Scalable and Memory-Efficient
Kernel Ridge Regression

Gustavo Chávez, Yang Liu, Pieter Ghysels, Xiaoye Sherry Li
Computational Research Division
Lawrence Berkeley National Laboratory
Berkeley, USA
{gichavez,liuyangzhuan,pghysels,xsli}@lbl.gov

Elizaveta Rebrova
Department of Mathematics
University of California, Los Angeles
Los Angeles, USA
rebrova@math.ucla.edu

Abstract—We present a scalable and memory-efficient framework for kernel ridge regression. We exploit the inherent rank deficiency of the kernel ridge regression matrix by constructing an approximation that relies on a hierarchy of low-rank factorizations of tunable accuracy, rather than leverage scores or other subsampling techniques. Without ever decompressing the kernel matrix approximation, we propose factorization and solve methods to compute the weight(s) for a given set of training and test data. We show that our method performs an optimal number of operations $O(r^2 n)$ with respect to the number of training samples $r$ due to the underlying numerical low-rank $(r)$ structure of the kernel matrix. Furthermore, each algorithm is also presented in the context of a massively parallel computer system, exploiting two levels of concurrency that take into account both shared-memory and distributed-memory inter-node parallelism. In addition, we present a variety of experiments using popular datasets – small, and large – to show that our approach achieves much-improved performance and memory footprint.

I. INTRODUCTION

Classical ridge regression is designed to find the linear hyperplane that approximates the data labels well, and at the same time does not have too large coefficients, namely

$$\arg\min_w \sum_{i=1}^{n} (y_i - x_i^T w)^2 + \lambda \|X^T w\|_2^2,$$

where $x_i$ are data points (rows of the $n \times d$ data matrix $X$), $y_i$’s are their labels, $y = (y_1, \ldots, y_n)$, $w \in \mathbb{R}^n$ is the normal vector to the target hyperplane, and $\lambda > 0$ is a hyperparameter of the method. It can be proved (see, for example, [17]) that the optimal $w$ is given by

$$\hat{w} = (XX^T + \lambda I)^{-1} y.$$  \hspace{1cm} (1)

The kernel trick idea is to implicitly embed data points $x_1, \ldots, x_n \in \mathbb{R}^d$ into a higher dimensional space, and summarize the information about this space by an $n \times n$ kernel matrix $K$, which replaces the matrix $XX^T$. This effectively substitutes the scalar product $x_i^T x_j$ by the element $K(x_i, x_j) = K_{ij}$, that represents the scalar product in some higher dimensional space.

Kernel methods turn out to be extremely efficient non-parametric methods across a variety of supervised learning problems and applications [19]. Indeed, with the kernel map, one can approximate any function or decision boundary arbitrarily well, given enough training data. In this paper, we focus on the use of kernel ridge regression (KRR) to solve classification problems. Algorithm 1 shows the steps needed for two-class classification.

Algorithm 1 Kernel ridge regression

**Input:** $X - n \times d$ train data matrix;  
$X' - m \times d$ test data matrix;  
y $\in \{\pm 1\}^n$ - train labels  

**Output:** $y' \in \{\pm 1\}^m$ - predicted test labels

1. Compute kernel matrix on the train data $K_{ij} = \mathcal{K}(x_i, x_j)$ where data points $x_i$ are the rows of $X$, $i = 1, \ldots, n$.
2. Compute weight vector $w$ by solving the linear system $w = (K + \lambda I)^{-1} y$
3. For each test data sample $x'_i \in X'$, compute kernel vector w.r.t. the train data $K'(i) = (\mathcal{K}(x_1, x'_i), \ldots, \mathcal{K}(x_n, x'_i))^T$
4. For each $x'_i \in X'$, predict its class label as $y'_i = \text{sign}(w^T K'(i))$

The memory and time required for the exact execution of Algorithm 1 are $O(n^2)$ and $O(n^3)$, respectively, with $n$ data points. This is due to the storage and inversion of the $n \times n$ dense positive definite matrix $A = K + \lambda I$. Throughout the text, $I$ denotes identity matrix of the proper size. This is prohibitively expensive when the dataset is large. On the other hand, since the classification is fully determined by only the sign of the scalar product between $w$ and the kernel vector (see Step 4 of the algorithm), the $w$ vector does not need to be computed with high accuracy. Therefore, there are ample opportunities to use approximation algorithms to compute $w$ in Step 2 of Algorithm 1.

There is a rich literature concerned with the acceleration of kernel methods, usually based on the efficient approximation of the kernel map. The most popular approach is to construct a low-rank matrix approximation of the kernel matrix. This includes Nyström-type methods [30], [15], [20], random feature maps (to approximate the kernel function directly [24] or
as a preconditioner [5], and hybrid methods like FALKON [27], where the Nyström method is combined with a good preconditioner to be used in conjugate gradients.

However, the numerical rank of the kernel matrix depends on parameters, which are, in turn, data-dependent. For example, the most popular Gaussian kernel matrix

\[ K_{ij} = K^\theta(X_i, X_j), \quad X_1, \ldots, X_n = \text{training data} \]

\[ K^\theta(x, y) = \exp\left(-\frac{1}{2} \frac{||x - y||^2}{h^2}\right), \quad (2) \]

is approximately low-rank only if the value of the hyperparameter \( h > 0 \) (radius) is sufficiently large, which might not be the best choice of \( h \) (see also the discussion in [29]).

