Linear-Time Gaussian Processes Using Binary Tree Kernels

Michael K. Cohen
University of Oxford
michael.cohen@eng.ox.ac.uk

Samuel Daulton
University of Oxford, Meta
sdaulton@fb.com

Michael A. Osborne
University of Oxford
mosb@robots.ox.ac.uk

Abstract

Gaussian processes (GPs) produce good probabilistic models of functions, but most GP kernels require $O((n+m)n^2)$ time, where n is the number of data points and m the number of predictive locations. We present a new kernel that allows for Gaussian process regression in $O(n+m)$ time. Our "binary tree" kernel places all data points on the leaves of a binary tree, with the kernel depending only on the depth of the deepest common ancestor. We can store the resulting kernel matrix in $O(n)$ space, as a sum of sparse rank-one matrices, and approximately invert the kernel matrix in $O(n)$ time. Sparse GP methods also offer $O(n)$ run time, but they predict less well than higher dimensional kernels. On a classic suite of regression tasks, we compare our kernel against Matérn, sparse, and sparse variational kernels. The binary tree GP assigns the highest likelihood to the test data on a plurality of datasets, usually achieves lower mean squared error than the sparse methods, and often ties or beats the Matérn GP. On large datasets, the binary tree GP is fastest, and much faster than a Matérn GP.

1 Introduction

Gaussian processes (GPs) can be used to perform regression with high-quality uncertainty estimates, but they are slow. Naïvely, GP regression requires $O(n^3 + n^2 m)$ computation time and $O(n^2)$ computation space when predicting at m locations given n data points [24]. A kernel matrix of size $n \times n$ must be inverted (or Cholesky decomposed), and then m matrix-vector multiplications must be done with that inverse matrix (or m linear solves with the Cholesky factors). A few methods that we will discuss later achieve $O(n^2 m)$ time complexity [21, 26].

With special kernels, GP regression can be faster and use less space. Inducing point methods, using z inducing points, allow regression to be done in $O(z^2(n+m))$ time and in $O(z^2 + zn)$ space [16, 17, 18, 9]. We will discuss the details of these inducing point kernels later, but they are kernels in their own right, not just approximations to other kernels. Unfortunately, if we have observed the function value at a location x, and have to predict the function value at that location x, these inducing point kernels do not automatically predict the observed value at x with very little uncertainty (unless x is one of our inducing points). This is the easiest possible prediction problem—simple recall. Inducing point methods fail it.

We present a new kernel, the *binary tree kernel*, that also allows for GP regression in $O(n+m)$ time and space (both model fitting and prediction). Unlike inducing point kernels, with the binary tree kernel, if x is both a training data point and a predictive location, it will produce an accurate and appropriately confident prediction. The fact that inducing point kernels fail this sanity check should make us suspicious of their performance in general.

A simple depiction of our kernel is shown in Figure 1, which we will define precisely in Section 3. First, we create a procedure for placing all data points on the leaves of a binary tree. Given the binary tree, the kernel between two points depends only on the depth of the deepest common ancestor.

Preprint. Under review.

Because very different tree structures are possible for the data, we can easily form an ensemble of diverse GP regression models. Figure 2 depicts a schematic sample from a binary tree kernel. Tree-based models are a well-known tool in machine learning [20].

On a standard suite of benchmark regression tasks [21], we show that, on average, our kernel achieves better negative log likelihood (NLL) than state-of-the-art sparse methods and conjugate-gradient-based "exact" methods, at lower computational cost in the big-data regime.

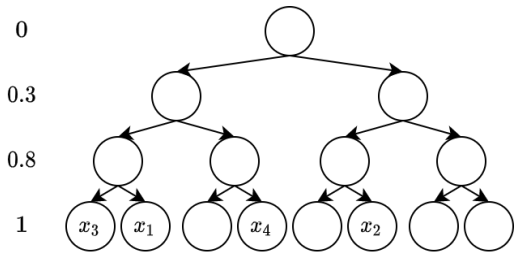

Figure 1: A binary tree kernel with four data points. In this example, $k(x_1, x_1) = 1$, $k(x_1, x_2) = 0$, $k(x_1, x_3) = 0.8$, and $k(x_1, x_4) = 0.3$.

There are not many limitations to using our kernel. The main limitation is that other kernels sometimes capture the relationships in the data better. We do not have a good procedure for understanding when data has more Matérn character or more binary tree character (except through running both and comparing training NLL). But given that the binary tree kernel usually outperforms the Matérn, we'll tentatively say the best first guess is that a new dataset has more binary tree character. One concrete limitation for some applications, like Bayesian Optimization, is that the posterior mean is piecewise-flat, so gradient-based heuristics for finding extrema would not work.

In contexts where a piecewise-flat posterior mean is suitable, we struggle to see when one would prefer a sparse or sparse variational GP to a binary tree kernel. The most thorough approach would be to run both and see which has a better training NLL, but if you had to pick one, the binary tree GP seems to be better performing and comparably fast. If minimizing mean-squared error is the objective, the Matérn kernel seems to do slightly better than the binary tree. If the dataset is small, and one needs a very fast prediction, a Matérn kernel may be the best option. But otherwise, if one cares about well-calibrated predictions, these initial results we present tentatively suggest using a binary tree kernel over the widely-used Matérn kernel.

The linear time and space complexity of the binary tree GP, with performance *exceeding* a "normal" kernel, could profoundly expand the viability of Gaussian process regression to larger datasets.

2 Preliminaries

Our problem setting is regression. Given a function $f : \mathcal{X} \to \mathbb{R}$, for some arbitrary set $\mathcal{X}$, we would like to predict $f(x)$ for various $x \in \mathcal{X}$. What we have are observations of $f(x)$ for various (other) $x \in \mathcal{X}$. Let $X \in \mathcal{X}^n$ be an n-tuple of elements of $\mathcal{X}$, and let $y \in \mathbb{R}^n$ be an n-tuple of real numbers, such that $y_i \sim f(X_i) + \mathcal{N}(0, \lambda)$, for $\lambda \in \mathbb{R}^{\geq 0}$. X and y comprise our training data.

With an m-tuple of test locations $X' \in \mathcal{X}^m$, let $y' \in \mathbb{R}^m$, with $y'_i = f(X'_i)$. y' is the ground truth for the target locations. Given training data, we would like to produce a distribution over $\mathbb{R}$ for each target location X'_i, such that it assigns high marginal probability to the unknown y'_i. Alternatively, we sometimes would like to produce point estimates $\hat{y}'_i$ in order to minimize the squared error $(\hat{y}'_i - y'_i)^2$.

A GP prior over functions is defined by a mean function $m : \mathcal{X} \to \mathbb{R}$, and a kernel $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$. The expected function value at a point x is defined to be $m(x)$, and the covariance of the function values at two points x_1 and x_2 is defined to be $k(x_1, x_2)$.

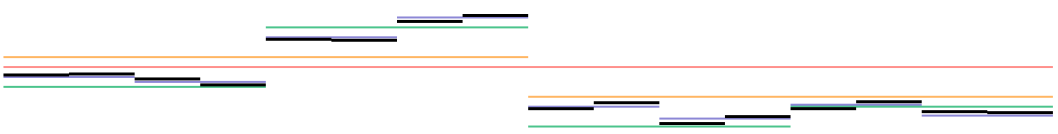

Figure 2: A schematic diagram of a function sampled from a binary tree kernel. The function is over the interval [0, 1], and points on the interval are placed onto the leaves of a depth-4 binary tree according to the first 4 bits of their binary expansion. The sampled function is in black. Purple represents the sample if the tree had depth 3, green depth 2, orange depth 1, and red depth 0.

Let $K_{XX} \in \mathbb{R}^{n \times n}$ be the matrix of kernel values $(K_{XX})_{ij} = k(X_i, X_j)$, and let $m_X \in \mathbb{R}^n$ be the vector of mean values $(m_X)_i = m(X_i)$. For a GP to be well-defined, the kernel must be such that K_{XX} is positive semidefinite for any $X \in \mathcal{X}^n$. For a point $x \in \mathcal{X}$, Let $K_{Xx} \in \mathbb{R}^n$ be the vector of kernel values: $(K_{Xx})_i = k(X_i, x)$, and let $K_{xX} = K_{Xx}^\top$. Let $\lambda \geq 0$ be the variance of observation noise. Let μ_x and σ_x^2 be the mean and variance of our posterior predictive distribution at x. Then, with $K_{XX}^{\lambda\text{inv}} = (K_{XX} + \lambda I)^{-1}$,

$$\mu_x := (y - m_X)^\top K_{XX}^{\lambda\text{inv}} K_{Xx} + m(x) \quad (1) \qquad \sigma_x^2 := k(x,x) - K_{xX} K_{XX}^{\lambda\text{inv}} K_{Xx} + \lambda. \quad (2)$$

See Williams and Rasmussen [24] for a derivation. We compute Equations 1 and 2 for all $x \in X'$.

3 Binary tree kernel

We now introduce the binary tree kernel. First, we encode our data points as binary strings. So we have $\mathcal{X} = \mathbb{B}^q$, where $\mathbb{B} = \{0, 1\}$, and $q \in \mathbb{N}$.

