A Linear-Time, Infinite-Dimensional Extension of any Finite-Dimensional Kernel

Michael K. Cohen MICHAEL.COHEN@ENG.OX.AC.UK
Department of Engineering Science
University of Oxford
Oxford, UK OX1 3PJ

Michael A. Osborne MOSB@ROBOTS.OX.AC.UK
Department of Engineering Science
University of Oxford
Oxford, UK OX1 3PJ

Abstract

Kernel methods allow infinite-dimensional function approximation, but a key problem is inverting a kernel matrix, generally taking $\Theta(n^2)$ space and $\Theta(n^3)$ time. What if we can only afford linear complexity? The usual answer is a finite-dimensional kernel, but such a kernel is often less expressive. We devise a method to extend any finite-dimensional kernel to an infinite-dimensional one, while maintaining a complexity of $O(n)$ time and space. We multiply any finite-dimensional kernel by Cohen et al.'s (2022) Binary Tree kernel, which is a linear-time, infinite-dimensional kernel. But to do linear-time regression with this product kernel, we must develop a novel matrix representation and algorithms for the relevant matrix operations. Generally, finite-dimensional kernel methods take $\Theta(\dim^2)$ time and $\Theta(\dim)$ space. Under assumptions about the data distribution, we achieve the same complexities; without those assumptions, we incur a multiplicative cost of $O(\dim)$. On a borrowed suite of experimental benchmarks, we test a model finite-dimensional kernel (a Sparse Gaussian Process Regression Kernel approximating a Matérn 3/2 kernel), and compare it to the performance of our infinite-dimensional extension at fixed memory budget. On one dataset, our method is worse, on two, comparable, and on nine, better, as judged by root mean squared error.

Keywords: Gaussian processes, kernel methods, decision trees, sparse matrix representations, fast linear algebra

1 Introduction

Gaussian processes (GPs) can be used to approximate unknown functions while quantifying uncertainty. But doing inference with a GP requires inverting a matrix. Absent any special trick, with n the number of data points, this takes $O(n^3)$ time—too much for many practical applications. So an active area of research is how to make GPs faster.

Usually, this involves using a finite-dimensional kernel (sometimes one that approximates the kernel we really care about), but finite-dimensional kernels are often less good at modelling the true underlying function. Cohen et al. (2022) developed an infinite-dimensional kernel for which the GP can be computed in log-linear time (actually in linear time, as we show). This GP is also of limited expressiveness, with a posterior mean that is piecewise flat. We show that one can use Cohen et al.'s (2022) "binary tree GP" to extend any finite-dimensional kernel to an infinite-dimensional one, while still running in linear time. The new kernel is the product of the two. Achieving this complexity requires developing a novel representation of certain linear operators; we call this representation a *tree matrix*. A tree matrix, which is stored in linear space, can be inverted or multiplied by a vector in linear time. In a tree matrix, the rows and columns are placed on the leaves of a binary tree, and computations are generally executed iteratively from leaves to root and/or vice versa. This is the core of our contribution and the origin of our formal results.

In general, extending the finite-dimensional kernel to infinite dimensions brings a multiplicative overhead equal to the number of dimensions. However, under certain assumptions about the data distribution, there is no increase in the asymptotic complexity; the extension is "for free".

Using a typical finite-dimensional kernel—a Sparse Gaussian Process Regression (SGPR) Kernel approximating a Matérn 3/2 kernel—we test our method empirically on a suite of regression benchmarks. Changing the dimension of the SGPR kernel changes the memory requirements, and our method uses more memory for a fixed dimension, so for a fair comparison, we compare the two methods' performance as a function of their memory usage. As measured by negative log likelihood, our method does worse on two datasets, comparably on two more, and robustly better on the other eight. As measured by root mean square error, our method does worse on one dataset, comparably on two, and robustly better on the other nine. This suite of datasets was copied from the literature (Wang et al., 2019), not cherry-picked.

2 Preliminaries

A GP defines a multivariate Gaussian distribution over function values at any finite set of points in a domain. The covariance between the function values at two different locations in the domain $\mathcal{X}$ is determined by a kernel function $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$. This will produce a consistent probability distribution, no matter which points in the domain are chosen, as long as the kernel k is positive semi-definite. That is, for a n-tuple of points $X \in \mathcal{X}^n$, the matrix K_{XX}, where $(K_{XX})_{ij} = k(X_i, X_j)$, must be positive semi-definite.

If you start with a multivariate Gaussian distribution, then once you have observed several coordinates, the resulting conditional distribution will be a new, lower-dimensional multivariate Gaussian. This is how a GP can be used for machine learning. Consider, without loss of generality, a GP with 0 mean, and a kernel k (the problem can be transformed if necessary). In particular, given an n-tuple of training locations $X \in \mathcal{X}^n$ and an n-tuple of (potentially noisily-observed) training targets $y \in \mathbb{R}^n$, and an m-tuple of target locations $X' \in \mathcal{X}^m$, the predictive targets can be modelled by the distribution $\mathcal{N}(\mu, \Sigma)$, where

$$\mu = K_{X'X}(K_{XX} + \lambda I_n)^{-1} y, \tag{1}$$

$$\Sigma = K_{X'X'} - K_{X'X}(K_{XX} + \lambda I_n)^{-1} K_{XX'} + \lambda I_m, \tag{2}$$

and λ is the variance of the observation noise.

Unfortunately, unless the structure of kernel allows for a special trick, computing these quantities to within a desired precision requires $O\big(n^2(n+m)\big)$ time and $O\big(n(n+m)\big)$ space, using standard algorithms for matrix multiplication, although the space requirements can be diminished by recomputing sections of the matrices repeatedly.

3 Low-dimensional kernels

The dimension of a kernel is the dimension of its reproducing kernel Hilbert space (RKHS). For a kernel $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$, the RKHS is defined as

$$\mathrm{RKHS}(k) = \mathrm{span}(\{k(\cdot, x) | x \in \mathcal{X}\}), \tag{3}$$

and then the dimension of this space is the cardinality of a basis set. A Gaussian process with a finite-dimensional kernel can be computed more efficiently if the number of data points exceeds the dimensionality.

One can show that any z-dimensional kernel is equivalent to Bayesian linear regression in $\mathbb{R}^z$ after transforming the data with some transform $f : \mathcal{X} \to \mathbb{R}^z$ (Rasmussen and Williams, 2006, pg. 96). This is also equivalent to GP regression with a dot product kernel $k(x, x') = x^\top x'$ in the transformed space $\mathbb{R}^z$. Using a dot product kernel, $K_{XX'} = XX'^\top$, so Equations 1 and 2 can be computed in $O(z^2(n + m))$ time in $O(z(n + m))$ space.

A popular low dimensional kernel is an inducing points kernel; this usually thought of as a Nyström approximation of a base kernel, but it is also a kernel in its own right (Williams and Seeger, 2000). For a base kernel k, and for inducing points $Z \in \mathcal{X}^z$, the inducing points kernel $k^Z(x, x') = K_{xZ}K_{ZZ}^{-1}K_{Zx'}$. Thus, the kernel matrix $K^Z_{XX'} = K_{XZ}K_{ZZ}^{-1}K_{ZX'}$ has rank at most z. And using the transform $f(x) = K_{ZZ}^{-1/2}K_{Zx}$, this kernel can be replaced with the dot product kernel in the transformed space. Sparse Gaussian Process Regression (SGPR) (Titsias, 2009) and Sparse Variational Gaussian Processes (SVGP) (Hensman et al., 2013) are both ways of using variational inference to optimize the inducing points (and in the case of the latter, also selecting a mean function). This is an approximation of the base kernel in the sense that as the number of inducing points increases, the resulting kernel approximates the base kernel (Titsias, 2009).

4 What does infinite-dimensional mean?

Many commonly used kernels are infinite-dimensional, like the radial basis function kernel or the Matérn family of kernels Rasmussen and Williams (2006, pg. 94). Inducing points kernels, on the other hand, have a finite dimensionality equal to the number of inducing points.

However, when an infinite-dimensional kernel is approximated with finite-precision arithmetic operations, it is technically finite-dimensional.

Proposition 1 (Finite Dimensionality) $\dim k \le |\mathcal{X}|$

Proof If $\mathrm{span}(s) = S$, then $\dim S \le |s|$. $\mathrm{span}(\{k(\cdot, x)|x \in \mathcal{X}\} = \mathrm{RKHS}(k)$, so $\dim k \le |\{k(\cdot, x) : x \in \mathcal{X}\}| = |\mathcal{X}|$. ■

Nobody really cares, because the dimensionality is exponential in the precision of the floating point numbers, while the time complexity is only quadratic in the precision of the floating point numbers, and the space complexity is only linear. ($O(n \log n)$-time algorithms exist for multiplication, but they are not used).

Therefore, for a kernel that is actually implemented on a computer, we need a slightly different definition of infinite dimensionality. A kernel is infinite-dimensional if

1. a dimensionality can be chosen that is arbitrarily high, and

2. the space complexity is $O\big(\log(\dim)\big)$, and the time complexity is $O\big(\log(\dim)^2\big)$

For a standard infinite-dimensional kernel (like an RBF kernel) implemented with finite precision arithmetic, an arbitrarily high dimension can be chosen by increasing the floating point precision; the dimension is exponential in the floating point precision, the space complexity is linear in the floating point precision, and the time complexity is quadratic in the floating point precision (using standard multiplication algorithms). By comparison, for a family of inducing points kernels, we can achieve an arbitrarily high dimensionality by adding inducing points, but the space complexity is $O(\dim^2)$ and the time complexity is $O(\dim^3)$, not even close.

5 Related Work

We have already discussed inducing points kernels, which are among the more widely-used fast kernels. We will now discuss other approaches to linear-time GP regression. This does not cover the whole of the GP literature, of course, or its application to machine learning. For a broader review, see Rasmussen and Williams (2006) or Liu et al. (2020).

GPs over one-dimensional domains are special. With certain kernels, one can track sufficient statistics at various points in the domain, such that these statistics screen off the left from the right; these are known as state space GPs (Särkkä, 2013). This allows for a reduction to a Kalman filtering problem, enabling linear-time computation (Hartikainen and Särkkä, 2010). Or, with any kernel, if the data points are evenly spaced, Cunningham et al. (2008) show how to exploit the Toeplitz structure of the kernel matrix to do log-linear time GP regression.

In domains with more than one dimension, Särkkä and Hartikainen (2012) extend their state space method, but it doesn't scale to high dimensions. Another approach, related to the inducing points methods, is an inducing frequencies method (Hensman et al., 2017). This method resembles Rahimi and Recht's (2007) Variational Fourier Features, but it optimizes those features according to a variational objective. Unfortunately, the number of frequencies required for a given approximation grows exponentially in the dimension of the domain. If the kernel can be written as the sum of kernels over one-dimensional domains, it is possible to extend these methods tractably to higher-dimensional domains. If the data is on a grid, and the kernel can be written as a product of a kernel over each coordinate, then the kernel matrix will have Kronecker structure, enabling fast GP regression (Saatçi, 2012).

