High-Dimensional Gaussian Process Regression with Soft Kernel Interpolation

Anonymous authors
Paper under double-blind review

Abstract

We introduce *soft kernel interpolation* (SoftKI), a method that combines aspects of Structured Kernel Interpolation (SKI) and variational inducing point methods, to achieve scalable Gaussian Process (GP) regression on high-dimensional datasets. SoftKI approximates a kernel via softmax interpolation from a smaller number of interpolation points learned by optimizing a combination of the SoftKI marginal log-likelihood (MLL), and when needed, an approximate MLL for improved numerical stability. Consequently, it can overcome the dimensionality scaling challenges that SKI faces when interpolating from a dense and static lattice while retaining the flexibility of variational methods to adapt inducing points to the dataset. We demonstrate the effectiveness of SoftKI across various examples and show that it is competitive with other approximated GP methods when the data dimensionality is modest (around 10).

1 Introduction

Gaussian processes (GPs) are flexible function approximators based on Bayesian inference. However, there are scaling concerns. For a dataset comprising n data points, constructing the $n \times n$ kernel (covariance) matrix required for GP inference incurs a space complexity of $\mathcal{O}(n^2)$. Solving the associated linear system for exact posterior inference via direct methods requires $\mathcal{O}(n^3)$ time complexity. These computational costs render exact GP inference challenging, even for modest n, although efforts have been made to accelerate exact GP inference (Wang et al., 2019; Gardner et al., 2021).

One approach to scalable GP regression is based in variational inference (Titsias, 2009; Hensman et al., 2013). These methods build a variational approximation of the posterior GP posterior by learning the locations of $m \ll n$ *inducing points* (Quinonero-Candela & Rasmussen, 2005; Snelson & Ghahramani, 2005). Inducing points and their corresponding *inducing variables* introduce latent variables with normal priors that form a low-rank approximation of the covariance structure in the original dataset. This approach improves the time complexity of posterior inference to $\mathcal{O}(m^2 n)$ for Sparse Gaussian Process Regression (SGPR) (Titsias, 2009) and $\mathcal{O}(m^3)$ for Stochastic Variational Gaussian Process Regression (SVGP) (Hensman et al., 2013), with the latter introducing additional variational parameters.

Another approach is based on *Structured Kernel Interpolation* (SKI) (Wilson & Nickisch, 2015) and its variants such as product kernel interpolation (SKIP) (Gardner et al., 2018) and Simplex-SKI (Kapoor et al., 2021). SKI-based methods achieve scalability by constructing computationally tractable approximate kernels via interpolation from a pre-computed and dense rectilinear grid of *interpolation points*. This structure enables fast matrix-vector multiplications (MVMs), which can be leveraged to scale inference via conjugate gradient (CG) methods. However, this approach also causes the time complexity of posterior inference to become explicitly dependent on the dimensionality d of the data (*e.g.*, $\mathcal{O}(n4^d + m \log m)$ for a MVM in SKI and $\mathcal{O}(d^2(n+m))$ for a MVM in Simplex-SKI). Moreover, the static grid does not have the flexibility of a variational method to adapt to the dataset at hand. This motivates us to search for accurate and scalable GP regression algorithms that can attain the best of both approaches.

In this paper, we introduce *soft kernel interpolation* (SoftKI), which combines aspects of inducing points and SKI to enable scalable GP regression on high-dimensional datasets. Our main observation is that while SKI

can support large numbers of interpolation points, it uses them in a sparse manner—only a few interpolation points contribute to the interpolated value of any single data point. Consequently, we should still be able to interpolate successfully provided that we place enough of them in good locations relative to the dataset, *i.e.*, learn their locations as in a variational approach. This change removes the explicit dependence of the cost of the method on the data dimensionality d. It also opens the door to more directly optimize with an approximate GP marginal log-likelihood (MLL) to learn the locations of interpolations points as opposed to a variational approximation that introduces latent variables with normal priors over the values observed at inducing point locations.

Our method approximates a kernel by interpolation from a softmax of $m \ll n$ learned interpolation points, hence the name *soft kernel interpolation* (Section 4.1). The interpolation points are learned by optimizing a combination of the SoftKI MLL, and when needed, an approximate MLL for improved numerical stability in single-precision floating point arithmetic (Section 4.2). It is able to leverage GPU acceleration and stochastic optimization for scalability. Since the kernel structure is dynamic, we rely on $m \ll n$ interpolation points to obtain a time complexity of $\mathcal{O}(m^2 n)$ and space complexity of $\mathcal{O}(mn)$ for posterior inference (Section 4.3). We evaluate SoftKI on a variety of datasets from the `UCI` repository (Kelly et al., 2017) and demonstrate that it achieves test root mean square error (RMSE) comparable to other inducing point methods for datasets with moderate dimensionality (approximately $d = 10$) (Section 5.1). To further explore the effectiveness of SoftKI in high dimensions, we apply SoftKI to molecule datasets (d in the hundreds to thousands) and show that SoftKI is scalable and competitive in these settings as well (Section 5.2). Lastly, we compare the interpolation points learned by SoftKI to the inducing points learned by SGPR and SVGP, and find that they are structurally different, suggesting that learned interpolation points provide another promising path for constructing approximate GPs (Section 5.3).

2 Background

Notation. Matrices will be denoted by upper-case boldface letters (e.g. $\mathbf{A} \in \mathbb{R}^{n \times m}$). When specifying a matrix entry-wise we write $\mathbf{A} = [g(i,j)]_{i,j}$, meaning $\mathbf{A}_{ij} = g(i,j)$. At times, given points $x_1, \ldots, x_n \in \mathbb{R}^d$, we stack them into $\mathbf{x} \in \mathbb{R}^{nd}$ by $\mathbf{x}^\top = (x_1^\top, \ldots, x_n^\top)$, so that entries $d(i-1)+1$ through di are the components of $\mathbf{x}_i$. Any function $f : \mathbb{R}^d \to \mathbb{R}$ extends column-wise via $f(\mathbf{x}) = (f(\mathbf{x}_1), \ldots, f(\mathbf{x}_n))^\top$.

2.1 Gaussian Processes

Let $k : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ be a positive semi-definite kernel function. A (centered) *Gaussian process* (GP) with kernel k is a distribution over functions $f : \mathbb{R}^d \to \mathbb{R}$ such that for any finite collection of inputs $\{x_i\}_{i=1}^n$, the vector of function values $f(\mathbf{x})$ is jointly Gaussian $f(\mathbf{x}) \sim \mathcal{N}(\mathbf{0}, \mathbf{K_{xx}})$, where the kernel matrix $\mathbf{K_{xx}} \in \mathbb{R}^{d \times d}$ has entries $[\mathbf{K_{xx}}]_{ij} = k(x_i, x_j)$. Given two collections $\{x_i\}_{i=1}^n$ and $\{x'_j\}_{j=1}^{n'}$, the cross-covariance matrix $\mathbf{K_{xx'}} \in \mathbb{R}^{d \times nd'}$ is defined by $[\mathbf{K_{xx'}}]_{ij} = k(x_i, x'_j)$.

To perform GP regression on the labeled dataset $\mathcal{D} := \{(x_i, y_i) : x_i \in \mathbb{R}^d, y_i \in \mathbb{R}\}_{i=1}^n$, we assume the data is generated as follows:

$$f(\mathbf{x}) \sim \mathcal{N}(\mathbf{0}, \mathbf{K_{xx}}) \tag{GP}$$

$$\mathbf{y} \,|\, f(\mathbf{x}) \sim \mathcal{N}(f(\mathbf{x}), \beta^2 \mathbf{I}) \tag{likelihood}$$

where f is a function sampled from a GP and each observation y_i is f evaluated at x_i perturbed by independent and identically distributed (i.i.d.) Gaussian noise with zero mean and variance β^2. The posterior predictive distribution has the following closed-form solution (Rasmussen & Williams, 2005)

$$p(f(*) \,|\, \mathbf{y}) = \mathcal{N}(\mathbf{K}_{*\mathbf{x}}(\mathbf{K_{xx}} + \mathbf{\Lambda})^{-1}\mathbf{y}, \mathbf{K}_{**} - \mathbf{K}_{*\mathbf{x}}(\mathbf{K_{xx}} + \mathbf{\Lambda})^{-1}\mathbf{K}_{\mathbf{x}*}) \tag{1}$$

where $\mathbf{\Lambda} = \beta^2 \mathbf{I}$. Using direct methods, the time complexity of inference is $\mathcal{O}(n^3)$ which is the complexity of solving the system of linear equations in n variables $(\mathbf{K_{xx}} + \mathbf{\Lambda})\alpha = \mathbf{y}$ for α so that the posterior mean (Equation 1) is $\mathbf{K}_{*\mathbf{x}}\alpha$.

A GP's *hyperparameters* θ include the noise β^2 and other parameters such as those involved in the definition of a kernel such as its *lengthscale* ℓ and *output scale* σ. Thus $\theta = (\beta, \ell, \sigma)$. The hyperparameters can be learned by maximizing the MLL of a GP

$$\log p(\mathbf{y} \mid \mathbf{x}; \theta) = \log \mathcal{N}(\mathbf{y} \mid \mathbf{0}, \mathbf{K_{xx}}(\theta) + \mathbf{\Lambda}(\theta)) \tag{2}$$

where $\mathcal{N}(\cdot \mid \mu, \mathbf{\Sigma})$ is notation for the probability density function (PDF) of a Gaussian distribution with mean μ and covariance $\mathbf{\Sigma}$ and we have explicitly indicated the dependence of $\mathbf{K_{xx}}$ and $\mathbf{\Lambda}$ on θ.

2.2 Sparse Gaussian Process Regression

Many scalable GP methods address the high computational cost of GP inference by approximating the kernel using a Nyström method (Williams & Seeger, 2000). This approach involves selecting a smaller set of m inducing points, $\mathbf{z} \subset \mathbf{x}$, to serve as representatives for the complete dataset. The original $n \times n$ kernel $\mathbf{K_{xx}}$ is then approximated as

$$\mathbf{K_{xx}} \approx \mathbf{K_{xz}} \mathbf{K_{zz}^{-1}} \mathbf{K_{zx}} \tag{3}$$

leading to an overall inference complexity of $\mathcal{O}(m^2 n)$.

Variational inducing point methods such as SGPR (Titsias, 2009) combines the Nyström GP kernel approximation with a variational optimization process so that the positions of the inducing points can be learned. The variational approximation introduces latent *inducing variables* u to model the values observed at inducing points $\mathbf{z}$ with joint distribution

$$\begin{pmatrix} u \\ f(\mathbf{x}) \end{pmatrix} \sim \mathcal{N}\left(\begin{pmatrix} \mathbf{0} \\ \mathbf{0} \end{pmatrix}, \begin{pmatrix} \mathbf{K_{zz}} & \mathbf{K_{zx}} \\ \mathbf{K_{xz}} & \mathbf{K_{xx}} \end{pmatrix} \right). \tag{4}$$

This leads to a posterior predictive distribution $q(f(*))$ which has closed form

$$q(f(*) \mid \mathbf{y}) = \mathcal{N}(f(*) \mid \mathbf{K_{*z}} \mathbf{C^{-1}} \mathbf{K_{zx}} \mathbf{\Lambda^{-1}} \mathbf{y}, \mathbf{K_{**}} - \mathbf{K_{*z}} (\mathbf{K_{zz}^{-1}} - \mathbf{C^{-1}}) \mathbf{K_{z*}}). \tag{5}$$

The time complexity of posterior inference is $\mathcal{O}(m^2 n + m^3)$ since it requires solving $\mathbf{C}\alpha = \mathbf{K_{zx}} \mathbf{\Lambda^{-1}} \mathbf{y}$ for α which is dominated by forming $\mathbf{C}$. The inducing points are learned by maximizing the evidence lower bound (ELBO)

$$\text{ELBO}(q) = \log \mathcal{N}(\mathbf{y} \mid \mathbf{0}, \mathbf{K_{xx}^{SGPR}} + \mathbf{\Lambda}) - \frac{1}{2} \operatorname{tr}(\mathbf{K_{xx}} - \mathbf{K_{xx}^{SGPR}}) \tag{6}$$

where $\mathbf{K_{xx}^{SGPR}} = \mathbf{K_{xz}} \mathbf{K_{zz}^{-1}} \mathbf{K_{zx}}$ is a Nyström approximation of $\mathbf{K_{xx}}$. Since the ELBO is a lower bound on the MLL $\log p(\mathbf{y} \mid \mathbf{x}; \theta)$, we can maximize the ELBO via gradient-based optimization as a proxy for maximizing $\log p(\mathbf{y} \mid \mathbf{x}; \theta)$ to learn the location of the inducing points $\mathbf{z}$ by treating it as a GP hyperparameter, *i.e.*, $\theta = (\beta, \ell, \sigma, \mathbf{z})$. The time complexity of computing the ELBO is $\mathcal{O}(m^2 n)$.

2.3 Structured Kernel Interpolation

SKI approaches scalable GP regression by approximating a large covariance matrix $\mathbf{K_{xx}}$ by interpolating from cleverly chosen interpolation points. In particular, SKI makes the approximation

$$\mathbf{K_{xx'}^{SKI}} = \mathbf{W_{xz}} \mathbf{K_{zz}} \mathbf{W_{zx'}} \approx \mathbf{K_{xx'}} \tag{7}$$

where $\mathbf{W_{xz}}$ is a sparse matrix of interpolation weights, $\mathbf{W_{zx}} = \mathbf{W_{xz}^\top}$, and $\mathbf{z}$ are interpolation points. The interpolation points can be interpreted as quasi-inducing points since

$$\mathbf{K_{xx'}^{SKI}} = (\mathbf{W_{xz}} \mathbf{K_{zz}}) \mathbf{K_{zz}^{-1}} (\mathbf{K_{zz}} \mathbf{W_{zx'}}) = \widehat{\mathbf{K}}_{\mathbf{xz}} \mathbf{K_{zz}^{-1}} \widehat{\mathbf{K}}_{\mathbf{zx'}} \tag{8}$$

where $\widehat{\mathbf{K}}_{\mathbf{xz}} = \mathbf{W_{xz}} \mathbf{K_{zz}}$. However, interpolation points are not associated with inducing variables since they are used for kernel interpolation rather than defining a variational approximation. The posterior predictive distribution is then the standard GP posterior predictive

$$p(f(*) \mid \mathbf{y}) \approx \mathcal{N}(\mathbf{K_{*x}} (\mathbf{K_{xx}^{SKI}} + \mathbf{\Lambda})^{-1} \mathbf{y}, \mathbf{K_{**}} - \mathbf{K_{*x}} (\mathbf{K_{xx}^{SKI}} + \mathbf{\Lambda})^{-1} \mathbf{K_{x*}}) \tag{9}$$

where $\mathbf{K_{xx}}$ is replaced with $\mathbf{K_{xx}^{SKI}}$. SKI uses linear CG to solve the linear system $(\mathbf{K_{xx}^{SKI}} + \mathbf{\Lambda})\alpha = \mathbf{y}$ for α. Since the time complexity of CG depends on access to fast MVMs, SKI makes two design choices that are compatible with CG. First, SKI uses cubic interpolation (Keys, 1981) weights so that $\mathbf{W_{xz}}$ has four non-zero entries in each row, i.e., $\mathbf{W_{xz}}$ is sparse. Second, SKI chooses the interpolation points $\mathbf{z}$ to be on a fixed rectilinear grid in $\mathbb{R}^d$ which induces multilevel Toeplitz structure in $\mathbf{K_{zz}}$ if $k(x, y)$ is a stationary kernel.