If $\mathcal{X} = \mathbb{R}^d$, we must map $\mathbb{R}^d \mapsto \mathbb{B}^q$. First, we rescale all points (training points and test points) to lie within the box $[0, 1]^d$. (If we have a stream of test points, and one lands outside of the box $[0, 1]^d$, we can either set K_{xX} to $\mathbf{0}$ for that point or we rescale and retrain in $O(n)$ time.) Then, for each $x \in [0, 1]^d$, for each dimension, we take the binary expansion up to some precision p, and for those $d \times p$ bits, we permute them using some fixed permutation. We call this permutation the bit order, and it is the same for all $x \in [0, 1]^d$. Note that now $q = dp$. See Figure 3 for an example. We optimize the bit order during training, and we can also form an ensemble of GPs using different bit orders.

For $x \in \mathbb{B}^q$, let $x^{\leq i}$ be the first i bits of x. $[\![\text{expression}]\!]$ evaluates to 1, if expression is true, otherwise 0. We now define the kernel:

Definition 1 (Binary Tree Kernel)**.** *Given a weight vector $w \in \mathbb{R}^q$, with $w \succeq 0$ and $||w||_1 = 1$,*

$$k_w(x_1, x_2) = \sum_{i=1}^{q} w_i \left[\!\left[x_1^{\leq i} = x_2^{\leq i} \right]\!\right]$$

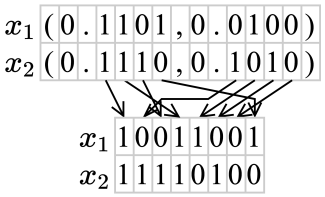

Figure 3: Function from $[0, 1]^2 \to \mathbb{B}^8$.

So the more leading bits shared by x_1 and x_2, the larger the covariance between the function values. Consider, for example, points x_1 and x_4 from Figure 1, where x_1 is (left, left, right), and x_4 is (left, right, right); they share only the first leading "bit". We train the weight vector w to maximize the likelihood of the training data.

Proposition 1 (Positive Semidefiniteness)**.** *For $X \in \mathcal{X}^n$, for $k = k_w$, $K_{XX} \succeq 0$.*

Proof. Let $s \in \bigcup_{r=1}^{q} \mathbb{B}^r$ be a binary string, and let $|s|$ be the length of s. Let $X_{[s]} \in \mathbb{R}^n$ with $(X_{[s]})_i = \left[\!\left[X_i^{\leq |s|} = s \right]\!\right]$. $X_{[s]} X_{[s]}^\top$ is clearly positive semidefinite. Finally,

$$K_{XX} = \sum_{i=1}^{q} \sum_{s \in \mathbb{B}^i} w_i X_{[s]} X_{[s]}^\top \quad (3)$$

and recall $w_i \geq 0$, so $K_{XX} \succeq 0$. □

4 Sparse rank one sum representation

In order to do GP regression in $O(n)$ space and time, we develop a "Sparse Rank One Sum" representation of linear operators (SROS). In SROS form, linear transformation of a vector can be done in $O(n)$ time instead of $O(n^2)$. We will store our kernel matrix and inverse kernel matrix in SROS form. The proof of Proposition 1 exemplifies representing a matrix as the sum of sparse rank one matrices. Note that each $X_{[s]}$ is sparse—if q is large, most $X_{[s]}$'s are the zero vector.

We now show how to interpret an SROS representation of an $n \times n$ matrix. Let $[n] = \{1, 2, ..., n\}$. For $m \in \mathbb{N}$, let $L : [m]^n \times [m]^n \times \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^{n \times n}$ construct a linear operator from four vectors.

Definition 2 (Linear Operator from Simple SROS Representation)**.** *Let $p, p' \in [m]^n$, and let $u, u' \in \mathbb{R}^n$. Let $u^{p=i} \in \mathbb{R}^n$ be the vector where $u^{p=i}_j = u_j [\![p_j = i]\!]$, and likewise for u' and p'. Then:*

$$L(p, p', u, u') \mapsto \sum_{i=1}^{m} (u^{p=i})((u')^{p'=i})^\top.$$

We depict Definition 2 in Figure 4. p and p' represent partitions over n elements: all elements with the same integer value in the vector p belong to the same partition. If $p = p'$ (which is almost always the case for us) and the elements of p, u, and u' were shuffled so that all elements in the same partition were next to each other, then $L(p, p', u, u')$ would be block diagonal. Note that $L(p, p', u, u')$ is not necessarily low rank. If p is the finest possible partition, and $p = p'$, $L(p, p', u, u')$ is diagonal. SROS matrices can be thought of as a generalization of two types of matrix that are famously amenable to fast computation: rank one matrices (all points in the same partition) and diagonal matrices (each point in its own partition).

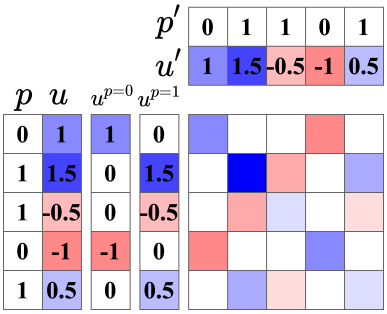

Figure 4: A matrix in standard form constructed from a matrix in SROS form. The large square depicts the matrix $L(p, p', u, u') \in \mathbb{R}^{5\times5}$ with elements colored by value. See $u^{p=0}$ for a color legend.

We now extend the definition of L to allow for multiple p, p', u, and u' vectors.

Definition 3 (Linear Operator from SROS Representation)**.** *Let $L : [m]^{n\times q} \times [m]^{n\times q} \times \mathbb{R}^{n\times q} \times \mathbb{R}^{n\times q} \to \mathbb{R}^{n\times n}$. Let $P, P' \in [m]^{n\times q}$, and let $U, U' \in \mathbb{R}^{n\times q}$. Let $P_{:,i}$, $U_{:,i}$, etc. be the i^{th} columns of the respective arrays. Then: $L(P, P', U, U') \mapsto \sum_{i=1}^{q} L(P_{:,i}, P'_{:,i}, U_{:,i}, U'_{:,i})$.*

Algorithm 1 performs linear transformation of a vector using SROS representation in $O(nq)$ time.

Algorithm 1 Linear Transformation with SROS Linear Operator. This can be vectorized on a Graphical Processing Unit (GPU), using e.g. `torch.Tensor.index_add_` for Line 5 and non-slice indexing for Line 6 [14]. Slight restructuring allows vectorization over $[q]$ as well.

Require: $P, P' \in [m]^{n\times q}, U, U' \in \mathbb{R}^{n\times q}, x \in \mathbb{R}^n$
Ensure: $y = L(P, P', U, U')x$
1: $y \leftarrow \mathbf{0} \in \mathbb{R}^n$
2: **for** $i \in [q]$ **do** ▷ $O(nq)$ time
3: $\quad p, p', u, u' \leftarrow P_{:,i}, P'_{:,i}, U_{:,i}, U'_{:,i}$
4: $\quad z \leftarrow \mathbf{0} \in \mathbb{R}^m$ ▷ z_i will store the dot product $((u')^{p'=i})^\top x^{p'=i}$
5: $\quad$ **for** $j \in [n]$ **do** $z_{p'_j} \leftarrow z_{p'_j} + u'_j x_j$ ▷ $O(n)$ time
6: $\quad$ **for** $j \in [n]$ **do** $y_j \leftarrow y_j + z_{p_j} u_j$ ▷ $O(n)$ time
return y

We now discuss how to approximately invert a certain kind of symmetric SROS matrix, but our methods could be extended to asymmetric matrices. First, we define a partial ordering over partitions. For two partitions p, p', we say $p' \leq p$ if p' is finer than or equal to p; that is, $p'_i = p'_j \implies p_i = p_j$. Using that partial ordering, a symmetric SROS matrix can be approximately inverted efficiently if for all $1 \leq i, j \leq q$, $P_{:,i} \leq P_{:,j}$ or $P_{:,j} \leq P_{:,i}$. As the reader may have recognized, our kernel matrix K_{XX} can be written as an SROS matrix with this property.

We will write symmetric SROS matrices in a slightly more convenient form. All $(u')^{p=i}$ must be a constant times $u^{p=i}$. We will store these constants in an array C. Let $L(P, C, U)$ be shorthand for $L(P, P, U, C \odot U)$, where $\odot$ denotes element-wise multiplication. For $L(P, C, U)$ to be symmetric, it must be the case that $P_{kj} = P_{lj} \implies C_{kj} = C_{lj}$. Then, all elements of U corresponding to a given $u^{p=i}$ are multiplied by the same constant. We now present an algorithm for calculating $(L(P, C, U) + \lambda I)^{-1}$, for $\lambda \neq 0$. We have not yet analyzed numerical sensitivity for $\lambda \to 0$, but we conjecture that all floating point numbers involved need to be stored to at least $\log_2(1/\lambda)$ bits. Without loss of generality, let $\lambda = 1$, and note $(L(P, C, U) + \lambda I)^{-1} = \lambda^{-1}(L(P, \lambda^{-1}C, U) + I)^{-1}$.