Recall that inducing points methods take $O(nz^2)$ time, where z is the number of inducing points, and the dimension of the kernel. If the inducing points are placed on a grid in a d-dimensional domain, and $z = p^d$, then K_{ZZ} has Kronecker structure, enabling inversion in $O(dz)$ space and $O(dz)$ time, and matrix-vector multiplication in $O(dz)$ time (Saatçi, 2012). This is an improvement on the quadratic dependence on z, but it is still not a logarithmic dependence, which would lead us to call it an infinite-dimensional kernel. Moreover, K_{ZX} does not have Kronecker structure. Wilson and Nickisch (2015) approximate K_{ZX} as a sparse matrix times K_{ZZ}. This amounts to approximating $k(x, \cdot)$ as a linear interpolation of various $k(x', \cdot)$, where the x's are nearby inducing points. Unfortunately, because the number

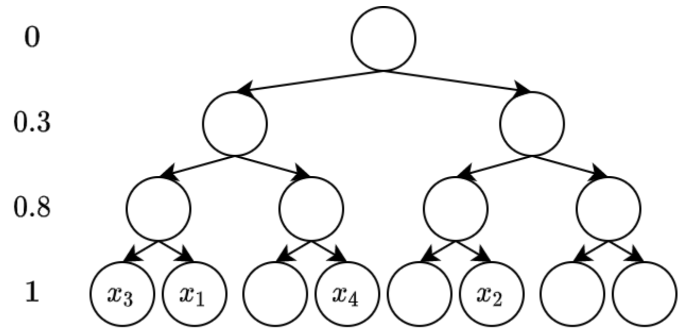

Figure 1: A binary tree kernel with four data points. In this example, $k(x_1, x_1) = 1$, $k(x_1, x_2) = 0$, $k(x_1, x_3) = 0.8$, and $k(x_1, x_4) = 0.3$.

of inducing points is exponential in the dimension, the Kronecker structure method doesn't scale to high-dimensional domains.

6 Binary Tree Kernel

For most infinite-dimensional kernels, GP regression takes $O(n^2(n+m)\log^2(\text{dim}))$ time, but Cohen et al. (2022) introduced a new infinite-dimensional kernel that takes only $O((n+m)\log(n+m)\log(\text{dim}))$ time. And on a suite of regression benchmarks taken from Wang et al. (2019), it actually performed slightly *better* than a standard Matérn 3/2 kernel, in terms of marginal likelihood assigned to the test set.

The binary tree kernel is over the space $\mathbb{B}^q$, where $\mathbb{B} = \{0, 1\}$, and q is the "depth" of the kernel, so the data must first be transformed into this space. This amounts to placing the data on the leaves of a depth-q binary tree, with any desired method. If the starting space is $[0, 1)^d$, they (and we, in our experiments) truncate the binary representation of data points, and shuffle the bits according to a learned permutation. Figure 1 is a diagram of the kernel, taken from Cohen et al. (2022).

Formally, for $x \in \mathbb{B}^q$, let $x^{\leq i}$ be the first i bits of x. And let $[\![\text{expression}]\!]$ evaluate to 1, if expression is true, otherwise 0. The binary tree kernel is defined:

Definition 2 (Binary Tree Kernel) *Given a weight vector $w \in \mathbb{R}^q$, with $w \succeq 0$,*

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

The dimension of this kernel is 2^q, provided $w_q > 0$, although Cohen et al. (2022) do not give a proof.

Proposition 3 (BTGP Dimensionality) *For $w \succ 0$, $\dim k_w = 2^q$.*

Proof Proposition 1 establishes that the dimensionality is no more than 2^q. So it suffices to identify 2^q linearly independent functions in $\text{span}(\{k_w(\cdot, x) : x \in \mathbb{B}^q\})$. We show that the function $[\![x = \cdot]\!]$ is in that span, for any x. These are clearly linearly independent, since they are nonzero at disjoint subsets of the domain.

Let $f_x^r(\cdot) = [\![x^{\leq r} = \cdot^{\leq r}]\!]$. We show by induction on r that this function is in $\mathrm{span}(\{k_w(\cdot, x) : x \in \mathbb{B}^q\})$. Starting with $r = 1$, consider $g_x^1(\cdot) = \sum_{x' : x^{\leq 1} = (x')^{\leq 1}} k_w(\cdot, x')$. This is positive when $\cdot^{\leq 1} = x^{\leq 1}$, 0 when it doesn't, and constant over $\cdot^{\leq 1} = x^{\leq 1}$, by symmetry over all bits beyond the first. So multiplying by a constant gives $f_x^1 \in \mathrm{span}(\{k_w(\cdot, x) : x \in \mathbb{B}^q\})$.

Now assuming that the inductive hypothesis holds for $r' < r$, we show that it also holds for r. Let $g_x^r(\cdot) = \sum_{x' : x^{\leq r} = (x')^{\leq r}} k_w(\cdot, x')$. For $\cdot^{\leq r} = x^{\leq r}$, $g_x^r(\cdot)$ is constant by symmetry, and strictly greater than its value elsewhere because $w_r > 0$. Where $\cdot^{\leq r} \neq x^{\leq r}$, $g_x^r(\cdot) = 2^{q-r} \sum_{r'=1}^{r-1} w_{r'} [\![\cdot^{\leq r'} = x^{\leq r'}]\!]$. By the inductive hypothesis, the r.h.s. is in the span, and by subtracting it from g_x^r, we get a function that is constant and nonzero where $\cdot^{\leq r} = x^{\leq r}$, and zero where $\cdot^{\leq r} \neq x^{\leq r}$. Therefore, f_x^r is in the span, completing the proof by induction. Finally, note that the function $[\![x = \cdot]\!]$ is equal to f_x^q. ■

Cohen et al.'s (2022) Binary Tree GP (BTGP) requires $O((n+m)q)$ space and $O((n+m)\log(n+m)q)$ time, and $q = \log(\mathrm{dim})$, so it qualifies as infinite-dimensional under our definition.[1]

7 Dot Binary Tree Kernel

We both improve and extend the Binary Tree GP. Our improvement is a more efficient algorithm for computing it. Instead of taking $\Theta((n+m)q\log(n+m))$ time, it takes only $O((n+m)q)$ time, and that is because it takes that long to read the data. Setting aside the memory requirements of the data, and the time requirements of reading it (alongside some extremely simple "write" operations), our algorithm implements BTGP in $O(n+m+q)$ time and space, not $\Theta((n+m)\log(n+m)q)$. For precious GPU space, that is the relevant complexity.

Our extension allows us to multiply the binary tree kernel by any finite-dimensional kernel, and efficiently compute the GP.

Definition 4 (Dot Binary Tree Kernel) $k_w^f(x_1, x_2) = k_w(x_1, x_2) f(x_1)^\top f(x_2)$.

This means we can construct the kernel $k_w^Z(x_1, x_2) = k_w(x_1, x_2) k^Z(x_1, x_2)$, for example. When f maps to $\mathbb{R}^z$, our Dot Binary Tree GP (DBTGP) can be computed in $O((n+m)(q+z^2))$ space and $O((n+m)(q+z^3))$ time. That is, both space and time are linear in the number of data points. If the binary tree is balanced, then a factor of z can be removed from each; in that case, we have the same complexity compared to the z-dimensional kernel on its own, but the kernel is now infinite-dimensional! Compared to BTGP, for which the posterior mean is always piecewise constant, the posterior mean of DBTGP is piecewise linear in the transformed z-dimensional space, so the latter is significantly more expressive. Because we can optimize w, we can recover the original finite-dimensional kernel k^Z, by setting $w = (1, 0, ..., 0)$, so DBTGP qualifies as an extension, not just a modification, of the original finite-dimensional kernel.

1. Technically, infinite dimensionality requires an infinite domain, so it is only in that circumstance that the binary tree kernel is infinite-dimensional, as with any kernel. If we do have an infinite domain, let it be $\mathbb{B}^* = \bigcup_{i=0}^{\infty} \mathbb{B}^i$, or else we map the original domain to that one. The binary tree GP requires approximating the infinite binary strings with their first q bits, but we can increase q arbitrarily at linear cost.

8 Tree Matrix Representation

We achieve this time and space complexity with a special representation of the kernel matrix K_{XX} and its inverse. We call it a Tree of Sparse Low Rank Matrices, or just a Tree Matrix. This matrix representation, and the fast operations it affords, might deserve a standalone paper, since its utility could go well beyond GPs.

The tree matrix representation is inspired by Cohen et al.'s (2022) Sparse Rank One Sum representation. This scheme represents a matrix as a sum of rank one matrices, where most of the rank one matrices are the outer product of very sparse vectors. Note that if v has 10 nonzero elements, then $vv^\top$ has 100, so this can be a very space-efficient way of representing linear transformations.

We now define an $m \times n$ tree matrix. Each row and each column of the matrix is assigned to a leaf of a proper binary tree. ("Proper" means that no node has exactly one child). Each node encodes a matrix, and the matrix as a whole is the sum of all the node matrices. Each leaf node contains a matrix which only has nonzero elements at the rows and columns belonging to that leaf, and which has rank no more than z. If a leaf node has no constituent rows or no constituent columns, the matrix is empty. Each leaf's matrix is represented as $VAV'^\top$, where $V \in \mathbb{R}^{m\times z}$, $V' \in \mathbb{R}^{n\times z}$, and $A \in \mathbb{R}^{z\times z}$. A row of V is only nonzero if the row belongs to that leaf, and a row of V' is only nonzero if that column belongs to that leaf. Because each row and column only belong to one leaf, the V- and V'-matrices for all the leaves can be stored in an $m \times z$ and an $n \times z$ array. All other nodes' matrices are represented as $VAV'^\top$, where $V_{\text{node}} = V_{\text{left child}} B_{\text{left}} + V_{\text{right child}} B_{\text{right}}$, and $V'_{\text{node}} = V'_{\text{left child}} B'_{\text{left}} + V'_{\text{right child}} B'_{\text{right}}$. Throughout the paper, when referring to a node "node", "leftc" and "rightc" will refer to its left and right children. For non-leaf nodes, only the A, B, and B' matrices are stored, all $z \times z$. Because the number of non-leaf nodes in a proper binary tree is less than the number of leaf nodes, this requires $O((m+n)z^2)$ space, regardless of the depth of the tree.

Formally, let each node be indexed by a natural number from 0 to $t-1$, where t is the number of nodes in the tree, which is twice the number of leaves minus one. Let 0 be the root node.

Definition 5 (Tree matrix) *A tree matrix $\mathcal{T} = (lc, rc, rl, cl, V, V', A, B_{left}, B_{right}, B'_{left}, B'_{right})$. The arrays lc and $rc \in [t]^t$ store the indices of the of the children of node i at their i^{th} elements. $rl \in [t]^m$ and $cl \in [t]^n$ are vectors that identifies which leaf each row and column belongs to. An $m \times z$ array V and an $n \times z$ array V' contain the V and V' matrices for all the leaf nodes. For a leaf node j, $V_j = V[rl = j]$ (in numpy notation), and $V'_j = V'[cl = j]$. The A, B, and B' matrices defined above are each stored in a $t \times z \times z$ array.*

For a symmetric tree matrix, all $V' = V$, all $B' = B$, and rl = cl. A symmetric tree matrix is depicted in Figure 2.