Putting the two together, the resulting $\mathbf{K_{xx}^{SKI}}$ is a structured kernel which enables fast MVMs. If we wish to have k distinct interpolation points for each dimension, the grid will contain $m = k^d$ interpolation points leading to a total complexity of GP inference of $\mathcal{O}(n4^d + m\log(m)) = \mathcal{O}(n4^d + dk^d\log(k))$ since there are $\mathcal{O}(4^d)$ nonzero entries per row of $\mathbf{W_{xz}}$ and a MVM with a Toeplitz structured kernel like $\mathbf{K_{zz}}$ can be done in $\mathcal{O}(m\log m)$ time via a Fast Fourier Transform (FFT) (Wilson, 2014). Thus, for low dimensional problems, SKI can provide significant speedups compared to vanilla GP inference and supports a larger number of interpolation points compared to SGPR.

3 Related Work

Beyond SGPR and vanilla SKI, a wide range of approximate GP models and variants have been developed that align closely with SoftKI.

Additional variational methods. Building on SGPR, SVGP (Hensman et al., 2013) extends the optimization process used for SGPR to use stochastic variational inference. This further reduces the computational cost of GP inference to $\mathcal{O}(m^3)$ since optimization is now over a variational distribution on $\mathbf{z}$, rather than the full posterior. Importantly SVGP is highly scalable to GPUs since its optimization strategy is amenable to minibatch optimization procedures. More recent research has introduced Variational Nearest Neighbor Gaussian Processes (VNNGP) (Wu et al., 2024), which replace the low-rank prior of SVGP with a sparse approximation of the precision matrix by retaining only correlations among each point's K nearest neighbors. By constructing a sparse Cholesky factor with at most $K + 1$ nonzeros per row, VNNGP can reduce the per-iteration cost of evaluating the SVGP objective Equation 6 to $\mathcal{O}\big((n_b + m_b)K^3\big)$ when minibatching over n_b data points and m_b inducing points.

Other variants of Structured Kernel Interpolation. While SKI can provide significant speedups over vanilla GP inference, its scaling in d restricts its usage to low-dimensional settings where $4^d < n$. To address this, Gardner et al. (2018) introduced SKIP, which optimizes SKI by expressing a d-dimensional kernel as a product of d one-dimensional kernels. This reduces MVM costs from $\mathcal{O}(dk^d\log k)$ to $\mathcal{O}(dm\log m)$, using m grid points per component kernel instead of k^d. However, SKIP is limited to dimensions roughly in the range of $d = 10$–30 for large datasets, as it requires substantial memory and may suffer from low-rank approximation errors.

Recent work on improving the scaling properties of SKI has been focused on efficient incorporation of simplicial interpolation, which when implemented carefully in the case of Simplex-GP (Kapoor et al., 2021) and Sparse grid GP (Yadav et al., 2023) can reduce the cost of a single MVM with $\mathbf{W_{xz}}$ down to $\mathcal{O}\left(nd^2\right)$. These methods differ in the grid construction used for interpolation with Simplex-SKI opting for a permutohedral lattice (Adams et al., 2010), while sparse grid-GP uses sparse grids and a bespoke recursive MVM algorithm, leading to $\mathbf{K_{zz}}$ MVM costs of $\mathcal{O}(d^2(n + m))$ and $\mathcal{O}\left(m(\log m)^d\right)$ respectively. While both of these methods greatly improve the scaling of SKI, they still feature a problematic dependency on d limiting their extension to arbitrarily high dimensional datasets.

4 Method

In this section, we introduce SoftKI (Algorithm 1). SoftKI takes the same starting point as SKI, namely that $\mathbf{K_{xx}^{SKI}} = \mathbf{W_{xz}}\mathbf{K_{zz}}\mathbf{W_{zx}}$ can be used as an approximation of $\mathbf{K_{xx}}$. However, we will deviate in that we will abandon the structure given by a lattice and opt to learn the locations of the interpolation points $\mathbf{z}$ instead. This raises several issues. First, we revisit the choice of interpolation scheme since we no longer assume a static structure (Section 4.1). Second, we require an efficient procedure for learning the interpolation points

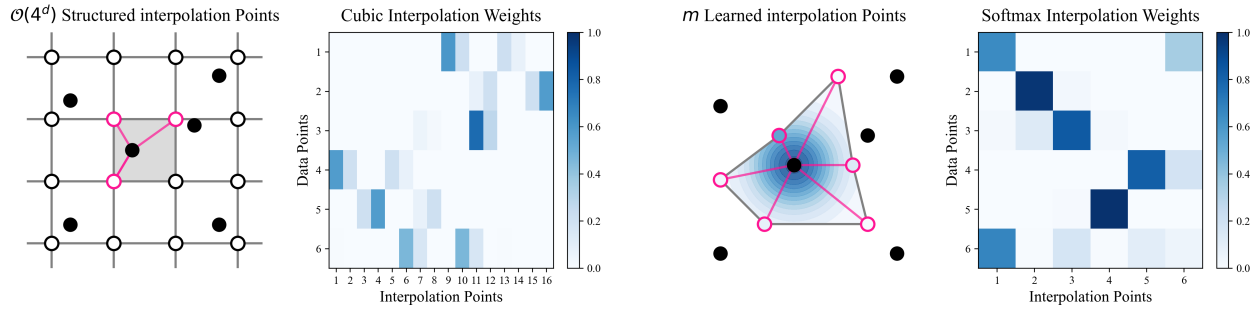


Figure 1: **(Structured & Soft Kernel Interpolation):** Comparison of interpolation procedure for local cubic interpolation during SKI and softmax interpolation during SoftKI. Here white points ∘ indicate interpolation points $z_i \in \mathbf{z}$ and black points • are data points $x_i \in \mathbf{x}$.

(Section 4.2). Third, we require an alternative route to recover a scalable GP since our approach removes structure in the kernel that was used in SKI for efficient inference (Section 4.3).

4.1 Soft Kernel Interpolation

Whereas a cubic (or higher-order) interpolation scheme is natural in SKI when using a static lattice structure, our situation is different since the locations of the interpolation points are now dynamic. In particular, we would ideally want our interpolation scheme to have the flexibility to adjust the contributions of interpolation points used to interpolate a kernel value in a data-dependent manner. To accomplish this, we propose *softmax interpolation weights*.

Definition 4.1 (Softmax interpolation weights)**.** Define *softmax interpolation weights* as the following matrix

$$\mathbf{\Sigma_{xz}} = \left[\frac{\exp\left(-\left\|x_i - z_j\right\|\right)}{\sum_{k=1}^{m} \exp\left(-\left\|x_i - z_k\right\|\right)} \right]_{ij} \tag{10}$$

to create the SoftKI kernel

$$\mathbf{K}_{\mathbf{xx}}^{\text{SoftKI}} = \mathbf{\Sigma_{xz}} \mathbf{K_{zz}} \mathbf{\Sigma_{zx}} \,. \tag{11}$$

Since a softmax can be interpreted as a probability distribution, we can view SoftKI as interpolating from a probabilistic mixture of interpolation points. Each input point x_i is interpolated from all points, with an exponential weighting favoring closer interpolation points (See Figure 1). Each row $\mathbf{\Sigma}_{x_i \mathbf{z}}$ represents the softmax interpolation weights between a single point x_i and all m interpolation points. Note that $\mathbf{\Sigma_{xz}}$ is not strictly sparse since softmax interpolation continuously assigns weights over each interpolation point for each x_i. However, as d increases, the weights for distant points become negligible. This leads to an effective sparsity, with each x_i predominantly influenced only by its nearest interpolation points during the reconstruction of $\mathbf{K_{xx}}$.

To add more flexibility to the model, the softmax interpolation scheme can be extended to contain a learnable temperature parameter T as below

$$\mathbf{\Sigma_{xz}} = \left[\frac{\exp\left(-\left\|x_i/T - z_j\right\|\right)}{\sum_{k=1}^{m} \exp\left(-\left\|x_i/T - z_k\right\|\right)} \right]_{ij} . \tag{12}$$

This additional hyperparameter is needed because unlike the SGPR kernel $\mathbf{K}_{\mathbf{xx}}^{\text{SGPR}} = \mathbf{K_{xz}} \mathbf{K_{zz}^{-1}} \mathbf{K_{zx}}$ where the lengthscale influences each term, the lengthscale only affects $\mathbf{K_{zz}}$ in the SoftKI kernel $\mathbf{K}_{\mathbf{xx}}^{\text{SoftKI}} = \mathbf{\Sigma_{xz}} \mathbf{K_{zz}} \mathbf{\Sigma_{zx}}$. The temperature acts as lengthscale-like hyperparameter on the interpolation scheme that controls the distance between the datapoints $\mathbf{x}$ and the interpolation points $\mathbf{z}$. When $T = 1$, we obtain the original scheme give in Equation 10. Analogous to how automatic relevance detection (ARD) (MacKay et al.,

Algorithm 1 SoftKI Regression. The procedure `kmeans` performs k-means clustering, `batch` splits the dataset into batches, and `softmax_interpolation` produces a softmax interpolation matrix (see Section 4.1).

Require: SoftKI GP hyperparameters $\theta = (\beta, \ell, \sigma, \mathbf{z} \in \mathbb{R}^{(m \times d)}, T)$, kernel function $k(x, y)$.
Require: Dataset $\mathcal{D} = \{\mathbf{x} \in \mathbb{R}^{(n \times d)}, \mathbf{y} \in \mathbb{R}^{(n \times 1)}\}$.
Require: Optimization hyperparameters: batch size b and learning rate η.
Ensure: Learned SoftKI coefficients α.

▷ Model Training

$\mathbf{z} \leftarrow \texttt{kmeans}(\mathbf{x}, m)$
for $i = 1$ **to** epochs **do**
  for $(\mathbf{x}_b, \mathbf{y}_b)$ in $\texttt{batch}(\mathcal{D}, b)$ **do**
    $\mathbf{\Sigma}_{\mathbf{x}_b\mathbf{z}} \leftarrow \texttt{softmax\_interpolation}(\mathbf{x}_b, \mathbf{z})$
    $\mathbf{K}_{\mathbf{zz}} \leftarrow [\, k(z_i, z_j)\,]_{ij}$
    $\mathbf{K}^{\text{SoftKI}}_{\mathbf{x}_b\mathbf{x}_b} \leftarrow \mathbf{\Sigma}_{\mathbf{x}_b\mathbf{z}} \mathbf{K}_{\mathbf{zz}} \mathbf{\Sigma}^{\top}_{\mathbf{x}_b\mathbf{z}}$
    $\theta \leftarrow \theta + \eta \nabla_\theta \log \widehat{p}\Big(\mathbf{y}_b \,\Big|\, \mathbf{x}_b;\, \mathbf{K}^{\text{SoftKI}}_{\mathbf{x}_b\mathbf{x}_b} + \mathbf{\Lambda}\Big)$ ▷ Stabilized MLL (Section 4.2)
  end for
end for

▷ Stabilized Inference (Section 4.3

$\mathbf{\Sigma}_{\mathbf{xz}} \leftarrow \texttt{softmax\_interpolation}(\mathbf{x}, \mathbf{z})$
$\mathbf{U}^{\top}_{\mathbf{zz}} \mathbf{U}_{\mathbf{zz}} \leftarrow \texttt{cholesky}(\mathbf{K}_{\mathbf{zz}})$
$\mathbf{Q}, \mathbf{R} \leftarrow \texttt{QR}\left(\begin{pmatrix} \mathbf{\Lambda}^{-1/2}\, \mathbf{\Sigma}_{\mathbf{xz}} \mathbf{K}_{\mathbf{zz}} \\ \mathbf{U}_{\mathbf{zz}} \end{pmatrix}\right)$
$\alpha \leftarrow \mathbf{R}^{-1}\, \mathbf{Q}^{\top} \begin{pmatrix} \mathbf{\Lambda}^{-1/2}\, \mathbf{y} \\ 0 \end{pmatrix}$
return α

1994) can be used to set lengthscales for different dimensions, we can also use a different temperature per dimension. When learning both temperatures and lengthscales, we cap the lengthscale range for more stable hyperparameter optimization since they have a push-and-pull effect.

4.2 Learning Interpolation Points

Similar to a SGPR, we can treat the interpolation points $\mathbf{z}$ as hyperparameters of a GP and learn them by optimizing an appropriate objective. We propose optimizing the MLL $\log p(\mathbf{y} \,|\, \mathbf{x}; \theta)$ with a method based on gradient descent for a SoftKI which for $\mathbf{D}_\theta = \mathbf{K}^{\text{SoftKI}}_{\mathbf{xx}}(\theta) + \mathbf{\Lambda}(\theta)$ has closed the form solution

$$\log p(\mathbf{y} \mid \mathbf{x}; \theta) = -\frac{1}{2}\Big[\mathbf{y}^{\top} \mathbf{D}_\theta^{-1} \mathbf{y} + \log\det(\mathbf{D}_\theta) + n \log(2\pi)\Big], \tag{13}$$

with derivative

$$\frac{\partial \log p(\mathbf{y} \mid \mathbf{x}; \theta)}{\partial \theta} = -\frac{1}{2}\Big[\mathbf{y}^{\top} \mathbf{D}_\theta^{-1} \frac{\partial \mathbf{D}_\theta}{\partial \theta} \mathbf{D}_\theta^{-1} \mathbf{y} + \operatorname{tr}\Big(\mathbf{D}_\theta^{-1} \frac{\partial \mathbf{D}_\theta}{\partial \theta}\Big)\Big]. \tag{14}$$

Each evaluation of $\log p(\mathbf{y} \,|\, \mathbf{x}; \theta)$ has time complexity $\mathcal{O}(m^2 n)$ and space complexity $\mathcal{O}(mn)$ where m is the number of interpolation points. This is tractable to compute only when $m \ll n$ because $\mathbf{K}^{\text{SoftKI}}_{\mathbf{xx}}$ is low-rank.

