
Probabilistic Programming with Gaussian Process Memoization

000
001
002
003
004
005
006
007
008
009
010
011
012
013
014
015
016
017
018
019
020
021
022
023
024
025
026
027
028
029
030
031
032
033
034
035
036
037
038
039
040
041
042
043
044
045
046
047
048
049
050
051
052
053
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

Gaussian Processes (GP) are widely used tools in statistics (Barry, 1986), machine learning (Neal, 1995; Williams and Barber, 1998; Kuss and Rasmussen, 2005; Rasmussen and Williams, 2006; Damianou and Lawrence, 2013), robotics (Ferris et al., 2006), computer vision (Kemmler et al., 2013), and scientific computation (Kennedy and O’Hagan, 2001; Schneider et al., 2008; Kwan et al., 2013). They are also central to probabilistic numerics, an emerging effort to develop more computationally efficient numerical procedures, and to Bayesian optimization, a family of meta-optimization techniques that are widely used to tune parameters for deep learning algorithms (Snoek et al., 2012; Gelbart et al., 2014). They have even seen use in artificial intelligence; for example, they provide the key technology behind a project that produces qualitative natural language descriptions of time series (Duvenaud et al., 2013; Lloyd et al., 2014).

This paper describes Gaussian process memoization, a technique for integrating GPs into a probabilistic programming language, and demonstrates its utility by re-implementing and extending state-of-the-art applications of the GP. Memoization, typically implemented by a procedure called `mem()`, is a classic higher-order programming technique in which a procedure is augmented with an input-output cache that is checked each time before the function is invoked. This prevents unnecessary recomputation, potentially saving time at the cost of increased storage requirements. Gaussian process memoization, implemented by the `gpmem()` procedure, generalizes this idea to include a statistical emulator that uses previously computed values as data in a statistical model that can cheaply forecast probable outputs. The covariance function for the Gaussian process is also allowed to be an arbitrary probabilistic program.

This paper presents three applications of `gpmem`: (i) a replication of (Neal, 1997) results on outlier rejection via hyper-parameter inference; (ii) a fully Bayesian extension to the Automated Statis-

054 tician project; and (iii) an implementation of Bayesian optimization via Thompson sampling. The
 055 first application can in principle be replicated in several other probabilistic languages embedding the
 056 proposal that is described in this paper. The remaining two applications rely on distinctive capabili-
 057 ties of Venture: support for fully Bayesian structure learning and language constructs for inference
 058 programming. All applications share a single 50-line Python library and require fewer than 20 lines
 059 of probabilistic code each.

061 2 Background on Gaussian Processes

063 Let the data be pairs of real-valued scalars $\{(x_i, y_i)\}_{i=1}^n$ (complete data will be denoted by column
 064 vectors \mathbf{x}, \mathbf{y}). GPs present a non-parametric way to express prior knowledge on the space of possible
 065 functions f modeling a regression relationship. Formally, a GP is an infinite-dimensional extension
 066 of the multivariate Gaussian distribution. Often one assumes the values \mathbf{y} are noisily measured, that
 067 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_{noise}^2 .
 068 In that case, the log-likelihood of a GP is

$$069 \log p(\mathbf{y}_{\text{noisy}} | \mathbf{x}) = -\frac{1}{2}\mathbf{y}^\top(\Sigma + \sigma_{\text{noise}}^2\mathbf{I})^{-1}\mathbf{y} - \frac{1}{2}\log|\Sigma + \sigma_{\text{noise}}^2\mathbf{I}| - \frac{n}{2}\log 2\pi \quad (1)$$

070 where n is the number of data points. Both log-likelihood and predictive posterior can be computed
 071 efficiently in a Venture SP with an algorithm that resorts to Cholesky factorization(Rasmussen and
 072 Williams, 2006, chap. 2) resulting in a computational complexity of $\mathcal{O}(n^3)$ in the number of data
 073 points.
 074

075 The covariance function (or kernel) of a GP governs high-level properties of the observed data such
 076 as linearity, periodicity and smoothness. It comes with few free parameters that we call hyper-
 077 parameters. Adjusting these results in minor changes, for example with regards to when two data
 078 points are treated similar. More drastically different covariance functions are achieved by changing
 079 the structure of the covariance function itself. Note that covariance function structures are compo-
 080 sitional: adding or multiplying two valid covariance functions results in another valid covariance
 081 function.
 082

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

$$087 \text{SE}(x, x') = \sigma^2 \exp\left(\frac{(x - x')^2}{2\ell}\right)$$

088 with $\sigma = 1$ and $\ell = 1$, can be instantiated as follows:
 089

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

093 There are two ways two view `g` as a “random function.” In the first view, the `assume` directive that
 094 instantiates `g` does not use any randomness—only the subsequent calls to `g` do—and coherence
 095 constraints are upheld by the interpreter by keeping track of which evaluations of `g` exist in the current
 096 execution trace. Namely, if the current trace contains evaluations of `g` at the points x_1, \dots, x_N with
 097 return values y_1, \dots, y_N , then the next evaluation of `g` (say, jointly at the points x_{N+1}, \dots, x_{N+n})
 098 will be distributed according to the joint conditional distribution
 099

$$100 P((g x_{N+1}), \dots, (g x_{N+n}) | (g x_i) = y_i \text{ for } i = 1, \dots, N).$$

101 In the second view, `g` is a randomly chosen deterministic function, chosen from the space of all
 102 deterministic real-valued functions; in this view, the `assume` directive contains *all* the randomness,
 103 and subsequent invocations of `g` are deterministic. The first view is procedural and is faithful to the
 104 computation that occurs behind the scenes in Venture. The second view is declarative and is faithful
 105 to notations like “ $g \sim P(g)$ ” which are often used in mathematical treatments. Because a model
 106 program could make arbitrarily many calls to `g`, and the joint distribution on the return values of the
 107

108 calls could have arbitrarily high entropy, it is not computationally possible in finite time to choose
 109 the entire function g all at once as in the second view. Thus, it stands to reason that any computa-
 110 tionally implementable notion of “nonparametric random functions” must involve incremental random
 111 choices in one way or another, and Gaussian processes in Venture are no exception.
 112

113 2.1 The `gpmem` construct

114 Memoization is the practice of storing previously computed values of a function so that future calls
 115 with the same inputs can be evaluated by lookup rather than recomputation. Although memoiza-
 116 tion does not change the semantics of a deterministic program, it does change that of a stochastic
 117 program (Goodman et al., 2008). The authors provide an intuitive example: let f be a function
 118 that flips a coin and return “head” or “tails”. The probability that two calls of f are equivalent is
 119 0.5. However, if the function call is memoized, it is 1. In fact, there is an infinite range of possible
 120 caching policies (specifications of when to use a stored value and when to recompute), each poten-
 121 tially having a different semantics. Any particular caching policy can be understood by random
 122 world semantics (Poole, 1993; Sato, 1995) over the stochastic program: each possible world corre-
 123 sponds to a mapping from function input sequence to function output sequence (McAllester et al.,
 124 2008). In Venture, these possible worlds are first-class objects, and correspond to the *probabilistic*
 125 *execution traces* (Mansinghka et al., 2014).