If a node's descendant leaves contain z or fewer rows (so we'll say the node contains that many rows) and z or fewer columns, it may as well be a leaf node, without any loss of expressivity. The sum of the matrices encoded by it and its descendants cannot have rank greater than z, so that sum can be assigned to the node in question. As we prove later, if the tree is α-weight-balanced, the tree can be pruned to have no more than $\alpha^{-1}n/z$ leaves. And if the weight-balancing by node is stochastic, but the expected balance at each node

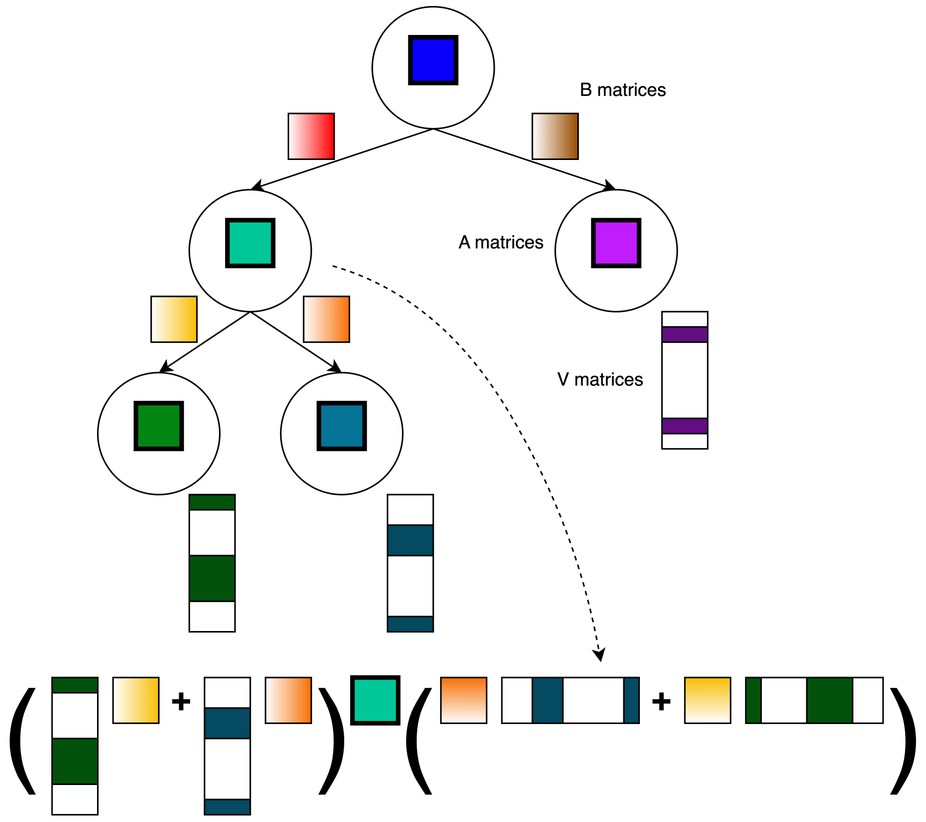

Figure 2: Depiction of a symmetric tree matrix. The visual equation represents the contribution of a single node, and the sum of the contributions of each node gives the matrix as a whole. The white patches of the V matrices represent all 0s.

exceeds α, then the expected number of leaves after pruning is no more than $\alpha^{-1}n/z$, even if the probability that the whole tree is α-weight balanced is quite small. In the balanced case, the tree matrix can be represented in $O(\alpha^{-1}nz)$ space.

For many computations involving a tree matrix, it will be helpful to partition the nodes into a list. Each element of the list is a set of nodes, and every node's parent belongs to an earlier element. Within such a set of nodes, important operations can be done in parallel for each node. Call this list of sets $Q \in \mathcal{P}([t])^q$, where q is an upper bound on the depth of the tree. In our implementation, the indices of each such set of nodes are contiguous, so the subsets of nodes can be represented by two integers—the start and end of the slice.

We now produce algorithms for key operations with tree matrices and note their computational complexity. We show in Algorithm 1 that matrix-vector multiplication can be done in $O(tz^2)$ time $\subset O(nz^2)$ time. All loops can be parallelized, which we do on a GPU, except for the loop over q iterations on Line 5.

For symmetric tree matrices, we develop algorithms for inversion and calculating the log determinant, after adding λI to the tree matrix, all in $O((n + tz)z^2)$ time and $O((n + tz)z)$ space. Without loss of generality, we let $\lambda = 1$, and scale the tree matrix if other values of λ are desired. We develop an algorithm for computing the diagonal of a symmetric tree

Algorithm 1 Matrix-vector multiplication with a tree matrix.

```
Require: a tree matrix $\mathcal{T}$, defined by $\mathrm{lc}, \mathrm{rc} \in [t]^t$, $\mathrm{rl} \in [t]^m$, $\mathrm{cl} \in [t]^n$, $V \in \mathbb{R}^{m \times z}$, $V' \in \mathbb{R}^{n \times z}$,
    $A, B_{\mathrm{left}}, B_{\mathrm{right}}, B'_{\mathrm{left}}, B'_{\mathrm{right}} \in \mathbb{R}^{t \times z \times z}$; $Q \in \mathcal{P}([t])^q$; $x \in \mathbb{R}^n$
Ensure: $y = \mathcal{T} x$
 1: $W' \leftarrow V' \odot x$        ▷ each row of $V'$ is multiplied by the corresponding element of $x$
 2: $D = \mathbf{0} \in \mathbb{R}^{t \times z}$
 3: for $j \in [n]$ do        ▷ parellelizable; $O(nz)$ time
 4:     $D_{\mathrm{cl}_j} \leftarrow D_{\mathrm{cl}_j} + W'_j$        ▷ For all leaf nodes $s$, $D_s$ now contains ${V'}_s^{\top} x$
 5: for nodeslice $\in$ reversed($Q$) do        ▷ leaves to root; $O(tz^2)$ time
 6:     for node $\in$ nodeslice do        ▷ parellelizable
 7:         $D_{\mathrm{node}} \leftarrow (B'_{\mathrm{left}})^{\top}_{\mathrm{node}} D_{\mathrm{lc}_{\mathrm{node}}} + (B'_{\mathrm{right}})^{\top}_{\mathrm{node}} D_{\mathrm{rc}_{\mathrm{node}}}$ ▷ $z \times z$ matrix-vector mult'cation
 8:                 ▷ $O(z^2)$ time; $D_{\mathrm{node}}$ now contains ${V'}^{\top}_{\mathrm{node}} x$
 9: for node $\in [t]$ do        ▷ parellelizable; $O(tz^2)$ time
10:     $D_{\mathrm{node}} \leftarrow A_{\mathrm{node}} D_{\mathrm{node}}$        ▷ $z \times z$ matrix-vector mult'cation; $D_{\mathrm{node}}$ now contains
    $A_{\mathrm{node}} {V'}^{\top}_{\mathrm{node}} x$
11: for nodeslice $\in Q$ do        ▷ root to leaves; $O(tz^2)$ time
12:     for node $\in$ nodeslice do        ▷ parellelizable
13:         $D_{\mathrm{lc}_{\mathrm{node}}} \leftarrow D_{\mathrm{lc}_{\mathrm{node}}} + (B_{\mathrm{left}})_{\mathrm{node}} D_{\mathrm{node}}$
14:         $D_{\mathrm{rc}_{\mathrm{node}}} \leftarrow D_{\mathrm{rc}_{\mathrm{node}}} + (B_{\mathrm{right}})_{\mathrm{node}} D_{\mathrm{node}}$
15:             ▷ leaves will contain their part of ancestor nodes' $B_{\mathrm{leaf}} \ldots B_{\mathrm{node}} A_{\mathrm{node}} {V'}^{\top}_{\mathrm{node}} x$
16: $W \leftarrow \mathbf{0} \in \mathbb{R}^{m \times z}$
17: for $i \in [m]$ do $W_i \leftarrow D_{\mathrm{rl}_i}$        ▷ parellelizable
18: return $y =$ sum-over-columns($W \odot V$)
```

matrix in the same time and space. And for two symmetric tree matrices with the same tree structure, we develop an algorithm for calculating the Frobenius inner product in the same time and space.

We now derive the algorithm that takes a tree matrix $\mathcal{T}$ and identifies another tree matrix T' such that $(\mathcal{T}+I)^{-1} = \mathcal{T}'+I$.

Theorem 6 (Inversion) *Almost everywhere, for a symmetric tree matrix $\mathcal{T}$, there exists another symmetric tree matrix $\mathcal{T}'$ such that $(\mathcal{T}+I)^{-1} = \mathcal{T}'+I$.*

Proof We construct such a tree matrix $\mathcal{T}'$. We can write $\mathcal{T} = \sum_{\mathrm{node} \in \mathrm{nodes}} V_{\mathrm{node}} A_{\mathrm{node}} V^T_{\mathrm{node}}$, and recall $V_{\mathrm{node}} = V_{\text{left child}} (B_{\mathrm{left}})_{\mathrm{node}} + V_{\text{right child}} (B_{\mathrm{right}})_{\mathrm{node}}$. Let $M_{\text{some nodes}} = I + \sum_{\mathrm{node} \in \text{some nodes}} V_{\mathrm{node}} A_{\mathrm{node}} V^T_{\mathrm{node}}$.

First, order the nodes so that all descendants of a node appear before it. Let S_{node} be the set of nodes that appear before the node. We calculate $M^{-1}_{S_{\mathrm{node}} \cup \{\mathrm{node}\}} - M^{-1}_{S_{\mathrm{node}}}$. Then, adding these terms together for every node amounts to building up the tree from the leaves to the root, and updating the inverse as we go. Using the Woodbury Matrix Identity, we have

$$(M_{S_{\mathrm{node}}} + V_{\mathrm{node}} A_{\mathrm{node}} V^{\top}_{\mathrm{node}})^{-1} = M^{-1}_{S_{\mathrm{node}}} - M^{-1}_{S_{\mathrm{node}}} V_{\mathrm{node}} (A^{-1}_{\mathrm{node}} + V^{\top}_{\mathrm{node}} M^{-1}_{S_{\mathrm{node}}} V_{\mathrm{node}})^{-1} V^{\top}_{\mathrm{node}} M^{-1}_{S_{\mathrm{node}}} \quad (4)$$

Letting $V'_{\text{node}} = M^{-1}_{S_{\text{node}}} V_{\text{node}}$, and $A'_{\text{node}} = -(A^{-1}_{\text{node}} + V^{\top}_{\text{node}} M^{-1}_{S_{\text{node}}} V_{\text{node}})^{-1}$, we have $M^{-1}_{S_{\text{node}} \cup \{\text{node}\}} - M^{-1}_{S_{\text{node}}} = V'_{\text{node}} A'_{\text{node}} {V'}^{\top}_{\text{node}}$. So we just need to calculate V'_{node} and A'_{node}.