By assumption, all columns of P are comparable with respect to the partial ordering above, so we can reorder the columns of P such that $P_{:,i} \geq P_{:,j}$ for $i < j$. The key identity that we use to develop our fast inversion algorithm is the Sherman–Morrison Formula:

$$(A + cuu^\top)^{-1} = A^{-1} - \frac{A^{-1}uu^\top A^{-1}}{c^{-1} + u^\top A^{-1}u} \tag{4}$$

Starting with $A = I$, we add the sparse rank one matrices iteratively, from the finest partition to the coarsest one, updating A^{-1} as we go. We represent $(L(P, C, U) + I)^{-1}$ in the form $I + L(P, C', U')$, so we write an algorithm that returns C' and U'. We can also quickly calculate $\log |L(P, C, U) + I|$ at the same time, using the matrix determinant lemma: $|A + cuu^\top| = (1 + cu^\top A^{-1}u)|A|$.

Theorem 1 (Fast Inversion). *For $P \in [m]^{n\times q}$ and $C, U \in \mathbb{R}^{n\times q}$, if $P_{:,i} \overset{\text{(is coarser than)}}{\geq} P_{:,j}$ for $i < j$, then there exists $C', U' \in \mathbb{R}^{n\times q}$, such that $(L(P, C, U) + I)^{-1} = I + L(P, C', U')$. There exists an algorithm for computing C' and U' that takes $O(nq^2)$ time.*

Proof. For $X \in \mathbb{R}^{n\times m}$, let $X_{:,i+1:m} \in \mathbb{R}^{n\times(m-i)}$ be columns $i+1$ through m of matrix X (inclusive). Let $A_i = I + L(P_{:,i+1:q}, C_{:,i+1:q}, U_{:,i+1:q})$, and $A_q = I$. Now suppose A_i^{-1} can be written as $I + L(P_{:,i+1:q}, C'_{:,i+1:q}, U'_{:,i+1:q})$ for some C' and U'. For the base case of $i = q$, this holds trivially. We show it also holds for $i - 1$, and we can compute $C'_{:,i:q}$, $U'_{:,i:q}$ in $O\big(n(q-i)\big)$ time. Let $p = P_{:,i}$, $u = U_{:,i}$, and $c = C_{:,i}$. Consider $u^{p=j}$, where each element is zero unless the corresponding element of p equals j. What do we know about the product $A_i^{-1}u^{p=j}$ (as seen in Equation 4)?

Because we have reordered the columns of P from coarse partitions to fine ones, all of the vectors generating the sparse rank one components of $L(P_{:,i+1:q}, C'_{:,i+1:q}, U'_{:,i+1:q})$ are from partitions that are equal to or finer than p. Thus, they are either zero everywhere $u^{p=j}$ is zero, or zero everywhere $u^{p=j}$ is nonzero. Vectors v of the second kind can be ignored, as $cvv^\top u^{p=j} = 0$. Thus, when multiplying $L(P_{:,i+1:q}, C'_{:,i+1:q}, U'_{:,i+1:q})$ by $u^{p=j}$, the only relevant vectors are filled with zeros except where the corresponding element of p equals j. So we can get rid of those rows of $P_{:,i+1:q}$, $C_{:,i+1:q}$, and $U_{:,i+1:q}$. Suppose there are n_j elements of p that equal j. Then $L(P_{:,i+1:q}, C'_{:,i+1:q}, U'_{:,i+1:q})u^{p=j}$ involves n_j rows, and can be computed in $O\big(n_j(q-i)\big)$ time. Moreover, this product, which we will call $(u')^{p=j}$, is only nonzero when the corresponding element of p equals j, so it has the same sparsity pattern as $u^{p=j}$. The other component of A_i^{-1} is the identity matrix, and $Iu^{p=j}$ clearly has the same sparsity as $u^{p=j}$. Thus, returning to Equation 4, when we add $u^{p=j}(u^{p=j})^\top$ to A_i, we update A_i^{-1} with an outer product of vectors whose sparsity pattern is the same as that of $u^{p=j}$.

For each j, A_i^{-1} need not be updated with each $u^{p=j}$ one at a time. For $j \neq j'$, $u^{p=j}$ and $u^{p=j'}$ are nonzero at separate indices, so $u^{p=j}$ and $(u')^{p=j'}$ are nonzero at separate indices, so the extra component of A_i^{-1} that appears after after the $u^{p=j'}$ update is irrelevant to the $u^{p=j}$ update. Since the $u^{p=j}$ update takes $O(n_j(q-i))$ time, all of them together take $O(\sum_j n_j(q-i))$ time, which equals $O(n(q-i))$ time. Calculating c'_i only involves computing the denominator in Equation 4, using a matrix-vector product already computed. So we can write $C'_{:,i:q}$ and $U'_{:,i:q}$ by adding a preceding column to $C'_{:,i+1:q}$ and $U'_{:,i+1:q}$, using the same partition p, and we can do it in $O(n(q-i))$ time.

Following the induction down to $i = 0$, we have $(L(P, C, U) + I)^{-1} = I + L(P, C', U')$, and a total time of $O(nq^2)$. □

Algorithm 2 also performs approximate inversion, which we prove in Appendix A, but differs slightly from the algorithm in the proof above. Algorithm 2 can take full advantage of a GPU speedup. In the special setting where all the columns of U are identical, observe that in Lines 10 and 11, the same computation is being repeated for all $k \in [i]$. Indeed, in this setting, this block of code can be modified to run in $O(n)$ time and space rather than $O(ni)$, making the whole algorithm run in $O(nq)$ time, as shown in Proposition 4 in Appendix A.

Algorithm 2 Inverse and determinant of $I+$ SROS Linear Operator. Lines 5 through 11 can all be easily vectorized on a GPU. Lines 5 and 10 require `torch.Tensor.index_add_` or equivalent, and lines 6 and 11 require non-slice indexing, which are not quite as fast as some GPU operations.

Require: $P \in [m]^{n \times q}, C, U \in \mathbb{R}^{n \times q}$
Ensure: $I + L(P, C', U') = (I + L(P, C, U))^{-1}; x = \log |I + L(P, C, U)|$
1: $x, C', U' \leftarrow 0, \mathbf{0} \in \mathbb{R}^{n \times q}, U$
2: **for** $i \in (q, q-1, ..., 1)$ **do** ▷ $O(nq^2)$ time
3: $\quad p, c, u, u' \leftarrow P_{:,i}, C_{:,i}, U_{:,i}, U'_{:,i}$
4: $\quad z \leftarrow \mathbf{0} \in \mathbb{R}^m$ ▷ z_i will store $c^{(i)}((u')^{p=i})^\top u^{p=i}$, where $c^{(i)} = c_k$ if $p_k = i$
5: $\quad$ **for** $j \in [n]$ **do** $z_{p_j} \leftarrow z_{p_j} + c_j u'_j u_j$ ▷ $O(n)$ time
6: $\quad$ **for** $j \in [n]$ **do** $C'_{ji} \leftarrow -c_j/(1 + z_{p_j})$ ▷ $O(n)$ time
7: $\quad$ **for** $i \in [m]$ **do** $x \leftarrow x + \log(1 + z_i)$ ▷ $O(n)$ time
8: $\quad$ **if** i > 0 **then**
9: $\quad\quad y \leftarrow \mathbf{0} \in \mathbb{R}^{n \times i}$ ▷ $O(ni)$ time
10: $\quad\quad$ **for** $j, k \in [n] \times [i-1]$ **do** $y_{p_j k} \leftarrow y_{p_j k} + u'_j U_{jk}$ ▷ $O(ni)$ time
11: $\quad\quad$ **for** $j, k \in [n] \times [i-1]$ **do** $U'_{jk} \leftarrow U'_{jk} + C'_{ji} u'_j y_{p_j k}$ ▷ $O(ni)$ time
return C', U', x

5 Binary tree Gaussian process

We now show that our kernel matrix K_{XX} can be written in SROS form, with P containing successively finer partitions. Thus, K_{XX} can be approximately inverted quickly, for use in Equations 1 and 2. Next, we'll show that we can efficiently optimize the log likelihood of the training data by tuning the weight vector w along with the bit order. The log likelihood can be calculated in $O(nq)$ time and the gradient w.r.t. w in $O(nq^2)$ time.

Recall Equation 3 in the proof of Proposition 1: $K_{XX} = \sum_{i=1}^{q} \sum_{s \in \mathbb{B}^i} w_i X_{[s]} X_{[s]}^\top$, where $X_{[s]} \in \mathbb{R}^n$ with $(X_{[s]})_i = \left[\left[X_i^{\leq |s|} = s \right]\right]$. So we will set $P_{:,i}$, $C_{:,i}$, and $U_{:,i}$, so that $L(P_{:,i}, C_{:,i}, U_{:,i}) = \sum_{s \in \mathbb{B}^i} w_i X_{[s]} X_{[s]}^\top$. Let $P_{:,i}$ partition the set of points X so that points are in the same partition if the first i bits match. Now, requiring the first $i+1$ bits to match is a stricter criterion than requiring the first i bits to match, so the $P_{:,i}$ grow successively finer. For any piece of the partition where the first i bits of the constituent points equals the bitstring s, the corresponding sparse rank one component of K_{XX} is $w_i X_{[s]} X_{[s]}^\top$. So let $U_{:,i} = \mathbf{1}^n$, and let $C_{:,i} = w_i \mathbf{1}^n$.

Proposition 2 (SROS Form Kernel)**.** *$K_{XX} = L(P, C, U)$, as defined above.*

This follows immediately from the definitions. To compute these partitions $P_{:,i}$, we sort X, which is a set of bit strings. And then we can easily compute which points have the same first i bits. This all takes $O(nq \log n)$ time. Now note that $U_{:,i} = U_{:,j}$ for all i, j, so $(K_{XX} + \lambda I)^{-1}$ and $|K_{XX} + \lambda I|$ can be computed in $O(nq)$ time, rather than $O(nq^2)$.