126 To transfer this idea to probabilistic programming, we now introduce a language construct called a
 127 *statistical memoizer*. Suppose we have a function f which can be evaluated but we wish to learn
 128 about the behavior of f using as few evaluations as possible. The statistical memoizer, which here
 129 we give the name `gpmem`, was motivated by this purpose. It produces two outputs:
 130

$$f \xrightarrow{\text{gpmem}} (f_{\text{probe}}, f_{\text{emu}}).$$

131 The function f_{probe} calls f and stores the output in a memo table, just as traditional memoization
 132 does. The function f_{emu} is an online statistical emulator which uses the memo table as its training
 133 data. A fully Bayesian emulator, modelling the true function f as a random function $f \sim P(f)$,
 134 would satisfy
 135

$$(f_{\text{emu}} x_1 \dots x_k) \sim P(f(x_1), \dots, f(x_k) \mid f(x) = (f x) \text{ for each } x \text{ in memo table}).$$

136 Different implementations of the statistical memoizer can have different prior distributions $P(f)$; in
 137 this paper, we deploy a Gaussian process prior (implemented as `gpmem` below). Note that we require
 138 the ability to sample f_{emu} jointly at multiple inputs because the values of $f(x_1), \dots, f(x_k)$ will in
 139 general be dependent. To illustrate the linguistic power of `gpmem` consider a simple model program
 140 that uses `gpmem` procedure abstraction, and a loop:
 141

```

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 // Hyper-parameter
8 assume theta = tag( quote( params ), 0, gamma(1,1))
9 assume (f_probe f_emu) = gpmem(f, make_squaredexp(1.0, theta))
10
11 for t...T:
12   f_probe( choose_next_point( f_emu ))
13   extract_answer( f_emu )
14
  
```

155 An equivalent model in typical statistics notation, written in a way that attempts to be as linguistically
 156 faithful as possible, is
 157

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

162 The linguistic constructs for abstraction are not present in statistics notation. The equation $x^{(t)} =$
 163 $\arg \max_x f^{(t)}(x)$ has to be inlined in statistics notation: while something like
 164

$$x^{(t)} \sim F(P^{(t)})$$

165 (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
 166 makes modularity, or the easy digestion of large but finely decomposable models, more difficult in
 167 statistics notation.
 168

169 Adding a single line to the program, such as
 170

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

172 after line 12 to infer the parameters of the emulator, would require a significant refactoring and
 173 elaboration of the statistics notation. One basic reason is that probabilistic programs are written pro-
 174 cedurally, whereas statistical notation is declarative; reasoning declaratively about the dynamics of a
 175 fundamentally procedural inference algorithm is often unwieldy due to the absence of programming
 176 constructs such as loops and mutable state.
 177

178 We implement `gpmem` by memoizing a target procedure in a wrapper that remembers previously
 179 computed values. This comes with interesting implications: from the standpoint of computation, a
 180 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
 181 to only allow evaluation at a specific set of inputs x (Alg. 1). Modelling the data set with a GP then
 182

Algorithm 1 Restricted Function

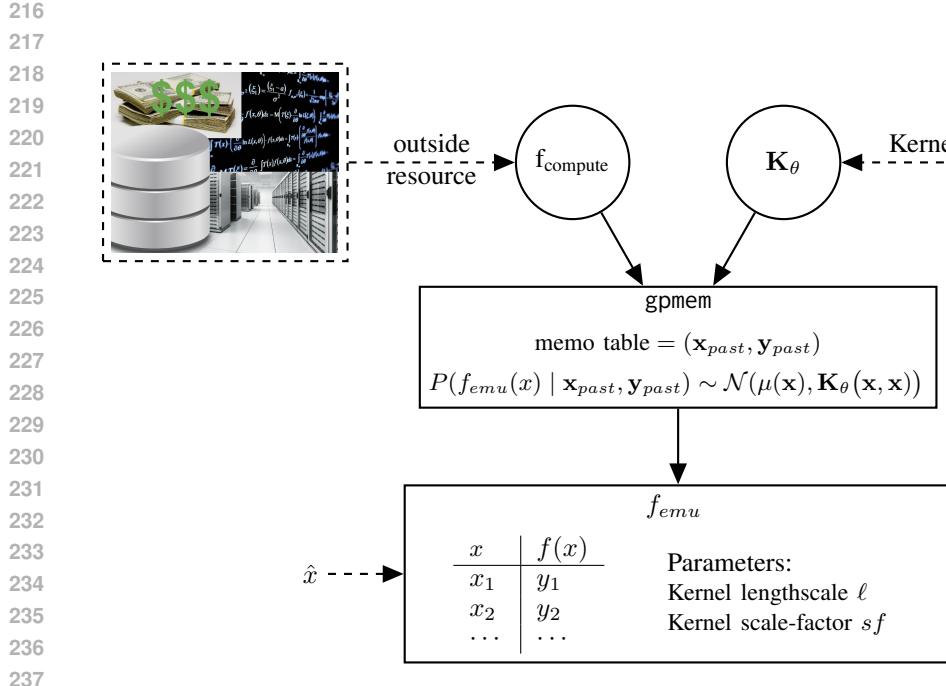
```
1: function  $f_{\text{RESTR}}(x)$ 
2:   if  $x \in \mathcal{D}$  then
3:     return  $\mathcal{D}[x]$ 
4:   else
5:     raise Exception('Illegal input')
6:   end if
7: end function
```

196
 197 amounts to trying to learn a smooth function f_{emu} ("emu" stands for "emulator") which extends f
 198 to its full domain. Indeed, if f_{restr} is a foreign procedure made available as a black-box to Venture,
 199 whose secret underlying pseudo code is in Alg. 1, then the `observe` code can be rewritten using
 200 `gpmem` as in Listing 1 (where here the data set D has keys $x[1], \dots, x[n]$):
 201

Listing 1: Observation with `gpmem`

```
202
203
204 1  assume (f_compute f_emu) = gpmem( f_restr )
205 2  for i=1 to n:
206 3    predict f_compute( x[i] )
207 4    infer mh(quote(hyper-parameters), one, 100)
208 5    sample (f_emu( array( 1, 2, 3 ))
```

210
 211 This rewriting has at least two benefits: (i) readability (in some cases), and (ii) amenability to active
 212 learning. As to (i), the statistical code of creating a Gaussian process is replaced with a memoization-
 213 like idiom, which will be more familiar to programmers. As to (ii), when using `gpmem`, it is quite
 214 easy to decide incrementally which data point to sample next: for example, the loop from $x[1]$ to
 215 $x[n]$ 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.



```

239 define f = proc( x ) {
240     exp(-0.1*abs(x-2)) * 
241     10* cos(0.4*x) + 0.2
242 }
243 assume (f_compute f_emu) = gpmem( f, K_theta )
244 sample f_emu( array( -20, ..., 20))

245
246 predict f_compute( 12.6)
247
248 sample f_emu( array( -20, ..., 20))

249
250
251 predict f_compute( -6.4)
252
253 sample f_emu( array( -20, ..., 20))

254
255 observe f_emu( -3.1) = 2.60
256 observe f_emu( 7.8) = -7.60
257 observe f_emu( 0.0) = 10.19
258
259 sample f_emu( array( -20, ..., 20))

260
261
262 infer mh(quote(hyper-parameter), one, 50)
263
264 sample f_emu( array( -20, ..., 20))
265
266
267
268
269

```

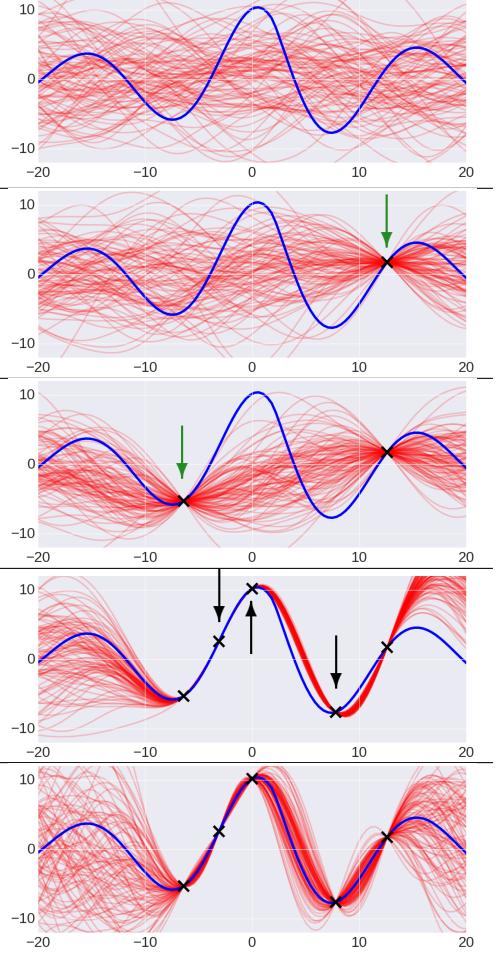
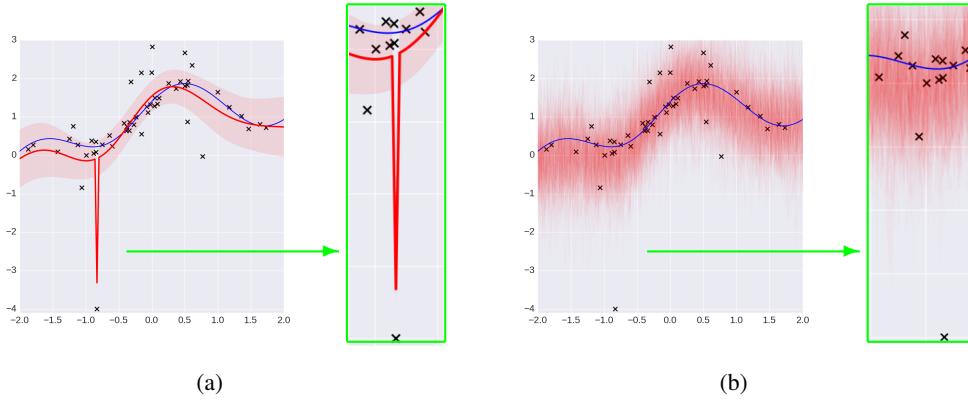


Figure 1: gpmem tutorial. The top shows a schematic of gpmem. $f_{com} = f_{compute}$ probes an outside resource. This can be expensive (top left). Every probe is memoized and improves the GP-based emulator. Below the schematic we see a movie of the evolution of gpmem's state of believe of the world given certain Venture directives.