Now recall that the nonzero rows of V_{leaf} are disjoint for the set of all leaves. And for any node, a row of V_{node} can only be nonzero if it is nonzero in one of its descendant leaves. Therefore, for any pair of nodes where neither is a descendant of the other, the nonzero rows of their V matrices are disjoint. Now, we show that a row of V' is nonzero only if it is nonzero in one of its descendant leaves, like for V. We proceed by induction, node by node, following the order selected above. For the first node, $M_{S_{\text{node}}} = I$, so $V'_{\text{node}} = V_{\text{node}}$. Now note that, by definition,

$$M^{-1}_{S_{\text{node}}} = I + \sum_{\text{node}' \in S_{\text{node}}} V'_{\text{node}'} A'_{\text{node}'} {V'}^{\top}_{\text{node}'}. \tag{5}$$

So for an arbitrary node "node",

$$V'_{\text{node}} = V_{\text{node}} + \sum_{\text{node}' \in S_{\text{node}}} V'_{\text{node}'} A'_{\text{node}'} {V'}^{\top}_{\text{node}'} V_{\text{node}}. \tag{6}$$

node$'$ is not an ancestor of node, given the constraints of the order. Thus, if node$'$ is not a descendant of node, then by the inductive hypothesis, the nonzero rows of $V'_{\text{node}'}$ and V_{node} are disjoint. Therefore ${V'}^{\top}_{\text{node}'} V_{\text{node}} = 0$, unless node$'$ is a descendant of node. So we can write

$$V'_{\text{node}} = V_{\text{node}} + \sum_{\text{node}' \in \text{descendants}(\text{node})} V'_{\text{node}'} A'_{\text{node}'} {V'}^{\top}_{\text{node}'} V_{\text{node}}. \tag{7}$$

Therefore, like V_{node}, V'_{node} can only have nonzero rows if it is nonzero in one of its descendant leaves, because all terms in the sum are left multiplied by a V' matrix that nullifies all other rows.

With that observation, we can efficiently compute and represent V' and A'. Letting *leftc* and *rightc* be the left and right children of node,

$$\begin{aligned}
V'_{\text{node}} &= M^{-1}_{S_{\text{node}}} V_{\text{node}} \\
&= (M^{-1}_{S_{\text{node}} \setminus \{\text{leftc},\text{rightc}\}} + V'_{\text{leftc}} A'_{\text{leftc}} {V'}^{\top}_{\text{leftc}} + V'_{\text{rightc}} A'_{\text{rightc}} {V'}^{\top}_{\text{rightc}}) V_{\text{node}} \\
&= (M^{-1}_{S_{\text{node}} \setminus \{\text{leftc},\text{rightc}\}} + V'_{\text{leftc}} A'_{\text{leftc}} {V'}^{\top}_{\text{leftc}} + V'_{\text{rightc}} A'_{\text{rightc}} {V'}^{\top}_{\text{rightc}}) * \\
&\qquad\qquad (V_{\text{leftc}} (B_{\text{left}})_{\text{node}} + V_{\text{rightc}} (B_{\text{right}})_{\text{node}}) \\
&\overset{(a)}{=} \sum_{\text{side} \in \{\text{left},\text{right}\}} M^{-1}_{S_{\text{node}} \setminus \{\text{leftc},\text{rightc}\}} V_{\text{sidec}} (B_{\text{side}})_{\text{node}} + V'_{\text{sidec}} A'_{\text{sidec}} {V'}^{\top}_{\text{sidec}} V_{\text{sidec}} (B_{\text{side}})_{\text{node}} \\
&\overset{(b)}{=} \sum_{\text{side} \in \{\text{left},\text{right}\}} M^{-1}_{S_{\text{sidec}}} V_{\text{sidec}} (B_{\text{side}})_{\text{node}} + V'_{\text{sidec}} A'_{\text{sidec}} {V'}^{\top}_{\text{sidec}} V_{\text{sidec}} (B_{\text{side}})_{\text{node}} \\
&= \sum_{\text{side} \in \{\text{left},\text{right}\}} V'_{\text{sidec}} (I + A'_{\text{sidec}} {V'}^{\top}_{\text{sidec}} V_{\text{sidec}}) (B_{\text{side}})_{\text{node}}
\end{aligned} \tag{8}$$

where (a) follows because neither rightc nor leftc is a descendant of the other, so ${V'}^{\top}_{\text{rightc}} V_{\text{leftc}} = 0$ and vice versa, and (b) follows because all descendants of sidec appear in both S_{sidec} and

$S_{\text{node}} \setminus \{\text{leftc}, \text{rightc}\}$, no ancestors of sidec appear in either, and all other terms in the inverse M matrix are irrelevant because the nonzero columns are disjoint with the nonzero rows of V_{sidec}.

Thus, we can write $(B'_{\text{left}})_{\text{node}} = (I + A'_{\text{leftc}} {V'}^{\top}_{\text{leftc}} V_{\text{leftc}})(B_{\text{left}})_{\text{node}}$, and likewise for the right, and then $V'_{\text{node}} = V'_{\text{leftc}}(B'_{\text{left}})_{\text{node}} + V'_{\text{rightc}}(B'_{\text{right}})_{\text{node}}$. The only term remaining to calculate is $V^{\top}_{\text{node}} V'_{\text{node}}$, which appears in the definitions of B' and A'.

$$\begin{aligned}
C_{\text{node}} &= V^{\top}_{\text{node}} V'_{\text{node}} \\
&= ((B_{\text{left}})^{\top}_{\text{node}} V^{\top}_{\text{leftc}} + (B_{\text{right}})^{\top}_{\text{node}} V^{\top}_{\text{rightc}})(V'_{\text{leftc}}(B'_{\text{left}})_{\text{node}} + V'_{\text{rightc}}(B'_{\text{right}})_{\text{node}}) \\
&= (B_{\text{left}})^{\top}_{\text{node}} V^{\top}_{\text{leftc}} V'_{\text{leftc}}(B'_{\text{left}})_{\text{node}} + (B_{\text{right}})^{\top}_{\text{node}} V^{\top}_{\text{rightc}} V'_{\text{rightc}}(B'_{\text{right}})_{\text{node}} \\
&= (B_{\text{left}})^{\top}_{\text{node}} C_{\text{leftc}}(B'_{\text{left}})_{\text{node}} + (B_{\text{right}})^{\top}_{\text{node}} C_{\text{rightc}}(B'_{\text{right}})_{\text{node}} \qquad (9)
\end{aligned}$$

And the first equation suffices when node is a leaf. Manipulating the definition of A' into a more numerically stable form gives

$$A'_{\text{node}} = -(A^{-1}_{\text{node}} + C_{\text{node}})^{-1} = -A_{\text{node}}(I_z + C_{\text{node}} A_{\text{node}})^{-1} \qquad (10)$$

Note that this matrix is invertible almost everywhere. (We really ought to define a measure, but any standard measure will do). Note that if all A matrices are positive semidefinite, one can show by induction that all A' and C matrices are as well, in which case, invertibility is guaranteed. Finally, we construct $\mathcal{T}'$ by replacing the A and B matrices of $\mathcal{T}$ with A' and B'. ■

Theorem 7 (Inversion Runtime) *The operator $\mathcal{T} + I$ can be inverted in $O((n + tz)z^2)$, in $O((n + tz)z)$ space.*

Algorithm 2 is the proof, which calculates the quantities in the proof of Theorem 6. Cohen et al.'s (2022) SROS representation can be employed for the same calculation if $z = 1$, but their algorithm for inversion takes $\Theta(nq)$ time, not $O(n)$ time.

We now give an algorithm for computing the diagonal of a tree matrix $\mathcal{T}$.

First, note $\text{diag}(\sum_{\text{nodes}} V_{\text{node}} A_{\text{node}} V^{\top}_{\text{node}}) = \sum_{\text{nodes}} \text{diag}(V_{\text{node}} A_{\text{node}} V^{\top}_{\text{node}})$. Second, note that if we expand $V_{\text{node}} A_{\text{node}} V^{\top}_{\text{node}} = (V_{\text{leftc}}(B_{\text{left}})_{\text{node}} + V_{\text{rightc}}(B_{\text{right}})_{\text{node}}) A_{\text{node}} (V_{\text{leftc}}(B_{\text{left}})_{\text{node}} + V_{\text{rightc}}(B_{\text{right}})_{\text{node}})^{\top}$, any term beginning with V_{leftc} and ending with $V^{\top}_{\text{rightc}}$ or the other way around contributes nothing to the diagonal, because the nonzero rows don't line up with the nonzero columns. Thus,

$$\text{diag}(V_{\text{node}} A_{\text{node}} V^{\top}_{\text{node}}) = \sum_{\text{side} \in \{\text{left}, \text{right}\}} \text{diag}(V_{\text{sidec}}(B_{\text{side}})_{\text{node}} A_{\text{node}} (B_{\text{side}})^{\top}_{\text{node}} V^{\top}_{\text{sidec}}). \qquad (11)$$

This allows us to write an algorithm that modifies the tree matrix while keeping the diagonal invariant. For a given node, we add $(B_{\text{left}})_{\text{node}} A_{\text{node}} (B_{\text{left}})^{\top}_{\text{node}}$ to the A matrix of its left child, do likewise for its right child, and then we zero its own A matrix. This and the trivial final step appear in Algorithm 3.

Finally, we show how to compute the Frobenius inner product between tree matrices that have the same tree structure.

