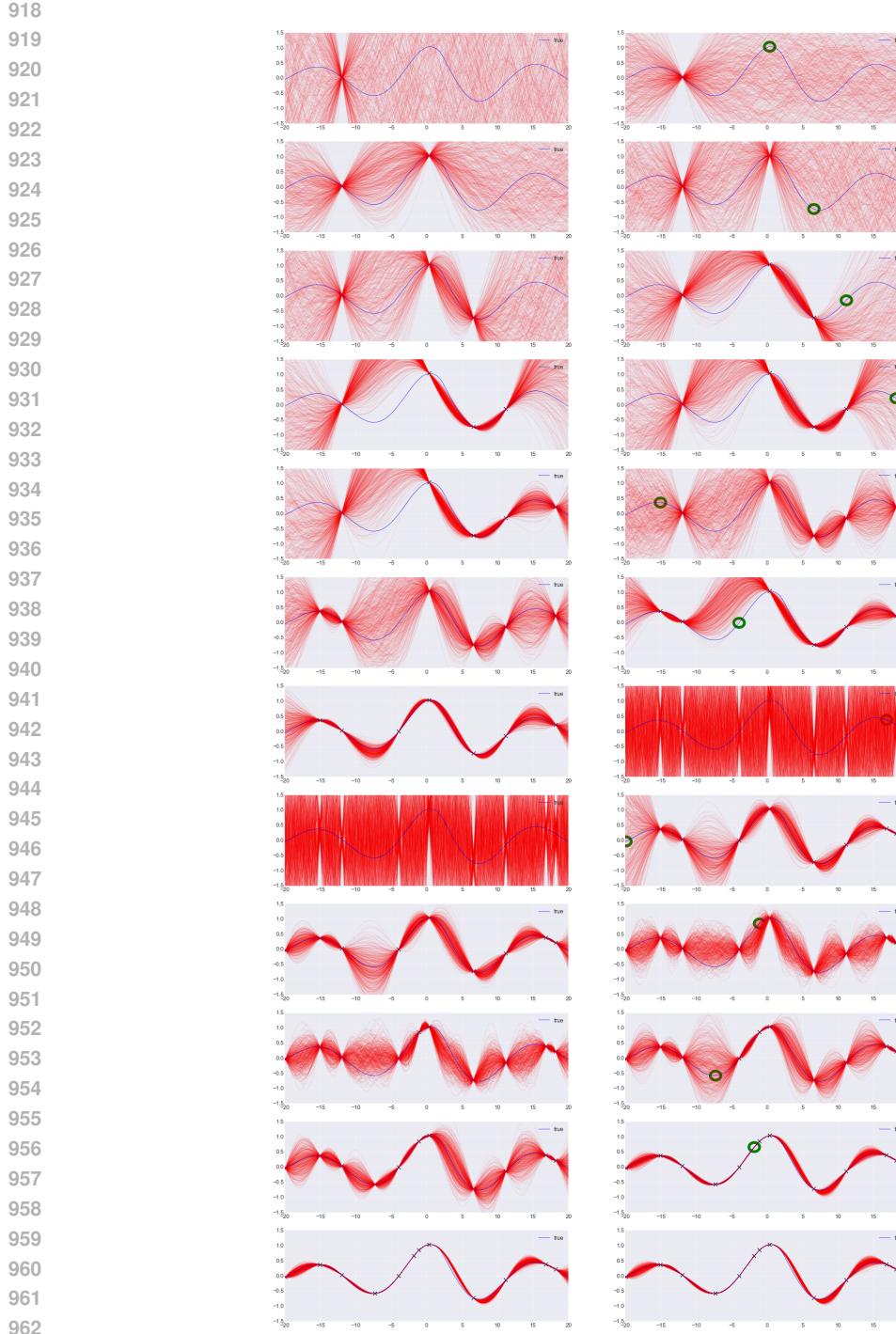
Listing 1: Code for Bayesian optimization using `gpmem`. In the loop, `V_compute` is called to probe the value of `V` at a new argument. The new argument, `(mc_argmax V_emu_pointwise mc_sampler)`, is a Monte Carlo estimate of the maximum pointwise sample of `V_emu` (itself a stochastic quantity), with the Monte Carlo samples being drawn in this case uniformly between -20 and 20. After each new call to `V_compute`, the Metropolis–Hastings algorithm is used to perform inference on the hyperparameters of the covariance function in the GP model in light of the new conditioning data. Once enough calls to `V_compute` have been made (in our case we stopped at 15 calls), we can inspect the full list of probed (a, r) pairs with `extract_stats`. The answer to our maximization problem is simply the pair having the highest r ; but our algorithm also learns more potentially useful information.

```

874 assume sf = tag(quote(hyper), 0, uniform_continuous(0, 10))
875 assume l = tag(quote(hyper), 1, uniform_continuous(0, 10))
876 assume se = make_squaredexp(sf, l)
877 assume blackbox_f = get_bayesopt_blackbox()
878 assume (f_compute, f_emu) = gpmem(blackbox_f, se)

879 define get_uniform_candidate = proc(prev_xs) {
880   uniform_continuous(-20, 20)
881 }
882
883 define mc_argmax = proc(func, prev_xs) {
884   // Monte Carlo estimator for the argmax of func.
885   run(do(
886     candidate_xs <- mapv(proc(i) {get_uniform_candidate(prev_xs)},
887                             linspace(0, 19, 20)),
888     candidate_ys <- mapv(func, candidate_xs),
889     lookup(candidate_xs, argmax_of_array(candidate_ys))))
890 }
891
892 define emulator_point_sample = proc(x) {
893   run(sample(lookup(
894     f_emu(array(x)),
895     0)))
896 }
897
898 infer repeat(15, do(pass,
899   // Phase 2: Call f_compute on the next probe point
900   predict f_compute(
901     mc_argmax(emulator_point_sample, '_)),
902   // Phase 1: Hyperparameter inference
903   mh(quote(hyper), one, 50)))
904
905
906
907
908
909
910
911
912
913
914
915
916
917