270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
Figure 2: (a) shows deterministic inference, that is optimization for an example of Neal's synthetic function with an extreme outlier (green).
Figure 2: (b) shows the posterior distribution learned by gpmem, which exhibits a bimodal histogram when sampling 100 times reflecting the two modes of data generation.

3 Applications

gpmem is an elegant linguistic framework for function learning-related tasks. The technique allows language constructs from programming to help express models which would be cumbersome to express in statistics notation. We will now illustrate this with three example applications.

3.1 Nonlinear regression in the presence of outliers

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

$$P(\boldsymbol{\theta} | \mathbf{D}, \mathbf{K}) = \frac{P(\mathbf{D} | \boldsymbol{\theta}, \mathbf{K})P(\boldsymbol{\theta} | \mathbf{K})}{P(\mathbf{D} | \mathbf{K})}. \quad (2)$$

Let the \mathbf{K} be the sum of a smoothing and a white noise (WN) kernel. For this case, Neal (1997) suggested the problem of outliers in data as a use-case for a hierarchical Bayesian treatment of Gaussian processes¹. The work suggests a hierarchical system of hyper-parameterization. Here, we draw hyper-parameters from a Γ distributions:

$$\ell^{(t)} \sim \Gamma(\alpha_1, \beta_1), \sigma^{(t)} \sim \Gamma(\alpha_2, \beta_2) \quad (3)$$

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

$$\alpha_1^{(t)} \sim \Gamma(\alpha_\alpha^1, \beta_\alpha^1), \alpha_2^{(t)} \sim \Gamma(\alpha_\alpha^2, \beta_\alpha^2), \dots \quad (4)$$

One can represent this kind of model using gpmem (Fig. 3). 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:

$$f(x) = 0.3 + 0.4x + 0.5 \sin(2.7x) + \frac{1.1}{(1+x^2)} + \eta \quad \text{with } \eta \sim \mathcal{N}(0, \sigma) \quad (5)$$

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-parameters by showing a comparison between σ (Fig. 2). 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 outliers.

¹In (Neal, 1997) the sum of an SE plus a constant kernel is used. We keep the WN kernel for illustrative purposes.

```

324
325
326
327
328
329
330
331
332
333 // Data and look-up function
334 define data = array(array(-1.87,0.13),..., array(1.67,0.81))
335 assume f_look_up = proc(index) {lookup( data, index)}
336
337
338 assume sf = tag(quote(hyper), 0, gamma(alpha_sf, beta_sf)))
339 assume l = tag(quote(hyper), 1, gamma(alpha_l, beta_l)))
340 assume sigma = tag(quote(hyper), 2, uniform_continuous(0, 2))
341
342 // The covariance function
343 assume se = make_squaredexp(sf, l)
344 assume wn = make_whitenoise(sigma)
345 assume composite_covariance = add_funcs(se, wn)
346
347 // Create a prober and emulator using gpmem
348 assume (f_compute, f_emu)
349 = gpmem(f_look_up, composite_covariance)
350
351 sample f_emu( array( -2, ..., 2))
352
353 // Observe all data points
354 for n ... N
355 observe f_emu(first(lookup(data,n)))
356 = second(lookup(data,n))
357 // Or: probe all data points
358 for n ... N
359 predict f_compute(first(lookup(data,n)))
360
361 sample f_emu( array( -2, ..., 2))
362
363 // Metropolis-Hastings
364 infer repeat( 100, do(
365 mh( quote(hyperhyper), one, 2),
366 mh( quote(hyper), one, 1)))
367
368 sample f_emu( array( -2, ..., 2))
369
370
371 // Optimization
372
373 infer map( quote(hyper), all, 0.01, 15)
374
375 sample f_emu( array( -2, ..., 2))
376
377
```

Hyper-Parameters for the Kernel:

Figure 3: Hyper-parameterization

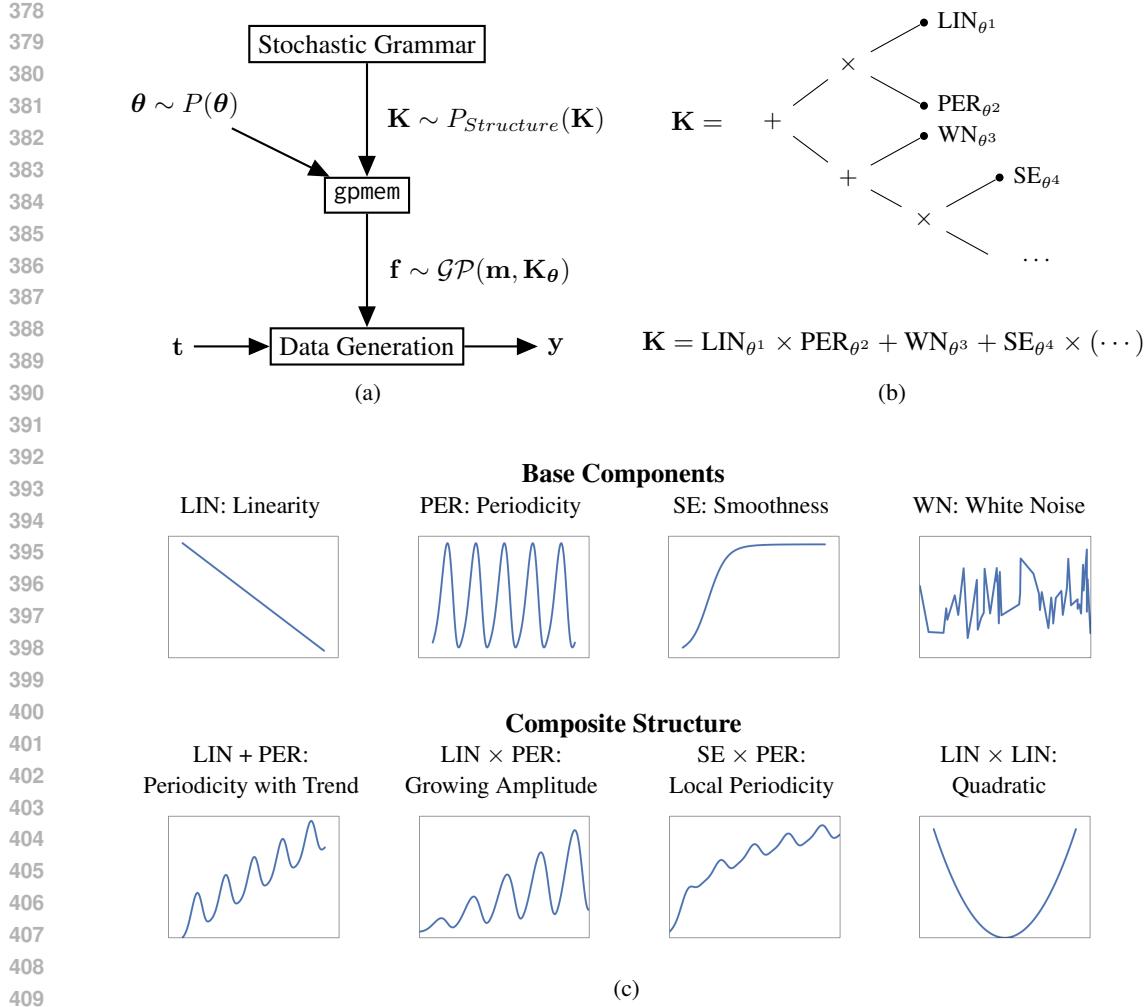


Figure 4: (a) Graphical description of Bayesian GP structure learning. (b) Composite structure. (c) The natural language interpretation of the structure.