Algorithm 2 Inverse and Log Determinant of Tree Matrix + I.

```
Require: a symmetric tree matrix $\mathcal{T}$, defined by leftc, rightc $\in [t]^t$, leaves $\in [t]^n$, $V \in \mathbb{R}^{n\times z}$,
    $A, B_{\text{left}}, B_{\text{right}}, \in \mathbb{R}^{t\times z\times z}$; $Q \in \mathcal{P}([t])^q$
Ensure: $\mathcal{T}' + I = (\mathcal{T} + I)^{-1}$; $x = \log|\mathcal{T} + I|$
 1: $B'_{\text{left}}, B'_{\text{right}}, A', C \leftarrow \mathbf{0} \in \mathbb{R}^{t\times z\times z}$
 2: for $i \in [n]$ do                                         ▷ parallelizable, $O(nz^2)$ time
 3:     $C_{\text{leaves}_i} \leftarrow C_{\text{leaves}_i}$ + outer-product$(V_i, V_i)$
 4: for leaf $\in$ leaves do                                    ▷ parallelizable; $O(tz^3)$ time
 5:     $A'_{\text{leaf}} \leftarrow -A_{\text{leaf}}(I_z + C_{\text{leaf}}A_{\text{leaf}})^{-1}$
 6: for nodeslice $\in$ reversed$(Q)$ do                         ▷ leaves to root; $O(tz^3)$ time
 7:     for node $\in$ nodeslice do                                          ▷ parellelizable
 8:         for side $\in$ {left, right} do
 9:             $(B'_{\text{side}})_{\text{node}} \leftarrow (I_z + A'_{\text{sidec}}C_{\text{sidec}})(B_{\text{side}})_{\text{node}}$          ▷ $O(z^3)$ time
10:             $C_{\text{node}} \leftarrow C_{\text{node}} + (B_{\text{side}})^\top_{\text{node}}C_{\text{sidec}_{\text{node}}}(B'_{\text{side}})_{\text{node}}$     ▷ $O(z^3)$ time
11:         $A'_{\text{node}} \leftarrow -A_{\text{node}}(I_z + C_{\text{node}}A_{\text{node}})^{-1}$                     ▷ $O(z^3)$ time
12: $x \leftarrow 0$
13: for node $\in [t]$ do                                      ▷ parallelizable; $O(tz^3)$ time
14:     $x \leftarrow x + \log|I_z + A_{\text{node}}C_{\text{node}}|$                                ▷ $O(z^3)$ time
15: $V'$, leftc$'$, rightc$'$, leaves$'$ $\leftarrow V$, leftc, rightc, leaves
16: return $\mathcal{T}'$, $x$
```

Algorithm 3 Diagonal of Tree Matrix.

```
Require: a symmetric tree matrix $\mathcal{T}$, defined by leftc, rightc $\in [t]^t$, leaves $\in [t]^n$, $V \in \mathbb{R}^{n\times z}$,
    $A, B_{\text{left}}, B_{\text{right}}, \in \mathbb{R}^{t\times z\times z}$; $Q \in \mathcal{P}([t])^q$
Ensure: $d = \text{diag}(\mathcal{T})$
 1: $H \leftarrow A$                                                     ▷ $O(tz^2)$ space
 2: for nodeslice $\in Q$ do                                   ▷ root to leaves; $O(tz^3)$ time
 3:     for node $\in$ nodeslice do                                          ▷ parellelizable
 4:         for side $\in$ {left, right} do
 5:             $H_{\text{sidec}} \leftarrow H_{\text{sidec}} + (B_{\text{side}})_{\text{node}}H_{\text{node}}(B_{\text{side}})^\top_{\text{node}}$
 6: $d \leftarrow \mathbf{0}_n$
 7: for $i \in [n]$ do                                         ▷ parallelizable; $O(nz^2)$ time
 8:     $d_i \leftarrow V_iH_{\text{leaves}_i}(V_i)^\top$
 9: return $d$
```

Theorem 8 (Frobenius Inner Product) *The Frobenius inner product between two tree matrices $\mathcal{T}$ and $\mathcal{T}'$ with the same tree structure can be computed in $O((n + tz)z^2)$ time.*

Proof We aim to compute $\mathrm{Tr}\left[\left(\sum_{\mathrm{node}\in\mathrm{nodes}} V_{\mathrm{node}} A_{\mathrm{node}} V_{\mathrm{node}}^\top\right)\left(\sum_{\mathrm{node}\in\mathrm{nodes}} V'_{\mathrm{node}} A'_{\mathrm{node}} (V'_{\mathrm{node}})^\top\right)\right]$ $= \mathrm{Tr}\left[\sum_{a,b\in\mathrm{nodes}} V_a A_a V_a^\top V_b' A_b' (V_b')^\top\right]$. But note that unless $a = b$, or a is a descendant of b, or vice versa, then $V_a^\top V_b' = 0$, because the nonzero rows of V_a and V_b' are disjoint. Using the cyclic property of the trace, and letting $C_{a,b} = V_a^\top V_b'$, we rewrite the expression as

$$\sum_{a\in\mathrm{nodes}} \mathrm{Tr}\left(A_a C_{a,a} A_a' C_{a,a}^\top\right) + \sum_{a,b\in\mathrm{nodes}: b \text{ desc of } a} \mathrm{Tr}\left(A_a C_{a,b} A_b' C_{a,b}^\top\right) + \sum_{a,b\in\mathrm{nodes}: a \text{ desc of } b} \mathrm{Tr}\left(A_a C_{a,b} A_b' C_{a,b}^\top\right)$$

We skip the calculation of the third term, because it is exactly the same as the second, except swapping A with A' and $C_{a,b}$ with $C_{a,b}^\top$. We begin with the calculation of $C_{a,a}$.

First, if a is a leaf, $V_a^\top V_a'$ can be computed in rz^2 time, where r is the number of rows belonging to the leaf. Over all the leaves, this takes $O(nz^2)$ time. Otherwise,

$$\begin{aligned} V_a^\top V_a' &= \left((B_{\mathrm{left}})_a^\top V_{\mathrm{leftc}_a}^\top + (B_{\mathrm{right}})_a^\top V_{\mathrm{rightc}_a}^\top\right)\left(V'_{\mathrm{leftc}_a}(B'_{\mathrm{left}})_a + V'_{\mathrm{rightc}_a}(B'_{\mathrm{right}})_a\right) \\ &= \sum_{\mathrm{side}\in\{\mathrm{left},\mathrm{right}\}} (B_{\mathrm{side}})_a^\top V_{\mathrm{sidec}_a}^\top V'_{\mathrm{sidec}_a}(B'_{\mathrm{side}})_a \\ &= \sum_{\mathrm{side}\in\{\mathrm{left},\mathrm{right}\}} (B_{\mathrm{side}})_a^\top C_{\mathrm{sidec}_a,\mathrm{sidec}_a}(B'_{\mathrm{side}})_a, \end{aligned} \tag{12}$$

because V_{rightc_a} and V'_{leftc_a} have disjoint nonzero rows, and vice versa. Iterating from leaves to root, each $C_{a,a}$ can be calculated from a's children in $O(z^3)$ time. Thus, the first term can be computed in $O(tz^3)$ time.

Turning to the second term, if b is a descendant on the left of a, $C_{a,b} = V_a^\top V_b' = \left((B_{\mathrm{left}})_a^\top V_{\mathrm{leftc}_a}^\top + (B_{\mathrm{right}})_a^\top V_{\mathrm{rightc}_a}^\top\right) V_b' = (B_{\mathrm{left}})_a^\top V_{\mathrm{leftc}_a}^\top V_b' = (B_{\mathrm{left}})_a^\top C_{\mathrm{leftc}_a,b}$. And likewise for the right. Now we transform the second term,

$$\sum_{a,b\in\mathrm{nodes}: b \text{ desc of } a} \mathrm{Tr}\left(A_a C_{a,b} A_b' C_{a,b}^\top\right) = \sum_{a\in\mathrm{nodes}} \mathrm{Tr}\left(A_a \sum_{b \text{ desc of } a} C_{a,b} A_b' C_{a,b}^\top\right) =: \sum_{a\in\mathrm{nodes}} \mathrm{Tr}\left(A_a E_a\right) \tag{13}$$

So our final task is to compute E_a. If a is a leaf, it has no descendants, so $E_a = 0$. Otherwise,

$$\begin{aligned} E_a = \sum_{b \text{ desc of } a} C_{a,b} A_b' C_{a,b}^\top &= \sum_{b \text{ left desc of } a} C_{a,b} A_b' C_{a,b}^\top + \sum_{b \text{ right desc of } a} C_{a,b} A_b' C_{a,b}^\top \\ &= \sum_{\mathrm{side}\in\{\mathrm{left},\mathrm{right}\}} \sum_{b \text{ side desc of } a} (B_{\mathrm{side}})_a^\top C_{\mathrm{sidec}_a,b} A_b' C_{\mathrm{sidec}_a,b}^\top (B_{\mathrm{side}})_a \\ &= \sum_{\mathrm{side}\in\{\mathrm{left},\mathrm{right}\}} (B_{\mathrm{side}})_a^\top \left(C_{\mathrm{sidec}_a,\mathrm{sidec}_a} A'_{\mathrm{sidec}_a} C_{\mathrm{sidec}_a,\mathrm{sidec}_a}^\top + \sum_{b \text{ desc of } \mathrm{sidec}_a} C_{\mathrm{sidec}_a,b} A_b' C_{\mathrm{sidec}_a,b}^\top\right)(B_{\mathrm{side}})_a \\ &= \sum_{\mathrm{side}\in\{\mathrm{left},\mathrm{right}\}} (B_{\mathrm{side}})_a^\top \left(C_{\mathrm{sidec}_a,\mathrm{sidec}_a} A'_{\mathrm{sidec}_a} C_{\mathrm{sidec}_a,\mathrm{sidec}_a}^\top + E_{\mathrm{sidec}_a}\right)(B_{\mathrm{side}})_a \end{aligned} \tag{14}$$

Iterating from leaves to root, each E_a can be calculated from a's children in $O(z^3)$ time, so the second (and third) terms can be computed in $O(tz^3)$ time, which completes the proof. ■

9 Tree Matrix Representation for DBTGP

We now show that the kernel matrix K_{XX} for the dot binary tree kernel can be written as a tree matrix in $O((n+m)q)$ time. For the kernel matrix $K_{XX'}$, we can simply take a submatrix of the matrix $K_{\text{concat}(X,X')\text{concat}(X,X')}$.

First, we must construct a binary tree with rows (each described by a binary string of length q) assigned to the leaves. The algorithm is very straightforward. Begin with only a root node, and all strings assigned to it. This node is "live". For i ranging from 1 to q, for each live node, if the i^{th} bits of the member strings differ, create two children, put the "0" strings on the left, and the "1" strings on the right, and make that node dead and its children live. This is written formally in Algorithm 4.

Algorithm 4 Create Binary Tree.

```
Require: $B \in \mathbb{B}^{n \times q}$: $n$ binary strings of length $q$
 1: $h \leftarrow \mathbf{0}^n$                              ▷ $h$ stores the home node of each string
 2: $a \leftarrow (0,)$                              ▷ $a$ stores a list of all nodes that are "alive"
 3: $m \leftarrow 1$                              ▷ $m$ is the number of nodes in the tree
 4: $l, r \leftarrow (\text{null},)$              ▷ $l$ and $r$ store the left and right children of each node
 5: for $j \in [q]$ do
 6:     $o, p \leftarrow \mathbf{0}^n$                  ▷ these will store a count of 0's and 1's
 7:     for $i \in [n]$ do                              ▷ parallelizable
 8:         if $B_{ij} = 0$ then $o_{h_i} \leftarrow o_{h_i} + 1$
 9:         else $p_{h_i} \leftarrow p_{h_i} + 1$
10:     for $k \in a$ do                              ▷ parallelizable
11:         if $o_k > 0$ and $p_k > 0$ then
12:             delete $k$ from $a$
13:             append $(m, m+1)$ to $a$
14:             $l_k, r_k \leftarrow m, m+1$
15:             $m \leftarrow m + 2$
16:             append $(\text{null}, \text{null})$ to $l$ and $r$
17:     for $i \in [n]$ do                              ▷ parallelizable
18:         if $h_i \notin a$ then $h_i \leftarrow l_{h_i}$ if $B_{ij} = 0$ else $r_{h_i}$
    return $h, l, r$
```

This takes $O((n+m)q)$ time when done for both train and test points, and the loop over $(n+m)$ is parallelizable. In the binary tree kernel k_w there is a weight for each i, so as we do this, we must keep track, for each node, how many bits of correspondence there are before its member strings must be split into seperate child nodes.

