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 $\mathcal{O}(r^2 n)$ with respect to the number of training samples (n) 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 provides sufficient accuracy in comparison with state-of-the-art methods and with the exact (i.e. non-approximated) kernel ridge regression method. For datasets, in the order of 10^6 data points, we show that our framework strong-scales to 10^3 cores. Finally, we provide a Python interface to the scikit-learn library so that scikit-learn can leverage our high-performance solver library to achieve 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

$$\text{argmin}_w \sum_{i=1}^{n} (y_i - X_i X^T w)^2 + \lambda \|X^T w\|_2^2,$$

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

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

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 $X_i X_j^T$ by the element $\mathcal{K}(X_i, X_j) = K_{ij}$, 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 $X'_i \in X'$, $i = 1, \ldots, m$, 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 $\mathcal{O}(n^2)$ and $\mathcal{O}(n^3)$, respectively, with n data points. This is due to the storage and inversion of the $n \times n$ dense positive definite matrix $A = K + \lambda I$. Throughput 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

$$K_{ij} = \mathcal{K}^g(X_i, X_j), \quad X_1, \ldots, X_n = \text{training data}$$
$$\mathcal{K}^g(x, y) = \exp\left(-\frac{1}{2}\frac{\|x-y\|_2^2}{h^2}\right). \tag{2}$$

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:

$$\mathcal{K}^l(x, y) = \exp\left(-\frac{1}{2}\frac{\|x-y\|_1}{h}\right), \tag{3}$$

$$\mathcal{K}^a(x, y) = \sum_{1 \le k_1 < \ldots < k_p \le d} \mathcal{K}^g(x_{k_1}, y_{k_1}) \cdots \mathcal{K}^g(x_{k_p}, y_{k_p}). \tag{4}$$

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 $\mathcal{K}$ is dropped without causing 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 ($\mathcal{H}$ [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 $\mathcal{H}$ 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 $\mathcal{O}(dn \log n)$ 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 $\mathcal{O}(n \log n)$ algorithm INV-ASKIT to perform a factorization of the approximate matrix [36], [37] (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 $\mathcal{O}(n^3)$ 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 $\mathcal{O}(n)$ 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 2×2 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

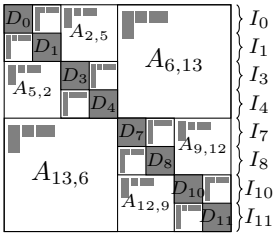

Fig. 1. Illustration of an HSS matrix using 4 levels. Diagonal blocks are partitioned recursively. Gray blocks denote the basis matrices.

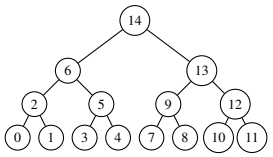

Fig. 2. Tree for Figure 1, using post-tordering. All nodes except the root store U_i and V_i. Leaves store D_i, non-leaves B_{ij}, B_{ji}

low-rank factorization $A_{ij} \approx U_i B_{ij} V_j^T$. Moreover, the column basis matrix U_i for a node i with children c_1 and c_2 in the hierarchy is defined as $U_i = [U_{c_1}\ 0\,;0\ U_{c_2}]\,\tilde{U}_i$, and hence only the smaller matrix $\tilde{U}_i$ is stored at node i. Only at the leaf nodes, where $U_i \equiv \tilde{U}_i$, 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, $U_i \equiv V_i$ and $B_{ij} \equiv B_{ji}$.

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 Ω. 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 $\mathcal{O}(rn)$ 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 [14], [1], 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 $\mathcal{H}$-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 $\mathcal{O}(n^2 r)$ (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 ν_{ann} nearest neighbors for each data point X_i. Finding exact nearest neighbors involves computing pairwise distances between all points and takes $\mathcal{O}(n^2 d)$ 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 ν_{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 $\mathcal{S}$.