3.2 Discovering qualitative structure from time series data

Inductive learning of symbolic expression for continuous-valued time series data is a hard task which has recently been tackled using a greedy search over the approximate posterior of the possible kernel compositions for GPs (Duvenaud et al., 2013; Lloyd et al., 2014)².

With gpmem we can provide a fully Bayesian treatment of this, previously unavailable, using a stochastic grammar (see Fig. 4).

We deploy a probabilistic context free grammar for our prior on structures. An input of non-composite kernels (base kernels) is supplied to generate a posterior distributions of composite structure to express local and global aspects of the data.

We approximate the following intractable integrals of the expectation for the prediction:

$$\mathbb{E}[y^* | x^*, \mathbf{D}, \mathbf{K}] = \iint f(x^*, \theta, \mathbf{K}) P(\theta | \mathbf{D}, \mathbf{K}) P(\mathbf{K} | \Omega, s, n) d\theta d\mathbf{K}. \quad (6)$$

²<http://www.automaticstatistician.com/>

432 This is done by sampling from the posterior probability distribution of the hyper-parameters and the
 433 possible kernel:
 434

$$435 \quad y^* \approx \frac{1}{T} \sum_{t=1}^T f(x^* | \boldsymbol{\theta}^{(t)}, \mathbf{K}^{(t)}). \quad (7)$$

437 In order to provide the sampling of the kernel, we introduce a stochastic process that simulates the
 438 grammar for algebraic expressions of covariance function algebra:
 439

$$440 \quad \mathbf{K}^{(t)} \sim P(\mathbf{K} | \boldsymbol{\Omega}, s, n) \quad (8)$$

442 Here, we start with the set of given base kernels and draw a random subset. For this subset of size
 443 n , we sample a set of possible operators $\boldsymbol{\Omega}$ combining base kernels. The marginal probability of a
 444 composite structure

$$445 \quad P(\mathbf{K} | \boldsymbol{\Omega}, s, n) = P(\boldsymbol{\Omega} | s, n) \times P(s | n) \times P(n), \quad (9)$$

446 is characterized by the prior $P(n)$ on the number of base kernels used, the probability of a uniformly
 447 chosen subset of the set of n possible covariance functions
 448

$$449 \quad P(s | n) = \frac{n!}{|s|!}, \quad (10)$$

451 and the probability of sampling a global or a local structure, which is given by a binomial distribu-
 452 tion:
 453

$$454 \quad P(\boldsymbol{\Omega} | s, n) = \binom{n}{r} p_{+ \times}^k (1 - p_{+ \times})^{n-k}. \quad (11)$$

456 Many equivalent covariance structures can be sampled due to covariance function algebra and equiv-
 457 alent representations with different parameterization (Lloyd et al., 2014). To inspect the posterior
 458 of these equivalent structures we convert each kernel expression into a sum of products and subse-
 459 quently simplify. All base kernels can be found in Appendix A, rules for this simplification can be
 460 found in appendix B. The code for learning of kernel structure is as follows:

461
 462
 463
 464
 465
 466
 467
 468
 469
 470
 471
 472
 473
 474
 475
 476
 477
 478
 479
 480
 481
 482
 483
 484
 485

```

486
487 // GRAMMAR FOR KERNEL STRUCTURE
488 1 assume kernels = list(se, wn, lin, per, rq) // defined as above
489
490 // prior on the number of kernels
491 2 assume p_number_k = uniform_structure(n)
492 3 assume sub_s = tag(quote(grammar), 0,
493                      subset(kernels, p_number_k))
494
495 5 assume grammar = proc(l) {
496     // kernel composition
497     if (size(l) <= 1)
498         { first(l) }
499     else { if (bernoulli())
500           { add_funcs(first(l), grammar(rest(l))) }
501           else { mult_funcs(first(l), grammar(rest(l))) }
502     }
503
504 13 assume K = tag(quote(grammar), 1, grammar(sub_s))
505
506 // Probe all data points
507 15 for n ... N
508     predict f_compute(get_data_xs(n))
509
510 // PERFORMING INFERENCE
511 17 infer repeat(2000, do(
512     mh(quote(grammar), one, 1),
513     for kernel ∈ K
514         mh(quote(hyperkernel), one, 1)))
515

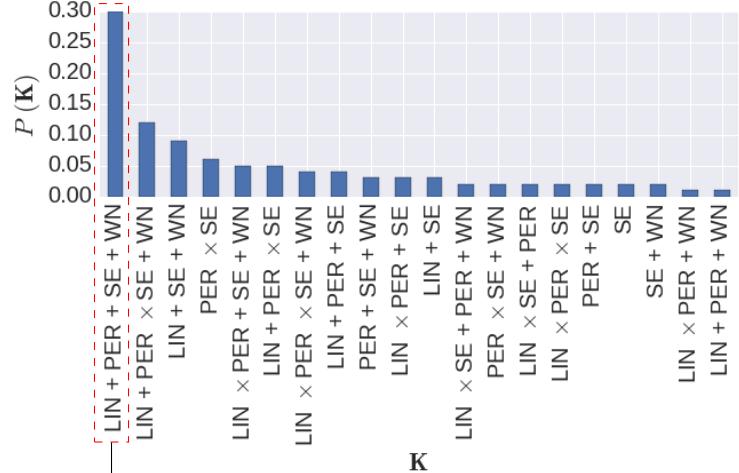
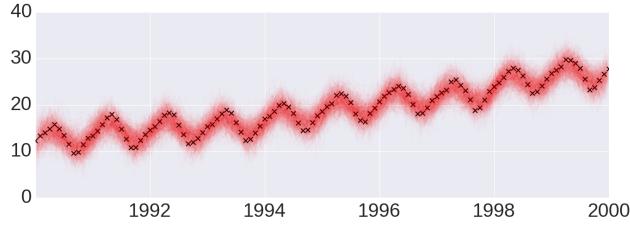
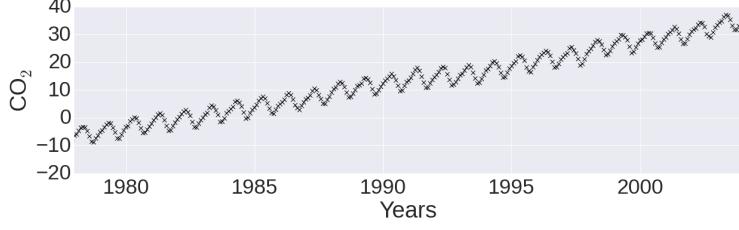
```

We defined the space of covariance structures in a way that allows us to produce results coherent with work presented in Automatic Statistician. For example, for the airline data set describing monthly totals of international airline passengers (Box et al., 1997, according to Duvenaud et al., 2013). Our most frequent 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 the most probable alternatives for a structural description both recover the data dynamics (Fig. 12 for the airline data set).

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 (see Fig. 5). We can also query this posterior distribution for the marginal of certain simple structures to occur. We demonstrate this in Fig. 6

3.3 Bayesian optimization

In this section we present a probabilistic meta-program for Bayesian optimization, along with optimization routines implemented as concrete applications of the meta-program. Figure 7 shows a probabilistic meta-program for Bayesian optimization. We use the term “meta-program” because this program defines a generic rule for combining abstract components to perform optimization. The components are themselves subprograms, whose implementations may vary. A traditional example of generic programming is the tree search meta-program of Norvig (1992), written in LISP, which we present alongside the optimization meta-program for comparison.



576
577
578
579
580
581
582

$$\boxed{\mathbf{K} = \text{LIN} + \text{PER} + \text{SE} + \text{WN}}$$

$$= 2.7^2(xx') + 5.6^2 \exp\left(\frac{2 \sin^2(\pi(x-x')/3.7)}{6.4^2}\right) + 0.4^2 \exp\left(-\frac{(x-x')^2}{2 \times 6.3^2}\right) + 1.9^2 \delta_{x,x'}$$

583
584
585
586
587
588

Qualitative Interpretation: The posterior peaks at a kernel structure with four additive components. Additive components hold globally, that is there are no higher level, qualitative aspects of the data that vary with the input space. The additive components are as follows: (i) a linearly increasing function or trend; (ii) a periodic function; (iii) a smooth function; and (iv) white noise.