Other popular kernels with similar properties include the Laplacian and ANOVA kernels defined as follows:

\[ K^l(x, y) = \exp\left(-\frac{1}{2} \frac{||x - y||_1}{h}\right), \quad (3) \]

\[ K^\alpha(x, y) = \sum_{1 \leq k_1 < \ldots < k_p \leq d} K^\alpha(x_{k_1}, y_{k_1}) \cdots K^\alpha(x_{k_p}, y_{k_p}). \quad (4) \]

Here \( p \) denotes the degree of the ANOVA kernel and \( x_k \) represents the \( k \)-th component of vector \( x \). Throughout this paper, the superscript of \( K \) is dropped without confusing confusion.

Some methods were proposed to overcome the fact that \( K \) is not necessarily approximately low-rank. For example, one can start with the initial splitting of the data into classes, so that between-classes interactions are represented by either sparse or low-rank parts of the kernel matrix (the examples include Memory Efficient Kernel Representation [28], Block Basis Factorization [29] and \( k \)-means kernel ridge regression [34]).

This inspires the main idea of the current work, namely, to approach the problem of kernel matrix approximation with the methods created for hierarchical (\( H \) [16]) matrix or hierarchical semi-separable matrix (HSS) representations [6]. This does not require \( K \) to be low-rank, but only some off-diagonal parts to be rank-deficient, at least, after some suitable preprocessing. Moreover, one has a freedom to choose this preprocessing (reordering of the rows and columns of \( K \)) in the best possible way. One combination of \( H \) and HSS matrix formats for the approximation of \( K \) (or \( A \)) was proposed in [25], using the STRUMPACK library for HSS-compression, factorization and solve, see more details below in Section II-A. The authors also compare various ways to perform preprocessing, and the one based on recursive 2-means clustering of the data points is claimed to ensure a better representation of \( K \) in the hierarchical format.

Another predecessor of the current work is the \( O(dn \log n) \) algorithm ASKIT, which uses a block-diagonal-plus-low-rank hierarchical matrix format to construct an approximate representation for the kernel matrix [21], and later the \( O(n \log n) \) algorithm INV-ASKIT to perform a factorization of the approximate matrix [15, 17] (to be used as a direct linear solver). The authors also propose a geometric (neighbor-based) way to find basis columns of the sub-blocks of \( K \) (needed for the hierarchical compression), see also [35] and below in Section II-B.

In the current work we improve the compression process proposed in [25] by a delicate use of the geometrical structure of kernel matrices (a variation of the approach proposed in [35]). Our new ideas are summarized below:

- We present a scalable way to approximate kernel matrices with optimal time and memory complexity, which is achieved by clustering and neighboring-based preprocessing techniques and the nested bases of the HSS format.
- We applied the above algorithm to classification problems using Kernel Ridge Regression. Our method achieves a similar classification error as the exact kernel ridge regression, and is much faster and more scalable than the \( O(n^3) \) exact Kernel Ridge Regression algorithm.
- We developed an efficient hyperparameter tuning method, taking advantage of the fact that recompression is not needed when tuning the regularization parameter (see Algorithm 5 and Section II-D).
- We developed a Python interface to scikit-learn classifiers and regressors, enabling faster time and distributed memory parallelism (in contrast to the shared-memory parallelism) for scikit-learn users (see Section III-E).

The rest of the paper is organized as follows: Section II-A presents a brief review of the data-sparse hierarchically semi-separable (HSS) matrix representation and the traditional approaches to construct an HSS representation of a square matrix \( A \in \mathbb{R}^{n \times n} \). Section II-B shows a faster sampling method exploiting the geometry of the data that defines the kernel matrix, where the samples are computed via approximate nearest neighbors. In Section II-C we present a new way to construct HSS matrices efficiently, using the neighboring points as the samples. In Section II-D we discuss the crucial ideas in the hyperparameter tuning process. Section III gives the experimental support for the proposed framework.

II. METHODS

Section II-A describes the HSS rank-structured matrix format and briefly discusses randomized HSS construction. In Section II-B we describe an \( O(n) \) algorithm to find approximate nearest neighbors of the data points, which is based on the construction of multiple randomized projection trees. Then in Section II-C we present an efficient HSS construction algorithm that relies on this nearest neighbor information, and which avoids the expensive random sampling phase.

A. Hierarchically Semi-Separable matrix representation

The rank-structured HSS matrix representation uses a hierarchical block \( 2 \times 2 \) partitioning of the matrix, where all off-diagonal blocks are compressed, or approximated, using a low-rank product, see Figure 1. Every node \( i \) in the HSS tree, illustrated in Figure 2, has a corresponding index set \( I_i \subset \{1, \ldots, n\} \). At the last level of the recursion, the diagonal blocks, i.e., \( A(I_i, I_i) = A_{ii} \) are stored as (small) dense matrices \( A_{ii} = D_i \). All off-diagonal blocks \( A_{ij} \) are compressed using a
low-rank factorization $A_{ij} ≈ U_i B_{ij} V_j^T$. Moreover, the column basis matrix $U_i$ for a node $i$ with children $c_1$ and $c_2$ in the hierarchy is defined as $U_i = [U_{c_1} \; 0 \; U_{c_2}] U_{i_1}$, and hence only the smaller matrix $U_{i_1}$ is stored at node $i$. Only at the leaf nodes, where $U_i ≡ U_{i_1}$, are the $U_i$ stored explicitly. A similar relation holds for the $V_i$ basis matrices, and is referred to as the nested basis property. For symmetric matrices, $U_i ≡ V_i$ and $B_{ij} ≡ B_{ji}$.