Then, to construct the tree matrix, the V matrix is simply the feature matrix, where the i^{th} row is the feature vector $f(x_i)$. Recall that for any leaf, V_{leaf} is the V matrix, but with all rows not belonging to the leaf set equal to 0. All B matrices are the identity matrix. And for all nodes, $A_{\text{node}} = I \sum_{j=\#\text{bits of correspondence(parent(node))}+1}^{\#\text{bits of correspondence(node)}} w_j$. Call this tree matrix $\mathcal{T}_{w,f}$.

Proposition 9 (Tree Matrix Kernel) *Letting $\mathbb{I}_i$ be vector where $(\mathbb{I}_i)_j = \delta_{ij}$, $\mathbb{I}_i^\top \mathcal{T}_{w,f}\, \mathbb{I}_j = k_w(x_i, x_j) f(x_i)^\top f(x_j)$.*

Proof For all nodes that are not a parent of the leaf containing i, $\mathbb{I}_i V_{\text{node}} = 0$, and for all nodes that are not a parent of the leaf containing j, $V_{\text{node}}^\top \mathbb{I}_j = 0$. For nodes that are a parent of both, since all B matrices are the identity, $\mathbb{I}_i V_{\text{node}} = f(x_i)^\top$ and $V_{\text{node}}^\top \mathbb{I}_j = f(x_j)$, so the contribution of each such node is $f(x_i)^\top f(x_j) \sum_{r=\#\text{bits of correspondence(parent(node))}+1}^{\#\text{bits of correspondence(node)}} w_r$. Summing over all such nodes gives $f(x_i)^\top f(x_j) \sum_{r=1}^{\#\text{bits of correspondence between } x_i \text{ and } x_j} w_r = f(x_i)^\top f(x_j) k_w(x_i, x_j)$. ■

We can now efficiently compute Equations 1 and 2. Recall,

$$\mu = K_{X'X}(K_{XX} + \lambda I_n)^{-1} y$$
$$\Sigma = K_{X'X'} - K_{X'X}(K_{XX} + \lambda I_n)^{-1} K_{XX'} + \lambda I_m$$

For μ, inverting $K_{XX} + \lambda I$ takes $O(tz^3)$ time, and then we do matrix-vector multiplication right to left in $O(tz^2)$ time. For Σ, we could represent it as a composition of tree matrices, and calculate matrix-vector multiplications Σv as $K_{X'X'}v - K_{X'X}(K_{XX} + \lambda I)^{-1} K_{XX'}v + \lambda v$, but we often want diag(Σ), independent predictive variances for each of the m points in X'. Borrowing a trick from Cohen et al. (2022), let $\tilde{X}$ be the concatenation of X and X', and then $\Sigma = (K_{\tilde{X}\tilde{X}} + \lambda I_{n+m})/K_{XX}$, where / denotes the Schur complement. Now note that the bottom right $m \times m$ block of $(K_{\tilde{X}\tilde{X}} + \lambda I_{n+m})^{-1} = ((K_{\tilde{X}\tilde{X}} + \lambda I_{n+m})/K_{XX})^{-1}$. So to represent Σ as a single tree matrix, we first construct $K_{\tilde{X}\tilde{X}}$ as a tree matrix, then we invert it (adding λI_{m+n}), then we take the submatrix corresponding to the bottom right $m \times m$ matrix, and then we invert it again. Finally, to get independent predictive variances, we use Algorithm 3 to compute the diagonal.

For the tree matrix $\mathcal{T}_{w,f}$ that we constructed, the number of leaves is the number of unique binary strings in the data, so it could be as large as $m + n$. This means that GP regression takes $O((n + m)(q + z^3))$ time and $O((n + m)(q + z^2))$ space. But if we prune the trees, GP regression only takes $O((n + m)q + (n + m + tz)z^2)$ time and $O((n + m)q + (n + m + tz)z)$ space, where t is the number of leaves. Recall that pruning can be done without changing the underlying linear operator, as long as none of the new leaves have more than z constituent rows or columns (unless they started with that many).

Cohen et al. (2022) find it helpful to do gradient-based optimization of the weight vector w. This is also possible for the dot binary tree kernel. The optimization target is to minimize the negative log likelihood of the training data. This minimand and its gradient can be calculated as follows:

$$\text{NLL}(w) = {}^1\!/_2 \left[y^\top (K_{XX}(w) + \lambda I_n)^{-1} y + \log |K_{XX}(w) + \lambda I_n| + n \log(2\pi) \right] \tag{15}$$

$$\frac{\partial \text{NLL}}{\partial w_i} = \frac{1}{2} \left[-v^\top \frac{\partial K_{XX}}{\partial w_i} v + \text{Tr}\left(\frac{\partial K_{XX}}{\partial w_i} (K_{XX}(w) + \lambda I_n)^{-1} \right) \right] \tag{16}$$

where $v = (K_{XX}(w) + \lambda I_n)^{-1}y$. This is why we devised an algorithm for calculating the Frobenius inner product of two binary tree matrices with the same tree structure. In our experiments, we use slightly different algorithms to calculate each term of the gradient to make it more efficient to compute for all elements of w. For both terms, we construct algorithms where we can "fold in" the A matrix for ${}^{\partial K_{XX}}/_{\partial w_i}$ last, because that is the only thing that changes for the different i. These can be found in the code linked in the experiments section.

10 Pruning

First, we discuss how we prune binary trees, and then we identify a condition such that the expected number of leaves on the pruned tree is $O(n/z)$. If any sibling leaves have z or fewer constituent rows between them and z or fewer constituent columns, they are pruned, and their constituent rows and columns are passed to their parent. For any new leaf, the sum of the matrix for that node and the matrices from all its former descendants has no more than z nonzero rows and columns, so it has rank no more than z, so we can easily find matrices V, A, and V', such that $VAV'^{\top}$ is equal to this sum. And if we are pruning a symmetric matrix, we can ensure $V = V'$. We omit the details of such a calculation. For $\mathcal{T}_{w,f}$, since the A and B matrices are all constant multiples of the identity, it can be done very efficiently.

Now, we analyze how many resulting leaves we can expect in the pruned tree. This depends on the distribution of the binary strings in the data. We begin with simple scenario: the data-distribution is equivalent to every bit being sampled from a Bernoulli(α) distribution, with $0 < \alpha \leq 1/2$. (If $\alpha = 0$, all the data ends up on the root node, so the number of resulting leaves is 1, without even pruning.) We assume q is infinite; lesser q will only potentially reduce the number of leaves, since we may have more than z copies of a single string in the data.

More concisely, let F_n be a random variable corresponding to the number of leaves resulting from the following stochastic process. Let the root node of a tree have the value n, then for any node with value $v > z$, create a left and right child, give the left child a value sampled from Binomial(v, α), give the right child a value of v minus that, and set the node's own value to 0. Repeat until all nodes have a value at most z. Let $f_\alpha(n) = \mathbb{E}_\alpha F_n$, which we abbreviate $f(n) = \mathbb{E} F_n$.

Theorem 10 (Basic Prunability) *$f(n) \leq \alpha^{-1}n/z$ for $n > z$.*

To prove Theorem 10, we require two lemmas. First:

Lemma 11 *For two probability distributions p_1 and p_2 over a finite set of elements $\mathcal{X} \subset \mathbb{R}$, if $p_2(x)/p_1(x) \leq p_2(y)/p_1(y)$ for all $x < y$, then $\mathbb{E}_{x \sim p_2} x \geq \mathbb{E}_{x \sim p_1} x$.*

Surely this is a special case of some known result, but we cannot find it. It is depicted in Figure 3.

Proof We prove this by induction over the number of elements of $\mathcal{X}$. If $|\,\mathcal{X}\,| = 2$, then let y be the larger element, and x the smaller. Now note that $p_2(x)/p_1(x) \leq 1$, because it is the minimal element, so if this ratio were greater than 1, then it would be for y as well, but then the sum of the p_2 probabilities would exceed one. Likewise $p_2(y)/p_1(y) \geq 1$. The rest of the $|\,\mathcal{X}\,| = 2$ case is trivial.

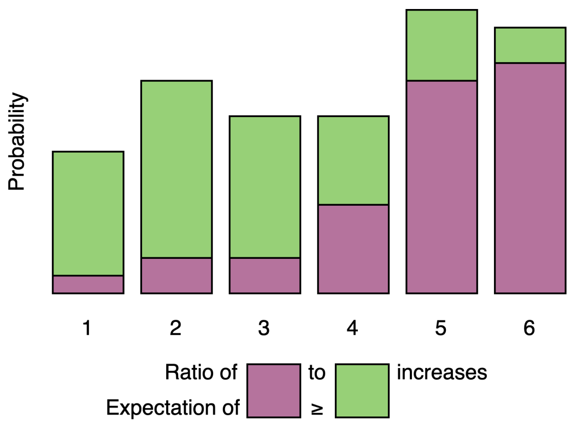

Figure 3: Diagram of Lemma 11

Now, assuming the lemma holds for $|\mathcal{X}| < s$, we show it holds for $|\mathcal{X}| = s$. Let $x^\star$ be the minimal element of $\mathcal{X}$, and let $\mathcal{X}' = \mathcal{X} \setminus \{x^\star\}$. We abbreviate $\mathbb{E}_{x\sim p_1}$ as $\mathbb{E}_1$, and likewise for p_2.

$\mathbb{E}_2 x = x^\star p_2(x^\star) + \mathbb{E}_2[x|x \in \mathcal{X}']p_2(\mathcal{X}') \geq x^\star p_2(x^\star) + \mathbb{E}_1[x|x \in \mathcal{X}']p_2(\mathcal{X}')$, by the inductive hypothesis, since the ratio of elements' probabilities is the same within a conditional distribution. As before, $p_2(x^\star)/p_1(x^\star) \leq 1$, because it is the minimal element, so if this ratio were greater than 1, then it would be for all elements, but then $\sum_x p_2(x) > 1$. That implies $p_2(\mathcal{X}')/p_1(\mathcal{X}') \geq 1$, because if not, $\sum_x p_2(x) < 1$. Now note that $x^\star < \mathbb{E}_1[x|x \in \mathcal{X}']$, because $x^\star < x$ for all $x \in \mathcal{X}'$.