To begin with, the data points are partitioned into the leaves of the random projection tree as $\{L_1, \ldots, L_k\} = \texttt{ConstructRPT}(\mathcal{X}, 6\nu_{ann})$ 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 5-11). 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 ν_{ann} *best* ones out of the union of ν_{ann} previously found neighbours and ν_{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 $\mathcal{S}$ 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: ν_{ann} – number of approximate neighbors to search
$\mathcal{X} = \{X_1, \ldots, X_n\}$ – data points, $X_i \in \mathbb{R}^d$
Output: $[\mathcal{N}, \mathcal{S}]$ – $n \times \nu_{ann}$ matrices with neighbors of each X_i and corresponding scores

1: $q = 0$ ▷ $q =$ quality of the neighbor approximation
2: $\mathcal{N}, \mathcal{S} =$ (), ()
3: **while** $iter < 30$ and $q < 0.99$ **do**
▷ construct random projection tree (RPT), see [10]
▷ $L_l =$ list of data points inside leaf l of the RPT
4: $\{L_1, \ldots, L_k\} = \texttt{ConstructRPT}(\mathcal{X}, 6\nu_{ann})$
5: **for** $l = 1$ **to** k **do** ▷ find ν_{ann} closest points for each X_i within its leaf
6: construct distance matrix $\mathcal{D}(i,j)$ for $i, j \in L_l$
7: **for** i **in** L_l **do**
8: $\mathcal{N}_{cur}(i,:) = \nu_{ann}$ elements in L_l with smallest $\mathcal{D}(i,:)$
9: $\mathcal{S}_{cur}(i,:) = \mathcal{D}(i, \mathcal{N}_{cur}(i,:))$
10: **end for**
11: **end for**
12: **for** $i = 1$ **to** n **do** ▷ keep ν_{ann} closest neighbors found over previous iterations
13: $\mathcal{N}(i,:) = \nu_{ann}$ elements in $\mathcal{N}(i,:) \cup \mathcal{N}_{cur}(i,:)$ with smallest scores $(\mathcal{S}(i,:) \cup \mathcal{S}_{cur}(i,:))$
14: $\mathcal{S}(i,:) =$ respective scores of $\mathcal{N}(i,:)$
15: **end for**
16: $\mathcal{X}_s = \{X_{i_1}, \ldots, X_{i_s}\}$ random sample from $\mathcal{X}$
17: **for** $j = 1$ **to** s **do** ▷ test ANN quality on $s = 100$ random data points
18: $\mathcal{N}_{exact}(i_j,:) = \nu_{ann}$ L_2-closest data points to $X_{i_j} \in \mathcal{X}_s$
19: $q = q + |\mathcal{N}(i_j,:) \cap \mathcal{N}_{exact}(i_j,:)|/\nu_{ann}$
20: **end for**
21: $q = q/s$ ▷ average over s points
22: **end while**

$\mathcal{O}(sn)$, 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 $\mathcal{O}(n)$ operations, instead of $\mathcal{O}(n^2)$ 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 ν_{ann} adaptively: if we observe that more basis columns are needed (that is, the quality of HSS compression was not good), we increase ν_{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 for