We briefly recall two distinct approaches for the construction of HSS matrices. The most straightforward technique, see [31], is to apply low-rank compression to row blocks $A(I_i, I_{\text{root}} \setminus I_i)$ and column blocks $A(I_{\text{root}} \setminus I_i, I_i)$ of $A$ directly, using either truncated singular value decompositions or the cheaper rank-revealing QR. The problem with this approach is the complexity, and the fact that the entire matrix $A$ needs to be explicitly formed. A second approach to constructing the HSS representation is through randomized sampling (also referred to as random projection), as introduced in [22]. The HSS representation is constructed from random samples $S = A \Omega$ with a tall and skinny random matrix $\Omega$. The goal of the random sampling is to reduce the problem to the much smaller sample matrix $S$, which is then used to approximate the HSS basis matrices $U_i$ and $V_i$ using the interpolative decomposition. The dense diagonal blocks $A_{ii}$ and the transfer matrices $B_{ij}$ are submatrices of $A$ and can be computed directly from $A$. Hence this approach is called a partially matrix-free method, since it requires a matrix times (multiple) vector product ($A\Omega$) as well as access to individual elements of $A$. When a fast matrix-vector product is available, this randomized construction algorithm has $O(rn)$ complexity, with $r$ the maximum HSS rank, i.e., the maximum rank over all off-diagonal blocks in the HSS hierarchy. The randomized sampling construction algorithm is implemented in the STRUMPACK library [13], and was used in previous published results for kernel matrix compression [25] and other applications [26]. In [25], the random sampling was performed with an $H$-matrix approximation of the kernel.

After construction, the HSS matrix can be factorized into a $ULV$ form [7], where $L$ is lower triangular and $U$ and $V$ are orthogonal. This factored form can be used to solve the linear system.

B. Approximate nearest neighbor search for faster sampling

The role of random projection $S = A\Omega$ is to approximate column bases (ranges) of sub-matrices of the matrix $A$. Using conventional matrix-matrix multiplication to compute $S$, the HSS construction costs $O(n^2 r)$ (for an $n \times n$ matrix of HSS rank $r$). Instead, we use a more sophisticated yet cheaper column sampling, based on a modified neighbor sampling procedure [21], [32]. The general idea is as follows. For the kernel matrix $A$ (formed based on data points with a kernel function that decays with distance, or, more generally, dissimilarity) we can use the distance (similarity) between the points to identify the dominating entries of the kernel matrix. Then, picking the columns indexed by these dominating nearest neighbor points, we expect to get a reasonable approximation to the column basis of a certain sub-matrix of $A$.

In the case of a Gaussian kernel [2], the largest entries of $A$ correspond to the nearest neighbors of the data points in Euclidean distance. So, the first step would be to find $\nu_{ann}$ nearest neighbors for each data point $X_i$. Finding exact nearest neighbors involves computing pairwise distances between all points and takes $O(n^2 d)$ operations. In order to get (log-)linear complexity for the whole process, we approximate neighbors of the data points instead. Following the approach from [21], we find approximate nearest neighbors (ANN) based on random projection trees. Random projection trees were proposed as a more robust analogue of kd-trees in [10] (like in kd-trees, the direction of the median split is chosen randomly, but the direction is not necessarily one of the coordinate dimensions, see [10] Section 2.3 for more details).

The ANN algorithm (Algorithm 2) operates on a collection $\mathcal{X} = \{X_1, \ldots, X_n\}$ of data points and returns the indices of closest $\nu_{ann}$ neighbors for each point as an $n \times \nu_{ann}$ matrix $\mathcal{N}$. In addition, the corresponding distances for each neighbor (called scores) is computed as $S$.

To begin with, the data points are partitioned into the leaves of the random projection tree as $\{L_1, \ldots, L_k\} = \text{ConstructRPT}(\mathcal{X}, 6\nu_{ann})$ where each leaf $L_i$ has approximately size $6\nu_{ann}$ (line 4). For all members of each leaf the exact nearest neighbors are found within this same leaf (line 5 [1]). Note that this reduces the complexity of the exact neighbor search significantly, given that the leaves are not too big.

Clearly, this selection misses all exact neighbors of the data points that end up in different leaves. So, the process is repeated iteratively: new random projection trees are constructed several times, and only the $\nu_{ann}$ best ones out of the union of $\nu_{ann}$ previously found neighbours and $\nu_{ann}$ currently found neighbours are kept as $\mathcal{N}$ (line 12-15), see also the illustration in Figure 3. We define these best neighbors as those with the shortest distance/score to the respective data points (those responsible for adding that specific neighbor point to the collection). We keep these scores $S$ for the future use in the HSS compression (see the details in Section II-C).

To estimate the quality of the approximate neighbors, we find the exact neighbors for a small constant number ($s = 100$, see line 16) of data points, which can be done with complexity...
Algorithm 2: ConstructANN: find approximate nearest neighbors

Input: \( \nu_{\text{ann}} \) – number of approximate neighbors to search
\( \mathcal{X} = \{X_1, \ldots, X_n\} \) – data points, \( X_i \in \mathbb{R}^d \)

Output: \([\mathcal{N}, S]\) – \( n \times \nu_{\text{ann}} \) matrices with neighbors of each \( X_i \) and corresponding scores