The training negative log likelihood of a GP is that of the corresponding multivariate Gaussian on the training data. So: $\text{NLL}(w) = \frac{1}{2}\left(y^\top (K_{XX}(w) + \lambda I)^{-1} y + \log |K_{XX}(w) + \lambda I| + n \log(2\pi)\right)$. This can be computed in $O(nq)$ time, since matrix-vector multiplication takes $O(nq)$ time for a matrix in SROS form. So if the bit order is unchanged, an optimization step can be done in $O(nq)$ time, and if the data needs to be resorted, then in $O(nq \log n)$ time. On the largest dataset we tested (House Electric), with $n \approx 1.3$ million and $q = 88$, sorting the data and computing P takes about 0.96 seconds on a GPU, and then calculating the negative log likelihood takes about another 1.08 seconds. We show in Appendix B how to compute $\nabla_w \text{NLL}$ in $O(nq^2)$ time.

To optimize the bit order and weight vector at the same time, we represent both with a single parameter vector $\theta \in \mathbb{R}_+^q$, with $||\theta||_\infty = 1$. To get the bit order from θ, we start with a default bit order and permute the bit order according to a permutation that would sort θ in descending order. To get the weight vector, we sort θ in descending order, add a 0 at the end, and compute the differences between adjacent elements. When there are ties in the elements of θ, the choice of bit order does not affect the negative log likelihood (or the kernel at all) because the relevant associated weight is 0. The negative

log likelihood is continuous with respect to θ, and when all values of θ are unique, it is differentiable with respect to θ. Letting $\theta = e^\phi / ||e^\phi||_\infty$, we minimize loss w.r.t. ϕ using BFGS [6].

To calculate the predictive mean at a list of predictive locations X', we first multiply y by $(K_{XX} + \lambda I)^{-1}$, and then we multiply that vector by $K_{XX'}$. We obtain both K_{XX} and $K_{XX'}$ in SROS form using the following procedure. Let $\tilde{X} = X \circ X'$ be the concatenation of the two tuples, now an $(n+m)$-tuple. Writing $K_{\tilde{X}\tilde{X}} = L(\tilde{P}, \tilde{C}, \tilde{U})$, the arrays on the r.h.s. can be computed in $O((n+m)q\log(n+m))$ time. Then, with P, C, and U being the first n rows of $\tilde{P}$, $\tilde{C}$, $\tilde{U}$, $K_{XX} = L(P, C, U)$. And letting P'' and U'' be the last m rows, $K_{XX'} = L(P, P'', C \odot U, U'')$. Thus, the predictive mean μ_x from Equation 1 can be computed at m locations in $O((n+m)q\log(n+m))$ time.

The predictive covariance matrix, which extends the predictive variance from Equation 2, is calculated $\Sigma_{X'} = K_{X'X'} + \lambda I_m - K_{X'X}(K_{XX} + \lambda I_n)^{-1} K_{XX'} = (K_{\tilde{X}\tilde{X}} + \lambda I_{m+n})/K_{XX}$, where $/$ denotes the Schur complement. From a property of block matrix inversion, the last m columns of the last m rows of $(K_{\tilde{X}\tilde{X}} + \lambda I)^{-1}$ equals $((K_{\tilde{X}\tilde{X}} + \lambda I_{m+n})/K_{XX})^{-1}$. So we get the predictive precision matrix in $O((n+m)q\log(n+m))$ time by inverting $K_{\tilde{X}\tilde{X}} + \lambda I$ and taking the bottom right $m \times m$ block. Then, we get the predictive covariance matrix by inverting that. This takes $O(mq^2)$ time, since it does not have the property of all the columns of U being equal. If we only want the diagonal elements of an SROS matrix (the independent predictive variances in this case), we can simply sum the rows of $C \odot U \odot U$ in $O(mq)$ time. Thus, in total, computing the independent predictive variances requires $O((n+m)q\log(n+m) + mq^2)$ time. See Algorithm 3.

Algorithm 3 GP Regression with a binary tree kernel.

Require: $X \in \mathbb{B}^{n\times q}$, $y \in \mathbb{R}^n$, $X' \in \mathbb{B}^{m\times q}$, $w \in \mathbb{R}^q$, $\lambda \in \mathbb{R}^+$

Ensure: $\mu_{X'}$ and $\sigma^2_{X'}$ are the predictive means and variances at X', and nll the training negative log likelihood.

$\tilde{X} \leftarrow X \circ X'$

$\tilde{X}^\uparrow, \text{perm} \leftarrow \text{Sort}(\tilde{X})$ ▷ The rows of X are sorted lexically from leading bit to trailing bit. $O((n+m)q\log(n+m))$ time.

for $i, j \in [n+m] \times [q]$ **do** $\tilde{P}^\uparrow_{ij} \leftarrow$ #of unique rows in $X^\uparrow_{1:i,1:j}$

▷ $M_{1:i,1:j}$ is the first i rows and j columns of M. $O((n+m)q)$ time.

$\tilde{P} \leftarrow \text{perm}^{-1}(P^\uparrow)$ ▷ This "unsorts" the input. $O((n+m)q)$ time.

$P, P' \leftarrow \tilde{P}_{1:n}, \tilde{P}_{n+1:n+m}$

$U, U', \tilde{U} \leftarrow \mathbf{1}^{n\times q}, \mathbf{1}^{m\times q}, \mathbf{1}^{(n+m)\times q}$

$C, \tilde{C} \leftarrow \mathbf{1}^n w^T, \mathbf{1}^{n+m} w^T$

$C^{-1}_\lambda, U^{-1}, \text{logdet}_\lambda \leftarrow \text{Invert}(P, \lambda^{-1}C, U)$ ▷ Uses Algorithm 2. Speedup to $O(nq)$ time because columns of $\lambda^{-1}U$ are identical.

$C^{-1}, \text{logdet} \leftarrow \lambda^{-1}C^{-1}_\lambda, \text{logdet}_\lambda + n\log(\lambda)$

$z \leftarrow \text{LinTransform}(P, P, U^{-1}, C^{-1} \odot U^{-1}, y) + \lambda^{-1} y$ ▷ Uses Algorithm 1 to compute the Woodbury vector. $O(nq)$ time.

$\mu_{X'} \leftarrow \text{LinTransform}(P', P, U', C \odot U, z)$ ▷ $O((n+m)q)$ time.

$\text{nll} \leftarrow (y^\top z + \text{logdet} + n\log(2\pi))/2$

$\tilde{C}^{\text{prec}}, \tilde{U}^{\text{prec}} \leftarrow \text{Invert}(\tilde{P}, \lambda^{-1}\tilde{C}, \lambda^{-1}\tilde{U})$ ▷ $O((n+m)q)$ time.

$C^{\text{prec}}, U^{\text{prec}} \leftarrow \tilde{C}^{\text{prec}}_{n+1:n+m}, \tilde{U}^{\text{prec}}_{n+1:n+m}$

$C^{\text{cov}}, U^{\text{cov}} \leftarrow \text{Invert}(P', C^{\text{prec}}, U^{\text{prec}})$ ▷ $O(mq^2)$ time; extra factor of q because columns of U^{prec} are not identical.

$\sigma^2_{X'} \leftarrow \lambda(\mathbf{1}^m + \text{SumEachRow}(C^{\text{cov}} \odot U^{\text{cov}} \odot U^{\text{cov}}))$ ▷ $O(mq)$ time.

return $\mu_{X'}, \sigma^2_{X'}, \text{nll}$

6 Related Work

All existing kernels of which we are aware for $O(n)$ time GP regression on unstructured data involve inducing points (related to the Nyström approximation [23]) or inducing frequencies. For a given set of inducing points Z, for some base kernel k, the inducing point kernel (in its most basic form) is the following, although subtle variants exist: $k^Z(x, x') = K_{xZ}K^{-1}_{ZZ}K_{Zx'}$ [16].

Sparse Gaussian Process Regression (SGPR) involves selecting Z, and then using k^Z (or a variant). Notably, $K_{XX}^Z = K_{XZ}K_{ZZ}^{-1}K_{ZX}$ is low rank, providing computational efficiency. The predictive mean and covariance have compact form, with observational noise λ: $\mu_x(Z) = K_{xZ}(\lambda K_{ZZ} + k_{ZX}k_{XZ})^{-1}K_{ZX}y$ and $\sigma_{xx'}^2(Z) = K_{xZ}(K_{ZZ} + \lambda^{-1}k_{ZX}k_{XZ})^{-1}K_{Zx'}$.

Titsias's [19] sparse variational kernel is also low rank and uses inducing points. The sparse variational GP (SVGP) is constructed as the solution to a variational inference problem. It depends on inducing points Z, data points X, and observed function values y. We have focused on Gaussian processes with 0 mean, but the SVGP method uses a nonzero prior mean along with a kernel: $m^{\text{SVGP}}(x) = \mu_x(Z)$ and $k^{\text{SVGP}}(x, x') = k(x, x') - K_{xZ}K_{ZZ}^{-1}K_{Zx'} + \sigma_{xx'}^2(Z)$. Given the dependence on X and y, this is not a true probability distribution over function space. The variational problem underlying this kernel also provides guidance in how to select the inducing points Z. For further discussion of the kernel underlying the SVGP method, see Wild et al. [22].