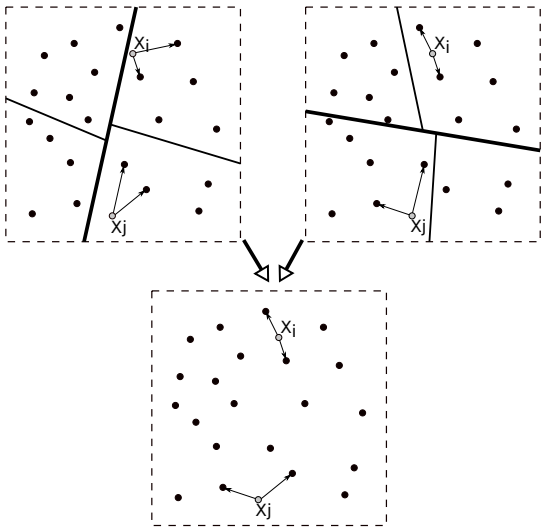

Fig. 3. Illustration of the approximate nearest neighbor search algorithm, see Algorithm 2. Multiple random projection trees are constructed (the 2 recursive partitions at the top), and in every tree, the search for the ν_{ann} nearest neighbors is restricted to the leaves, which are of size $\sim 6\nu_{ann}$. For data point X_j, the $\nu_{ann} = 2$ 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 $\mathcal{O}(r^2 n)$. 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_τ can be computed directly from S_τ (line 13) using an interpolative decomposition [9]: $[X, J] = ID(Y, \varepsilon)$, such that $Y = Y(:, J)X + \mathcal{O}(\varepsilon)$, where $Y(:, J)$ is a subset of the columns of Y. However, when the interpolative decomposition picks almost all columns of Y (S_τ^T in line 13), or, more precisely when $|J| > d_{\text{max}} - p$, 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 [25], 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 λ, we developed an autotuning training framework to search for an optimal setting of h and λ, 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 λ changes we can avoid recompression, which is costly. Thus, our tuning strategy is to

Algorithm 3 HSScompression: construction of an HSS matrix using approximate nearest neighbors

Input: $\{X_1, \ldots, X_n\}$ – data points, $X_i \in \mathbb{R}^d$
$\mathcal{N}$, $\mathcal{S}$ – $n \times \nu_{ann}$ neighbors and scores for each X_i
$K_{ij} = \mathcal{K}(X_i, X_j)$ – kernel matrix
HSS cluster tree
p – oversampling parameter, ε – compression tolerance
Output: $\tilde{K}$ – HSS of K, defined by D_τ, U_τ, B_{ν_1,ν_2}

1: **for** node τ in the HSS tree, in postorder **do**
2: **if** τ is a leaf **then**
3: $D_\tau = K(I_\tau, I_\tau)$
4: $d_{\text{max}} = |I_\tau| + p$
5: $N_\tau = d_{\text{max}}$ elements of $\cup_{i \in I_\tau}(\mathcal{N}(i, :)) \setminus I_\tau$ with smallest scores
6: $S_\tau = K(I_\tau, N_\tau)$ ▷ local sample matrix
7: **else** ▷ let ν_1 and ν_2 be two children of τ
8: $B_{\nu_1,\nu_2} = B_{\nu_2,\nu_1} = K(\tilde{I}_{\nu_1}, \tilde{I}_{\nu_2})$
9: $d_{\text{max}} = |\tilde{I}_{\nu_1}| + |\tilde{I}_{\nu_2}| + p$
10: $N_\tau = d_{\text{max}}$ elements of $(N_{\nu_1} \cup N_{\nu_2}) \setminus (I_{\nu_1} \cup I_{\nu_2})$ with smallest scores
11: $S_\tau = \begin{bmatrix} K(\tilde{I}_{\nu_1}, N_\tau) \\ K(\tilde{I}_{\nu_2}, N_\tau) \end{bmatrix}$
12: **end if**
13: $[U_\tau, J_\tau] = ID(S_\tau^T, \varepsilon)$
14: **if** ID failed to reach tolerance ε **then**
15: Failed to compress HSS ▷ restart, see Alg. 4
16: **end if**
17: **if** τ is a leaf **then**
18: $\tilde{I}_\tau = I_\tau(J_\tau)$
19: **else**
20: $\tilde{I}_\tau = [\tilde{I}_{\nu_1} \tilde{I}_{\nu_2}](J_\tau)$
21: **end if**
22: **end for**

Algorithm 4 HSS compression with adaptive ν_{ann} selection

Input: $\mathcal{X} = \{X_i\}_{i=1}^n$ – data points, $X_i \in \mathbb{R}^d$
$K_{ij} = \mathcal{K}(X_i, X_j)$ – kernel matrix
ν_{ann} – number of approximate nearest neighbors
p – oversampling parameter, ε – compression tolerance
Output: $\tilde{K}$ – HSS of K, defined by D_τ, U_τ, B_{ν_1,ν_2}

1: construct HSS tree via 2-means, cobble, PCA, etc.
2: **while** not successfully compressed **do**
3: $[\mathcal{N}, \mathcal{S}] =$ `ConstructANN`($\mathcal{X}$, ν_{ann})
4: $\tilde{K} =$ `HSScompression`($\mathcal{X}$, $\mathcal{N}$, $\mathcal{S}$, K, tree, p, ε)
5: $\nu_{ann} = 2 \cdot \nu_{ann}$
6: **end while**

search for multiple λ values for each selected h value. To tune h, we use an off-the-shelf black-box optimization package called OpenTuner [4], which takes as inputs an objective function `ComputeCerror` 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 λ values. Note that the `c-error` of the given hyperparameters is defined at line 11 of Algorithm 5. For each new λ, we update $\tilde{A}$ by adding it to $\tilde{K}$'s diagonal entries (line 4), and perform ULV factorization and solve for each new λ (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.