Hutchinson's pseudoloss. Occasionally, we observe that the MLL is numerically unstable in single-precision floating point arithmetic (see Appendix B.1 for further details). To help stabilize the MLL in these situations, we use an approximate MLL based on work in approximate GP theory (Gardner et al., 2018; Maddox et al., 2021; Wenger et al., 2023) that identify efficiently computable estimates of the log determinant (Equation 13) and trace term (Equation 14) in the GP MLL. In more detail, these methods combine stochastic trace estimation via the Hutchinson's trace estimator (Girard, 1989; Hutchinson, 1989) with blocked conjugate gradients so that the overall cost remains quadratic in the size of the kernel. Maddox et al. (2022) term this approximate MLL the *Hutchinson's pseudoloss* and show that it remains stable under low-precision conditions.

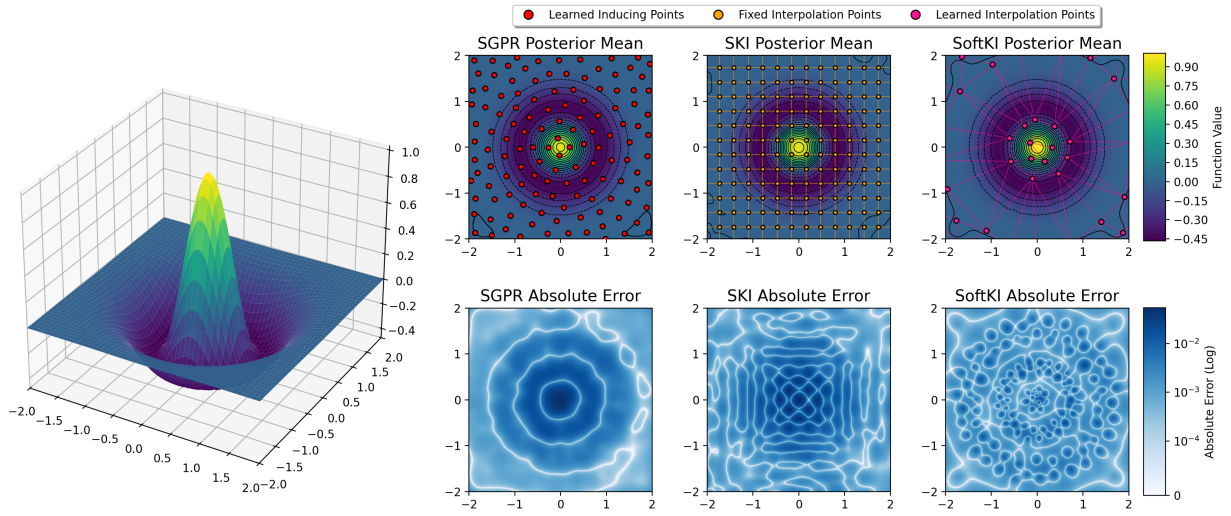


Figure 2: **(Inducing vs. Interpolation Points):** Comparison of inducing points learned by SGPR, static rectilinear lattice points in SKI, and adaptive interpolation points learned by SoftKI on the Ricker wavelet (Ricker, 1953) after 100 epochs of hyperparameter optimization. Contour plots of the posterior mean are paired with each method's absolute-error. Each method achieves comparable accuracy: SGPR's inducing points follow the sample distribution, and SoftKI's interpolation points adapt to the true function's local geometry. Additional experiment details can be found in Appendix C.1

Definition 4.2 (Hutchinson Pseudoloss)**.** Let $\mathbf{u}_0, \mathbf{u}_1, \ldots, \mathbf{u}_l$ be the solutions obtained by using block conjugate gradients for the system $\mathbf{D}_\theta\big(\mathbf{u}_0\ \mathbf{u}_1 \ldots \mathbf{u}_l\big) = \big(\mathbf{y}\ \mathbf{z}_1 \ldots \mathbf{z}_l\big)$, where each $\mathbf{z}_j$ is a Gaussian random vector normalized to unit length. The Hutchinson pseudoloss approximation of the GP marginal log likelihood in Equation 13 is given as

$$\log \tilde{p}(\mathbf{y} \,|\, \mathbf{x}; \theta) = -\frac{1}{2}\left[\mathbf{u}_0^\top \mathbf{D}_\theta \mathbf{u}_0 + \frac{1}{l}\sum_{j=1}^{l} \mathbf{u}_j^\top (\mathbf{D}_\theta \mathbf{z}_j)\right] \tag{15}$$

with derivative

$$\frac{\partial \log \tilde{p}(\mathbf{y} \,|\, \mathbf{x}; \theta)}{\partial \theta} = -\frac{1}{2}\left[\mathbf{u}_0^\top \frac{\partial \mathbf{D}_\theta}{\partial \theta}\mathbf{u}_0 + \frac{1}{l}\sum_{j=1}^{l} \mathbf{u}_j^\top \frac{\partial \mathbf{D}_\theta}{\partial \theta}\mathbf{z}_j\right] \tag{16}$$

By computing Equation 15 as described above as opposed to explicitly computing a stochastic Lanczos quadrature approximation (Ubaru et al., 2017), the gradient can also be efficiently computed using back propagation.

Stabilized MLL. We combine SoftKI's MLL with Hutchinson's pseudoloss to arrive at the objective

$$\log \hat{p}(\mathbf{y} \mid \mathbf{x}; \theta) = \begin{cases} \log p(\mathbf{y} \mid \mathbf{x}; \theta) & \text{when stable} \\ \log \tilde{p}(\mathbf{y} \mid \mathbf{x}; \theta) & \text{otherwise} \end{cases} \tag{17}$$

for optimizing SoftKI's hyperparameters. In practice, this means that we default to using SoftKI's MLL and fallback to Hutchinson's pseudoloss when numeric instability is encountered. This enables SoftKI to accurately recover approximations of the MLL, even when the current positioning of interpolation points would otherwise make the direct computation of the MLL unstable, while relying on the exact MLL as much as possible.

Stochastic optimization. Instead of computing $\nabla \log \hat{p}(\mathbf{y} \mid \mathbf{x}; \theta)$ on the entire dataset $\mathbf{x}$, we can compute $\nabla \log \hat{p}(\mathbf{y} \mid \mathbf{x}_b; \theta)$ on a minibatch of data $\mathbf{x}_b$ of size b to perform stochastic optimization (see line 7 of Algorithm 1). The stochastic estimate provides an unbiased estimator of the gradient. In this way, SoftKI can leverage powerful stochastic optimization techniques used to train neural networks such as Adam (Kingma & Ba, 2014) and hardware acceleration such as graphics processing units (GPUs). The time complexity of evaluating $\nabla \log \hat{p}(\mathbf{y} \mid \mathbf{x}_b; \theta)$ is cheap, costing $\mathcal{O}(b^2 m)$ operations with space complexity $\mathcal{O}(bm)$. The lower space complexity makes it easier to use GPUs since they have more memory constraints compared to CPUs. We emphasize that we are performing stochastic gradient descent and not stochastic variational inference as in SVGP (Hensman et al., 2013). In particular, we do not define a distribution on the interpolation points $\mathbf{z}$ nor make a variational approximation.

4.3 Posterior Inference

Because $\mathbf{K}_{\mathbf{xx}}^{\text{SoftKI}}$ is low-rank, we can use the matrix inversion lemma to rewrite the posterior predictive of SoftKI as

$$p(f(*) \mid \mathbf{y}) = \mathcal{N}(\widehat{\mathbf{K}}_{*\mathbf{z}} \widehat{\mathbf{C}}^{-1} \widehat{\mathbf{K}}_{\mathbf{zx}} \mathbf{\Lambda}^{-1} \mathbf{y}, \mathbf{K}_{**}^{\text{SoftKI}} - \mathbf{K}_{*\mathbf{x}}^{\text{SoftKI}} (\mathbf{\Lambda}^{-1} - \mathbf{\Lambda}^{-1} \widehat{\mathbf{K}}_{\mathbf{xz}} \widehat{\mathbf{C}}^{-1} \widehat{\mathbf{K}}_{\mathbf{zx}} \mathbf{\Lambda}^{-1}) \mathbf{K}_{\mathbf{x}*}^{\text{SoftKI}}) \tag{18}$$

where $\widehat{\mathbf{C}} = \mathbf{K}_{\mathbf{zz}} + \widehat{\mathbf{K}}_{\mathbf{zx}} \mathbf{\Lambda}^{-1} \widehat{\mathbf{K}}_{\mathbf{xz}}$ (see Appendix A). To perform posterior mean inference, we solve

$$\widehat{\mathbf{C}} \alpha = \widehat{\mathbf{K}}_{\mathbf{zx}} \mathbf{\Lambda}^{-1} \mathbf{y} \tag{19}$$

for weights α. Note that this is the posterior mean of a SGPR with $\mathbf{C}$ replaced with $\widehat{\mathbf{C}}$ and $\mathbf{K}_{\mathbf{xz}}$ replaced with $\widehat{\mathbf{K}}_{\mathbf{xz}}$. Moreover, note that $\widehat{\mathbf{C}}$ is $\mathbf{C}$ with $\mathbf{K}_{\mathbf{xz}}$ replaced with $\widehat{\mathbf{K}}_{\mathbf{xz}}$. Since $\widehat{\mathbf{C}}$ is a $m \times m$ matrix, the solution of the system of linear equations has time complexity $\mathcal{O}(m^3)$. The formation of $\widehat{\mathbf{C}}$ requires the multiplication of a $m \times n$ matrix with a $n \times m$ matrix which has time complexity $\mathcal{O}(m^2 n)$. Thus the complexity of SoftKI posterior mean inference is $\mathcal{O}(m^2 n)$ since it is dominated by the formation of $\widehat{\mathbf{C}}$. We refer the reader to the supplementary material for discussion of the posterior covariance (Appendix A).

Solving with QR. Unfortunately, solving Equation 19 for α can be numerically unstable. Foster et al. (2009) introduce a stable QR solver approach for a Subset of Regressors (SoR) GP (Smola & Bartlett, 2000) which we adapt to a SoftKI. Define the block matrix

$$\mathbf{A} = \begin{pmatrix} \mathbf{\Lambda}^{-1/2} \widehat{\mathbf{K}}_{\mathbf{xz}} \\ \mathbf{U}_{\mathbf{zz}} \end{pmatrix} \tag{20}$$

where $\mathbf{U}_{\mathbf{zz}}^\top \mathbf{U}_{\mathbf{zz}} = \mathbf{K}_{\mathbf{zz}}$ is the upper triangular Cholesky decomposition of $\mathbf{K}_{\mathbf{zz}}$ so that

$$\mathbf{A}^\top \mathbf{A} = \widehat{\mathbf{K}}_{\mathbf{zx}} \mathbf{\Lambda}^{-1} \widehat{\mathbf{K}}_{\mathbf{xz}} + \mathbf{K}_{\mathbf{zz}} = \widehat{\mathbf{C}} \,. \tag{21}$$

Let $\mathbf{QR} = \mathbf{A}$ be the QR decomposition of $\mathbf{A}$ so that $\mathbf{Q}$ is a $(n+m) \times m$ orthonormal matrix and $\mathbf{R}$ is $m \times m$ right triangular matrix. Then

$$\begin{aligned} \widehat{\mathbf{C}} \alpha &= \widehat{\mathbf{K}}_{\mathbf{zx}} \mathbf{\Lambda}^{-1} \mathbf{y} && (\iff) \\ (\mathbf{QR})^\top (\mathbf{QR}) \alpha &= (\mathbf{QR})^\top \begin{pmatrix} \mathbf{\Lambda}^{-1/2} \mathbf{y} \\ 0 \end{pmatrix} && (\iff) \\ \mathbf{R} \alpha &= \mathbf{Q}^\top \begin{pmatrix} \mathbf{\Lambda}^{-1/2} \mathbf{y} \\ 0 \end{pmatrix} . && (22) \end{aligned}$$

We thus solve Equation 22 for α via a triangular solve in $\mathcal{O}(m^2)$ time. For additional experiments demonstrating the inability of other linear solvers to offer comparable accuracy to the QR-stabilized solve detailed in this section, see Appendix B.2.