An inducing point kernel with z inducing points produces a z-dimensional reproducing kernel Hilbert space (RKHS). The dimensionality of the RKHS relates to the expressivity of the kernel. Whereas an inducing point method buys a z-dimensional RKHS for the price of $O(z^2n)$ time and $O(zn)$ space, the binary tree kernel produces a 2^q-dimensional RKHS in $O(qn)$ time and space—an exponential improvement. (Observe that we can find 2^q linearly independent functions of the form $k(\cdot, x)$—one for each of the 2^q leaves x might belong to.) Wilson and Nickisch [25] develop a method for speeding up inducing point methods significantly, especially in low-dimensional settings.

Lázaro-Gredilla et al. [11] propose an inducing frequencies kernel: given a set of m inducing vectors s_i, $k(x, x') = 1/m \sum_{i=1}^m \cos(2\pi s_i^\top (x - x'))$. Dutordoir et al. [5] propose an inducing frequencies kernel for low dimensional data, in which $k(x, x')$ is a special function of $x^\top x'$.

On one-dimensional data, filtering/smoothing methods perform Bayesian inference over functions in $O(n)$ time [10, 8]. A few non-$O(n)$ methods bear mentioning. We are not the first to consider a kernel over points on the leaves of a tree [13, 12], but their methods take $O(n^3)$ time. On certain kinds of structured data, Toeplitz solvers achieve $O(n^2)$ time complexity [26]. Cutajar et al.'s [2], Wang et al.'s [21], and others' use of a conjugate gradients solver to replace inversion/factorization has unclear time complexity between $O(n^2)$ and $O(n^3)$, depending on the kernel matrix spectrum. [1]

7 Experiments

In Table 1, we compare our binary tree kernel and a binary tree ensemble (see Appendix C for details on the ensemble) against three baseline methods: exact GP regression using a Matérn kernel, sparse Gaussian process regression (SGPR) [18], and a stochastic variational Gaussian process (SVGP) [9]. We evaluate our method on the same open-access UCI datasets [3] as Wang et al. [21], using their same training, validation, and test partitions, and we compare against the baseline results they report. For the binary tree (BT) kernels, we use $p = \min(8, \lfloor 150/d \rfloor + 1)$, and recall $q = pd$. We set $\lambda = 1/n$. We train the bit order and weights to minimize training NLL. For the binary tree ensemble (BTE), we use 20 kernels. For the Matérn kernel, we use Blackbox Matrix-Matrix multiplication (BBMM) [7], which uses the conjugate gradients method to calculate $(K_{XX} + \lambda I)^{-1}$. SGPR uses 512 data points and SVGP uses 1,024 inducing points. We report the mean and two standard errors across 3 replications with different dataset splits. For further experimental details, see Appendix C. BTE achieves the best NLL on 6/12 datasets, and best RMSE on 5/12 datasets (including some ties). Out of the 4 largest datasets, BT/BTE is fastest on 3. The run times are plotted in Figure 5. The code is included in the supplementary material, and we will open source our code upon publication.

BT performs noticeably worse than BTE for test NLL on CTSlice due to over-fitting. There are enough degrees of freedom when optimizing the bit order ($d = 385$) that BT kernel can over-fit to the training data. The ensemble over multiple bit orders is much more robust.

[1] The conjugate gradients method takes $O(n^2\sqrt{\kappa})$ time, where κ is the condition number of the kernel matrix. Poggio et al. [15] say "claims about the condition number of a random matrix A should also apply to kernel matrices with random data." If they mean a Wishart random matrix (which it should be if, e.g., $k(x, x') = x^\top x'$), that would be the square of the condition number of the corresponding Gaussian random matrix, which grows as $O(n)$ [1]. Putting it all together, we get $O(n^3)$ for conjugate gradients. We don't know how quickly preconditioning can reduce the condition number.

Dataset	n	d	BTE	BT	Matérn (BBMM)	SGPR	SVGP
PoleTele	9,600	26	**−0.625** ± 0.035	−0.490 ± 0.040	−0.180 ± 0.036	−0.094 ± 0.008	−0.001 ± 0.008
Elevators	10,623	18	0.649 ± 0.032	0.646 ± 0.023	0.619 ± 0.054	0.580 ± 0.060	**0.519** ± 0.022
Bike	11,122	17	**−0.708** ± 0.433	**−0.806** ± 0.273	0.119 ± 0.044	0.291 ± 0.032	0.272 ± 0.018
Kin40k	25,600	8	0.869 ± 0.004	0.881 ± 0.008	**−0.258** ± 0.084	0.087 ± 0.067	0.236 ± 0.077
Protein	29,267	9	**0.781** ± 0.023	0.845 ± 0.026	1.018 ± 0.056	0.970 ± 0.010	1.035 ± 0.006
KeggDir	31,248	20	−1.031 ± 0.020	−1.029 ± 0.021	−0.199 ± 0.381	**−1.123** ± 0.016	−0.940 ± 0.020
CTslice	34,240	385	**−2.527** ± 0.147	−1.092 ± 0.147	−0.894 ± 0.188	−0.073 ± 0.097	1.422 ± 0.005
KEGGU	40,708	27	−0.667 ± 0.007	−0.667 ± 0.007	−0.419 ± 0.027	**−0.984** ± 0.012	−0.666 ± 0.007
3DRoad	278,319	3	**−0.251** ± 0.009	**−0.252** ± 0.006	0.909 ± 0.001	0.943 ± 0.002	0.697 ± 0.002
Song	329,820	90	1.330 ± 0.003	1.331 ± 0.003	**1.206** ± 0.024	1.213 ± 0.003	1.417 ± 0.000
Buzz	373,280	77	1.198 ± 0.003	1.198 ± 0.003	0.267 ± 0.028	**0.106** ± 0.008	0.224 ± 0.050
HouseElec	1,311,539	11	**−2.569** ± 0.006	−2.492 ± 0.012	−0.152 ± 0.001	—	−1.010 ± 0.039
PoleTele	9,600	26	**0.154** ± 0.006	0.161 ± 0.004	**0.151** ± 0.012	0.217 ± 0.002	0.215 ± 0.002
Elevators	10,623	18	0.478 ± 0.021	0.476 ± 0.018	**0.394** ± 0.006	0.437 ± 0.018	**0.399** ± 0.009
Bike	11,122	17	**0.118** ± 0.057	**0.103** ± 0.029	0.220 ± 0.002	0.362 ± 0.004	0.303 ± 0.004
Kin40k	25,600	8	0.580 ± 0.003	0.587 ± 0.006	**0.099** ± 0.001	0.273 ± 0.025	0.268 ± 0.022
Protein	29,267	9	0.608 ± 0.008	0.623 ± 0.011	**0.536** ± 0.012	0.656 ± 0.010	0.668 ± 0.005
KeggDir	31,248	20	**0.086** ± 0.003	**0.086** ± 0.003	**0.086** ± 0.005	0.104 ± 0.003	0.096 ± 0.001
CTslice	34,240	385	**0.116** ± 0.009	0.132 ± 0.009	0.262 ± 0.448	0.218 ± 0.011	1.003 ± 0.005
KEGGU	40,708	27	0.120 ± 0.001	0.121 ± 0.001	**0.118** ± 0.000	0.130 ± 0.001	0.124 ± 0.002
3DRoad	278,319	3	0.187 ± 0.002	0.186 ± 0.001	**0.101** ± 0.007	0.661 ± 0.010	0.481 ± 0.002
Song	329,820	90	0.914 ± 0.003	0.916 ± 0.003	**0.807** ± 0.024	**0.803** ± 0.002	0.998 ± 0.000
Buzz	373,280	77	0.801 ± 0.002	0.801 ± 0.002	**0.288** ± 0.018	**0.300** ± 0.004	**0.304** ± 0.012
HouseElec	1,311,539	11	**0.029** ± 0.001	**0.029** ± 0.001	0.055 ± 0.000	—	0.084 ± 0.005
PoleTele	9,600	26	5.16 ± 0.58		**0.69** ± 0.018	1.16 ± 0.34	1.15 ± 0.068
Elevators	10,623	18	2.6 ± 0.19		**0.68** ± 0.012	1.16 ± 0.38	1.27 ± 0.092
Bike	11,122	17	2.68 ± 0.15		**0.69** ± 0.015	1.17 ± 0.38	1.28 ± 0.093
Kin40k	25,600	8	1.44 ± 0.028		**0.71** ± 0.045	1.62 ± 0.96	3.26 ± 0.23
Protein	29,267	9	2.92 ± 0.2		**0.8** ± 0.17	2.27 ± 0.9	3.31 ± 0.27
KeggDir	31,248	20	7.14 ± 0.39		**0.85** ± 0.1	2.2 ± 1.09	3.8 ± 0.38
CTslice	34,240	385	52.01 ± 0.92		**3.32** ± 5.0	**2.16** ± 0.99	3.87 ± 0.34
KEGGU	40,708	27	7.46 ± 0.54		6.32* ± 0.41*	**2.22** ± 1.05	4.78 ± 0.4
3DRoad	278,319	3	**1.93** ± 0.12		126.37* ± 20.92*	12.01 ± 5.51	34.09 ± 3.19
Song	329,820	90	31.87 ± 3.79		33.79* ± 10.45*	**7.89** ± 3.12	39.55 ± 3.08
Buzz	373,280	77	**20.18** ± 6.66		571.15* ± 66.34*	**29.25** ± 18.33	46.35 ± 2.93
HouseElec	1,311,539	11	**118.41** ± 3.93		575.64* ± 6.94*	—	367.71 ± 4.7

Table 1: NLL (top), RMSE (middle), and run time in minutes (bottom) on regression datasets, using a single GPU (Tesla V100-SXM2-16GB for BT and BTE and Tesla V100-SXM2-32GB for the other methods). The asterisk indicates an estimate of the time from the reported training time on 8 GPUS, assuming linear speedup in number of GPUs and independent noise in training times per GPU.