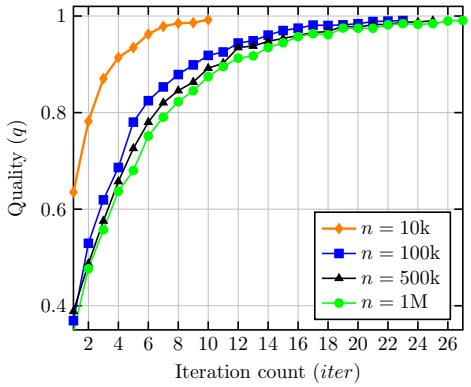

Fig. 4. History of ANN quality q (defined in Algorithm 2, lines 19 and 21), using the SUSY datasets.

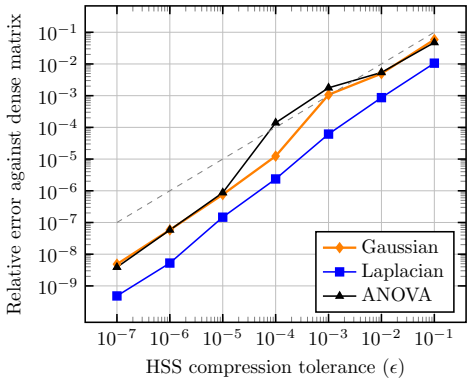

Fig. 5. The relative error between the original (dense) kernel matrix and its corresponding HSS approximation is always smaller than ε.

B. Robustness of kernel matrix approximation

To demonstrate the robustness of the HSS approximation with respect to the relative compression tolerance ε, 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 $\|\tilde{A} - A\|_F / \|A\|_F < \varepsilon$ 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 [18]. 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 ε as compared to the *exact* (single-precision, dense representation) kernel ridge regression matrix.

C. Comparison with other methods

Comparisons with FALKON [27] and INV-ASKIT [36] (more precisely, a similar implementation in the STRUMPACK

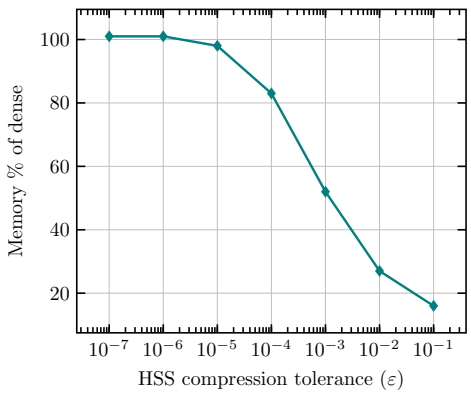

Fig. 6. Memory savings (in percentage) compared to the dense storage of the kernel matrix for different ε.

Algorithm 5 Modeling stage and hyperparameter tuning.

Input: $[h_{min}, h_{max}]$ – 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
$\mathcal{X}_{train}$, $\mathcal{X}_{test}$ – training and testing data sets
Output: h^*, λ^* – optimal hyperparameters after $(n_h \times n_\lambda)$ evaluations.