589
590
591
592
593

Figure 5: Posterior of structure and qualitative, human interpretable reading. We take the raw data (top), compute a posterior distribution on structures (bar plot). We take the peak of this distribution (LIN+PER+SE+WN) and show its human readable interpretation (left of bar plot). Below the bar plot show one sample of this structure with corresponding parameters. On the bottom, we sample from a GP with this kernel-hyper-parameter combination.

594
595
596
597
598
599

600 $P(\mathbf{K}) \approx \frac{1}{N} \sum_{n=1}^N f(\mathbf{K}_n)$ where $f(\mathbf{K}_n) = \begin{cases} 1, & \text{if } \mathbf{K}_{global} \in \mathbf{K}_n, \\ 0, & \text{otherwise.} \end{cases}$



601 $P(\mathbf{K}_a \wedge \mathbf{K}_b) \approx \frac{1}{N} \sum_{n=1}^N f(\mathbf{K}_n)$ where $f(\mathbf{K}_n) = \begin{cases} 1, & \text{if } \mathbf{K}_a \text{ and } \mathbf{K}_b \in \mathbf{K}_n, \\ 0, & \text{otherwise.} \end{cases}$

602 $P(\mathbf{K}_a \vee \mathbf{K}_b) = P(\mathbf{K}_a) + P(\mathbf{K}_b) - P(\mathbf{K}_a \wedge \mathbf{K}_b)$

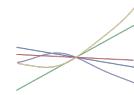
603
604
605
606
607
608

609 What is the probability of a trend, a recurring pattern **and** noise in the data?

610 $P((\text{LIN} \vee \text{LIN} \times \text{SE}) \wedge (\text{PER} \vee \text{PER} \times \text{SE} \vee \text{PER} \times \text{LIN}) \wedge (\text{WN} \vee \text{LIN} \times \text{WN})) = 0.36$

611
612
613

614 Is there a trend?
615 $P(\text{LIN} \vee \text{LIN} \times \text{SE}) = 0.65$

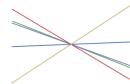


616 Is there noise?
617 $P(\text{WN} \vee \text{LIN} \times \text{WN}) = 0.75$

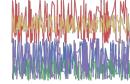


618
619
620

621 A linear trend?
622 $P(\text{LIN}) = 0.63$
623 A smooth trend?
624 $P(\text{LIN} \times \text{SE}) = 0.02$

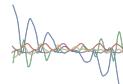


625 Heteroskedastic noise?
626 $P(\text{LIN} \times \text{WN}) = 0$
627 White noise?
628 $P(\text{WN}) = 0.75$



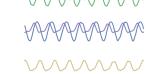
629
630

631 Is there repeating structure?
632 $P(\text{PER} \vee \text{PER} \times \text{SE} \vee \text{PER} \times \text{LIN}) = 0.73$



633

634 PER = 0.32



635 PER × SE = 0.34



636 PER × LIN = 0.07



637
638
639

640 Figure 6: We can query the data if some logical statements are probable to be true, for example, is it
641 true that there is a trend?

642
643
644
645
646
647

```

648
649
650
651
652
653
654
655
656
657
658
659
660
661
662 (defun tree-search (states goal-p successors combiner)
663   "Find a state that satisfies goal-p. Start with states,
664   and search according to successors and combiner."
665   (cond ((null states) fail)
666         ((funcall goal-p (first states)) (first states))
667         (t (tree-search
668             (funcall combiner
669                 (funcall successors (first states))
670                 (rest states))
671                 goal-p successors combiner))))
```

```

672
673
674 (define optimize
675   (lambda (probe do-search search-state-box
676             post-probe-inference extract-answer finished?)
677     (let loop ()
678       (if (finished?)
679           (extract-answer)
680           (begin
681             (do-search)
682             (if (contents-changed? search-state-box)
683                 (predict (probe ,(contents search-state-box))))
684                 (post-probe-inference)
685                 (loop)))))))
```

Figure 7: Top: Norvig's LISP meta-program for tree search. Bottom: A probabilistic meta-program for Bayesian optimization, written in VentureScheme, an abstract, Scheme-like syntax that can be used for Venture.

```

690
691
692
693
694
695
696
697
698
699
700
701
```

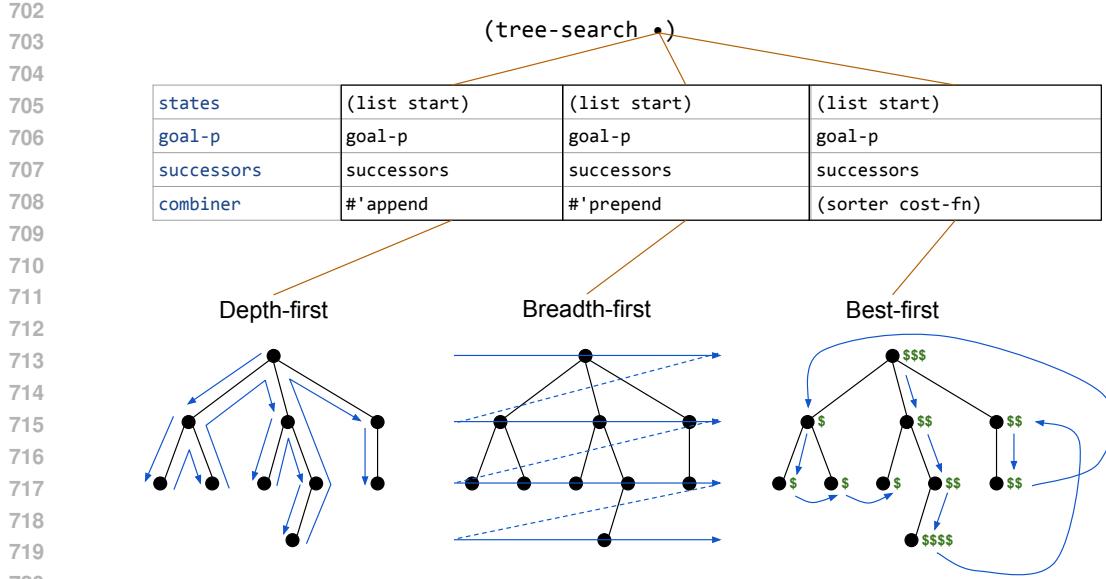


Figure 8: Invocations of Norvig’s tree search meta-procedure with different arguments to perform a variety of different tree searches.

Figure 8 shows the flexibility of Norvig’s meta-program. The procedure `tree-search` can be turned into a concrete implementation of depth-first, breadth-first or best-first search (among others) by simply supplying the appropriate combiner and set of initial states. Similarly, the meta-procedure `optimize` can make use of a variety of statistical emulators (e.g., GP-based emulators or polynomial regressions), a variety of acquisition heuristics (e.g., randomized grid search or Gaussian drift) and even a variety of rules for choosing an optimum based on the final state of the program (e.g., return the best probe so far, or return the analytically determined optimum of the emulator).

The specifications for the arguments to the meta-procedure `optimize` in Figures 7 are as follows:

- **probe:** A procedure for querying the true function f (and storing its value in the appropriate table, if caching is desired).
- **do_search:** A procedure to search for a new probe point and, if one is found, store it in `search_state_box`.
- **search_state_box:** A container for the next value to be probed. In each iteration of the loop, if the contents of `search_state_box` have changed, a new probe is performed.
- **post_probe_inference:** Any inference instructions which should be run after each probe, such as inferring hyperparameters.
- **extract_answer:** A procedure to produce an approximate optimum, based on the probes and inference that have been done so far.
- **finished?:** A procedure to decide whether optimization should be truncated here or should continue.

The meta-program is quite simple: Until `finished?` decides that optimization is finished, we choose new probe points and call `probe` on them. After each probe, we perform post-probe inference. Once optimization is finished, we call `extract_answer` to obtain an approximate optimum.