```



963
964
965
966
967
968
969
970
971
Figure 8: Dynamics of Thompson sampling in Venture. The blue curve is the true function V , and the red region is a blending of 100 samples of the curve generated (jointly) by a GP-based emulator V_{emu} . The left and right columns show the state of V_{emu} before and after hyperparameter inference is run on the new data, respectively. (We can see, for example, that after the seventh probe point, the Metropolis–Hastings sampler chose a “crazy” set of hyperparameters, which was corrected at the next inference step.) In the right column, the next chosen probe point is circled in green. Each successive probe point a is the (stochastic) maximum of V_{emu} , sampled pointwise and conditioned on the values of the previously probed points. Note that probes tend to happen at points either where the value of V_{emu} is high, or where V_{emu} has high uncertainty.

972 **5 Conclusion**
 973

974 We have shown Venture GPs. We have introduced novel stochastic processes for a probabilistic
 975 programming language. We showed how flexible non-parametric models can be treated in Venture
 976 in only a few lines of code. We evaluated our contribution on a range of hard problems for state-of-
 977 the-art Bayesian non-parametrics. Venture GPs showed competitive performance in all of them.
 978

979 **Appendix**
 980

981 **A Covariance Functions**
 982

983 SE and WN are defined in the text above, for completeness we will introduce the covariance:
 984

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

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

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

992 **B Covariance Simplification**
 993

994 SE × SE	→ SE
995 {SE, PER, C, WN} × WN	→ WN
996 LIN + LIN	→ LIN
997 {SE, PER, C, WN, LIN} × C	→ {SE, PER, C, WN, LIN}

998 Rule 1 is derived as follows:
 999

$$\begin{aligned} 1000 \quad \sigma_c^2 \exp\left(-\frac{(x - x')^2}{2\ell_c^2}\right) &= \sigma_a^2 \exp\left(-\frac{(x - x')^2}{2\ell_a^2}\right) \times \sigma_b^2 \exp\left(-\frac{(x - x')^2}{2\ell_b^2}\right) \\ 1001 \quad &= \sigma_c^2 \exp\left(-\frac{(x - x')^2}{2\ell_a^2}\right) \times \exp\left(-\frac{(x - x')^2}{2\ell_b^2}\right) \\ 1002 \quad &= \sigma_c^2 \exp\left(-\frac{(x - x')^2}{2\ell_a^2} - \frac{(x - x')^2}{2\ell_b^2}\right) \\ 1003 \quad &= \sigma_c^2 \exp\left(-\frac{(x - x')^2}{2\ell_c^2}\right) \end{aligned} \quad (27)$$