5 Experiments

We compare the performance of SoftKIs against popular scalable GPs on selected data sets from the `UCI` data set repository (Kelly et al., 2017), a common GP benchmark (Section 5.1). Next, we test SoftKIs on

	n	d	SoftKI m=512	SGPR m=512	SVGP m=1024
3droad	391386	3	0.583 ± 0.01	-	**0.389 ± 0.001**
Kin40k	36000	8	0.169 ± 0.008	0.177 ± 0.005	**0.165 ± 0.005**
Protein	41157	9	**0.596 ± 0.016**	0.601 ± 0.015	0.607 ± 0.012
Houseelectric	1844352	11	**0.047 ± 0.001**	-	**0.047 ± 0.0**
Bike	15641	17	**0.062 ± 0.001**	0.099 ± 0.003	0.084 ± 0.006
Elevators	14939	18	**0.36 ± 0.006**	0.393 ± 0.007	0.384 ± 0.008
Keggdirected	43944	20	**0.08 ± 0.005**	0.251 ± 0.13	0.082 ± 0.004
Pol	13500	26	**0.075 ± 0.002**	0.108 ± 0.001	0.122 ± 0.002
Keggundirected	57247	27	**0.115 ± 0.004**	0.132 ± 0.021	0.121 ± 0.007
Buzz	524925	77	**0.24 ± 0.001**	-	0.25 ± 0.002
Song	270000	90	**0.777 ± 0.004**	-	0.794 ± 0.006
Slice	48150	385	**0.022 ± 0.006**	0.435 ± 0.003	0.082 ± 0.001

Table 1: Test RMSE on `UCI` datasets. Best results are bolded.

	n	d	SoftKI m=512	SGPR m=512	SVGP m=1024
3droad	391386	3	0.953 ± 0.041	-	**0.597 ± 0.008**
Kin40k	36000	8	0.055 ± 0.043	**-0.105 ± 0.01**	-0.082 ± 0.007
Protein	41157	9	**0.905 ± 0.026**	1.029 ± 0.01	1.047 ± 0.009
Houseelectric	1844352	11	1.274 ± 1.425	-	**-1.492 ± 0.007**
Bike	15641	17	**-0.379 ± 0.016**	-0.322 ± 0.01	-0.68 ± 0.017
Elevators	14939	18	**0.406 ± 0.021**	0.581 ± 0.008	0.586 ± 0.009
Keggdirected	43944	20	0.421 ± 0.063	2.355 ± 3.875	**-0.934 ± 0.017**
Pol	13500	26	**-0.71 ± 0.028**	-0.538 ± 0.003	-0.394 ± 0.012
Keggundirected	57247	27	0.302 ± 0.136	-0.56 ± 0.108	**-0.59 ± 0.021**
Buzz	524925	77	0.276 ± 0.348	-	**0.13 ± 0.008**
Song	270000	90	**1.179 ± 0.008**	-	1.288 ± 0.002
Slice	48150	385	1.258 ± 0.149	1.295 ± 0.003	**-0.662 ± 0.002**

Table 2: Test NLL on `UCI` datasets. Best results are bolded.

high-dimensional molecule data sets from the domain of computational chemistry (Section 5.2). Finally, we explore the interpolation points learned by SoftKI (Section 5.3).

5.1 Benchmark on UCI Regression

We evaluate the efficacy of a SoftKI against other scalable GP methods on data sets of varying size n and data dimensionality d from the `UCI` repository (Kelly et al., 2017). We choose SGPR and SVGP as two scalable GP methods since these methods can be applied in a relatively blackbox fashion, and thus, can be applied to many data sets. It is not possible to apply SKI to most datasets that we test on due to scalability issues with data dimensionality, and so we omit it. For additional comparisons to other alternative SKI architectures, see Appendix C.3.

Experiment details. For this experiment, we split the data set into 0.9 for training and 0.1 for testing. We standardize the data to have mean 0 and standard deviation 1 using the training data set. We use a Matérn 3/2 kernel and a learnable output scale. We choose $m = 512$ inducing points for SoftKI. Following Wang et al. (2019), we use $m = 512$ for SGPR and $m = 1024$ for SVGP. We learn model hyperparameters for SoftKI by maximizing Hutchinson's pseudoloss, and for SGPR and SVGP by maximizing the ELBO.

We perform 50 epochs of training using the Adam optimizer (Kingma & Ba, 2014) for all methods with a learning rate of $\eta = 0.01$. The learning rate for SGPR is $\eta = 0.1$ since we are not performing batching. We use a default implementation of SGPR and SVGP from GPyTorch. For SoftKI and SVGP, we use a

	n	d	SoftKI m=512	SGPR m=512	SVGP m=1024
Ac-ala3-nhme	76598	126	**0.79 ± 0.01**	0.847 ± 0.006	0.836 ± 0.005
Dha	62777	168	0.886 ± 0.012	0.897 ± 0.012	**0.882 ± 0.01**
Stachyose	24544	261	**0.363 ± 0.012**	0.638 ± 0.003	0.555 ± 0.001
At-at	18000	354	**0.528 ± 0.011**	0.589 ± 0.008	0.558 ± 0.009
At-at-cg-cg	9137	354	0.502 ± 0.007	0.563 ± 0.024	**0.461 ± 0.026**
Buckyball-catcher	5491	444	**0.153 ± 0.006**	0.394 ± 0.011	0.307 ± 0.015
Double-walled-nanotube	4528	1110	**0.031 ± 0.001**	1.001 ± 0.048	0.045 ± 0.0

Table 3: Test RMSE on `MD22` datasets. Best results are bolded.

	n	d	SoftKI m=512	SGPR m=512	SVGP m=1024
Ac-ala3-nhme	76598	126	**1.188 ± 0.014**	1.364 ± 0.003	1.356 ± 0.002
Dha	62777	168	**1.299 ± 0.014**	1.418 ± 0.006	1.409 ± 0.007
Stachyose	24544	261	**0.522 ± 0.059**	1.118 ± 0.003	1.012 ± 0.003
At-at	18000	354	**0.795 ± 0.007**	1.037 ± 0.004	1.003 ± 0.008
At-at-cg-cg	9137	354	**0.742 ± 0.01**	0.987 ± 0.017	0.8 ± 0.016
Buckyball-catcher	5491	444	**-0.278 ± 0.061**	0.69 ± 0.009	0.449 ± 0.015
Double-walled-nanotube	4528	1110	**-1.551 ± 0.049**	1.52 ± 0.021	-0.954 ± 0.009

Table 4: Test NLL on `MD22` datasets. Best results are bolded.

minibatch size of 1024. We report the average test RMSE (Table 1) and NLL (Table 2) for each method across three seeds. Additional timing information is given in Appendix C.2.

Results. We observe that SoftKI has competitive test RMSE performance compared to SGPR and SVGP, exceeding them when the dimension is modest ($d \approx 10$). For the test NLL, we observe cases where SoftKI's test NLL is smaller or larger relative to the test RMSE. As a reminder, the NLL for a GP method consists of two components: the RMSE and the complexity of the model. Consequently, a small test NLL relative to the test RMSE indicates a relatively small amount of noise in the dataset. Conversely, a large test NLL relative to the test RMSE indicates a large amount of noise in the dataset. We can see instances of this in the `bike` dataset where SoftKI achieves comparable test RMSE to SGPR and SVGP, but results in larger NLL. This indicates that SoftKI's kernel is more complex compared to the variational GP kernels.

5.2 High-Dimensional Molecule Dataset

Since SoftKI's are performant on large dimensional data sets from the `UCI` repository, we also test the performance on molecular potential energy surface data on the `MD22` (Chmiela et al., 2023) dataset. These are high-dimensional datasets that give the geometric coordinates of atomic nuclei in biomolecules and their respective energies.

Experiment details. For consistency, we keep the experimental setup the same as for `UCI` regression but up the number of epochs of training to 200 due to slower convergence on these datasets. We still use the Matérn kernel and not a molecule-specific kernel that incorporates additional information such as the atomic number of each atom (*e.g.*, a Hydrogen atom) or invariances (*e.g.*, rotational invariance). We standardize the data set to have mean 0 and standard deviation 1. We note that in an actual application of GP regression to this setting, we may only opt to center the targets to be mean 0. This is because energy is a relative number that can be arbitrarily shifted, whereas scaling the distances between atoms will affect the physics.

Results. Table 3 and Table 4 compares the test RMSE and test NLL of various GP models trained on `MD22`. We see that SoftKI is a competitive method on these datasets, especially on the test NLL. GPs that fit forces have been successfully applied to fit such data sets to chemical accuracy. In our case, we do not fit derivative information. It would be an interesting direction of future work to extend SoftKI to handle derivatives.

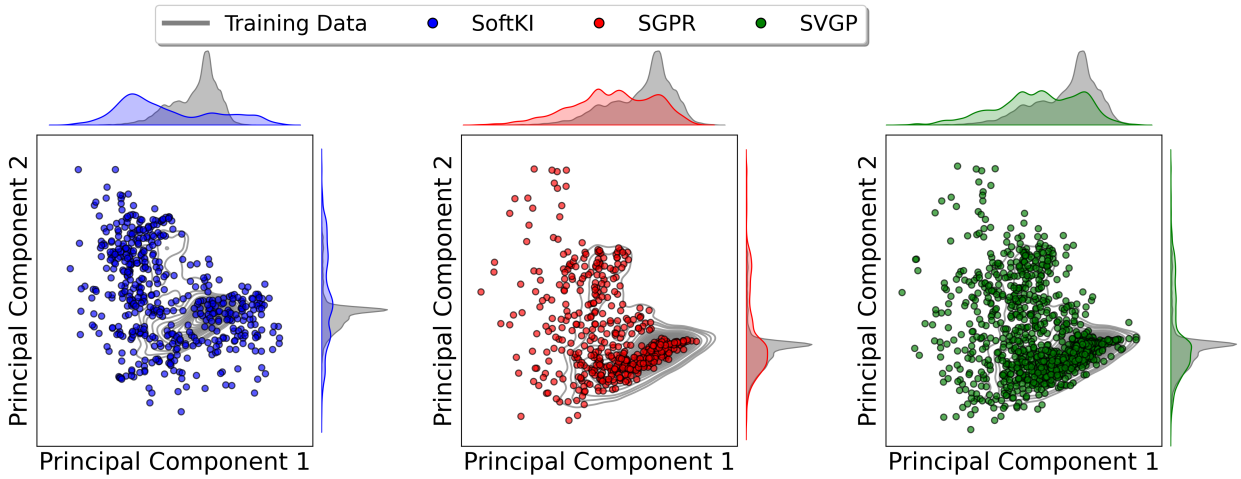


Figure 3: **(Interpolation vs. Inducing points–*High Dimensional Setting*):** In this comparison the `slice` ($d = 385$) dataset's PCA embedding is presented, along with a kernel density estimate of the embedded training data to more clearly highlight the placement of the learned interpolation points learned by SoftKI, and inducing points learned by SGPR and SVGP relative to the distribution of the data. Importantly note that in this example SVGP is run with twice as many (1024) inducing points compared to the other methods. Appendix C.4 contains the PCA analysis for the remaining `UCI` datasets.

5.3 Learned Interpolation Points

Because a SoftKI's predictive performance is determined by where it places its learned interpolation points, we examine their arrangement for the `slice` dataset in Figure 3. We apply a two-component PCA embedding to both the training dataset and the interpolation/inducing points $\mathbf{z}$ learned by SoftKI, SGPR, and SVGP. To assess how effectively each method captures the overall data distribution, we plot the PCA embeddings of the inducing points overlaid on the kernel density estimation (KDE) contour levels of the training points in the PCA space, along with the marginal distributions along each axis.

In general, we see that the interpolation point locations learned by SoftKI are more spread out compared to the inducing points learned by SGPR and SVGP. We believe that this is because the prior on inducing variables linking the locations of inducing points to the data in SGPR and SVGP are normally distributed whereas there are no such assumptions in the case of SoftKI. Moreover, we optimize a stabilized MLL that is related to an exact GP's MLL as opposed to an ELBO. Thus, while there are similarities in that both SoftKI and SGPR begin with a Nyström approximation of the covariance matrix, interpolation points and inducing points are subtly different. Additional interpolation point analysis, including PCA analysis for other datasets in the `UCI` repository, are included in Appendix C.4.

6 Conclusion

In this paper, we introduce SoftKI, an approximate GP designed for regression on large and high-dimensional datasets. SoftKI combines aspects of SKI and inducing points methods to retain the benefits of kernel interpolation while also scaling to higher dimensional datasets. We have tested SoftKI on a variety of datasets and shown that it is possible to perform kernel interpolation in high dimensional spaces in a way that is competitive with other approximate GP abstractions that leverage inducing points. Moreover, we find that the interpolation points learned by SoftKI are structurally different from the inducing points learned in SGRP and SVGP. This suggests that methods that learn interpolation points provide another promising path for constructing approximate GPs. Additionally, exploring methods to enforce stricter sparsity in the

interpolation matrices, such as through thresholding, could enable the use of sparse matrix data structures further improving the cost of inference.

References

Andrew Adams, Jongmin Baek, and Myers Davis. Fast high-dimensional filtering using the permutohedral lattice. *Comput. Graph. Forum*, 29:753–762, 05 2010. doi: 10.1111/j.1467-8659.2009.01645.x.

Stefan Chmiela, Valentin Vassilev-Galindo, Oliver T Unke, Adil Kabylda, Huziel E Sauceda, Alexandre Tkatchenko, and Klaus-Robert Müller. Accurate global machine learning force fields for molecules with hundreds of atoms. *Science Advances*, 9(2):eadf0873, 2023.

Leslie Foster, Alex Waagen, Nabeela Aijaz, Michael Hurley, Apolonio Luis, Joel Rinsky, Chandrika Satyavolu, Michael J. Way, Paul Gazis, and Ashok Srivastava. Stable and efficient gaussian process calculations. *Journal of Machine Learning Research*, 10(31):857–882, 2009. URL `http://jmlr.org/papers/v10/foster09a.html`.

Jacob R. Gardner, Geoff Pleiss, Ruihan Wu, Kilian Q. Weinberger, and Andrew Gordon Wilson. Product kernel interpolation for scalable gaussian processes, 2018. URL `https://arxiv.org/abs/1802.08903`.

Jacob R. Gardner, Geoff Pleiss, David Bindel, Kilian Q. Weinberger, and Andrew Gordon Wilson. Gpytorch: Blackbox matrix-matrix gaussian process inference with gpu acceleration, 2021.

A. Girard. A fast 'monte-carlo cross-validation' procedure for large least squares problems with noisy data. *Numer. Math.*, 56(1):1–23, January 1989. ISSN 0029-599X. doi: 10.1007/BF01395775. URL `https://doi.org/10.1007/BF01395775`.

James Hensman, Nicolo Fusi, and Neil D Lawrence. Gaussian processes for big data. *arXiv preprint arXiv:1309.6835*, 2013.

M.F. Hutchinson. A stochastic estimator of the trace of the influence matrix for laplacian smoothing splines. *Communication in Statistics- Simulation and Computation*, 18:1059–1076, 01 1989. doi: 10.1080/03610919008812866.

Sanyam Kapoor, Marc Finzi, Ke Alexander Wang, and Andrew Gordon Gordon Wilson. Skiing on simplices: Kernel interpolation on the permutohedral lattice for scalable gaussian processes. In *International Conference on Machine Learning*, pp. 5279–5289. PMLR, 2021.

Markelle Kelly, Rachel Longjohn, and Kolby Nottingham. The uci machine learning repository. `https://archive.ics.uci.edu`, 2017.

R. Keys. Cubic convolution interpolation for digital image processing. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 29(6):1153–1160, 1981. doi: 10.1109/TASSP.1981.1163711.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

David JC MacKay et al. Bayesian nonlinear modeling for the prediction competition. *ASHRAE transactions*, 100(2):1053–1062, 1994.