8 Discussion

We have proven that the binary tree kernel GP is scalable. Our empirical results suggest that it often outpredicts not just other scalable methods, but even the popular Matérn GP. If the results in this paper replicate in other domains, it could obviate wide usage of classic GP kernels like the Matérn kernel, as well as inducing point kernels. Sometimes, our kernel fails to capture patterns in the data; some functions' values simply do not covary this way. But other kernels we tested seemed to fail like that even more.

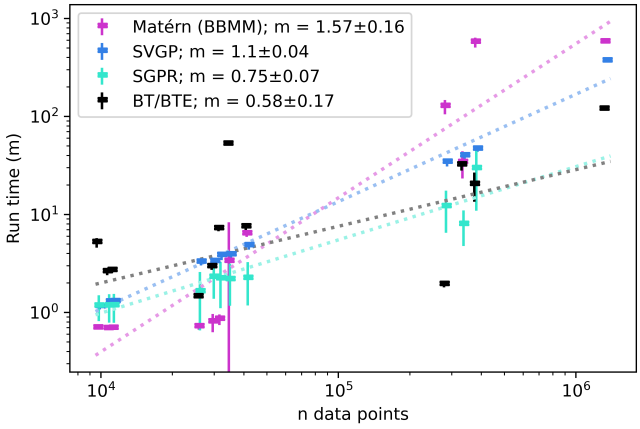

Figure 5: Run times given dataset size. For BT, the trendline is calculated controlling for $\log(q)$ with affine regression, and then setting $q = 150$. The slope w.r.t. $\log(q)$ is 2.82 ± 1.06. Theoretically, all slopes are too low except for that of SVGP, presumably because of overhead in the small-data regime.

The SROS representation of linear operators, which we developed for this kernel, is conceptually general, and could potentially be useful in other domains where linear algebra is applied. Low-rank approximations are widely used to speed up linear transformations, and SROS representations might be an alternative in many settings.

Our contributions to linear algebra and kernel design may significantly increase the size of data sets on which GPs can do state-of-the-art modelling.

References

[1] Zizhong Chen and Jack J Dongarra. Condition numbers of Gaussian random matrices. *SIAM Journal on Matrix Analysis and Applications*, 27(3):603–620, 2005.

[2] Kurt Cutajar, Michael Osborne, John Cunningham, and Maurizio Filippone. Preconditioning kernel matrices. In *International conference on machine learning*, pages 2529–2538. PMLR, 2016.

[3] Dheeru Dua and Casey Graff. UCI machine learning repository, 2017. URL `http://archive.ics.uci.edu/ml`.

[4] Philippe Duchon, Philippe Flajolet, Guy Louchard, and Gilles Schaeffer. Boltzmann samplers for the random generation of combinatorial structures. *Comb. Probab. Comput.*, 13(4–5): 577–625, jul 2004. ISSN 0963-5483. doi: 10.1017/S0963548304006315.

[5] Vincent Dutordoir, Nicolas Durrande, and James Hensman. Sparse Gaussian processes with spherical harmonic features. In *International Conference on Machine Learning*, pages 2793–2802. PMLR, 2020.

[6] Roger Fletcher. *Practical methods of optimization*. John Wiley & Sons, New York, 2nd edition, 2013.

[7] Jacob Gardner, Geoff Pleiss, Kilian Q Weinberger, David Bindel, and Andrew G Wilson. Gpytorch: Blackbox matrix-matrix Gaussian process inference with gpu acceleration. *Advances in neural information processing systems*, 31, 2018.

[8] Jouni Hartikainen and Simo Särkkä. Kalman filtering and smoothing solutions to temporal Gaussian process regression models. In *2010 IEEE international workshop on machine learning for signal processing*, pages 379–384. IEEE, 2010.

[9] James Hensman, Nicolò Fusi, and Neil D. Lawrence. Gaussian processes for big data. In *Proceedings of the Twenty-Ninth Conference on Uncertainty in Artificial Intelligence*, UAI'13, page 282–290, Arlington, Virginia, USA, 2013. AUAI Press.

[10] RE Kalman. A new approach to linear filtering and prediction problems. *Journal of Basic Engineering*, 83(1):95–108, 1960.

[11] Miguel Lázaro-Gredilla, Joaquin Quinonero-Candela, Carl Edward Rasmussen, and Aníbal R Figueiras-Vidal. Sparse spectrum Gaussian process regression. *The Journal of Machine Learning Research*, 11:1865–1881, 2010.

[12] Julien-Charles Lévesque, Audrey Durand, Christian Gagné, and Robert Sabourin. Bayesian optimization for conditional hyperparameter spaces. In *2017 International Joint Conference on Neural Networks (IJCNN)*, pages 286–293, 2017.

[13] Xingchen Ma and Matthew Blaschko. Additive tree-structured covariance function for conditional parameter spaces in Bayesian optimization. In Silvia Chiappa and Roberto Calandra, editors, *Proceedings of the Twenty Third International Conference on Artificial Intelligence and Statistics*, volume 108 of *Proceedings of Machine Learning Research*, pages 1015–1025. PMLR, 26–28 Aug 2020.

[14] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. URL `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`.

[15] Tomaso Poggio, Gil Kur, and Andrzej Banburski. Double descent in the condition number. *arXiv preprint arXiv:1912.06190*, 2019.

[16] Joaquin Quinonero-Candela and Carl Edward Rasmussen. A unifying view of sparse approximate Gaussian process regression. *The Journal of Machine Learning Research*, 6:1939–1959, 2005.

[17] Edward Snelson and Zoubin Ghahramani. Sparse Gaussian processes using pseudo-inputs. In Y. Weiss, B. Schölkopf, and J. Platt, editors, *Advances in Neural Information Processing Systems*, volume 18. MIT Press, 2005.

[18] Michalis Titsias. Variational learning of inducing variables in sparse Gaussian processes. In David van Dyk and Max Welling, editors, *Proceedings of the Twelth International Conference on Artificial Intelligence and Statistics*, volume 5 of *Proceedings of Machine Learning Research*, pages 567–574, Hilton Clearwater Beach Resort, Clearwater Beach, Florida USA, 16–18 Apr 2009. PMLR. URL `https://proceedings.mlr.press/v5/titsias09a.html`.

[19] Michalis Titsias. Variational learning of inducing variables in sparse Gaussian processes. In *Artificial intelligence and statistics*, pages 567–574. PMLR, 2009.

[20] William N Venables and Brian D Ripley. Tree-based methods. In *Modern applied statistics with S-Plus*, pages 303–327. Springer, 1999.

[21] Ke Wang, Geoff Pleiss, Jacob Gardner, Stephen Tyree, Kilian Q Weinberger, and Andrew Gordon Wilson. Exact Gaussian processes on a million data points. *Advances in Neural Information Processing Systems*, 32, 2019.

[22] Veit Wild, Motonobu Kanagawa, and Dino Sejdinovic. Connections and equivalences between the Nyström method and sparse variational Gaussian processes. *arXiv preprint arXiv:2106.01121*, 2021.

[23] Christopher Williams and Matthias Seeger. Using the Nyström method to speed up kernel machines. In T. Leen, T. Dietterich, and V. Tresp, editors, *Advances in Neural Information Processing Systems*, volume 13. MIT Press, 2000. URL `https://proceedings.neurips.cc/paper/2000/file/19de10adbaa1b2ee13f77f679fa1483a-Paper.pdf`.

[24] Christopher K Williams and Carl Edward Rasmussen. *Gaussian processes for machine learning*, volume 2. MIT press Cambridge, MA, 2006.

[25] Andrew Wilson and Hannes Nickisch. Kernel interpolation for scalable structured gaussian processes (kiss-gp). In *International conference on machine learning*, pages 1775–1784. PMLR, 2015.

[26] Yunong Zhang, William E Leithead, and Douglas J Leith. Time-series Gaussian process regression based on Toeplitz computation of $O(n^2)$ operations and $O(n)$-level storage. In *Proceedings of the 44th IEEE Conference on Decision and Control*, pages 3711–3716. IEEE, 2005.

A Correctness of Algorithm 2

In this section, we show that Algorithm 2 performs approximate inversion. And then we show how to modify the algorithm in the setting were all columns of U are identical, for a factor of q speedup.