1. \( q = 0 \)
2. \( \mathcal{N}, S = (0, 0) \)
3. while \( \text{iter} < 30 \) and \( q < 0.99 \) do
   4. \( \{L_1, \ldots, L_k\} = \text{ConstructRPT} (\mathcal{X}, 6\nu_{\text{ann}}) \)
   5. for \( i = 1 \) to \( k \) do
      6. construct distance matrix \( D(i,j) \) for \( i, j \in L_i \)
      7. for \( i \) in \( L_i \) do
         8. \( \mathcal{N}_{\text{cur}}(i,:) = \nu_{\text{ann}} \) elements in \( L_i \) with smallest \( D(i,:) \)
         9. \( S_{\text{cur}}(i,:) = D(i, \mathcal{N}_{\text{cur}}(i,:)) \)
      10. end for
   11. end for
   12. for \( i = 1 \) to \( n \) do
      13. \( \mathcal{N}(i,:) = \nu_{\text{ann}} \) elements in \( \mathcal{N}(i,:) \cup \mathcal{N}_{\text{cur}}(i,:) \) with smallest scores \( (S(i,:) \cup S_{\text{cur}}(i,:)) \)
      14. \( S(i,:) = \) respective scores of \( \mathcal{N}(i,:) \)
   15. end for
   16. \( \mathcal{X}_s = \{X_{i1}, \ldots, X_{is}\} \) random sample from \( \mathcal{X} \)
   17. for \( j = 1 \) to \( s \) do
      18. \( \mathcal{N}_{\text{exact}}(i,j,:) = \nu_{\text{ann}} L_2\)-closest data points to \( X_{ij} \in \mathcal{X}_s \)
      19. \( q = q + |\mathcal{N}(i,:) \cap \mathcal{N}_{\text{exact}}(i,j,:)|/\nu_{\text{ann}} \)
   20. end for
21. \( q = q/s \)
   22. end while

\( O(sn) \), and compare those with the list of approximate neighbors, see line 19. We iterate until 99% average quality \( q \) (line 21) is reached, or until 30 iterations, thus, taking \( O(n) \) operations, instead of \( O(n^2) \) for exact neighbor search for all datapoints.

Another feature of the ANN search is the possibility of parallelization at least in the construction of the distance matrices within leaves. For robustness, we choose the number of neighbors \( \nu_{\text{ann}} \) adaptively: if we observe that more basis columns are needed (that is, the quality of HSS compression was not good), we increase \( \nu_{\text{ann}} \). The adaptive ANN quantity results in a more robust approximation process and differentiates our approach from [21] and used for the classification task in [37]. The latter one just adds random columns in the situation when there are not enough neighbors.

Some theoretical analysis of the performance of random projection trees performance in finding approximate neighbors on the datasets with low intrinsic dimension is shown in [11]. There exists a variety of alternative approaches to find approximate nearest neighbors (e.g., see [2], [3] and references therein). It might be an interesting direction for the future work to compare various approaches for nearest neighbor approximation to be used in the column basis construction. However, our experiments on smaller datasets (when searching for the exact neighbors is not prohibitively expensive) show that we are able to achieve enough precision to be as effective as the approximate nearest neighbors, see Algorithm 2. Multiple random projection trees are constructed (the 2 recursive partitions at the top), and in every tree, the search for the \( \nu_{\text{ann}} \) nearest neighbors is restricted to the leaves, which are of size \( \sim 6\nu_{\text{ann}} \). For data point \( X_j \), the \( \nu_{\text{ann}} = 2 \) closest neighbors are found in different random trees. The result (bottom) after two iterations is found by merging the approximate neighbors of the two random projection trees, see lines 12 to 14 in Algorithm 2.

Fig. 3. Illustration of the approximate nearest neighbor search algorithm, see Algorithm 2. Multiple random projection trees are constructed (the 2 recursive partitions at the top), and in every tree, the search for the \( \nu_{\text{ann}} \) nearest neighbors is restricted to the leaves, which are of size \( \sim 6\nu_{\text{ann}} \). For data point \( X_j \), the \( \nu_{\text{ann}} = 2 \) closest neighbors are found in different random trees. The result (bottom) after two iterations is found by merging the approximate neighbors of the two random projection trees, see lines 12 to 14 in Algorithm 2.
the prediction purposes as we would be with exact neighbors.

C. HSS construction using approximate nearest neighbor information

We present a new approach, Algorithm 3, to construct an HSS matrix representation, relying on the geometry of the data that defines the matrix to be compressed, and reducing the cost compared to the earlier HSS construction approaches mentioned in Section II-A. This algorithm follows the same outline as the randomized algorithm from [22], but instead of constructing $S$ through random projection, $S$ is formed from columns of $K$ (lines 6 and 11 of Algorithm 3). Like the algorithm from [22] (excluding the random projection step), Algorithm 3 has a computational complexity of $O(r^2 n)$. The columns used are those corresponding to the nearest neighbors to all data points in the current node of the HSS tree (lines 5 and 10), as computed by Algorithm 2.

The algorithm traverses the tree in a bottom-up fashion. At the leaf level, it selects the important rows (called skeletons) for each leaf using columns corresponding to the nearest neighbors of each point in that leaf; at higher levels of the HSS tree, the skeletons are selected by essentially using the columns corresponding to the union of the nearest neighbors from the two children tree nodes which ensures the nested basis property of HSS.