1: **function** [c_error, λ^*] = ComputeCerror(h)
2: $\tilde{K} = \texttt{HSScompression}(\mathcal{K}(\mathcal{X}_{train}, \mathcal{X}_{train}), h)$ ▷ see Algorithms 2 and 3
3: **for** $j \in \{1, \ldots, n_\lambda\}$ **do**
4: $\tilde{A} = \tilde{K} + \lambda_j I$
5: $[U, L, V] = \texttt{ULVfactorization}(\tilde{A})$ ▷ ULV factorization, $\tilde{A} = ULV$
6: $W(:, j) = (ULV)^{-1} y$ ▷ matrix of weights W, size(W) = $[n, n_\lambda]$
7: **end for**
8: $K' = \mathcal{K}(\mathcal{X}_{test}, \mathcal{X}_{train})$ ▷ size(K') = $[m, n]$
9: $Y = K' \cdot W$ ▷ matrix of predictions Y (blocking), size(Y) = $[m, n_\lambda]$
10: **for** $j \in \{1, \ldots, n_\lambda\}$ **do**
11: c-errors(j) = mean(y != sign(Y_j)) ▷ element-wise sign comparison
12: **end for**
13: k = argmin(c-errors) ▷ index of the smallest c_error
14: **return** c-errors(k), $\Lambda(k)$
15: **end function**
16: h^* = OpenTuner(ComputeCerror, h_{min}, h_{max}, n_h)
17: [c_error, λ^*] = ComputeCerror(h^*)
18: **return** h^*, λ^*

library) are reported in Table I. The c_error metric is defined in Algorithm 5, line 14. Hyperparameter search was performed for h and λ 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 II. For all datasets, HSS-ANN can achieve up to 6× 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),

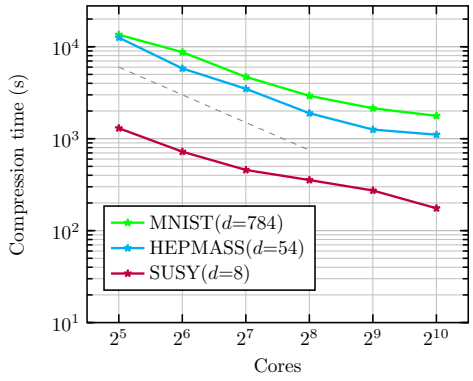

Fig. 7. Strong scaling $n = 10^6$.

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×, 3.5×, 4.8× and 12.1×, 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].

Dataset	FALKON [27]					c_error		
	h	λ	c_error	h	λ	**HSS-ANN**	**INV-ASKIT [36]**	**Exact**
SUSY	4.35	5.28E-04	**25.5**	1.30	3.11E+00	**24.3**	**24.3**	**24.3**
COVTYPE	4.71	4.81E-06	**4.7**	1.89	5.85E+00	**3.5**	**3.5**	**3.4**
GAS	3.83	1.89E-04	**2.2**	1.25	2.24E+00	**0.3**	**0.3**	**0.3**
MNIST	8.60	1.10E-04	**6.3**	5.77	2.11E-01	**2.3**	**2.3**	**2.3**
HEPMASS	6.98	3.49E-04	**9.5**	3.54	4.28E+00	**9.3**	**9.0**	**9.3**
LETTER	7.63	4.83E-05	**0.6**	0.60	4.83E+00	**0.1**	**0.1**	**0.1**
PENDIGITS	4.05	2.60E-05	**0.6**	0.94	7.32E-01	**0.3**	**0.3**	**0.3**

TABLE I

C_ERROR (DEFINED IN ALGORITHM 5, LINE 14) FOR DIFFERENT METHODS, TRAINING $n = 10^4$, TEST AND VALIDATION $m = 10^3$. THE SAME VALUES FOR λ AND h ARE USED FOR THE HSS APPROXIMATION, THE INV-ASKIT [36] EXPERIMENTS AND THE EXACT KERNEL MATRIX.

Dataset	HSS-ANN Time (s)	HSS-ANN Mem (MB)	INV-ASKIT Time (s)	INV-ASKIT Mem (MB)
SUSY	69.2	243.7	104.1	120.6
COVTYPE	23.5	149.1	110.1	138.1
GAS	13.7	93.7	75.8	83.2
MNIST	74.1	305.2	426.3	374.0
HEPMASS	82.2	354.1	55.7	67.4
LETTER	49.5	221.8	69.7	333.1
PENDIGITS	27.7	202.7	65.3	157.6

TABLE II

FIT TIME AND MEMORY WITH HSS-ANN AND INV-ASKIT, TRAINING $n = 10^4$, TEST AND VALIDATION $m = 10^3$.