So now we can consider the set $\{x^\star, \mathbb{E}_1[x|x \in \mathcal{X}']\}$, and two probability distributions p_1 and p_2 over those two elements, where $p_i(\mathbb{E}_1[x|x \in \mathcal{X}']) = p_i(\mathcal{X}')$. Since we have shown the lemma holds when the set has two elements, $x^\star p_2(x^\star) + \mathbb{E}_1[x|x \in \mathcal{X}']p_2(\mathcal{X}') \geq x^\star p_1(x^\star) + \mathbb{E}_1[x|x \in \mathcal{X}']p_1(\mathcal{X}')$, and this is just $\mathbb{E}_1 x$, which completes the proof by induction. ■

Lemma 12 (Conditional Binomial) *For $k \sim Binom(n, \alpha)$, for $z \leq n$, $\mathbb{E}[k|k \leq z] \geq \alpha z$.*

The intuition for this is that equality holds for $n = z$, and as n increases, the conditional distribution only gets more skewed to the right.

Proof We prove this by induction on n. First, let $n = z$. $\mathbb{E}[k|k \leq z] = \mathbb{E}\, k = \alpha n = \alpha z$. (The expectation of a binomial distribution can easily be derived using the moment generating function). Now, letting p_n denote the conditional distribution $p_{\text{Binom}}(k|k \leq z)$, for $n > z$, we assume an inductive hypothesis—$\mathbb{E}_{n-1} k \geq \alpha z$—and we compare $p_{n-1}(k)$ to $p_n(k)$. Let $\beta = 1 - \alpha$.

Letting $\tilde{p}_n(k) = \binom{n}{k}\alpha^k\beta^{n-k}$, we have $p_n(k) = \tilde{p}_n(k)/\sum_{j=1}^{z}\tilde{p}_n(j)$ for the conditional distribution. Observe $\tilde{p}_n(k)/\tilde{p}_{n-1}(k) = \beta\binom{n}{k}/\binom{n-1}{k} = \beta n/(n-k)$. Therefore, $p_n(k)/p_{n-1}(k) = \beta n/(n-k)$, so $\frac{p_n(k)/p_{n-1}(k)}{p_n(k-1)/p_{n-1}(k-1)} = (n-k+1)/(n-k) > 1$.

This means we can apply Lemma 11, so we have $\mathbb{E}_n k \geq \mathbb{E}_{n-1} k \geq \alpha z$. ■

Proof [Proof of Theorem 10] First, we identify a recurrence relation in $f(n)$. Observe that $F_n = 1$ for $n \leq z$, so $f(n) = 1$ there as well. And for $n > z$, $f(n) = \mathbb{E}_{k\sim\text{Binom}(n,\alpha)} f(k) + f(n-k)$. We construct a function g and show inductively that $f \leq g$.

Let $g(n) = \alpha^{-1}n/z$ for $n > z$ and $= 1$ for $n \leq z$. For the base case of $n \leq z$, $f(n) = g(n)$ clearly. Now assuming that for $k < n$, $f(k) \leq g(k)$, we show that $f(n) \leq g(n)$.

$f(n) = \mathbb{E}_{k\sim\text{Binom}(n,\alpha)} f(k) + f(n-k) \leq \mathbb{E}_{k\sim\text{Binom}(n,\alpha)} g(k) + g(n-k)$, by the inductive hypothesis. Splitting into two conditional expectations, $\mathbb{E}\, g(k) = \mathbb{E}[g(k) \mid k > z]p(k > z) + \mathbb{E}[g(k) \mid k \leq z]p(k \leq z)$. Given the definition of g, this evaluates to $\alpha^{-1}z^{-1}\,\mathbb{E}[k \mid k > z]p(k > z) + \alpha^{-1}z^{-1}(\alpha z)p(k \leq z)$. Applying Lemma 12, that quantity is $\leq \alpha^{-1}z^{-1}\,\mathbb{E}[k \mid k > z]p(k > z) + \alpha^{-1}z^{-1}\,\mathbb{E}[k \mid k \leq z]p(k \leq z) = \alpha^{-1}z^{-1}\,\mathbb{E}[k]$.

Likewise,

$$\begin{aligned}
&\mathbb{E}_{k\sim\text{Binom}(n,\alpha)}\, g(n-k)\\
&= \mathbb{E}_{k'\sim\text{Binom}(n,1-\alpha)}\, g(k')\\
&= \mathbb{E}_{1-\alpha}[g(k')|k' > z]p_{1-\alpha}(k' > z) + \mathbb{E}_{1-\alpha}[g(k')|k' \leq z]p_{1-\alpha}(k' \leq z)\\
&= \alpha^{-1}z^{-1}\,\mathbb{E}_{1-\alpha}[k' \mid k' > z]p_{1-\alpha}(k' > z) + (1-\alpha)^{-1}z^{-1}((1-\alpha)z)p_{1-\alpha}(k' \leq z)\\
&\leq \alpha^{-1}z^{-1}\,\mathbb{E}_{1-\alpha}[k' \mid k' > z]p_{1-\alpha}(k' > z) + (1-\alpha)^{-1}z^{-1}\,\mathbb{E}_{1-\alpha}[k' \mid k' \leq z]p_{1-\alpha}(k' \leq z)\\
&\leq \alpha^{-1}z^{-1}\,\mathbb{E}_{1-\alpha}[k' \mid k' > z]p_{1-\alpha}(k' > z) + \alpha^{-1}z^{-1}\,\mathbb{E}_{1-\alpha}[k' \mid k' \leq z]p_{1-\alpha}(k' \leq z)\\
&= \alpha^{-1}z^{-1}\,\mathbb{E}_{1-\alpha}[k'] = \alpha^{-1}z^{-1}\,\mathbb{E}[n-k] \qquad (17)
\end{aligned}$$

Therefore, $\mathbb{E}_{k\sim\text{Binom}(n,\alpha)}\, g(k) + g(n-k) \leq \alpha^{-1}z^{-1}(\mathbb{E}[k] + \mathbb{E}[n-k]) = \alpha^{-1}z^{-1}n$, which completes the proof that $f(n) \leq \alpha^{-1}z^{-1}n$ for $n > z$. ∎

Now, we extend Theorem 10 by relaxing the assumptions. First, by symmetry, for $\alpha \geq 1/2$, $f_\alpha(n) \leq (1-\alpha)^{-1}n/z$. And now instead of having a constant α for every Bernoulli distribution, let every bit be sampled from Bernoulli($\beta(s)$), where s is the string of already-sampled bits, and $\alpha \leq \beta(s) \leq 1-\alpha$. We would like to bound $\mathbb{E}_\beta\, F_n$.

Theorem 13 (Prunability) *$\mathbb{E}_\beta\, F_n \leq \alpha^{-1}n/z$ for $n > z$.*

Proof Because β depends on the address of the node in question, we cannot immediately produce a recurrence relation. Instead, we quantify over all possible functions β that have the property described above. So let $f'_\alpha(n) = \sup_{\beta':\mathbb{B}^*\to[\alpha,1-\alpha]} \mathbb{E}_{\beta'}\, F_n$, where $\mathcal{X}^*$ is the Kleene-* operator. Now, the maximum number of descendant leaves of a node does not depend on its address.

First, of course, $\mathbb{E}_\beta\, F_n \leq f'_\alpha(n)$. And now we show $f'_\alpha(n) \leq g(n)$, as defined in the proof of Theorem 10, which we do by induction. For the base case of $n \leq z$, $f'_\alpha(n) = g(n) = 1$. Now we assume that the inductive hypothesis holds for $k < n$, and show that it holds for n as well.

We can split the supremum over functions into a supremum over the function's value at a particular string and the supremum over all functions consistent with that value. So $f'_\alpha(n) = \sup_{\gamma\in[\alpha,1-\alpha]} \sup_{\beta':\mathbb{B}^*\to[\alpha,1-\alpha]\text{s.t.}\beta'(\emptyset)=\gamma} \mathbb{E}_{\beta'}\, F_n = \sup_{\gamma\in[\alpha,1-\alpha]} \mathbb{E}_{k\sim\text{Binom}(n,\gamma)}\, f'_\alpha(k) + f'_\alpha(n-k)$. By symmetry, we can restrict $\gamma \in [\alpha, 1/2]$ without reducing the supremum. By the inductive hypothesis, $f'_\alpha(n) \leq \sup_{\gamma\in[\alpha,1/2]} \mathbb{E}_{k\sim\text{Binom}(n,\gamma)}\, g(k) + g(n-k)$. The proof proceeds exactly as before, except all p's and $\mathbb{E}$'s are with respect to $k \sim \text{Binom}(n, \gamma)$ instead of $\text{Binom}(n, \alpha)$, and when we apply Lemma 12, instead of immediately saying $\alpha z \leq \mathbb{E}[k|k \leq z]$, we say $\alpha z \leq \gamma z \leq \mathbb{E}[k|k \leq z]$. ■

Therefore, under the conditions of the Prunability Theorem, the expected number of leaves $t \leq \alpha^{-1}(n + m)/z$, so the expected space requirements of the algorithm are $O((n+m)(q+\alpha^{-1}z))$ and the expected run time is $O((n+m)(q+\alpha^{-1}z^2))$. In terms of n, m, and z, this is the same asymptotic complexity that a general z-dimensional kernel achieves.

11 Experiments

We duplicate the experimental setup of Wang et al. (2019) and Cohen et al. (2022)—we evaluate our method on the same 12 regression datasets (Dua and Graff, 2017), using the same three train/test splits, and we look at the root mean square error of predictions for the test data alongside the mean negative log likelihood assigned to individual points in the test data. We adopt Cohen et al.'s (2022) method for optimizing the weight vector w, with minor modifications.[2] Of course, the gradient of the negative log likelihood with respect to the weight vector has a completely different form than it does in Cohen et al. (2022). The benchmarks used by these papers have data $\in \mathbb{R}^d$, so we also adopt Cohen et al.'s (2022) method for heuristically optimizing the mapping from $\mathbb{R}^d$ to $\mathbb{B}^q$. The only major deviation from Cohen et al. (2022) is that we do not run twenty training runs from different initializations and pick the one with the lowest training loss; we just run it once.

The Dot Binary Tree Kernel can be constructed with any finite-dimensional kernel as a base, so for our experiments we pick a model finite-dimensional kernel that is commonly used: an inducing points kernel, where the inducing points are optimized minimize the negative log likelihood of the training data plus a variational penalty (SGPR). And following the benchmarks in Wang et al. (2019) and Cohen et al. (2022), the SGPR kernel's base kernel is a Matérn 3/2 kernel.

It takes more memory to build our DBTGP from an SGPR kernel with some number of inducing points that just use an SGPR kernel with that number of inducing points. Under the conditions of the Prunability Theorem, the overhead of DBTGP compared to SGPR is only a multiplicative constant, but even so, doubling the number of inducing points also has only constant overhead. So to make a fair comparison, we evaluate DBTGP and SGPR with different numbers of inducing points, and compare performance as a function of the memory usage of each method. The code is available at `https://tinyurl.com/dbtgp-code` and `https://tinyurl.com/sgpr-code`.

For NLL and RMSE evaluation, see Figures 4 and 5, respectively. In terms of NLL, on two datasets—kin40k and keggundirected—SGPR does robustly better than DBTGP. On two datasets—elevators and song—performance is comparable. And on the other eight datasets, DBTGP is robustly better. For houseelectric and slice, DBTGP's improvement is more than a nat per prediction; for 3droad, it is more than two nats; for bike, it is one to three nats.