The row skeletons and the basis matrices $U_r$ can be computed directly from $S_r$ (line 13) using an interpolative decomposition [9]: $[X, J] = ID(Y, \varepsilon)$, such that $Y = Y(:, J) X + O(\varepsilon)$, where $Y(:, J)$ is a subset of the columns of $Y$. However, when the interpolative decomposition picks almost all columns of $Y$ (line 13), or, more precisely when $|J| > d_{\max} - p$, we conclude that likely not enough columns of $K$ were used. In that case, the algorithm is restarted with more approximate nearest neighbors, see Algorithm 4. In [5], it is shown that using an HSS tree defined from a clustering algorithm applied to the input data can drastically reduce the ranks of the off-diagonal blocks. One can use for instance a recursive k-means clustering, with $k = 2$ to obtain a binary cluster tree, see line 1 in Algorithm 4. In Algorithm 3 only the near-field (neighboring) interactions are considered for the kernel approximation. For a more general approach, far-field interactions can be included through so-called proxy points [33]. However, from our experience we conclude that this is not required for good approximation of typical kernel matrices with a practical compression tolerance.

D. Training and hyperparameter tuning

Since the accuracy of kernel ridge regression heavily depends on its two hyperparameters $h$ and $\lambda$, we developed an autotuning training framework to search for an optimal setting of $h$ and $\lambda$, see Algorithm 5. Recall that when we construct the HSS format for the shifted kernel matrix $A = K + \lambda I$, we only compress the off-diagonal blocks, and store the diagonal explicitly in a set of (dense) diagonal blocks. When $h$ changes we need to re-compress, but when $\lambda$ changes we can avoid recompression, which is costly. Thus, our tuning strategy is to

<table>
<thead>
<tr>
<th>Algorithm 3 HSS compression: construction of an HSS matrix using approximate nearest neighbors</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> ${X_1, \ldots, X_n}$ – data points, $X_i \in \mathbb{R}^d$</td>
</tr>
<tr>
<td>$\mathcal{N}$, $\mathcal{S}$ – $n \times \nu_{ann}$ neighbors and scores for each $X_i$</td>
</tr>
<tr>
<td>$K_{ij} = K(X_i, X_j)$ – kernel matrix</td>
</tr>
<tr>
<td><strong>HSS cluster tree</strong></td>
</tr>
<tr>
<td>$p$ – oversampling parameter, $\varepsilon$ – compression tolerance</td>
</tr>
<tr>
<td><strong>Output:</strong> $\hat{K}$ – HSS of $K$, defined by $D_\tau$, $U_\tau$, $B_{\nu_1, \nu_2}$</td>
</tr>
<tr>
<td>1: for node $\tau$ in the HSS tree, in postorder do</td>
</tr>
<tr>
<td>2: if $\tau$ is a leaf then</td>
</tr>
<tr>
<td>3: $D_\tau = K(I_\tau, I_\tau)$</td>
</tr>
<tr>
<td>4: $d_{\max} =</td>
</tr>
<tr>
<td>5: $N_\tau = d_{\max}$ elements of $\bigcup_{i \in I_\tau} (\mathcal{N}(i, :)) \setminus I_\tau$ with smallest scores</td>
</tr>
<tr>
<td>6: $S_\tau = K(I_\tau, N_\tau)$ $\triangleright$ local sample matrix</td>
</tr>
<tr>
<td>7: else $\triangleright$ let $\nu_1$ and $\nu_2$ be two children of $\tau$</td>
</tr>
<tr>
<td>8: $B_{\nu_1, \nu_2} = B_{\nu_2, \nu_1} = K(I_{\nu_1}, I_{\nu_2})$</td>
</tr>
<tr>
<td>9: $d_{\max} =</td>
</tr>
<tr>
<td>10: $N_\tau = d_{\max}$ elements of $(N_{\nu_1} \cup N_{\nu_2}) \setminus (I_{\nu_1} \cup I_{\nu_2})$ with smallest scores</td>
</tr>
<tr>
<td>11: $S_\tau = \begin{bmatrix} K(I_{\nu_1}, N_{\tau}) \ K(I_{\nu_2}, N_{\tau}) \end{bmatrix}$</td>
</tr>
<tr>
<td>12: end if</td>
</tr>
<tr>
<td>13: $[U_\tau, J_\tau] = ID(S_\tau^T, \varepsilon)$</td>
</tr>
<tr>
<td>14: if ID failed to reach tolerance $\varepsilon$ then</td>
</tr>
<tr>
<td>15: Failed to compress HSS $\triangleright$ restart, see Alg. 4</td>
</tr>
<tr>
<td>16: end if</td>
</tr>
<tr>
<td>17: if $\tau$ is a leaf then</td>
</tr>
<tr>
<td>18: $\hat{I}<em>\tau = I</em>\tau(J_\tau)$</td>
</tr>
<tr>
<td>19: else</td>
</tr>
<tr>
<td>20: $\hat{I}_\tau = <a href="J_%5Ctau">I_{\nu_1}, I_{\nu_2}</a>$</td>
</tr>
<tr>
<td>21: end if</td>
</tr>
<tr>
<td>22: end for</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Algorithm 4 HSS compression with adaptive $\nu_{ann}$ selection</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> $\mathcal{X} = {X_1}_{i=1}^n$ – data points, $X_i \in \mathbb{R}^d$</td>
</tr>
<tr>
<td>$K_{ij} = K(X_i, X_j)$ – kernel matrix</td>
</tr>
<tr>
<td>$\nu_{ann}$ – number of approximate nearest neighbors</td>
</tr>
<tr>
<td>$p$ – oversampling parameter, $\varepsilon$ – compression tolerance</td>
</tr>
<tr>
<td><strong>Output:</strong> $\hat{K}$ – HSS of $K$, defined by $D_\tau$, $U_\tau$, $B_{\nu_1, \nu_2}$</td>
</tr>
<tr>
<td>1: construct HSS tree via 2-means, cobble, PCA, etc.</td>
</tr>
<tr>
<td>2: while not successfully compressed do</td>
</tr>
<tr>
<td>3: $[\mathcal{N}, \mathcal{S}] = \text{ConstructANN}(\mathcal{X}, \nu_{ann})$</td>
</tr>
<tr>
<td>4: $\hat{K} = \text{HSScompression}(\mathcal{X}, \mathcal{N}, \mathcal{S}, \mathcal{K}, \text{tree}, \varepsilon)$</td>
</tr>
<tr>
<td>5: $\nu_{ann} = 2 \cdot \nu_{ann}$</td>
</tr>
<tr>
<td>6: end while</td>
</tr>
</tbody>
</table>
search for multiple $\lambda$ values for each selected $h$ value. To tune $h$, we use an off-the-shelf black-box optimization package called OpenTuner \cite{openTuner}, which takes as inputs an objective function $\text{ComputeError}$ and a budget $n_h$ (number of function evaluations) and performs internal optimization by varying $h$. For different $h$ values, OpenTuner calls the objective function, which performs a matrix compression operation (line 2 of Algorithm 5) and tries multiple $\lambda$ values. Note that the $c$-error of the given hyperparameters is defined at line 11 of Algorithm 5. For each new $\lambda$, we update $\tilde{A}$ by adding it to $K$’s diagonal entries (line 4), and perform ULV factorization and solve for each new $\lambda$ (lines 5 and 6). These operations are much cheaper than compression.