	SUSY ($n = 4.5M$)		COVTYPE ($n = 0.5M$)	
Cores	32	512	32	512
HSS construction	1,759.3	185.9	67.0	17.2
Factorization	181.1	25.8	35.7	5.6
Solve	2.7	0.7	0.2	0.1
Fit total (this work)	1,943.2	212.4	102.9	23.0
Fit total in [25]	3,532.1	748.6	495.5	276.9
Speed-up vs [25]	**1.8**×	**3.5**×	**4.8**×	**12.1**×

TABLE III

PERFORMANCE BREAKDOWN (IN SECONDS) WITH DIFFERENT NUMBER OF PROCESSORS, AND COMPARISON WITH [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 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 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.

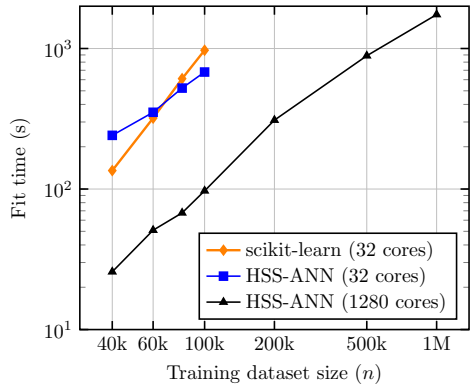

Fig. 8. Time comparison between scikit-learn and HSS-ANN using the SUSY datasets.

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 $\mathcal{O}(n^3)$ 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).

IV. CONCLUSIONS

This 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 $\mathcal{O}(n\sqrt{n})$ training time, and with a similar ($\mathcal{O}(r^2 n \log n)$) approach [25] – which requires a

higher memory footprint due to the need for an intermediate $\mathcal{H}$ 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 ε. Furthermore, this method compares very favorably against the current kernel ridge regression implementation in scikit-learn, which is implemented with $\mathcal{O}(n^3)$ 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

[1] STRUMPACK website. http://portal.nersc.gov/project/sparse/strumpack/.

[2] Alexandr Andoni and Piotr Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Communications of the ACM*, 51(1):117, 2008.

[3] Alexandr Andoni and Ilya Razenshteyn. Optimal data-dependent hashing for approximate near neighbors. In *Proceedings of the forty-seventh annual ACM symposium on Theory of computing*, pages 793–801. ACM, 2015.

[4] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. OpenTuner: An extensible framework for program autotuning. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 303–316, New York, NY, USA, 2014. ACM.

[5] Haim Avron, Kenneth L. Clarkson, and David P. Woodruff. Faster kernel ridge regression using sketching and preconditioning. *SIAM Journal on Matrix Analysis and Applications*, 38(4):1116–1138, 2017.

[6] S. Chandrasekaran, M. Gu, and W. Lyons. A fast adaptive solver for hierarchically semiseparable representations. *CALCOLO*, 42(3):171–185, Dec 2005.

[7] Shiv Chandrasekaran, Ming Gu, and Timothy Pals. A fast ULV decomposition solver for hierarchically semiseparable representations. *SIAM Journal on Matrix Analysis and Applications*, 28(3):603–622, 2006.

[8] Jie Chen, Haim Avron, and Vikas Sindhwani. Hierarchically compositional kernels for scalable nonparametric learning. *J. Mach. Learn. Res.*, 18(1):2214–2255, January 2017.

[9] Hongwei Cheng, Zydrunas Gimbutas, Per-Gunnar Martinsson, and Vladimir Rokhlin. On the compression of low rank matrices. *SIAM Journal on Scientific Computing*, 26(4):1389–1404, 2005.

[10] Sanjoy Dasgupta and Yoav Freund. Random projection trees for vector quantization. *IEEE Transactions on Information Theory*, 55(7):3229–3242, 2009.

[11] Sanjoy Dasgupta and Kaushik Sinha. Randomized partition trees for exact nearest neighbor search. In *Conference on Learning Theory*, pages 317–337, 2013.

[12] Diego García-Gil, Julián Luengo, Salvador García, and Francisco Herrera. Enabling smart data: Noise filtering in big data classification. *Information Sciences*, 479:135–152, 2019.

[13] Aurélien Géron. *Hands-on machine learning with Scikit-Learn and TensorFlow: concepts, tools, and techniques to build intelligent systems*. " O'Reilly Media, Inc.", 2017.