Wesley J. Maddox, Sanyam Kapoor, and Andrew Gordon Wilson. When are iterative gaussian processes reliably accurate?, 2021. URL `https://arxiv.org/abs/2112.15246`.

Wesley J Maddox, Andres Potapcynski, and Andrew Gordon Wilson. Low-precision arithmetic for fast gaussian processes. In *Uncertainty in Artificial Intelligence*, pp. 1306–1316. PMLR, 2022.

Leland McInnes, John Healy, and James Melville. Umap: Uniform manifold approximation and projection for dimension reduction, 2020. URL `https://arxiv.org/abs/1802.03426`.

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.

Carl Edward Rasmussen and Christopher K. I. Williams. *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press, 2005. ISBN 026218253X.

Norman Ricker. The form and laws of propagation of seismic wavelets. *Geophysics*, 18(1):10–40, 1953. doi: 10.1190/1.1437843.

Alex Smola and Peter Bartlett. Sparse greedy gaussian process regression. In T. Leen, T. Dietterich, and V. Tresp (eds.), *Advances in Neural Information Processing Systems*, volume 13. MIT Press, 2000. URL https://proceedings.neurips.cc/paper_files/paper/2000/file/3214a6d842cc69597f9edf26df552e43-Paper.pdf.

Edward Snelson and Zoubin Ghahramani. Sparse gaussian processes using pseudo-inputs. *Advances in neural information processing systems*, 18, 2005.

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

Shashanka Ubaru, Jie Chen, and Yousef Saad. Fast Estimation of $tr(f(A))$ via Stochastic Lanczos Quadrature. *SIAM Journal on Matrix Analysis and Applications*, 38(4):1075–1099, January 2017. ISSN 0895-4798, 1095-7162. doi: 10.1137/16M1104974.

Ke Alexander Wang, Geoff Pleiss, Jacob R. Gardner, Stephen Tyree, Kilian Q. Weinberger, and Andrew Gordon Wilson. Exact gaussian processes on a million data points, 2019. URL https://arxiv.org/abs/1903.08114.

Jonathan Wenger, Geoff Pleiss, Marvin Pförtner, Philipp Hennig, and John P. Cunningham. Posterior and computational uncertainty in gaussian processes, 2023. URL https://arxiv.org/abs/2205.15449.

Christopher K. I. Williams and Matthias Seeger. Using the nyström method to speed up kernel machines. In *Proceedings of the 13th International Conference on Neural Information Processing Systems*, NIPS'00, pp. 661–667, Cambridge, MA, USA, 2000. MIT Press.

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

Andrew Gordon Wilson. *Covariance kernels for fast automatic pattern discovery and extrapolation with Gaussian processes*. PhD thesis, University of Cambridge Cambridge, UK, 2014.

Luhuan Wu, Geoff Pleiss, and John Cunningham. Variational nearest neighbor gaussian process, 2024. URL https://arxiv.org/abs/2202.01694.

Mohit Yadav, Daniel Sheldon, and Cameron Musco. Kernel interpolation with sparse grids, 2023. URL https://arxiv.org/abs/2305.14451.

A SoftKI Posterior Predictive Derivation

As a reminder, the SoftKI posterior predictive is

$$p(f_{\mathbf{z}}(*) \mid y) = \mathcal{N}(\widehat{\mathbf{K}}_{*\mathbf{z}}\widehat{\mathbf{C}}^{-1}\widehat{\mathbf{K}}_{\mathbf{zx}}\mathbf{\Lambda}^{-1}\mathbf{y}, \mathbf{K}_{**}^{\text{SoftKI}} - \mathbf{K}_{*\mathbf{x}}^{\text{SoftKI}}(\mathbf{\Lambda}^{-1} - \mathbf{\Lambda}^{-1}\widehat{\mathbf{K}}_{\mathbf{xz}}\widehat{\mathbf{C}}^{-1}\widehat{\mathbf{K}}_{\mathbf{zx}}\mathbf{\Lambda}^{-1})\mathbf{K}_{\mathbf{x}*}^{\text{SoftKI}}) \tag{23}$$

where $\widehat{\mathbf{C}} = \mathbf{K_{zz}} + \mathbf{K_{zx}}\mathbf{\Lambda}^{-1}\mathbf{K_{xz}}$. To derive this, we first recall how to apply the matrix inversion lemma to the posterior mean of a SGPR, which as a reminder uses the covariance approximation $\mathbf{K_{xx}} \approx \mathbf{K}_{\mathbf{xx}}^{\text{SGPR}} = \mathbf{K_{xz}}\mathbf{K}_{\mathbf{zz}}^{-1}\mathbf{K_{zx}}$

Lemma A.1 (Matrix inversion with GPs)**.**

$$\mathbf{K}_{*\mathbf{z}}(\mathbf{K_{zz}} + \mathbf{K_{zx}}\mathbf{\Lambda}^{-1}\mathbf{K_{xz}})^{-1}\mathbf{K_{zx}}\mathbf{\Lambda}^{-1} = \mathbf{K}_{*\mathbf{x}}^{SGPR}(\mathbf{K}_{\mathbf{xx}}^{SGPR} + \mathbf{\Lambda})^{-1} \tag{24}$$

Proof.

$$
\begin{aligned}
&\mathbf{K}_{\mathbf{xx}}^{\mathrm{SGPR}} + \mathbf{\Lambda} = \mathbf{K}_{\mathbf{xx}}^{\mathrm{SGPR}} + \mathbf{\Lambda} && \text{(identity)}\\
\iff &\mathbf{I} = (\mathbf{K}_{\mathbf{xx}}^{\mathrm{SGPR}} + \mathbf{\Lambda})^{-1}\mathbf{K}_{\mathbf{xx}}^{\mathrm{SGPR}} + (\mathbf{K}_{\mathbf{xx}}^{\mathrm{SGPR}} + \mathbf{\Lambda})^{-1}\mathbf{\Lambda} && \text{(mult by } (\mathbf{K}_{\mathbf{xx}}^{\mathrm{SGPR}} + \mathbf{\Lambda})^{-1})\\
\iff &\mathbf{I} - (\mathbf{K}_{\mathbf{xx}}^{\mathrm{SGPR}} + \mathbf{\Lambda})^{-1}\mathbf{K}_{\mathbf{xx}}^{\mathrm{SGPR}} = (\mathbf{K}_{\mathbf{xx}}^{\mathrm{SGPR}} + \mathbf{\Lambda})^{-1}\mathbf{\Lambda} && \text{(rearrange)}\\
\iff &\mathbf{K}_{*\mathbf{x}}^{\mathrm{SGPR}} - \mathbf{K}_{*\mathbf{x}}^{\mathrm{SGPR}}(\mathbf{K}_{\mathbf{xx}}^{\mathrm{SGPR}} + \mathbf{\Lambda})^{-1}\mathbf{K}_{\mathbf{xx}}^{\mathrm{SGPR}} = \mathbf{K}_{*\mathbf{x}}^{\mathrm{SGPR}}(\mathbf{K}_{\mathbf{xx}}^{\mathrm{SGPR}} + \mathbf{\Lambda})^{-1}\mathbf{\Lambda} \\
& && \text{(mult both sides by } \mathbf{K}_{*\mathbf{x}}^{\mathrm{SGPR}} \text{ on left)}\\
\iff &\mathbf{K}_{*\mathbf{z}}\mathbf{K}_{\mathbf{zz}}^{-1}\mathbf{K}_{\mathbf{zx}} - \mathbf{K}_{*\mathbf{z}}\mathbf{K}_{\mathbf{zz}}^{-1}\mathbf{K}_{\mathbf{zx}}(\mathbf{K}_{\mathbf{xz}}\mathbf{K}_{\mathbf{zz}}^{-1}\mathbf{K}_{\mathbf{zx}} + \mathbf{\Lambda})^{-1}\mathbf{K}_{\mathbf{xz}}\mathbf{K}_{\mathbf{zz}}^{-1}\mathbf{K}_{\mathbf{zx}} = \mathbf{K}_{*\mathbf{x}}^{\mathrm{SGPR}}(\mathbf{K}_{\mathbf{xx}}^{\mathrm{SGPR}} + \mathbf{\Lambda})^{-1}\mathbf{\Lambda} \\
& && \text{(defn } \mathbf{K}_{\mathbf{xx}}^{\mathrm{SGPR}})\\
\iff &\mathbf{K}_{*\mathbf{z}}(\mathbf{K}_{\mathbf{zz}}^{-1} - \mathbf{K}_{\mathbf{zz}}^{-1}\mathbf{K}_{\mathbf{zx}}(\mathbf{K}_{\mathbf{xz}}\mathbf{K}_{\mathbf{zz}}^{-1}\mathbf{K}_{\mathbf{zx}} + \mathbf{\Lambda})^{-1}\mathbf{K}_{\mathbf{xz}}\mathbf{K}_{\mathbf{zz}}^{-1})\mathbf{K}_{\mathbf{zx}} = \mathbf{K}_{*\mathbf{x}}^{\mathrm{SGPR}}(\mathbf{K}_{\mathbf{xx}}^{\mathrm{SGPR}} + \mathbf{\Lambda})^{-1}\mathbf{\Lambda} && \text{(factor)}\\
\iff &\mathbf{K}_{*\mathbf{z}}(\mathbf{K}_{\mathbf{zz}} + \mathbf{K}_{\mathbf{zx}}\mathbf{\Lambda}^{-1}\mathbf{K}_{\mathbf{xz}})^{-1}\mathbf{K}_{\mathbf{zx}} = \mathbf{K}_{*\mathbf{x}}^{\mathrm{SGPR}}(\mathbf{K}_{\mathbf{xx}}^{\mathrm{SGPR}} + \mathbf{\Lambda})^{-1}\mathbf{\Lambda} && \text{(matrix inversion lemma)}\\
\iff &\mathbf{K}_{*\mathbf{z}}(\mathbf{K}_{\mathbf{zz}} + \mathbf{K}_{\mathbf{zx}}\mathbf{\Lambda}^{-1}\mathbf{K}_{\mathbf{xz}})^{-1}\mathbf{K}_{\mathbf{zx}}\mathbf{\Lambda}^{-1} = \mathbf{K}_{*\mathbf{x}}^{\mathrm{SGPR}}(\mathbf{K}_{\mathbf{xx}}^{\mathrm{SGPR}} + \mathbf{\Lambda})^{-1}\,. && \text{(mult } \mathbf{\Lambda}^{-1} \text{ on right)}
\end{aligned}
$$

□

Now, we extend the matrix inversion lemma to SoftKI.

Lemma A.2 (Matrix inversion with interpolation)**.**

$$\widehat{\mathbf{K}}_{*\mathbf{z}}(\mathbf{K}_{\mathbf{zz}} + \widehat{\mathbf{K}}_{\mathbf{zx}}\mathbf{\Lambda}^{-1}\widehat{\mathbf{K}}_{\mathbf{xz}})^{-1}\widehat{\mathbf{K}}_{\mathbf{zx}}\mathbf{\Lambda}^{-1} = \mathbf{K}_{*\mathbf{x}}^{SoftKI}(\mathbf{K}_{\mathbf{xx}}^{SoftKI} + \mathbf{\Lambda})^{-1} \tag{25}$$

Proof. Recall $\mathbf{K}_{**}^{\mathrm{SoftKI}} = \mathbf{\Sigma}_{\mathbf{xz}}\mathbf{K}_{\mathbf{zz}}\mathbf{\Sigma}_{\mathbf{zx}} = \mathbf{\Sigma}_{\mathbf{xz}}\mathbf{K}_{\mathbf{zz}}\mathbf{K}_{\mathbf{zz}}^{-1}\mathbf{K}_{\mathbf{zz}}\mathbf{\Sigma}_{\mathbf{zx}} = \widehat{\mathbf{K}}_{\mathbf{xz}}\mathbf{K}_{\mathbf{zz}}^{-1}\widehat{\mathbf{K}}_{\mathbf{zx}}$. The result follows by applying Lemma A with $\mathbf{K}_{\mathbf{xz}}$ replaced with $\widehat{\mathbf{K}}_{\mathbf{xz}}$. □

The posterior covariance is computed by a simple application of the matrix inversion lemma. To compute it, observe that the same procedure for solving the posterior mean can also be used for solving the posterior covariance by replacing $\mathbf{y}$ with $\mathbf{K}_{\mathbf{x}*}^{\mathrm{SoftKI}}$. More concretely,

$$
\begin{aligned}
&\mathbf{K}_{**}^{\mathrm{SoftKI}} - \mathbf{K}_{*\mathbf{x}}^{\mathrm{SoftKI}}(\mathbf{\Lambda}^{-1} - \mathbf{\Lambda}^{-1}\widehat{\mathbf{K}}_{\mathbf{xz}}\widehat{\mathbf{C}}^{-1}\widehat{\mathbf{K}}_{\mathbf{zx}}\mathbf{\Lambda}^{-1})\mathbf{K}_{\mathbf{x}*}^{\mathrm{SoftKI}} && (26)\\
&= \mathbf{K}_{**}^{\mathrm{SoftKI}} - \mathbf{K}_{*\mathbf{x}}^{\mathrm{SoftKI}}\mathbf{\Lambda}^{-1}\mathbf{K}_{\mathbf{x}*}^{\mathrm{SoftKI}} + \mathbf{K}_{*\mathbf{x}}^{\mathrm{SoftKI}}\mathbf{\Lambda}^{-1}\widehat{\mathbf{K}}_{\mathbf{xz}}\widehat{\mathbf{C}}^{-1}\widehat{\mathbf{K}}_{\mathbf{zx}}\mathbf{\Lambda}^{-1}\mathbf{K}_{\mathbf{x}*}^{\mathrm{SoftKI}} && (27)\\
&= \mathbf{K}_{**}^{\mathrm{SoftKI}} - \mathbf{K}_{*\mathbf{x}}^{\mathrm{SoftKI}}\mathbf{\Lambda}^{-1}\mathbf{K}_{\mathbf{x}*}^{\mathrm{SoftKI}} + \mathbf{K}_{*\mathbf{x}}^{\mathrm{SoftKI}}\mathbf{\Lambda}^{-1}\widehat{\mathbf{K}}_{\mathbf{xz}}\alpha_* && (28)
\end{aligned}
$$

where

$$\widehat{\mathbf{C}}\alpha_* = \widehat{\mathbf{K}}_{\mathbf{zx}}\mathbf{\Lambda}^{-1}\mathbf{K}_{\mathbf{x}*}^{\mathrm{SoftKI}} \tag{29}$$

can be solved for α_* in the same way we solved for the posterior mean (with $\mathbf{y}$ instead of $\mathbf{K}_{\mathbf{x}*}^{\mathrm{SoftKI}}$). Since this depends on the inference point $*$, we cannot precompute the result ahead of time as we could with the posterior mean. Nevertheless, the intermediate results of the QR decomposition can be computed once during posterior mean inference and reused for the covariance prediction. Thus, the time complexity of posterior covariance inference is $\mathcal{O}(m^2 n)$ per a test point.