We begin by writing the exact form of $(u')^{p=j}$ from the proof of Theorem 1, and the corresponding $c'^{(j)}$. Recall the Sherman–Morrison Formula:

$$(A + cuu^\top)^{-1} = A^{-1} - \frac{A^{-1}uu^\top A^{-1}}{c^{-1} + u^\top A^{-1}u}$$

At the time that we update A with the rank one matrix $c^{(j)}u^{p=j}{u^{p=j}}^\top$, what does A equal? Let c and u originate from the i^{th} column of C and U. So our update is $C^{(j)}_{:,i}U^{P_{:,i}=j}_{:,i}{U^{P_{:,i}=j}_{:,i}}^\top$.

Then $A \leftarrow A_{i+1} = I + \sum_{k=i+1}^{q}\sum_{\ell\in[m]} C^{(\ell)}_{:,k}U^{P_{:,k}=\ell}_{:,k}{U^{P_{:,k}=\ell}_{:,k}}^\top$, recalling m is the largest integer in P. And likewise, all the relevant entries of C' and U' have been calculated for A^{-1}. So we have $A^{-1}_{i+1} = I + \sum_{k=i+1}^{q}\sum_{\ell\in[m]} {C'}^{(\ell)}_{:,k}{U'}^{P_{:,k}=\ell}_{:,k}{{U'}^{P_{:,k}=\ell}_{:,k}}^\top$.

The main computation we need to do is $A^{-1}u$. We'll say that $k_\ell \sqsubseteq i_j$ if set j from partition $P_{:,i}$ is a superset of set ℓ from partition $P_{:,k}$. That is, the rows where $P_{:,k}$ takes the value ℓ are a subset of the rows where $P_{:,i}$ takes the value j. The key simplification we use is that if $k_\ell \not\sqsubseteq i_j$, then ${U^{P_{:,i}=j}_{:,i}}^\top {U'}^{P_{:,k}=\ell}_{:,k} = 0$. The rows at which those two vectors have nonzero elements are disjoint. This logic is explained in the proof of Theorem 1 without all the notation.

So we let

$$ {U'}^{P_{:,i}=j}_{:,i} = A^{-1}_{i+1} U^{P_{:,i}=j}_{:,i} = \left(I + \sum_{k=i+1}^{q} \sum_{\ell \in [m]} C'^{(\ell)}_{:,k} {U'}^{P_{:,k}=\ell}_{:,k} {U'}^{P_{:,k}=\ell^\top}_{:,k} \right) U^{P_{:,i}=j}_{:,i} \tag{5} $$

$$ = \left(I + \sum_{k=i+1}^{q} \sum_{\ell : k_\ell \sqsubseteq i_j} C'^{(\ell)}_{:,k} {U'}^{P_{:,k}=\ell}_{:,k} {U'}^{P_{:,k}=\ell^\top}_{:,k} \right) U^{P_{:,i}=j}_{:,i} \tag{6} $$

$$ = U^{P_{:,i}=j}_{:,i} + \sum_{k=i+1}^{q} \sum_{\ell : k_\ell \sqsubseteq i_j} C'^{(\ell)}_{:,k} {U'}^{P_{:,k}=\ell}_{:,k} {U'}^{P_{:,k}=\ell^\top}_{:,k} U^{P_{:,k}=\ell}_{:,i} \tag{7} $$

Equation 7 follows because the terms in the dot product ${U'}^{P_{:,k}=\ell^\top}_{:,k} U^{P_{:,i}=j}_{:,i}$ are only nonzero when $P_{:,k} = \ell$ and $P_{:,i} = j$, but the latter is implied by $k_\ell \sqsubseteq i_j$, so this is equivalent to the elements where $P_{:,k} = \ell$. So ${U'}^{P_{:,k}=\ell^\top}_{:,k} U^{P_{:,i}=j}_{:,i} = {U'}^{P_{:,k}=\ell^\top}_{:,k} U^{P_{:,k}=\ell}_{:,i}$, which gives us Equation 7. Then, we let

$$ C'^{(j)}_{:,i} = \frac{-1}{1/C^{(j)}_{:,i} + {U'}^{P_{:,i}=j^\top}_{:,i} U^{P_{:,i}=j}_{:,i}} \tag{8} $$

Proposition 3 (Correctness of Algorithm 2)**.** *In Algorithm 2, U' and C' take the values defined in Equations 7 and 8.*

Proof. We'll assume that for $k > i$, ${C'}_{:,k}$ and ${U'}_{:,k}$ have the right values, and we'll show that ${C'}_{:,i}$ and ${U'}_{:,i}$ get the right values. Starting with $U'_{:,q}$, the $\sum_{k=q+1}^{q}$ in Equation 7 is empty, so $U'_{:,q} = U_{:,q}$. In Algorithm 2, U' is initialized to U, and the q^{th} column is never updated.

Now we see that ${C'}_{:,i}$ is correct assuming ${U'}_{:,i}$ is. In Line 5, we ensure $z_j = C^{(j)}_{:,i} {U'}^{P_{:,i}=j^\top}_{:,i} U^{P_{:,i}=j}_{:,i}$. A dot product is the sum of elementwise multiplications, and one can inspect that each such multiplication gets added to the right slot in z. Then, Line 6 implements Equation 8, with numerator and denominator multiplied by $C^{(j)}_{:,i}$. (It stores the same value in multiple locations).

Now we turn to ${U'}_{:,i}$. ${U'}_{:,i}$ is updated in every preceding loop. It starts out initialized to $U_{:,i}$, which accounts for the first term in Equation 7. In the sum from $k = i + 1$ to q, each term is accounted for in a separate loop of the algorithm. We check that each term gets added at some point. So consider the term $\sum_{\ell : k_\ell \sqsubseteq i_j} C'^{(\ell)}_{:,k} {U'}^{P_{:,k}=\ell}_{:,k} {U'}^{P_{:,k}=\ell^\top}_{:,k} U^{P_{:,i}=j}_{:,i}$.

This term gets added to $U_{:,i}$ when i from Algorithm 2 equals k from Equation 7, and when k from Algorithm 2 equals i from Equation 7. We are very sorry about this correspondence, but it would have to happen either here or above in the discussion of C'. Observe that in Line 10, $y_{p_j k}$ takes the value ${(U')}^{P_{:,i}=p_j^\top}_{:,i} U^{P_{:,i}=p_j}_{:,k}$. Then, in Line 11, $y_{p_j k}$ gets multiplied by ${(U')}^{P_{:,i}=p_j}_{:,i}$ and $C'^{(p_j)}_{:,i}$, and added to $U'_{:,k}$. Swapping the i's and k's, and letting p_j from Algorithm 2 equal ℓ from Equation 7, Lines 10 and 11 add to $U'_{:,k}$ the terms in Equation 7.

Thus, doing induction from $i = q$ down to 1, Algorithm 2 assigns $C'_{:,i}$ and $U'_{:,i}$ the correct values. □

Now, we modify Algorithm 2 for the setting where all the columns $U_{:,i}$ are the same, allowing a speedup of $O(q)$.

Proposition 4 ($O(nq)$ inversion)**.** *Algorithm 4 performs approximate inversion in $O(nq)$ time.*

Proof. The fact that Algorithm 4 runs in $O(nq)$ time is easily verified. Up to Line 12, Algorithm 4 is the same as Algorithm 2, except U has been replaced with $u(\mathbf{1}^q)^\top$. So all we have to show is that Lines 13-16 produce the same result that Lines 10-12 would have.

Algorithm 4 Inverse and determinant of $I+$ SROS Linear Operator, in which all columns of U are the same.

Require: $P \in [m]^{n\times q}, C \in \mathbb{R}^{n\times q}, u \in \mathbb{R}^n$
Ensure: $I + L(P, C', U') = (I + L(P, C, u(\mathbf{1}^q)^\top))^{-1}$; $x = \log|I + L(P, C, u(\mathbf{1}^q)^\top)|$

$x, C', U' \leftarrow 0, \mathbf{0} \in \mathbb{R}^{n\times q}, u(\mathbf{1}^q)^\top$
for $i \in (q, q-1, ..., 1)$ **do** ▷ $O(nq)$ time
  $p, c, u' \leftarrow P_{:,i}, C_{:,i}, U'_{:,i}$
  $z \leftarrow \mathbf{0} \in \mathbb{R}^m$ ▷ z_i will store $c^{(i)}((u')^{p=i})^\top u^{p=i}$, where $c_k = c^{(i)}$ if $p_k = i$
  for $j \in [n]$ **do** $z_{p_j} \leftarrow z_{p_j} + c_j u'_j u_j$ ▷ $O(n)$ time
  for $j \in [n]$ **do** $C'_{ji} \leftarrow -c_j/(1 + z_{p_j})$ ▷ $O(n)$ time
  for $i \in [m]$ **do** $x \leftarrow x + \log(1 + z_i)$ ▷ $O(n)$ time
  if i > 0 **then**
    if False **then** ▷ This block is the slow version. What follows below is equivalent.
      $y \leftarrow \mathbf{0} \in \mathbb{R}^{n\times i}$
      for $j, k \in [n] \times [i-1]$ **do** $y_{p_j k} \leftarrow y_{p_j k} + u'_j u_j$
      for $j, k \in [n] \times [i-1]$ **do** $U'_{jk} \leftarrow U'_{jk} + C'_{ji} u'_j y_{p_j k}$
    $y \leftarrow \mathbf{0} \in \mathbb{R}^n$ ▷ $O(n)$ time
    $U'_{:,(i-1)} \leftarrow U'_{:,i}$ ▷ $O(n)$ time
    for $j \in [n]$ **do** $y_{p_j} \leftarrow y_{p_j} + u'_j u_j$ ▷ $O(n)$ time
    for $j \in [n]$ **do** $U'_{j(i-1)} \leftarrow U'_{j(i-1)} + C'_{ji} u'_j y_{p_j}$ ▷ $O(n)$ time
return C', U', x

Observe that at the start of the algorithm, $U'_{:,k}$ and $U'_{:,k+1}$ are initialized to the same value. Observe that Lines 11 and 12 repeat the same computation $i-1$ times. So in Lines 11 and 12, $U'_{:,k}$ and $U'_{:,k+1}$ are updated by the same amount if $i > k+1$, and they aren't updated at all if $i \le k$. So the difference between $U'_{:,k}$ and $U'_{:,k+1}$ comes from only the former being updated when $i = k+1$.