[14] Pieter Ghysels, Xiaoye S. Li, Christopher Gorman, and François-Henry Rouet. A robust parallel preconditioner for indefinite systems using hierarchical matrices and randomized sampling. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 897–906. IEEE, 2017.

[15] Alex Gittens and Michael W. Mahoney. Revisiting the Nyström method for improved large-scale machine learning. *The Journal of Machine Learning Research*, 17(1):3977–4041, 2016.

[16] Wolfgang Hackbusch. A Sparse Matrix Arithmetic Based on $\mathcal{H}$-Matrices. Part I: Introduction to $\mathcal{H}$-Matrices. *Computing*, 62(2):89–108, 1999.

[17] T. Hastie, R. Tibshirani, and J. Friedman. Unsupervised learning. In *The elements of statistical learning*, pages 485–585. Springer, New York, 2009.

[18] Kenneth L. Ho and Leslie. Greengard. A fast direct solver for structured linear systems by recursive skeletonization. *SIAM Journal on Scientific Computing*, 34(5):A2507–A2532, 2012.

[19] Thomas Hofmann, Bernhard Schölkopf, and Alexander J Smola. Kernel methods in machine learning. *The annals of statistics*, pages 1171–1220, 2008.

[20] Sanjiv Kumar, Mehryar Mohri, and Ameet Talwalkar. Sampling methods for the Nyström method. *Journal of Machine Learning Research*, 13(Apr):981–1006, 2012.

[21] William B. March, Bo Xiao, and George Biros. ASKIT: approximate skeletonization kernel-independent treecode in high dimensions. *SIAM J. Scientific Computing*, 37(2):1089–1110, 2015.

[22] Per-Gunnar Martinsson. A fast randomized algorithm for computing a hierarchically semiseparable representation of a matrix. *SIAM Journal on Matrix Analysis and Applications*, 32(4):1251–1274, 2011.

[23] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[24] Ali Rahimi and Benjamin Recht. Random features for large-scale kernel machines. In *Advances in neural information processing systems*, pages 1177–1184, 2008.

[25] Elizaveta Rebrova, Gustavo Chávez, Yang Liu, Pieter Ghysels, and Xiaoye S. Li. A study of clustering techniques and hierarchical matrix formats for kernel ridge regression. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2018.

[26] François-Henry Rouet, Xiaoye S. Li, Pieter Ghysels, and Artem Napov. A distributed-memory package for dense hierarchically semi-separable matrix computations using randomization. *ACM Transactions on Mathematical Software (TOMS)*, 42(4):27, 2016.

[27] Alessandro Rudi, Luigi Carratino, and Lorenzo Rosasco. FALKON: An optimal large scale kernel method. In *Advances in Neural Information Processing Systems*, pages 3888–3898, 2017.

[28] Si Si, Cho-Jui Hsieh, and Inderjit S. Dhillon. Memory efficient kernel approximation. *The Journal of Machine Learning Research*, 18(1):682–713, 2017.

[29] Ruoxi Wang, Yingzhou Li, Michael W. Mahoney, and Eric Darve. Structured block basis factorization for scalable kernel matrix evaluation. *CoRR*, abs/1505.00398, 2015.

[30] Christopher K. I. Williams and Matthias Seeger. Using the Nyström method to speed up kernel machines. In *Advances in neural information processing systems*, pages 682–688, 2001.

[31] Jianlin Xia, Shivkumar Chandrasekaran, Ming Gu, and Xiaoye S. Li. Fast algorithms for hierarchically semiseparable matrices. *Numerical Linear Algebra with Applications*, 17(6):953–976, 2010.

[32] Bo Xiao and George Biros. Parallel algorithms for nearest neighbor search problems in high dimensions. *SIAM Journal on Scientific Computing*, 38(5):667–699, 2016.

[33] Xin Ye, Jianlin Xia, and Lexing Ying. Analytical low-rank compression via proxy point selection. *arXiv preprint arXiv:1903.08821*, 2019.