B Algorithmic Choices

In this section, we examine various design choices we made for SoftKI. First, we discuss the rationale behind using a stochastic trace estimation scheme for computing the MLL during stochastic optimization (Section B.1). Next, we motivate choice of linear solver, comparing the use alternative linear solvers for the posterior inference procedure described in Section 4.3.

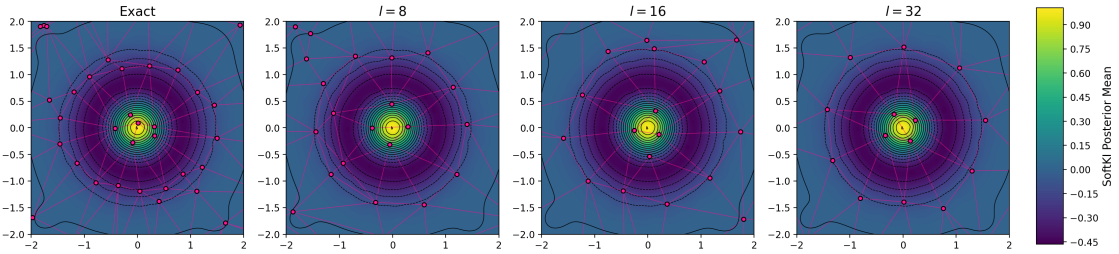


Figure 4: Comparison of Ricker wavelet surface reconstruction and interpolation points learned using the SoftKI MLL and Hutchinson's pseudoloss with varying numbers of probes.

UCI Dataset			Test RMSE		Test NLL	
dataset	n	d	$\log \hat{p}(\mathbf{y} \mid \mathbf{x}; \theta)$	$\log \tilde{p}(\mathbf{y} \mid \mathbf{x}; \theta)$	$\log \hat{p}(\mathbf{y} \mid \mathbf{x}; \theta)$	$\log \tilde{p}(\mathbf{y} \mid \mathbf{x}; \theta)$
3droad	391386	3	0.583 ± 0.01	**0.541 ± 0.005**	**0.953 ± 0.041**	1.004 ± 0.175
Kin40k	36000	8	**0.169 ± 0.008**	0.183 ± 0.006	**0.055 ± 0.043**	0.074 ± 0.026
Protein	41157	9	**0.596 ± 0.016**	0.602 ± 0.016	**0.905 ± 0.026**	0.914 ± 0.029
Houseelectric	1844352	11	**0.047 ± 0.0**	0.049 ± 0.002	**0.451 ± 0.029**	1.275 ± 1.103
Bike	15641	17	**0.062 ± 0.001**	0.073 ± 0.005	**-0.379 ± 0.016**	-0.07 ± 0.216
Elevators	14939	18	**0.36 ± 0.006**	0.361 ± 0.007	0.406 ± 0.021	**0.405 ± 0.024**
Keggdirected	43944	20	**0.08 ± 0.005**	0.082 ± 0.004	**0.421 ± 0.063**	0.5 ± 0.232
Pol	13500	26	**0.075 ± 0.002**	0.084 ± 0.001	-0.71 ± 0.028	**-0.744 ± 0.07**
Keggundirected	57247	27	**0.115 ± 0.004**	0.118 ± 0.004	**0.302 ± 0.136**	0.315 ± 0.017
Buzz	524925	77	**0.24 ± 0.001**	**0.24 ± 0.0**	**0.276 ± 0.348**	0.29 ± 0.33
Song	270000	90	**0.777 ± 0.004**	**0.777 ± 0.004**	1.179 ± 0.008	**1.178 ± 0.007**
Slice	48150	385	**0.022 ± 0.006**	0.045 ± 0.004	1.258 ± 0.149	**0.502 ± 0.137**

Table 5: Effect on SoftKI RMSE and NLL using $\log \hat{p}(\mathbf{y} \mid \mathbf{x}; \theta)$ vs. $\log \tilde{p}(\mathbf{y} \mid \mathbf{x}; \theta)$.

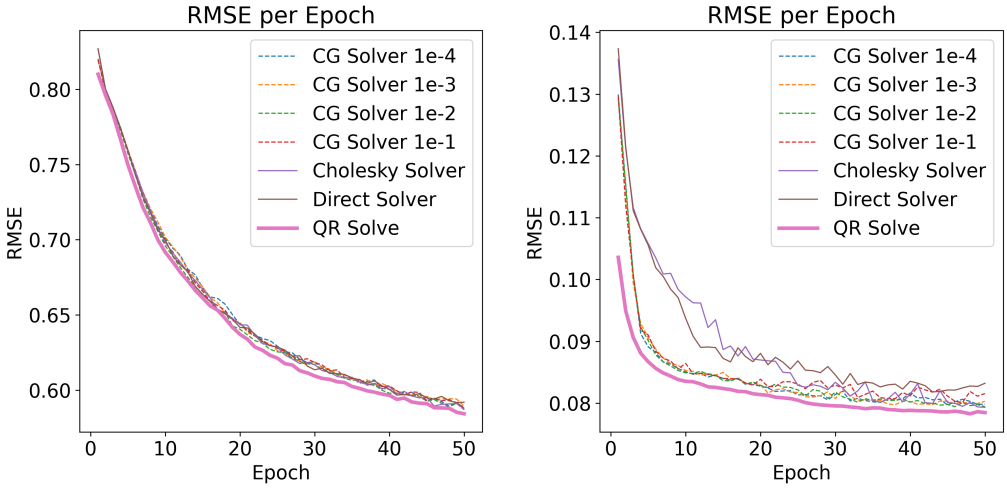
B.1 Effect of Hutchinson's Pseudoloss

In Section 4.2, we advocated for the use of Hutchinson's pseudoloss to overcome numerical stability issues that arise when calculating the exact MLL. This adjustment is specifically to address situations where the matrix $\mathbf{K_{zz}}$ is not positive semi-definite (PSD) in `float32` precision. In other approximate GP model using `gpytorch`, this challenge is typically met by performing a Cholesky decomposition followed by an efficient low-rank computation of the log determinant through the matrix determinant lemma. In our experience even when using versions of the Cholesky decomposition that add additional jitter along the diagonal there are still situations where $\mathbf{K_{zz}}$ can be poorly conditioned, particularity when n is large. We conjecture that this behavior originates from situations arising where the learned interpolation points of a SoftKI coincide at similar positions driving the effective rank of $\mathbf{K_{zz}}$ down. In these situations, Hutchinson's pseudoloss offers a more stable alternative because it does not directly rely on the matrix being invertible.

Figure 4 compares the results obtained with the SoftKI MLL and Hutchinson's pseudoloss for varying numbers of probes on the Ricker wavelet surface. In Table 5 we replicate the `UCI` experiment of Section 5.1 using our stabilized MLL $\log \hat{p}(\mathbf{y} \mid \mathbf{x}; \theta)$ and Hutchinson's pseudoloss $\log \tilde{p}(\mathbf{y} \mid \mathbf{x}; \theta)$. For most situations, the stabilized MLL and Hutchinson's pseudoloss give similar results, indicating that the MLL is largely used. The results differ on some datasets that produce noisy $\mathbf{K}_{**}^{\text{SoftKI}} + \mathbf{\Lambda}$ which can cause the Cholesky decomposition to fail.

B.2 Alternative Methods for Posterior Inference

Section 4.3 details the adaptation of the QR stabilized linear solve for approximate kernels (Foster et al., 2009) to the SoftKI setting. In this section, we provide empirical evidence illustrating how other linear



(a) Protein.

(b) Keggdirected.

Figure 5: Test RMSE of SoftKI models trained using different linear solvers on the `keggundirected` dataset (*left*) and the `protein` dataset (*right*).

Method	Learning rate (η)	Kernel	Starting noise (β)	Inducing points (m)	Test RMSE
SGPR	0.1	Matern $\nu = 1.5$	0	128 K-means centroids	6×10^{-3}
SKI	0.1	Matern $\nu = 1.5$	0.1	20×20 grid	6×10^{-3}
SoftKI	0.5	Matern $\nu = 1.5$	0.5	128 K-means centroids	2×10^{-3}

solvers fail when confronted with datasets that generate noisy kernels. We focus on two datasets, `protein` and `keggundirected`, which have previously caused instability in Hutchinson's pseudoloss computation.

As a reminder the QR stabilized linear solve is motivated by the challenge of solving the linear system:

$$\hat{\mathbf{C}}\,\boldsymbol{\alpha} = \hat{\mathbf{K}}_{\mathbf{zx}}\,\boldsymbol{\Lambda}^{-1}\,\mathbf{y}, \tag{30}$$

where $\hat{\mathbf{C}}$ is the estimated (potentially noisy) kernel matrix $\hat{\mathbf{C}} = \mathbf{K}_{\mathrm{zz}} + \hat{\mathbf{K}}_{\mathbf{zx}}\boldsymbol{\Lambda}^{-1}\hat{\mathbf{K}}_{\mathrm{xz}}$, and $\hat{\mathbf{K}}_{\mathrm{xz}} = \boldsymbol{\Sigma}_{\mathrm{xz}}\mathbf{K}_{\mathrm{zz}}$ is the interpolated kernel between interpolations points $\mathbf{z}$ and training points $\mathbf{x}$.

In this example we evaluate a direct solver, the Cholesky decomposition method, and CG with convergence tolerances set to 1×10^{-1}, 1×10^{-2}, 1×10^{-3}, and 1×10^{-4}. Despite adjusting the tolerances for the CG method, our experiments revealed that all solvers—except for the QR-stabilized routine—resulted in training instability. Figure 5 depicts the test RMSE performance across the different solvers. The direct and Cholesky solvers, while more stable than the CG solvers, failed to produce reliable solutions. Similarly, the CG method did not converge to acceptable solutions within the tested tolerance levels. In contrast, the QR-stabilized solver consistently produced stable and accurate solutions justifying its choice as a correctional measure that stabilizes a SoftKI on difficult problems.

C Additional Experiments

We provide supplemental data for experiments in the main text (Section C.2). We also report additional comparisons with SKI-based methods (Section C.3). Finally, we provide additional data regarding SoftKI's interpolation points (Section C.4).

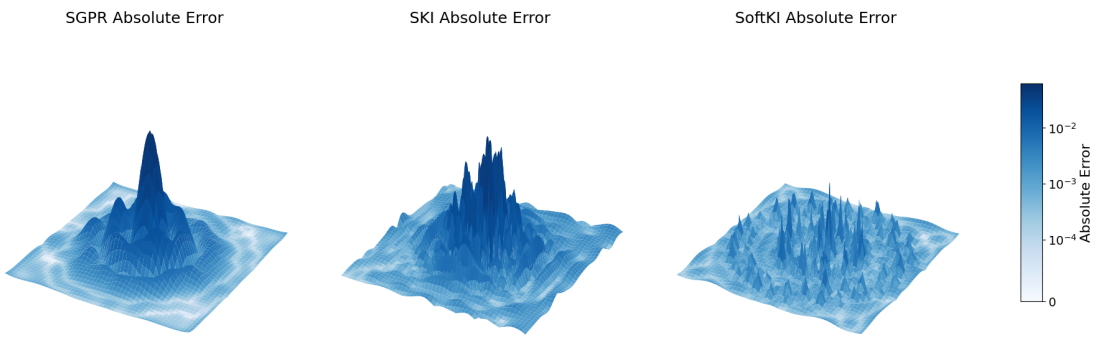


Figure 6: **(Absolute Error Surfaces):** Comparison of the absolute error on the training set domain from Figure 2 visualized as surfaces.

dataset	n	d	SoftKI	SGPR	SVGP
3droad	391386	3	**5.586 ± 0.12**	-	8.529 ± 0.216
Kin40k	36000	8	0.453 ± 0.012	**0.015 ± 0.002**	0.823 ± 0.01
Protein	41157	9	0.52 ± 0.006	**0.015 ± 0.003**	0.904 ± 0.02
Houseelectric	1844352	11	**26.066 ± 1.335**	-	40.33 ± 1.182
Bike	15641	17	0.227 ± 0.011	**0.013 ± 0.002**	0.367 ± 0.006
Elevators	14939	18	0.178 ± 0.004	**0.013 ± 0.002**	0.352 ± 0.046
Keggdirected	43944	20	0.554 ± 0.004	**0.016 ± 0.0**	0.95 ± 0.011
Pol	13500	26	0.171 ± 0.01	**0.013 ± 0.002**	0.325 ± 0.017
Keggundirected	57247	27	0.722 ± 0.043	**0.018 ± 0.0**	1.217 ± 0.047
Buzz	524925	77	**7.107 ± 0.053**	-	10.974 ± 0.131
Song	270000	90	**3.047 ± 0.055**	-	5.851 ± 0.185
Slice	48150	385	1.363 ± 0.004	0.019 ± 0.002	**1.069 ± 0.014**

Table 6: Timing per epoch of hyperparameter optimization in seconds.

C.1 Ricker Wavelet Experiment

In Section 4.2 we present an illustrative comparison of SGPR, SKI and SoftKI on the task of learning the Ricker wavelet. Figure 6 visualizes the absolute error given as surfaces. Each model was trained for 100 epochs using the `Adam` optimizer together with a `StepLR` learning-rate scheduler. The dataset is comprised of $3,000$ training points and 200 test points. We obtain the initial inducing and interpolation locations for both SGPR and SoftKI by clustering the training inputs with `KMeans`.

C.2 Supplemental to Experiments

Hardware details. We run all experiments on a single Nvidia RTX 3090 GPU which has 24Gb of VRAM. A GPU with more VRAM can support larger datasets. Our machine uses an Intel i9-10900X CPU at 3.70GHz with 10 cores. This primarily affects the timing of SoftKI and SVGP which use batched hyperparameter optimization, and thus, move data on and off the GPU more frequently than SGPR.