In practice, this strategy saves a considerable amount of time during both the fit stage (compression, factorization, and solve) and in the prediction phase (line 9 of Algorithm 5) because we do not evaluate the weight vectors one at a time as in the classical algorithm (matrix-vector multiplication which is memory-bound). Instead, we fit a set of weights $W$ (matrix-matrix multiplication at line 9 which is compute-bound). This blocking of weights also heavily reuses the (expensive) kernel evaluation, which is flop intensive in the case of the kernel since it requires a transcendental function evaluation.

### III. Experiments

#### A. Efficiency of approximate nearest neighbor search

The quality and efficiency of the approximate nearest neighbor search, Algorithm 2, is first demonstrated via application of the algorithm with $\nu_{ann} = 128$ to subsets of the SUSY dataset of different sizes. As shown in Figure 4, ANN converges typically in less than 30 iterations to reach a quality of 99%. Moreover, the iteration count depends weakly on the problem size.

#### B. Robustness of kernel matrix approximation

To demonstrate the robustness of the HSS approximation with respect to the relative compression tolerance $\epsilon$, we randomly select $n = 10^3$ samples from the SUSY dataset and compute the HSS approximations for the Gaussian, Laplacian and ANOVA ($p = 2$) kernels with $h = 3$, $\lambda = 4.1$. Figure 5 shows that the relative error of the approximation satisfies $\|A - A\|_F/\|A\|_F < \epsilon$ using the proposed HSS-ANN algorithm. As described in Algorithm 4, the number of ANN required for any given tolerance is automatically chosen to meet this error bound. It is worth mentioning that there are no theoretical guarantees for the HSS approximation quality using either the proposed ANN algorithm or the more rigorous proxy point method \cite{proxy_points}. However, due to the exponential decay of many existing kernels, the HSS-ANN algorithm can achieve reasonably good accuracies as demonstrated by Figure 5. Moreover, our results in Section III-C show that the HSS approximation retains almost the same prediction accuracy as the exact algorithm.

Figure 6 shows the savings in memory (which translate to computational efficiency) using the Gaussian kernel while selecting a larger $\epsilon$ as compared to the exact (single-precision, dense representation) kernel ridge regression matrix.

#### C. Comparison with other methods

Comparisons with FALKON \cite{falkon} and INV-ASKIT \cite{invaskit} (more precisely, a similar implementation in the STRUMPACK
Algorithm 5: Modeling stage and hyperparameter tuning.

```
Input: \([h_{\text{min}}, h_{\text{max}}]\) – range of values for kernel parameter \(h\), \(n_h\) – number of \(h\) trials
\( \Lambda = [\lambda_1, \lambda_2, \ldots, \lambda_{n_\lambda}] \) – vector of regularization parameters
\( \mathcal{X}_{\text{train}}, \mathcal{X}_{\text{test}} \) – training and testing data sets

Output: \(h^*, \lambda^*\) – optimal hyperparameters after \((n_h \times n_\lambda)\) evaluations.

1: function \([c_{\text{error}}, \lambda^*] = \text{ComputeError}(h)\)  \(\triangleright\) see Algorithms 2 and 3
2: \( \hat{K} = \text{HSScompression}(K(\mathcal{X}_{\text{train}}, \mathcal{X}_{\text{train}}), h)\)
3: for \(j \in \{1, \ldots, n_\lambda\}\) do
4: \( \hat{\Lambda} = \hat{K} + \lambda_j I\)
5: \( [U, L, V] = \text{ULVfactorization}(\hat{A})\)
6: \( W(:,j) = (ULV)^{-1} y\)
7: end for
8: \( K' = K(\mathcal{X}_{\text{test}}, \mathcal{X}_{\text{train}})\)
9: \( Y = K' \cdot W\)
10: for \(j \in \{1, \ldots, n_\lambda\}\) do
11: \( \text{c-errors}(j) = \text{mean}(y ! = \text{sign}(Y_j))\)
12: end for
13: \( k = \arg\min(c_{\text{errors}})\)
14: return \(c_{\text{errors}}(k), \Lambda(k)\)
15: end function
16: \( h^* = \text{OpenTuner}(\text{ComputeError}, h_{\text{min}}, h_{\text{max}}, n_h)\)
17: \([c_{\text{error}}, \lambda^*] = \text{ComputeError}(h^*)\)
18: return \(h^*, \lambda^*\)
```