[34] Yang You, James Demmel, Cho-Jui Hsieh, and Richard Vuduc. Accurate, fast and scalable kernel ridge regression on parallel and distributed systems. In *Proceedings of the 2018 International Conference on*

Supercomputing, ICS '18, pages 307–317, New York, NY, USA, 2018. ACM.

[35] Chenhan D. Yu, James Levitt, Severin Reiz, and George Biros. Geometry-oblivious FMM for compressing dense SPD matrices. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 53:1–53:14, 2017.

[36] Chenhan D. Yu, William B. March, and George Biros. An $N \log N$ parallel fast direct solver for kernel matrices. In *2017 IEEE International Parallel and Distributed Processing Symposium, IPDPS*, pages 886–896, 2017.

[37] Chenhan D. Yu, William B. March, Bo Xiao, and George Biros. INV-ASKIT: A parallel fast direct solver for kernel matrices. In *2016 IEEE International Parallel and Distributed Processing Symposium, IPDPS*, pages 161–171, 2016.

APPENDIX

```python
import numpy as np
import ctypes
from sklearn.base import BaseEstimator, ClassifierMixin
from sklearn.utils.validation import check_X_y, check_array, check_is_fitted
from sklearn.utils.multiclass 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 'ANOVA' (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_create_kernel_double(
                ctypes.c_int(X.shape[0]), ctypes.c_int(X.shape[1]),
                ctypes.c_void_p(X.ctypes.data), ctypes.c_double(self.h),
                ctypes.c_double(self.lam), ctypes.c_int(p), ctypes.c_int(ktype))
        # ...
        if self.approximation is 'HSS':
            if self.mpi:
                if X.dtype == np.float64:
                    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(X.ctypes.data),
                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])]

    def decision_function(self, X):
        # ...
```

Listing 1. Python scikit-learn interface.

```python
#!/bin/python3
# Build STRUMPACK as a shared library: -DBUILD_SHARED_LIBS=ON
# Add CMAKE_INSTALL_PREFIX/lib/ to your LD_LIBRARY_PATH
# Add CMAKE_INSTALL_PREFIX/include/python/ to your PYTHONPATH
import sys, numpy as np, STRUMPACKKernel as sp
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()

# parse input parameters
fname = './data/susy_10Kn'
h = 1.3     # kernel width
lam = 3.11  # regularization parameter
p = 1       # ANOVA kernel degree

# read training and testing data from comma-separated value files
prec = np.float32
train_points = np.genfromtxt(fname + '_train.csv', delimiter=",", dtype=prec)
train_labels = np.genfromtxt(fname + '_train_label.csv', delimiter=",", dtype=prec)
test_points = np.genfromtxt(fname + '_test.csv', delimiter=",", dtype=prec)
test_labels = np.genfromtxt(fname + '_test_label.csv', delimiter=",", dtype=prec)
n, d = train_points.shape
m = test_points.shape[0]

# kernel ridge regression classification using HSS approximation of the kernel
K = sp.STRUMPACKKernel(h, lam, p, kernel='rbf', approximation='HSS', mpi=True, argv=sys.arg)
K.fit(train_points, train_labels)
pred = K_HSS.predict(test_points)

# check quality, labels are -1 or +1
def quality(p, l):
    return 100.*(m - sum(p[i]*l[i] < 0 for i in range(m))) / m
if rank == 0:
    print('HSS KernelRR quality =', quality(pred, test_labels), '%')
    print('classes:', K_HSS.classes_)

# optionally fine-tune the model using scikit-learns gridsearch, with cross-validation
from sklearn.model_selection import GridSearchCV #, RandomizedSearchCV,
grid_list = {"h": np.logspace(-2, 2, num=3), "lam": np.logspace(-2, 2, num=3)}
grid_search = GridSearchCV(K, param_grid=grid_list, cv=3)
grid_search.fit(train_points, train_labels.round())
if rank == 0:
    print('best_params_ =', grid_search.best_params_)
pred_gs = grid_search.predict(test_points)
if rank == 0:
    print('HSS KernelRR grid search quality =', quality(pred_gs, test_labels), '%')
```

Listing 2. Example usage of the Python scikit-learn interface to leverage the STRUMPACK fast solvers capability.