Timing. Table 6 compares the average training time per epoch in seconds to for hyperparameter optimization of SoftKI vs SVGP. We do not include SGPR since it is much faster but does not support batched stochastic optimization, and thus, is limited to smaller datasets. We observe that SVGP and SoftKI have similar performance characteristics due to the mini-batch gradient descent approach that both SVGP and SoftKI employ. SVGP requires slightly more compute since it additionally learns the parameters of a variational Gaussian distribution.

	dataset	n	d	Exact	SoftKI	Skip	Simplex-SKI
RMSE	Protein	41157	9	0.511 ± 0.009	0.652 ± 0.012	0.817 ± 0.012	**0.571 ± 0.003**
	Houseelectric	1844352	11	0.054 ± 0.000	**0.052 ± 0.0**	-	0.079 ± 0.002
	Elevators	14939	18	0.399 ± 0.011	**0.423 ± 0.011**	0.447 ± 0.037	0.510 ± 0.018
	Keggdirected	43944	20	0.083 ± 0.001	**0.089 ± 0.001**	0.487 ± 0.005	0.095 ± 0.002
NLL	Protein	41157	9	0.960 ± 0.003	**0.992 ± 0.017**	1.213 ± 0.020	1.406 ± 0.048
	Houseelectric	1844352	11	0.207 ± 0.001	**0.251 ± 0.005**	-	0.756 ± 0.075
	Elevators	14939	18	0.626 ± 0.043	**0.372 ± 0.004**	0.869 ± 0.074	1.600 ± 0.020
	Keggdirected	43944	20	0.838 ± 0.031	**0.254 ± 0.156**	0.996 ± 0.013	0.797 ± 0.031

Table 7: Comparison with SKI-based methods. Numbers for Exact, Skip and Simplex-SKI are taken from (Kapoor et al., 2021). Best approximate GP numbers are bolded.

C.3 Comparison with SKI-based Methods

As we mentioned in the main text, it is not possible to apply SKI to a majority of the datasets in our experiments due to dimensionality scaling issues. Nevertheless, there are some datasets with smaller dimensionality that SKI-variants such as SKIP (Gardner et al., 2018) and Simplex-SKI (Kapoor et al., 2021) can be applied to.

To compare against these methods, we attempt to replicate the experimental settings reported by (Kapoor et al., 2021) so that we can directly compare against their reported numbers. The reason for doing so is because Simplex-SKI relies on custom Cuda kernels for an efficient implementation. Consequently, it is difficult to replicate on more current hardware and software stack as both the underlying GPU architectures and PyTorch have evolved. From this perspective, we view the higher-level PyTorch and GPyTorch implementation of SoftKI as one practical strength since it can rely on these frameworks to abstract away these details.

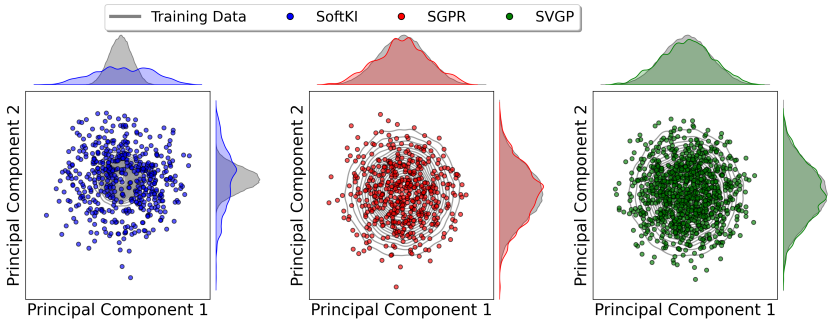
In the original Simplex-SKI experiments, the methods are tested on 4/9, 2/9, and 4/9 splits for training, validation, and testing respectively. For SoftKI, we simply train on 4/9 of the dataset, discard the validation set, and test on the rest. We report the test RMSE and test NLL achieved in Table 7. We find that SoftKI is competitive with Simplex-SKI.

C.4 Additional Experiments on Interpolation Points

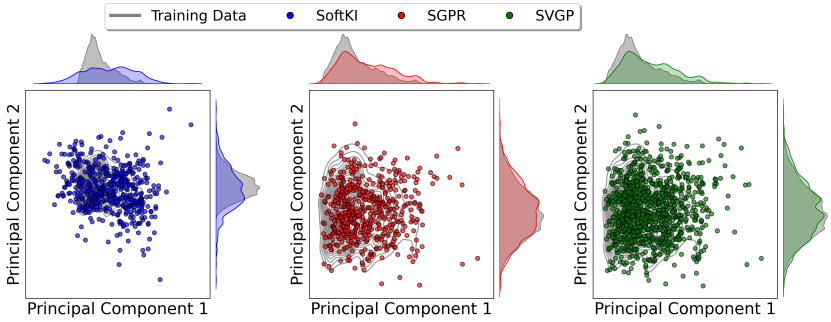
Figure 7 gives additional results on the PCA analysis for the `UCI` dataset. We observe the general trend that the interpolation points learned by SoftKI tend to be more spread out compared to the inducing points learned by SGRP and SVGP. This highlights that the structure interpolation points and inducing points are different, although they both have their start in a Nyström approximation.

Figure 8 visualizes the projection of the interpolation/inducing points via UMAP (McInnes et al., 2020) learned by each GP method on selected `UCI` datasets. As a reminder, UMAP attempts to preserve the distances between the points in the higher-dimensional manifold in the 2D plane. We make a couple of observations. First, the interpolation points learned by SoftKI are more spread out compared to the inducing points learned by SGPR and SVGP. Second, the patterns in the interpolation points learned by SoftKI confirm our intuition that high-dimensional spaces can benefit from fewer interpolation points. If a full lattice were required, then there would be less discernible structure in the projection.

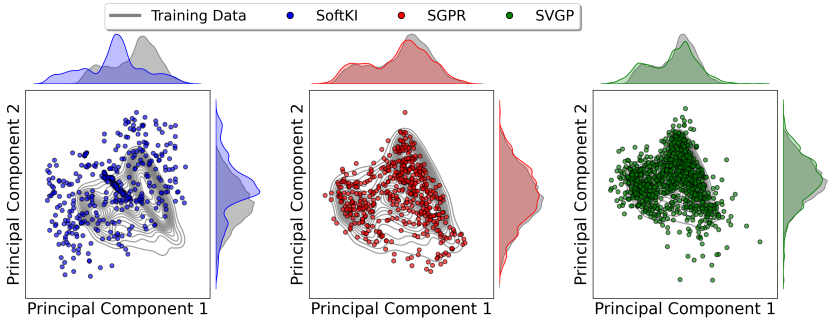
Figure 9 illustrates the effect of the number of interpolation points on the test RMSE of SoftKI. We find that increasing the number of interpolation points is helpful for improving test RMSE performance on some datasets (*e.g.*, `kin40k` and `pol`) while it is detrimental for other datasets (*e.g.*, `elevators` and `song`). In general, we expect that increasing the number of interpolation points should improve the performance of SoftKI. Thus, it is somewhat surprising that increasing the number of interpolation points did not help performance on some datasets. As a reminder, datasets such as `elevators` and `song` are challenging in general for GP models including SoftKI. Thus, we interpret the decrease in performance of SoftKI as



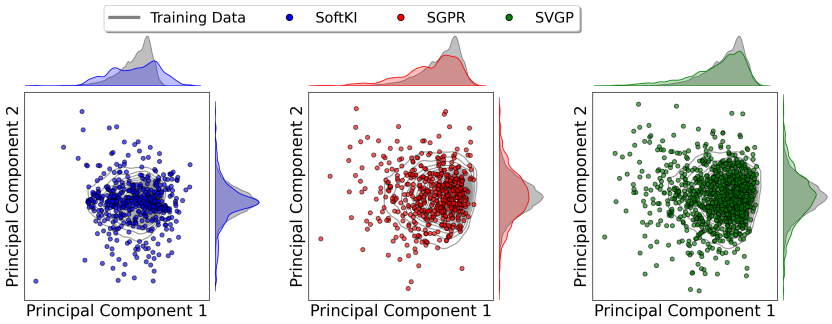
(a) `kin40k`



(b) `protein`

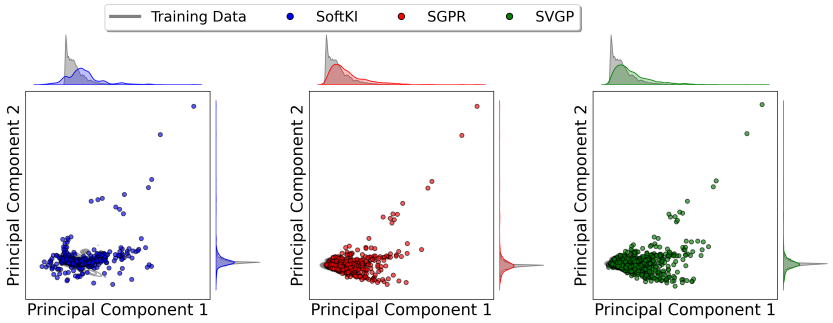


(c) `bike`

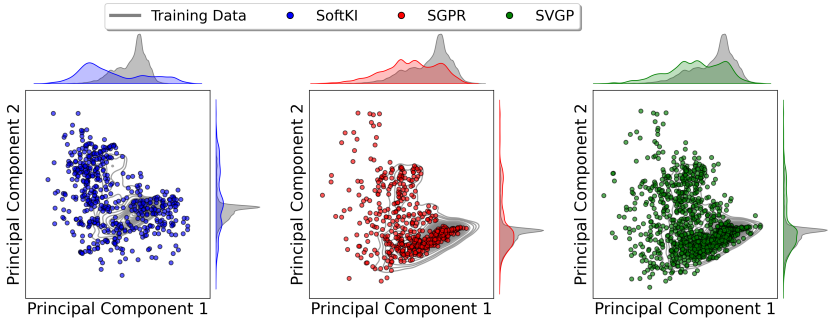


(d) `elevators`

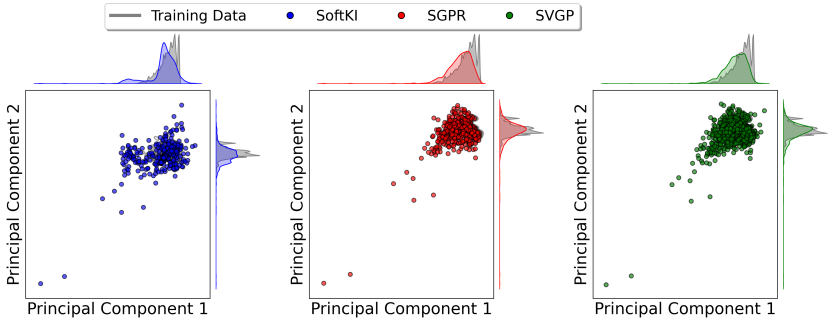
overfitting, especially since SoftKI explains less of the data as noise compared to another interpolation point method such as SVGP.



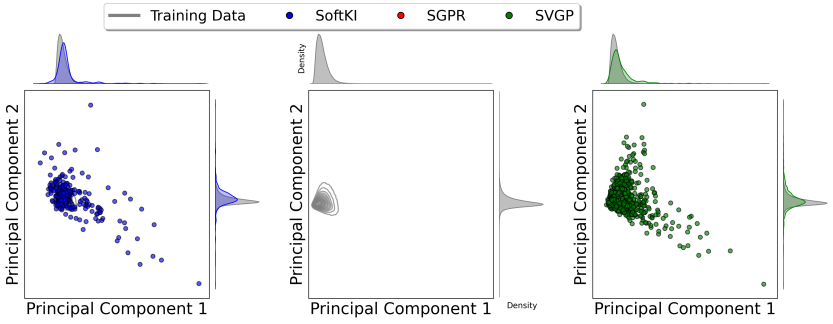
(e) `keggdirected`



(f) `pol`



(g) `keggundirected`

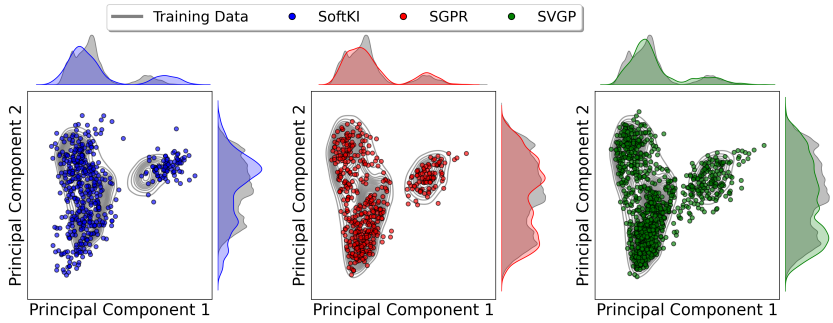


(h) `song`

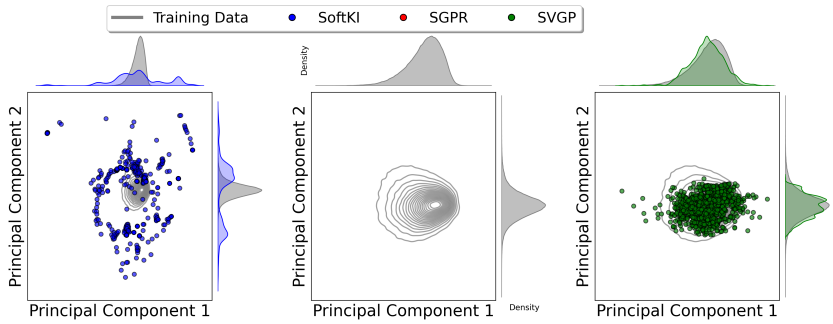
C.5 Lengthscale and Temperature

Figure 10 gives the histograms of the lengthscales learned by various methods in the experimention in Section 5.1. Figure 11 gives the histogram of the temperatures learned by SoftKI. On datasets such as

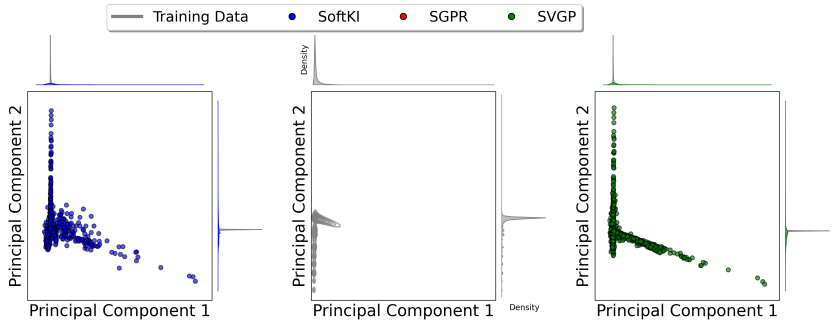
`song` and `slice`, we see that the majority of SoftKI's lengthscales for each dimension reach the maximum lengthscale of 5. The dimensionality-specific variation is instead captured in the temperature.