Thompson Sampling

We introduce Thompson sampling, the basic solution strategy underlying the Bayesian optimization with `gpmem`. Thompson sampling (Thompson 1933) is a widely used Bayesian framework for addressing the trade-off between exploration and exploitation in multi-armed (or continuum-armed) bandit problems. We cast the multi-armed bandit problem as a one-state Markov decision process,

756 and describe how Thompson sampling can be used to choose actions for that Markov Decision
757 Processes (MDP).
758

759 Here, an agent is to take a sequence of actions a_1, a_2, \dots from a (possibly infinite) set of possible
760 actions \mathcal{A} . After each action, a reward $r \in \mathbb{R}$ is received, according to an unknown conditional
761 distribution $P_{\text{true}}(r | a)$. The agent's goal is to maximize the total reward received for all actions in
762 an online manner. In Thompson sampling, the agent accomplishes this by placing a prior distribution
763 $P(\vartheta)$ on the possible "contexts" $\vartheta \in \Theta$. Here a context is a believed model of the conditional
764 distributions $\{P(r | a)\}_{a \in \mathcal{A}}$, or at least, a believed statistic of these conditional distributions which
765 is sufficient for deciding an action a . If actions are chosen so as to maximize expected reward, then
766 one such sufficient statistic is the believed conditional mean $V(a | \vartheta) = \mathbb{E}[r | a; \vartheta]$, which can be
767 viewed as a believed value function. For consistency with what follows, we will assume our context
768 ϑ takes the form $(\theta, \mathbf{a}_{\text{past}}, \mathbf{r}_{\text{past}})$ where \mathbf{a}_{past} is the vector of past actions, \mathbf{r}_{past} is the vector of their
769 rewards, and θ (the "semicontext") contains any other information that is included in the context.
770

In this setup, Thompson sampling has the following steps:

771 **Algorithm 2** Thompson sampling.

772 Repeat as long as desired:

- 773 1. **Sample.** Sample a semicontext $\theta \sim P(\theta)$.
774 2. **Search (and act).** Choose an action $a \in \mathcal{A}$ which (approximately) maximizes $V(a | \vartheta) =$
775 $\mathbb{E}[r | a; \vartheta] = \mathbb{E}[r | a; \theta, \mathbf{a}_{\text{past}}, \mathbf{r}_{\text{past}}]$.
776 3. **Update.** Let r_{true} be the reward received for action a . Update the believed distribution on
777 θ , i.e., $P(\theta) \leftarrow P_{\text{new}}(\theta)$ where $P_{\text{new}}(\theta) = P(\theta | a \mapsto r_{\text{true}})$.
-

781
782 Note that when $\mathbb{E}[r | a; \vartheta]$ (under the sampled value of θ for some points a) is far from the true value
783 $\mathbb{E}_{P_{\text{true}}}[r | a]$, the chosen action a may be far from optimal, but the information gained by probing
784 action a will improve the belief ϑ . This amounts to "exploration." When $\mathbb{E}[r | a; \vartheta]$ is close to the
785 true value except at points a for which $\mathbb{E}[r | a; \vartheta]$ is low, exploration will be less likely to occur, but
786 the chosen actions a will tend to receive high rewards. This amounts to "exploitation." The trade-off
787 between exploration and exploitation is illustrated in Figure 9. Roughly speaking, exploration will
788 happen until the context ϑ is reasonably sure that the unexplored actions are probably not optimal,
789 at which time the Thompson sampler will exploit by choosing actions in regions it knows to have
790 high value.

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

802 **A Mathematical Specification**

803 Our mathematical specification assert the following properties:
804

- 805 • The regression function has a Gaussian process prior.
806 • The actions $a_1, a_2, \dots \in \mathcal{A}$ are chosen by a Metropolis-like search strategy with Gaussian
807 drift proposals.
808 • The hyperparameters of the Gaussian process are inferred using Metropolis–Hastings sam-
809 pling after each action.

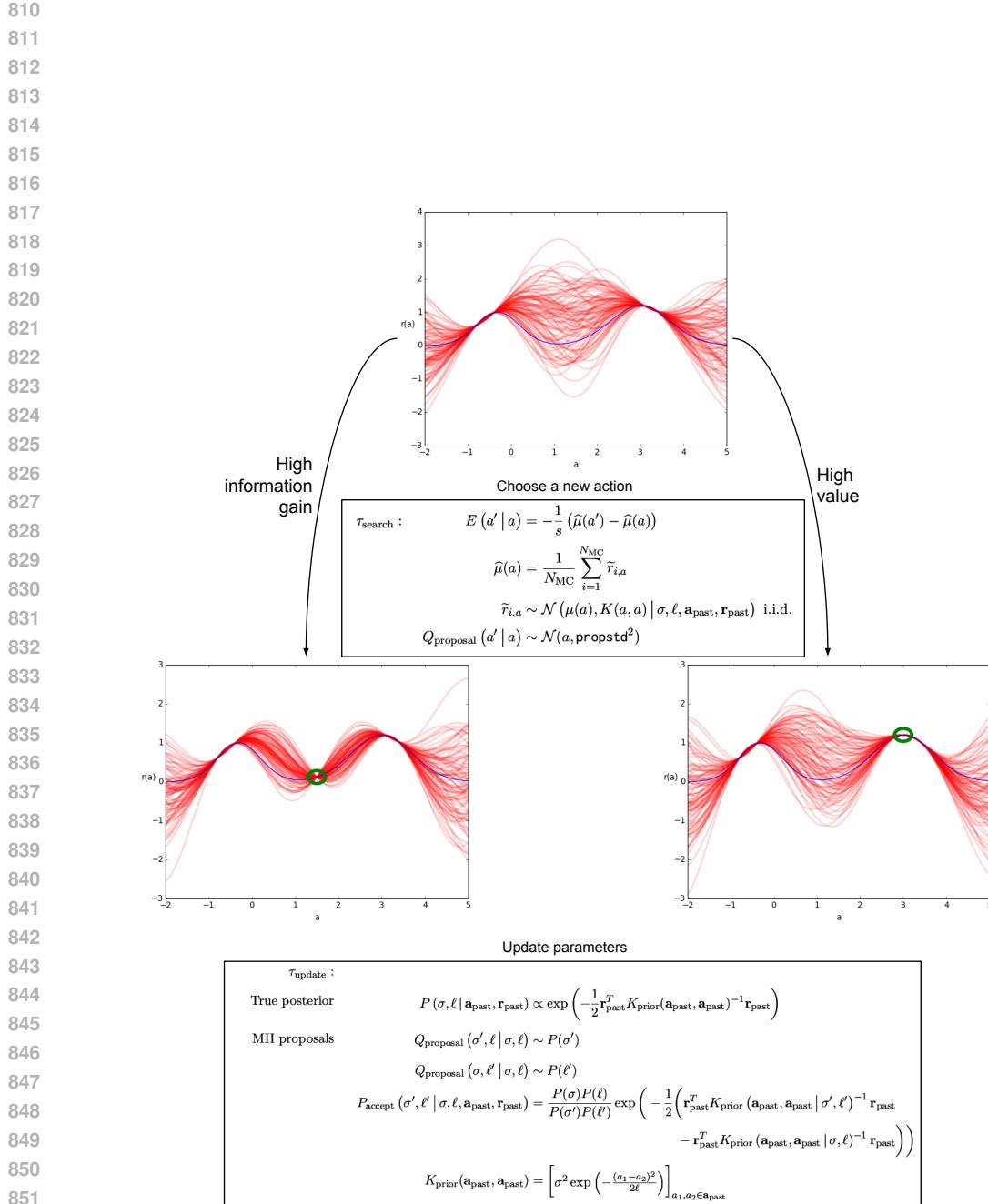


Figure 9: Two possible actions (in green) for an iteration of Thompson sampling. The believed distribution on the value function V is depicted in red. In this example, the true reward function is deterministic, and is drawn in blue. The action on the right receives a high reward, while the action on 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 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 $\mathcal{A} \rightarrow \mathbb{R}$ and the covariance K is a computable (symmetric, positive-semidefinite) function $\mathcal{A} \times \mathcal{A} \rightarrow \mathbb{R}$. 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, w.l.o.g. we set $\mu_{\text{prior}} \equiv 0$. K_{prior} is a procedure to be used as the prior covariance function, parameterized by η .

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):³

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

Note that the context space Θ is not a finite-dimensional parametric family, since the vectors \mathbf{a}_{past} and \mathbf{r}_{past} grow as more samples are taken. Θ is, however, 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 $P(\theta | \mathbf{a}_{\text{past}}, \mathbf{r}_{\text{past}})$. 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 $P(\sigma, \ell | \mathbf{a}_{\text{past}}, \mathbf{r}_{\text{past}})$ for the stationary distribution. We make MH proposals to one variable at a time, using the prior as proposal distribution:

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

and

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

The MH acceptance probability for such a proposal is

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

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

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

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 of steps is the new action a .