1004 Rule 3 is derived as follows:
 1005

$$\theta_c(x \times x') = \theta_a(x \times x') + \theta_b(x \times x') \quad (28)$$

1006 For stationary kernels that only depend on the lag vector between x and x' it holds that multiplying
 1007 such a kernel with a WN kernel we get another WN kernel. Take for example the SE kernel:
 1008

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

1009 Multiplying any kernel with a constant obviously changes only the scale parameter of a kernel.
 1010

1011
 1012
 1013
 1014
 1015
 1016
 1017
 1018
 1019
 1020
 1021
 1022
 1023
 1024
 1025

1026 **References**
1027

- 1028 Andrieu, C., De Freitas, N., Doucet, A., and Jordan, M. I. (2003). An introduction to MCMC for
1029 machine learning. *Machine learning*, 50(1-2):5–43.
1030 Box, G. E. P., Jenkins, G. M., and Reinsel, G. C. (1997). *Time series analysis: forecasting and*
1031 *control*.
1032 Duvenaud, D., Lloyd, J. R., Grosse, R., Tenenbaum, J., and Ghahramani, Z. (2013). Structure
1033 discovery in nonparametric regression through compositional kernel search. In *Proceedings of*
1034 *the 30th International Conference on Machine Learning (ICML-13)*, pages 1166–1174.
1035 Ghahramani, Z. (2015). Probabilistic machine learning and artificial intelligence. *Nature*,
1036 521(7553):452–459.
1037 Goodman, N. D .and Mansinghka, V. K., Roy, D., Bonawitz, K., and Tenenbaum, J. (2008). Church:
1038 A language for generative models. In *Proceedings of the 24th Conference on Uncertainty in*
1039 *Artificial Intelligence, UAI 2008*, pages 220–229.
1040 Lloyd, J. R., Duvenaud, D., Grosse, R., Tenenbaum, J., and Ghahramani, Z. (2014). Automatic
1041 construction and natural-language description of nonparametric regression models. In *Twenty-*
1042 *Eighth AAAI Conference on Artificial Intelligence*.
1043 Mansinghka, V. K., Selsam, D., and Perov, Y. (2014). Venture: a higher-order probabilistic pro-
1044 gramming platform with programmable inference. *arXiv preprint arXiv:1404.0099*.
1045 McAllester, D., Milch, B., and Goodman, N. D. (2008). Random-world semantics and syntactic
1046 independence for expressive languages. Technical report.
1047 Metropolis, N., Rosenbluth, A. W., Rosenbluth, M. N., Teller, A. H., and Teller, E. (1953). Equation
1048 of state calculations by fast computing machines. *The Journal of Chemical Physics*, 21(6):1087–
1049 1092.
1050 Neal, R. M. (1997). Monte carlo implementation of gaussian process models for bayesian regression
1051 and classification. *arXiv preprint physics/9701026*.
1052 Poole, D. (1993). Probabilistic horn abduction and bayesian networks. *Artificial Intelligence*,
1053 64(1):81–129.
1054 Rasmussen, C. E. and Williams, C. K. I. (2006). *Gaussian Processes for Machine Learning (Adap-*
1055 *tive Computation and Machine Learning)*. The MIT Press.
1056 Sato, T. (1995). A statistical learning method for logic programs with distribution semantics. In *In*
1057 *Proceedings of the 12th International Conference on Logic Programming*. Citeseer.
1058 Snoek, J., Larochelle, H., and Adams, R. P. (2012). Practical bayesian optimization of machine
1059 learning algorithms. In *Advances in Neural Information Processing Systems*, pages 2951–2959.
1060 Thompson, W. R. (1933). On the likelihood that one unknown probability exceeds another in view
1061 of the evidence of two samples. *Biometrika*, pages 285–294.
1062
1063

1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079