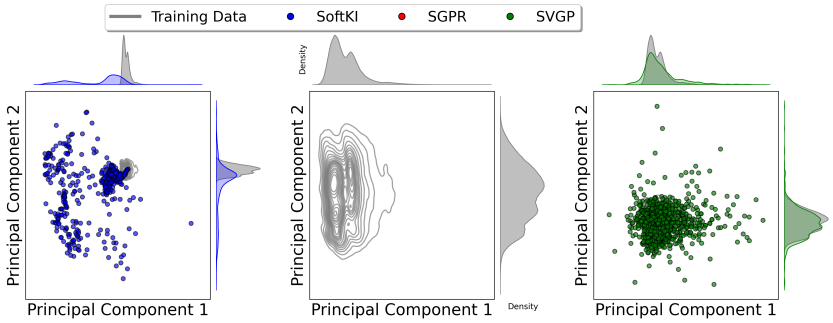
(i) `slice`



(j) `3droad`

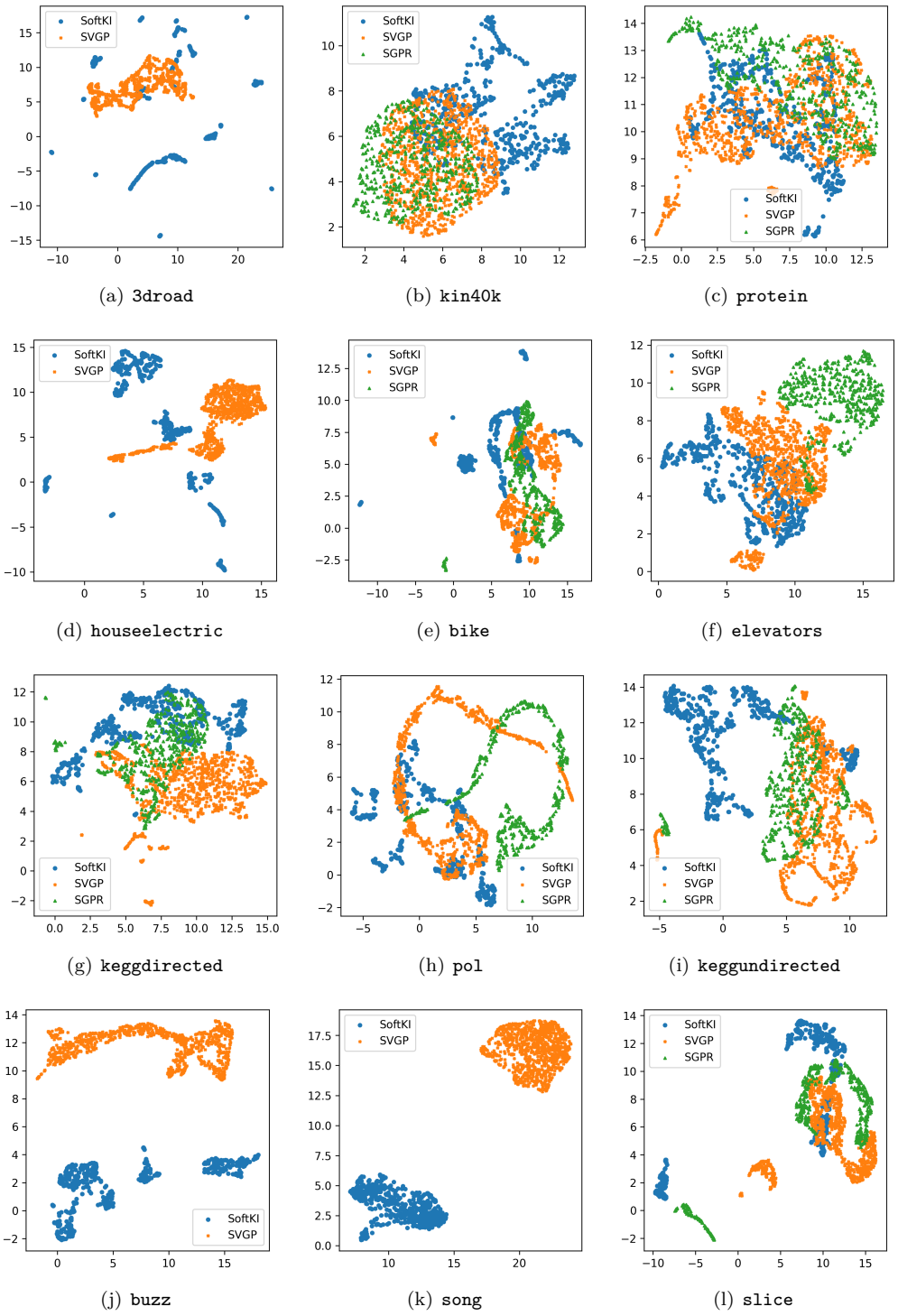


(k) `buzz`



(l) `houseelectric`

Figure 7: Interpolation and inducing points learned by each method as projected by PCA.



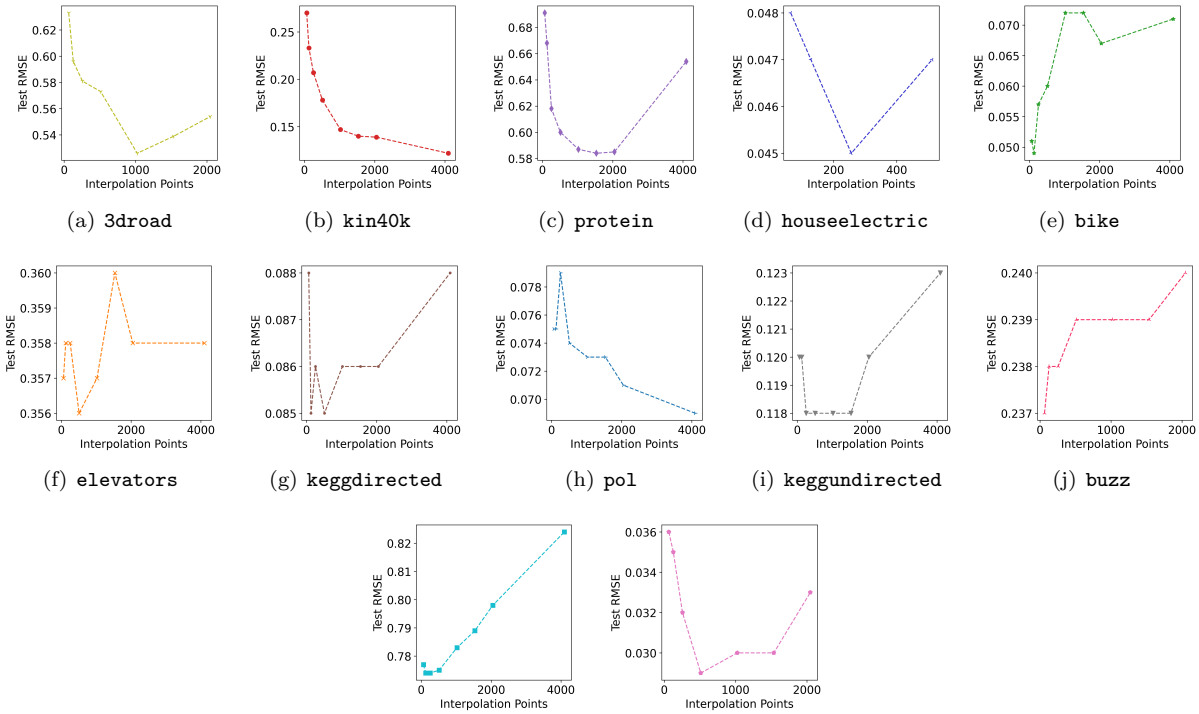
(a) `3droad` (b) `kin40k` (c) `protein`

(d) `houseelectric` (e) `bike` (f) `elevators`

(g) `keggdirected` (h) `pol` (i) `keggundirected`

(j) `buzz` (k) `song` (l) `slice`

Figure 8: Interpolation and inducing points learned by each method as projected by UMAP.

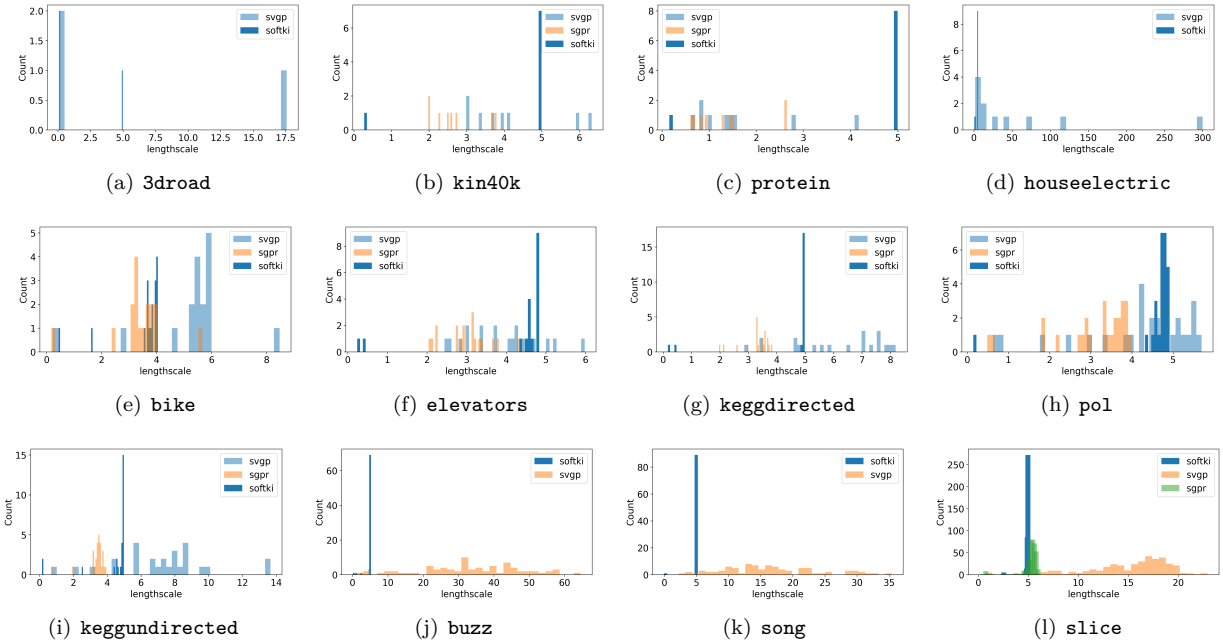


(a) `3droad` (b) `kin40k` (c) `protein` (d) `houseelectric` (e) `bike`

(f) `elevators` (g) `keggdirected` (h) `pol` (i) `keggundirected` (j) `buzz`

(k) `song` (l) `slice`

Figure 9: Test RMSE performance as a function of the number of interpolation points.

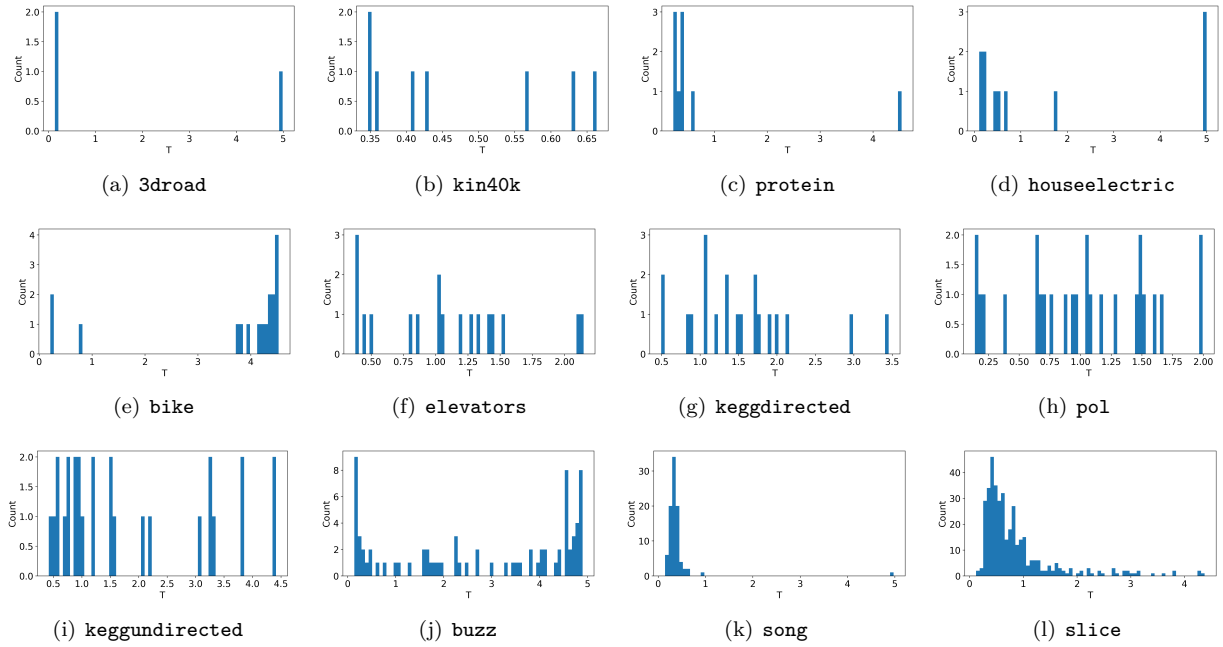


(a) `3droad` (b) `kin40k` (c) `protein` (d) `houseelectric`

(e) `bike` (f) `elevators` (g) `keggdirected` (h) `pol`

(i) `keggundirected` (j) `buzz` (k) `song` (l) `slice`

Figure 10: Lengthscales.



(a) `3droad` (b) `kin40k` (c) `protein` (d) `houseelectric`

(e) `bike` (f) `elevators` (g) `keggdirected` (h) `pol`

(i) `keggundirected` (j) `buzz` (k) `song` (l) `slice`

Figure 11: Temperatures.