The \(c_{\text{error}}\) metric is defined in Algorithm 5 line 14. Hyperparameter search was performed for \(h\) and \(\lambda\) of the Gaussian kernel (for all codes), with a budget of ten OpenTuner iterations. We use the same hyperparameters for the HSS-ANN experiments as for the Exact column to show that the approximation error introduced by HSS-ANN is minimal with respect to an exact kernel ridge regression method (i.e., using only dense linear algebra, and without any subsampling such as leverage scores). This error also indicates the best attainable error with kernel ridge regression. The classification error is computed with one-fold cross validation, and the training set is comprised of a random subset of the original dataset of size \(10^4\), whereas the validation and testing dataset represent \(10^3\) disjoint samples.

The total fit time and memory of HSS-ANN and INV-ASKIT are listed in Table I. For all datasets, HSS-ANN can achieve up to \(6\times\) speedup compared to INV-ASKIT for the datasets of size \(n = 10^4\). Although HSS-ANN may have slightly higher memory usage than INV-ASKIT, the memory complexity of HSS-ANN is \(\mathcal{O}(nr)\) as opposed to \(\mathcal{O}(nr \log n)\) for INV-ASKIT. Therefore for larger datasets, the memory requirement of HSS-ANN will be preferable.

Using the complete SUSY dataset, 5 million points, partitioned into 80% training, 10% validation, 10% test, and 10 evaluations of OpenTuner we get a classifier with 20.28% validation error and 20.29% test error (\(h = 0.55, \lambda = 10\)), numbers which are comparable to those reported in the literature [8], [12], [27]. Our approximation utilizes all the data in the training set \(n\) to build the kernel ridge regression matrix of size \(n \times n\) (i.e., without subsampling to prune data), and is able to utilize shared and distributed memory parallelism, in this case, \(2^{10} = 1024\) cores.

Comparisons with [25] are reported in Table III. Numerical experiments show significant speedups in favor of the framework presented here: \(1.8 \times, 3.5 \times, 4.8 \times\) and \(12.1 \times\), respectively. A key difference against the work under comparison is that our framework does not rely on the construction of an \(\mathcal{O}(n \log n)\) \(\mathcal{H}\)-matrix approximation, to compute an \(\mathcal{O}(n)\) HSS matrix approximation, but it constructs the HSS matrix in one pass, which leads to a decrease in overall the memory footprint of the kernel ridge regression matrix approximation. Numerical experiments utilize the same dataset, hyperparameters and the computational environment as reported in the work [25].
D. Parallel performance and large-scale prediction

The following large-scale experiments were performed at NERSC’s Cori supercomputer. Each compute node of Cori has two sockets, with a 16-core Intel Xeon E5-2698 v3 (“Haswell”) processor at 2.3 GHz per socket, and 128 GB DDR4 memory. We leverage both distributed, and shared-memory parallelism with MPI and OpenMP, respectively.

The strong scaling experiments are comprised of a random subset of one million samples for training data, and we report on the compression time of the matrix. Figure [7] shows the compression time as we increase the number of processors. A decrease in time is seen up to 210 processors.

E. Scikit-learn compatible Python interface

Our kernel algorithms are developed in C++, but we provide a C interface, and on top of that a Python interface compatible with the scikit-learn [23] classifiers and regressors. The Python interface class STRUMPACKKernel derives from BaseEstimator and ClassifierMixin, which are the base classes for all scikit-learn estimators and classifiers. The STRUMPACKKernel class implements fit, predict and decision_function member functions. See Listing [1] in the appendix for an illustration of this Python interface. Note that the interface can also be used for multi-class classification through the scikit-learn One-Vs-One or One-Vs-All estimators or with scikit-learn hyperparameter optimization algorithms for grid search or random search with cross validation, see for instance [13]. This is illustrated in Listing [2] in the appendix with a distributed memory Python example using the SUSY dataset.

We compare the performance of the standard scikit-learn (shared memory) kernel ridge regression with that of HSS-ANN (shared and distributed memory) using the SUSY datasets. As can be seen from Figure [8], scikit-learn relies on the $O(n^3)$ Cholesky factorization from LAPACK, while HSS-ANN can attain a much lower computational complexity. In addition, we can attain significantly reduced computation time with the distributed-memory implementation of HSS-ANN (that was tested with 3.07 GHz 1280 POWER9 cores of the Summit supercomputer).

### D. Parallel performance and large-scale prediction

The following large-scale experiments were performed at NERSC’s Cori supercomputer. Each compute node of Cori has two sockets, with a 16-core Intel Xeon E5-2698 v3 (“Haswell”) processor at 2.3 GHz per socket, and 128 GB DDR4 memory. We leverage both distributed, and shared-memory parallelism with MPI and OpenMP, respectively.

The strong scaling experiments are comprised of a random subset of one million samples for training data, and we report on the compression time of the matrix. Figure [7] shows the compression time as we increase the number of processors. A decrease in time is seen up to $2^{10}$ processors.