2. Like Cohen et al. (2022), we use Adam optimization, but we also add annealing (calculating the gradient at a nearby location, with noise decaying harmonically). We regularly check the loss, and if it has gone up, we revert and decrease the learning rate and annealing noise. And whereas Cohen et al. (2022) require $||w||_1 = 1$, we only add quadratic penalty for $||w||_1$ exceeding 1.

This improvement is not incremental. In terms of RMSE, on one dataset—kin40k—SGPR does robustly better than DBTGP. On two datasets—the same ones as before—performance is comparable. And on the other nine datasets, DBTGP is robustly better. Following Wang et al. (2019) and Cohen et al. (2022), the targets are transformed so that the training targets have zero mean and unit variance. So to get a sense of scale, an RMSE of 1 could be obtained by always guessing 0. With the exception of keggundirected for NLL, DBTGP matches or exceeds the performance of BTGP, sometimes substantially.

12 Conclusion

We have developed a new matrix representation suited for fast GP regression, which allows us to extend any finite-dimensional GP to an infinite-dimensional one with little to no increase in asymptotic complexity. In particular, our method still achieves linear complexity in the number of data points. We found empirically that this tended to improve predictive performance at a fixed memory budget.

Acknowledgements

Thank you to the Oxford Man Institute for funding this work and to Alexander Terenin for his helpful feedback.

References

Michael K. Cohen, Samuel Daulton, and Michael A Osborne. Log-linear-time Gaussian processes using binary tree kernels. In *Advances in Neural Information Processing Systems*, volume 35, pages 8118–8129, 2022. URL `https://proceedings.neurips.cc/paper_files/paper/2022/file/359ddb9caccb4c54cc915dceeacf4892-Paper-Conference.pdf`.

John P Cunningham, Krishna V Shenoy, and Maneesh Sahani. Fast Gaussian process methods for point process intensity estimation. In *Proceedings of the 25th international conference on Machine learning*, pages 192–199, 2008.

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

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.

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.

James Hensman, Nicolas Durrande, and Arno Solin. Variational Fourier features for Gaussian processes. *J. Mach. Learn. Res.*, 18(1):5537–5588, 2017.

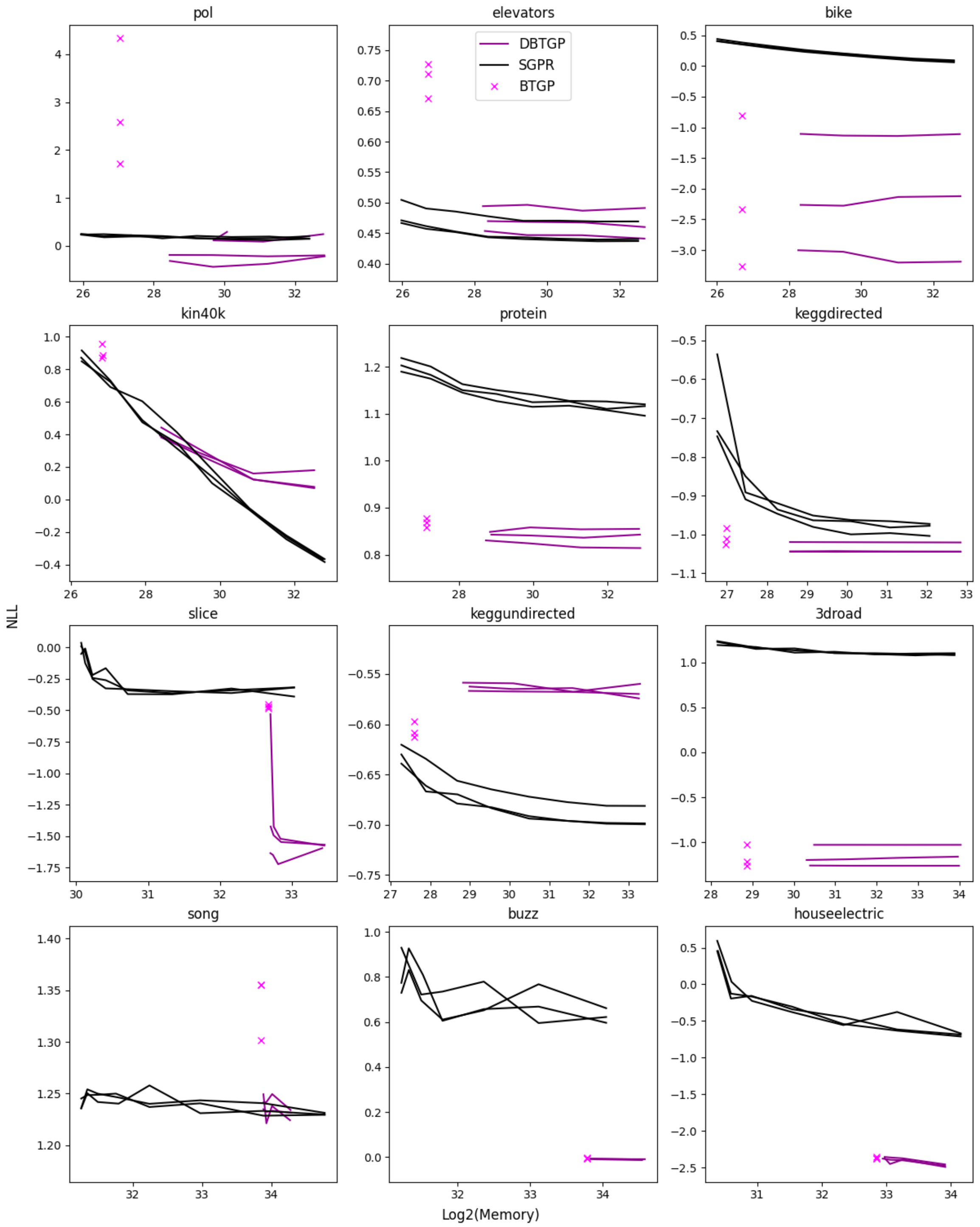

Figure 4: Test negative log likelihood as a function of memory usage for DBTGP (purple), SGPR (black), and BTGP (magenta). The three lines for each method correspond to three different train/test splits.

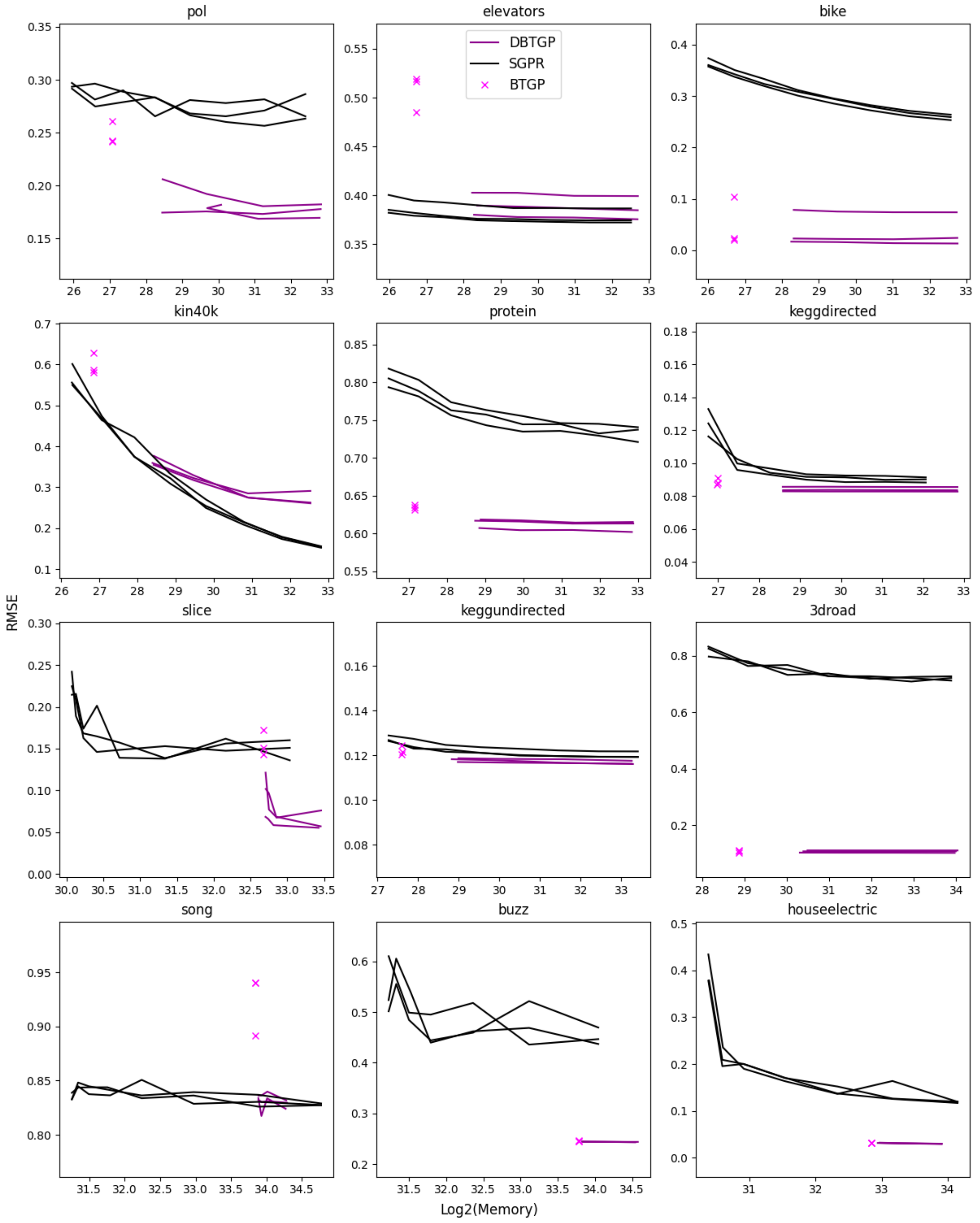

Figure 5: Test root mean square error as a function of memory usage for DBTGP (purple) and SGPR (black), and BTGP (magenta). The three lines for each method correspond to three different train/test splits.

Haitao Liu, Yew-Soon Ong, Xiaobo Shen, and Jianfei Cai. When Gaussian process meets big data: A review of scalable GPs. *IEEE transactions on neural networks and learning systems*, 31(11):4405–4423, 2020.

Ali Rahimi and Benjamin Recht. Random features for large-scale kernel machines. *Advances in neural information processing systems*, 20, 2007.

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

Yunus Saatçi. *Scalable inference for structured Gaussian process models*. PhD thesis, Citeseer, 2012.

Simo Särkkä. State space representation of Gaussian processes. *Aalto University, Espoo, Finland. Pg*, pages 22–23, 2013.

Simo Särkkä and Jouni Hartikainen. Infinite-dimensional Kalman filtering approach to spatio-temporal Gaussian process regression. In *Artificial Intelligence and Statistics*, pages 993–1001. PMLR, 2012.

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

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.

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`.

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.