³Here, 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 $[K(a_i, a'_j)]_{1 \leq i \leq n, 1 \leq j \leq n'}$.

918 The Markov chain's initial state is the most recent action, and the proposal distribution is Gaussian
 919 drift:

$$Q_{\text{proposal}}(a' | a) \sim \mathcal{N}(a, \text{propstd}^2),$$

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

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

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

$$E(a' | a) = -\frac{1}{s} (\hat{\mu}(a') - \hat{\mu}(a))$$

924 where

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

925 and

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

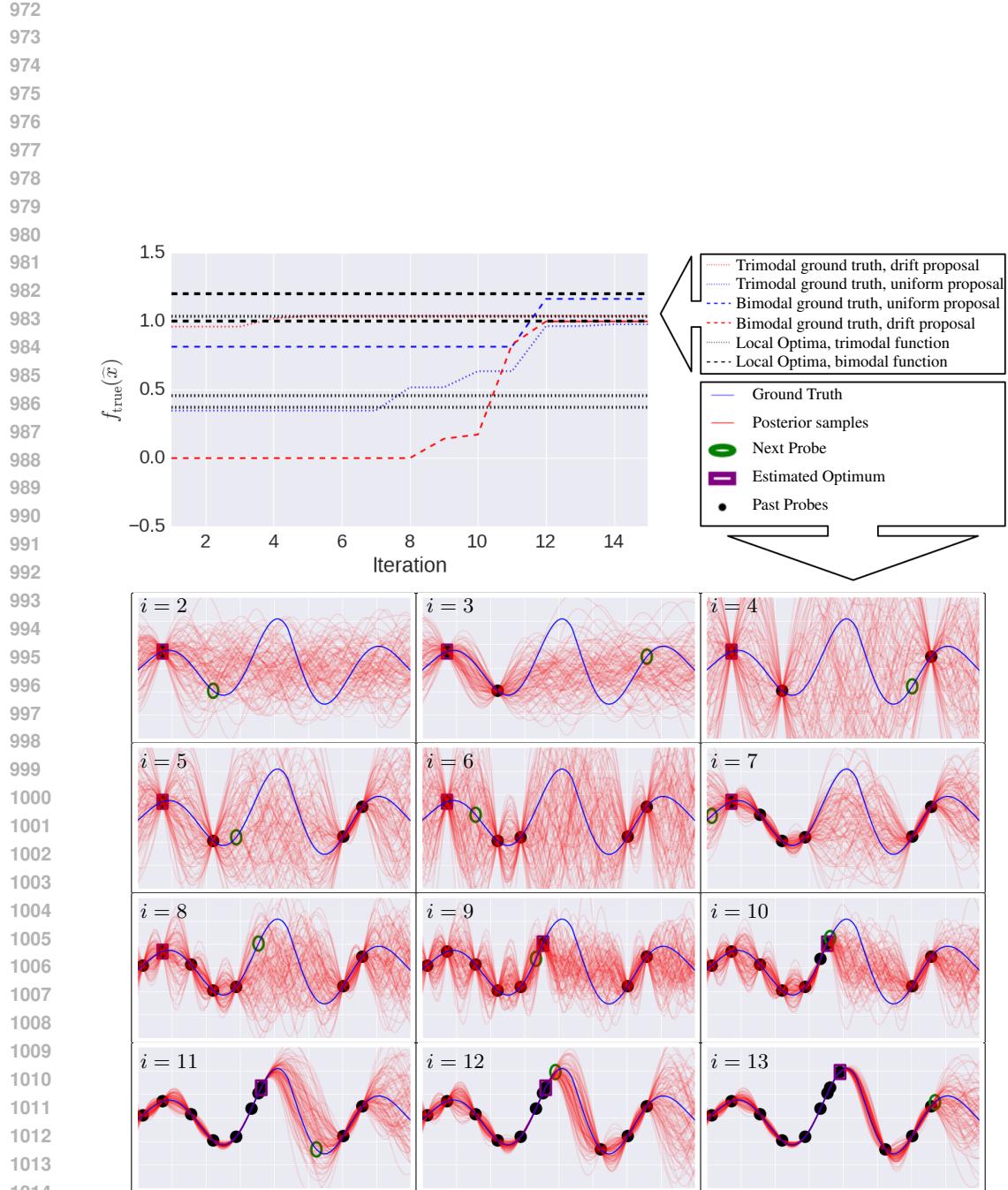
926 and $\{\tilde{r}_{i,a}\}_{i=1}^{N_{\text{avg}}}$ are i.i.d. for a fixed a . Here the temperature parameter $s \geq 0$ and the population size
 927 N_{avg} are specified ahead of time. Proposals of estimated value higher than that of the current action
 928 are always accepted, while proposals of estimated value lower than that of the current action are
 929 accepted with a probability that decays exponentially with respect to the difference in value. The
 930 rate of the decay is determined by the temperature parameter s , where high temperature corresponds
 931 to generous acceptance probabilities. For $s = 0$, all proposals of lower value are rejected; for
 932 $s = \infty$, all proposals are accepted. For points a at which the posterior mean $\mu(a)$ is low but the
 933 posterior variance $K(a, a)$ is high, it is possible (especially when N_{avg} is small) to draw a “wild”
 934 value of $\hat{\mu}(a)$, resulting in a favorable acceptance probability.

935 Indeed, taking an action a with low estimated value but high uncertainty serves the useful function
 936 of improving the accuracy of the estimated value function at points near a (see Figure 9).^{4,5} We see
 937 a complete probabilistic program with `gpmem` implementing Bayesian optimization with Thompson
 938 Sampling below (Listing 2).

939
 940
 941
 942
 943
 944
 945
 946
 947
 948
 949
 950
 951
 952
 953
 954
 955
 956
 957
 958
 959
 960
 961
 962
 963
 964
 965
 966
 967
 968
 969

⁴At least, this is true when we use a smoothing prior covariance function such as the squared exponential.

⁵For this reason, we consider the sensitivity of $\hat{\mu}$ to uncertainty to be a desirable property; indeed, this is why we use $\hat{\mu}$ rather than the exact posterior mean μ .



1015 Figure 10: Top: the estimated optimum over time. Blue and Red represent optimization with uniform
1016 and Gaussian drift proposals. Black lines indicate the local optima of the true functions. Bottom: a
1017 sequence of actions. Depicted are iterations 7-12 with uniform proposals.

1018
1019
1020
1021
1022
1023
1024
1025

```

1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079

    Listing 2: Bayesian optimization using gpmem

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

6 // A naive estimate of the argmax of the given function
7 define mc_argmax = proc(func) {
8     candidate_xs = mapv(proc(i) {uniform_continuous(-20, 20)},
9                          arange(20));
10    candidate_ys = mapv(func, candidate_xs);
11    lookup(candidate_xs, argmax_of_array(candidate_ys))
12};

13 // Shortcut to sample the emulator at a single point without packing
14 // and unpacking arrays
15 define emulate_pointwise = proc(x) {
16     run(sample(lookup(f_emulate(array(unquote(x))), 0)))
17};

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

```

4 Discussion

We provided gpmem, an elegant linguistic framework for function learning-related tasks such as Bayesian optimization and GP kernel structure learning. We highlighted how gpmem overcomes 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. We evaluated our contribution on a range of hard problems for state-of-the-art Bayesian nonparametrics.