Therefore, Line 14 initializes $U'_{:,k}$ to $U'_{:,k+1}$. Then Lines 15 and 16 update $U'_{:,k}$ with the appropriate difference; they copy Lines 11 and 12, setting k to $i-1$. □

B Gradient of loss with respect to weights

In this section, we show how to calculate ∇_w NLL in $O(nq^2)$ time. Recall:

$$\text{NLL}(w) = \frac{1}{2}\left(y^\top (K_{XX}(w) + \lambda I)^{-1} y + \log|K_{XX}(w) + \lambda I| + n\log(2\pi)\right). \tag{9}$$

Differentiating gives:

$$\frac{\partial \text{NLL}}{\partial w_i} = \frac{1}{2}\left[-y^\top (K_{XX} + \lambda)^{-1} \frac{\partial K_{XX}}{\partial w_i}(K_{XX} + \lambda)^{-1} y + \text{Tr}\left(\frac{\partial K_{XX}}{\partial w_i}(K_{XX} + \lambda)^{-1}\right)\right] \tag{10}$$

We begin by evaluuating $\partial K_{XX}/\partial w_i$. Recall from Proposition 2 that $K_{XX} = L(P, C, U)$, where $C = \mathbf{1}^n w^\top$, and $U = \mathbf{1}^{n\times q}$. It follows easily from the definition of L that the elements of $L(P, C, U)$ are linear in the elements of C. So,

$$\frac{\partial K_{XX}}{\partial w_i} = L(P_{:,i}, \mathbf{1}^n, \mathbf{1}^n) \tag{11}$$

Algorithm 3 shows how to calculate $(K_{XX}+\lambda)^{-1}$ in $O(nq)$ time, and represent it as $L(P, C^{-1}, U^{-1})$. (Recall C^{-1} and U^{-1} are not true inverses; we just the notation to denote their purpose.) Now we turn to the question of how to calculate $\text{Tr}[L(P_{:,i}, \mathbf{1}^n, \mathbf{1}^n) L(P, C^{-1}, U^{-1})]$. We are considering symmetric matrices, so the trace of the product is the sum of the elements of the elementwise product. We expand and simplify:

$$T := \operatorname{Tr}\left[L(P_{:,i}, \mathbf{1}^n, \mathbf{1}^n) L(P, C^{-1}, U^{-1})\right] \tag{12}$$

$$= \sum_{j=1}^{q} \operatorname{Tr}\left[L(P_{:,i}, \mathbf{1}^n, \mathbf{1}^n) L(P_{:,j}, C_{:,j}^{-1}, U_{:,j}^{-1})\right] \tag{13}$$

$$\overset{(a)}{=} \sum_{j=1}^{q} \operatorname{Tr}\left[L(P_{:,\max(i,j)}, \mathbf{1}^n, \mathbf{1}^n) L(P_{:,\max(i,j)}, C_{:,j}^{-1}, U_{:,j}^{-1})\right] \tag{14}$$

$$\overset{(b)}{=} \sum_{j=1}^{q} \sum_{\text{elements}} L(P_{:,\max(i,j)}, C_{:,j}^{-1}, U_{:,j}^{-1}) \tag{15}$$

$$\overset{(c)}{=} \sum_{j=1}^{q} \sum_{\ell=1}^{\max(P_{:,\max(i,j)})} \sum_{\text{elements}} (C_{:,j}^{-1})^{(\ell)} (U_{:,j}^{-1})^{P_{:,\max(i,j)}=\ell} (U_{:,j}^{-1})^{P_{:,\max(i,j)}=\ell\top} \tag{16}$$

$$= \sum_{j=1}^{q} \sum_{\ell=1}^{\max(P_{:,\max(i,j)})} (C_{:,j}^{-1})^{(\ell)} ||(U_{:,j}^{-1})^{P_{:,\max(i,j)}=\ell}||_1^2 \tag{17}$$

where (a) follows from the fact that the $(i,j)^{\text{th}}$ element of $L(p,c,u)$ is zero unless $p_i = p_j$, in which case, it is $c_i u_i u_j$; when multiplying elementwise by $L(p', c', u')$, where p' is a finer partition, $L(p,c,u) \odot L(p',c',u') = L(p',c,u) \odot L(p',c',u')$, because $L(p,c,u)$ and $L(p',c,u)$ only differ on elements where $L(p',c',u')$ is 0 anyway. In the context of (a), $P_{:,\max(i,j)}$ is a finer partition than $P_{:,\min(i,j)}$. (b) follows because we are doing elementwise multiplication between the two matrices; anywhere $L(P_{:,\max(i,j)}, \mathbf{1}^n, \mathbf{1}^n)$ is 0, $L(P_{:,\max(i,j)}, C_{:,j}^{-1}, U_{:,j}^{-1})$ is already 0, and elsewhere, multiplying elements by 1 does not effect the matrix. (c) follows from the construction of L.

It is straightforward to compute this in $O(nq)$ time. See Algorithm 5, which runs in $O(n)$ time and can be iterated over the q terms in Equation 17.

Algorithm 5 Calculate $\sum_{\ell=1}^{\max(p)} c^{(\ell)} ||u^{p=\ell}||_1^2$.

Require: $p \in [m]^n$, $c, u \in \mathbb{R}^n$
Ensure: $x = \sum_{\ell=1}^{m} c^{(\ell)} ||u^{p=\ell}||_1^2$
1: $y \leftarrow \mathbf{0}^m$
2: **for** $j \in [n]$ **do** $y_{p_j} \leftarrow y_{p_j} + \sqrt{c_j} u_j$ ▷ $\sqrt{c_j}$ may be imaginary, but it will later be squared. With modifications, we could avoid complex types.
3: $x \leftarrow 0$
4: **for** $j \in [m]$ **do** $x \leftarrow x + y_j^2$
return x

Now we can see that Equation 10 can be computed for all w_i in $O(nq^2)$ time. First, $(K_{XX} + \lambda)^{-1}$ can be computed in $O(nq)$ time, in the form $L(P, C^{-1}, U^{-1})$, as shown in Algorithm 3. Then, $z = (K_{XX} + \lambda)^{-1} y$ can be computed in $O(nq)$ time, also as shown in Algorithm 3. Then, for each $i \in [q]$, we can calculate $-z^\top \frac{\partial K_{XX}}{\partial w_i} z = -z^\top L(P_{:,i}, \mathbf{1}^n, \mathbf{1}^n) z$ in $O(n)$ time. Thus, handling the first term in Equation 10 for all i takes a total of $O(nq)$ time. As just shown, the second term can be computed in $O(nq)$ time for each i, giving a total run time of $O(nq^2)$.

C Experimental Details

We initialize 160 random bit orders. For each one, we initialize three weight vectors w: uniform, uniform except the last bit is 0.5, and uniform except the last bit is 0.9. Out of these 480 initializations, we draw 20 samples via Boltzman sampling [4] on the log likelihood of the training data (after standardizing the values to have zero mean and unit variance). Then, we optimize the weights and bit order with BFGS as described in Section 5, using line search with Wofle conditions, with no extra gradient computations during line search. This allows fewer calculations of the gradient relative to

the cheaper calculation of the loss. The BT column in Table 1 refers to the performance of the binary tree kernel, using the weights and bit order that gave the lowest training NLL out of these 20 trained models.

BTE produces a Gaussian mixture model at each predictive location, mixing over the predictive Gaussians produced by each of these 20 trained models. The relative weights of each Gaussian in the mixture depends on the training NLL of the model that produced it. We weight the models according to the softmax of the *per-data-point* NLL with a temperature of 0.01.

We follow the same train/test/validation splits as Wang et al. [21], but we never use the validation set, which the methods we compare against need. Thus, we could add the validation data to the training data for the binary tree kernel and call it a fair comparison, but we didn't do this, so as not to confuse the origin of the binary tree kernel's success.

D Societal Impacts

More accurate machine learning models allow for better decision making. Improvements in decision making can lead to improved societal outcomes in many applications such as early disease detection. However when predictions of a machine learning algorithm are confident, we may be more compelled to act on them, which could lead to increased risk in delicate settings. Therefore, confident and wrong predictions, resulting from poor and confident generalization due to overfitting, can lead to worse outcomes. When using binary tree kernels for Gaussian processes, overfitting is possible. That said, Gaussian processes do quantify their uncertainty very naturally, which can be helpful for knowing when to take their predictions with a grain, or a heap, of salt. As with all machine learning models, what outcomes are predicted and how those predictions are used are ultimately decided by to the practitioner, and those decisions contribute to whether the model impacts society in positive or negative ways.