### E. Scikit-learn compatible Python interface

Our kernel algorithms are developed in C++, but we provide a C interface, and on top of that a Python interface compatible with the scikit-learn [23] classifiers and regressors. The Python interface class STRUMPACKKernel derives from BaseEstimator and ClassifierMixin, which are the base classes for all scikit-learn estimators and classifiers. The STRUMPACKKernel class implements fit, predict and decision_function member functions. See Listing [1] in

<table>
<thead>
<tr>
<th>Dataset</th>
<th>HSS-ANN</th>
<th>HSS-ANN</th>
<th>INV-ASKIT</th>
<th>INV-ASKIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUSY</td>
<td>69.2</td>
<td>243.7</td>
<td>104.1</td>
<td>120.6</td>
</tr>
<tr>
<td>COVTYPE</td>
<td>25.5</td>
<td>149.1</td>
<td>110.1</td>
<td>138.1</td>
</tr>
<tr>
<td>GAS</td>
<td>13.7</td>
<td>93.7</td>
<td>75.8</td>
<td>85.2</td>
</tr>
<tr>
<td>MNIST</td>
<td>74.1</td>
<td>305.2</td>
<td>426.3</td>
<td>374.0</td>
</tr>
<tr>
<td>HEPMASS</td>
<td>82.2</td>
<td>354.1</td>
<td>55.7</td>
<td>67.4</td>
</tr>
<tr>
<td>LETTER</td>
<td>49.5</td>
<td>221.8</td>
<td>69.7</td>
<td>333.1</td>
</tr>
<tr>
<td>PENDIGITS</td>
<td>27.7</td>
<td>202.7</td>
<td>65.3</td>
<td>157.6</td>
</tr>
</tbody>
</table>

### PERFORMANCE BREAKDOWN (IN SECONDS) WITH DIFFERENT NUMBER OF PROCESSES, AND COMPARISON WITH [23].

<table>
<thead>
<tr>
<th>Dataset</th>
<th>SUSY (n = 4.5M)</th>
<th>COVTYPE (n = 0.5M)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cores</td>
<td>32</td>
<td>512</td>
</tr>
<tr>
<td>HSS construction</td>
<td>1,759.3</td>
<td>185.9</td>
</tr>
<tr>
<td>Factorization</td>
<td>181.1</td>
<td>29.8</td>
</tr>
<tr>
<td>Solve</td>
<td>2.7</td>
<td>0.7</td>
</tr>
<tr>
<td>Fit total (this work)</td>
<td>1,943.2</td>
<td>212.4</td>
</tr>
<tr>
<td>Fit total in [23]</td>
<td>3,532.1</td>
<td>748.6</td>
</tr>
<tr>
<td>Speed-up vs [23]</td>
<td>1.8x</td>
<td>3.5x</td>
</tr>
</tbody>
</table>

### TABLE III

Fig. 8. Time comparison between scikit-learn and HSS-ANN using the SUSY datasets.

The work presents a framework for kernel ridge regression that is scalable and memory efficient. It is scalable in terms of an optimal number of operations and in its ability to utilize a massively parallel computer system. It is memory efficient as it creates an approximation with optimal memory footprint. We present comparisons with a state-of-the-art Nyström based method [27] with near-optimal $O(n\sqrt{n})$ training time, and with a similar ($O(\sqrt{n}\log n)$) approach [25] – which requires a...
higher memory footprint due to the need for an intermediate $H$ representation, that in this work is removed by virtue of the use of approximate nearest neighbors. Numerical experiments in a distributed memory environment show that our implementation is able to reduce the time to solution by effectively utilizing more hardware and that it is possible to select an upper bound of the approximation error against a fully dense kernel ridge matrix with a single tunable parameter $\epsilon$. Furthermore, this method compares very favorably against the current kernel ridge regression implementation in scikit-learn, which is implemented with $O(n^3)$ Cholesky decomposition. We provide an interface to scikit-learn in order to easily leverage our software within scikit-learn at a much-improved performance and memory footprint.

ACKNOWLEDGMENTS

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. We used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility operated under Contract No. DE-AC02-05CH11231, and resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725. E. Rebrova also acknowledges sponsorship by Capital Fund Management.

REFERENCES

import numpy as np
from sklearn.base import BaseEstimator, ClassifierMixin
from sklearn.datasets import check_X_y,
    check_array, check_is_fitted
from sklearn.utils import unique_labels
sp = ctypes.cdll.LoadLibrary('@CMAKE_INSTALL_PREFIX@
/lib/libstrumpack.so')  # path set by CMake

class STRUMPACKKernel(BaseEstimator, ClassifierMixin):
    # kernel: 'rbf' ('Gauss'), 'Laplace' or 'ANOVAs' (degree p)
    # approximation: 'HSS' or 'HODLR'
    # MPI is supported through mpi4py
    def __init__(self, h=1., lam=4., p=1, kernel='rbf',
        approximation='HSS', mpi=False, argv=None):
        ...
        def fit(self, X, y):
            ...
            if X.dtype == np.float64:
                self.K_ = sp.
                    STRUMPACK_kernel_fit_HSS_MPI_double(
                        self.K_, ctypes.c_void_p(y.
                            ctypes.data), ctypes.c_int(argc),
                        argv)  # ...
                    return self
            def predict(self, X):
                ...
                if X.dtype == np.float64:
                    sp.
                        STRUMPACK_kernel_predict_double(
                            self.K_, ctypes.c_int(X.shape[0]),
                        ctypes.c_void_p(prediction.ctypes.
                            data))  # ...
                    return [self.classes_[0] if prediction[i] <
                        0.0 else self.classes_[1]
                        for i in range(X.shape[0])]

Listing 1. Python scikit-learn interface.

Listing 2. Example usage of the Python scikit-learn interface.