1080 **Appendix**
 1081

1082 **A Covariance Functions**
 1083

1084
 1085 $SE = \sigma^2 \exp\left(-\frac{(x - x')^2}{2\ell^2}\right)$ (12)
 1086

1087 $LIN = \sigma^2(xx')$ (13)
 1088

1089 $C = \sigma^2$ (14)
 1090

1091 $WN = \sigma^2 \delta_{x,x'}$ (15)
 1092

1093 $RQ = \sigma^2 \left(1 + \frac{(x - x')^2}{2\alpha\ell^2}\right)^{-\alpha}$ (16)
 1094

1095 $PER = \sigma^2 \exp\left(\frac{2\sin^2(\pi(x - x')/p)}{\ell^2}\right).$ (17)

1096 **B Covariance Simplification**
 1097

1098 $SE \times SE \rightarrow SE$
 1099 $\{SE, PER, C, WN\} \times WN \rightarrow WN$
 1100 $LIN + LIN \rightarrow LIN$
 1101 $\{SE, PER, C, WN, LIN\} \times C \rightarrow \{SE, PER, C, WN, LIN\}$

1102 Rule 1 is derived as follows:
 1103

1104 $\sigma_c^2 \exp\left(-\frac{(x - x')^2}{2\ell_c^2}\right) = \sigma_a^2 \exp\left(-\frac{(x - x')^2}{2\ell_a^2}\right) \times \sigma_b^2 \exp\left(-\frac{(x - x')^2}{2\ell_b^2}\right)$
 1105
 1106 $= \sigma_c^2 \exp\left(-\frac{(x - x')^2}{2\ell_a^2}\right) \times \exp\left(-\frac{(x - x')^2}{2\ell_b^2}\right)$
 1107
 1108 $= \sigma_c^2 \exp\left(-\frac{(x - x')^2}{2\ell_a^2} - \frac{(x - x')^2}{2\ell_b^2}\right)$
 1109
 1110 $= \sigma_c^2 \exp\left(-\frac{(x - x')^2}{2\ell_c^2}\right)$
 1111
 1112

1113 For stationary kernels that only depend on the lag vector between x and x' it holds that multiplying
 1114 such a kernel with a WN kernel we get another WN kernel (Rule 2). Take for example the SE kernel:
 1115

1116 $\sigma_a^2 \exp\left(-\frac{(x - x')^2}{2\ell_c^2}\right) \times \sigma_b \delta_{x,x'} = \sigma_a \sigma_b \delta_{x,x'}$ (19)
 1117

1118 Rule 3 is derived as follows:
 1119

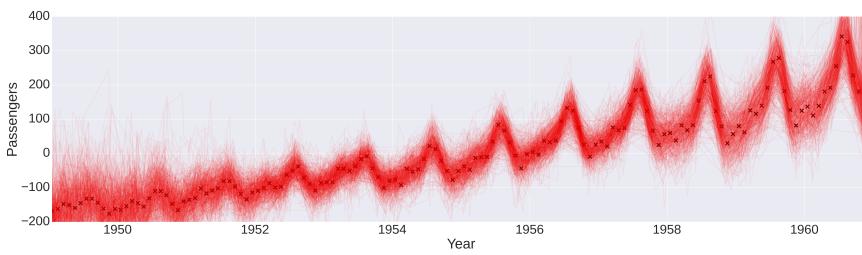
1120 $\theta_c(x \times x') = \theta_a(x \times x') + \theta_b(x \times x')$ (20)
 1121

1122 Multiplying any kernel with a constant obviously changes only the scale parameter of a kernel (Rule
 1123 4).
 1124

1125 **C Additional Structure Learning Results**
 1126

1127 Below we depict additional results for the airline data set using a hypothesis space of LIN, PER, SE
 1128 and RQ covariance functions as base kernels for the composition. Fig 11 shows the training data
 1129 and posterior samples drawn from the GP with posterior composite structure.
 1130

1131 Fig. 12 shows why it advantageous to be Bayesian here. We see two possible hypotheses for the
 1132 composite structure of \mathbf{K} .
 1133



1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
Figure 11: Data (black x) and posterior samples (red) for the airline data set.



1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
Figure 12: We see two possible hypotheses for the composite structure of K . (a) Most frequent sample drawn from the posterior on structure. We have found two global components. First, a smooth trend ($LIN \times SE$) 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.

References

- Barry, D. (1986). Nonparametric bayesian regression. *The Annals of Statistics*, 14(3):934–953.
- Box, G. E. P., Jenkins, G. M., and Reinsel, G. C. (1997). *Time series analysis: forecasting and control*.
- Damianou, A. and Lawrence, N. (2013). Deep gaussian processes. In *Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS)*, pages 207–215.
- Duvenaud, D., Lloyd, J. R., Grosse, R., Tenenbaum, J., and Ghahramani, Z. (2013). Structure discovery in nonparametric regression through compositional kernel search. In *Proceedings of the International Conference on Machine Learning (ICML)*, pages 1166–1174.
- Ferris, B., Haehnel, D., and Fox, D. (2006). Gaussian processes for signal strength-based location estimation. In *Proceedings of the Conference on Robotics Science and Systems*. Citeseer.
- Gelbart, M. A., Snoek, J., and Adams, R. P. (2014). Bayesian optimization with unknown constraints. *arXiv preprint arXiv:1403.5607*.
- Goodman, N. D .and Mansinghka, V. K., Roy, D., Bonawitz, K., and Tenenbaum, J. (2008). Church: A language for generative models. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 220–229.
- Kemmler, M., Rodner, E., Wacker, E., and Denzler, J. (2013). One-class classification with gaussian processes. *Pattern Recognition*, 46(12):3507–3518.
- Kennedy, M. C. and O'Hagan, A. (2001). Bayesian calibration of computer models. *Journal of the Royal Statistical Society. Series B, Statistical Methodology*, pages 425–464.
- Kuss, M. and Rasmussen, C. E. (2005). Assessing approximate inference for binary gaussian process classification. *The Journal of Machine Learning Research*, 6:1679–1704.
- Kwan, J., Bhattacharya, S., Heitmann, K., and Habib, S. (2013). Cosmic emulation: The concentration-mass relation for wcdm universes. *The Astrophysical Journal*, 768(2):123.
- Lloyd, J. R., Duvenaud, D., Grosse, R., Tenenbaum, J., and Ghahramani, Z. (2014). Automatic construction and natural-language description of nonparametric regression models. In *Proceedings of the Conference on Artificial Intelligence (AAAI)*.
- Mansinghka, V. K., Selsam, D., and Perov, Y. (2014). Venture: a higher-order probabilistic programming platform with programmable inference. *arXiv preprint arXiv:1404.0099*.

- 1188 McAllester, D., Milch, B., and Goodman, N. D. (2008). Random-world semantics and syntactic
1189 independence for expressive languages. Technical report.
1190
- 1191 Neal, R. M. (1995). *Bayesian Learning for Neural Networks*. PhD thesis, University of Toronto.
1192
- 1193 Neal, R. M. (1997). Monte carlo implementation of gaussian process models for bayesian regression
and classification. *arXiv preprint physics/9701026*.
- 1194 Norvig, P. (1992). *Paradigms of artificial intelligence programming: case studies in Common LISP*.
1195 Morgan Kaufmann.
- 1196 Poole, D. (1993). Probabilistic horn abduction and bayesian networks. *Artificial Intelligence*,
1197 64(1):81–129.
- 1198 Rasmussen, C. E. and Williams, C. K. I. (2006). *Gaussian Processes for Machine Learning (Adap-
1199 tive Computation and Machine Learning)*. The MIT Press.
- 1200 Sato, T. (1995). A statistical learning method for logic programs with distribution semantics. In *In
1201 Proceedings of the International Conference on Logic Programming*. Citeseer.
- 1202 Schneider, M. D., Knox, L., Habib, S., Heitmann, K., Higdon, D., and Nakhleh, C. (2008). Simula-
1203 tions and cosmological inference: A statistical model for power spectra means and covariances.
1204 *Physical Review D*, 78(6):063529.
- 1205 Snoek, J., Larochelle, H., and Adams, R. P. (2012). Practical bayesian optimization of machine
1206 learning algorithms. In *Advances in Neural Information Processing Systems (NIPS)*, pages 2951–
1207 2959.
- 1208 Thompson, W. R. (1933). On the likelihood that one unknown probability exceeds another in view
1209 of the evidence of two samples. *Biometrika*, pages 285–294.
- 1210 Williams, C. K. I. and Barber, D. (1998). Bayesian classification with gaussian processes. *IEEE
1211 Transactions on Pattern Analysis and Machine Intelligence*, 20(12):1342–1351.
- 1212
- 1213
- 1214
- 1215
- 1216
- 1217
- 1218
- 1219
- 1220
- 1221
- 1222
- 1223
- 1224
- 1225
- 1226
- 1227
- 1228
- 1229
- 1230
- 1231
- 1232
- 1233
- 1234
- 1235
- 1236
- 1237
- 1238
- 1239
- 1240
- 1241