Parallel Software for Million-scale Exact Kernel Regression

Yu Chen
William & Mary
Williamsburg, VA, USA
ychen39@wm.edu

Lucca Skon
William & Mary
Williamsburg, VA, USA
lskon@wm.edu

James R. McCombs
Indiana University
Bloomington, IN, USA
jmccombs@iu.edu

Zhenming Liu
William & Mary
Williamsburg, VA, USA
zliu20@wm.edu

Andreas Stathopoulos
William & Mary
Williamsburg, VA, USA
andreas@cs.wm.edu

ABSTRACT

We present the design and the implementation of a kernel principal component regression software that handles training datasets with a million or more observations. Kernel regressions are nonlinear and interpretable models that have wide downstream applications, and are shown to have a close connection to deep learning. Nevertheless, the exact regression of large-scale kernel models using currently available software has been notoriously difficult because it is both compute and memory intensive and it requires extensive tuning of hyperparameters.

While in computational science distributed computing and iterative methods have been a mainstay of large scale software, they have not been widely adopted in kernel learning. Our software leverages existing high performance computing (HPC) techniques and develops new ones that address cross-cutting constraints between HPC and learning algorithms. It integrates three major components: (a) a state-of-the-art parallel eigenvalue iterative solver, (b) a block matrix-vector multiplication routine that employs both multi-threading and distributed memory parallelism and can be performed on-the-fly under limited memory, and (c) a software pipeline consisting of Python front-ends that control the HPC backbone and the hyperparameter optimization through a boosting optimizer. We perform feasibility studies by running the entire ImageNet dataset and a large asset pricing dataset.

CCS CONCEPTS

• **Computing methodologies** → **Machine learning**; **Parallel algorithms**; • **Mathematics of computing** → *Solvers*.

KEYWORDS

kernel principal component regression, SVD, parallel algorithm, classification, machine learning, software tools, boosting

ACM Reference Format:
Yu Chen, Lucca Skon, James R. McCombs, Zhenming Liu, and Andreas Stathopoulos. 2023. Parallel Software for Million-scale Exact Kernel Regression. In *2023 International Conference on Supercomputing (ICS '23), June 21–23, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3577193.3593737

ICS '23, June 21–23, 2023, Orlando, FL, USA

ACM ISBN 979-8-4007-0056-9/23/06.
https://doi.org/10.1145/3577193.3593737

1 INTRODUCTION

Kernel learning refers to a set of learning algorithms that map the original features to a possibly infinite dimensional space and use them to learn a model with tractable/convex objectives. For example, kernel ridge and kernel principal component regression learn a linear model in the feature map, whereas the kernel support vector machine maximizes the margins of training data represented by the feature map. Because kernel learning algorithms usually enjoy sound theoretical properties, they had been extensively used in a wide range of areas such as (medical) image recognition [9, 27, 43], bioinformatics [26, 36], asset pricing [50], recommendation systems [18, 33], smart cities [55, 57], etc. Although in recent years some major downstream "users" move to use deep-learning-based models, applications that require interpretable models and robust reproducibility (e.g., different "random seeds" will not result in models with different performance) still heavily utilize kernel techniques. In addition, it was discovered recently that a neural net with infinite width is equivalent to kernel regression using the so-called neural tangent kernel (NTK) [25, 38]. Therefore, kernel techniques remain relevant for both specialized applications and a better understanding of deep learning.

This work presents the design and implementation of a software package that solves large-scale kernel principal component regressions (KPCR). Kernel principal regression and kernel ridge regression (KRR) are the two most widely used linear kernel models and they often deliver similar performance [13]. We choose KPCR over KRR because the former is more interpretable, especially when the dimension of the kept kernel map is small but the software and techniques developed here can be easily extended to KRR, which would require only a switch to a linear solver instead of an eigensolver.

Scaling up KPCR has been challenging for two reasons: (a) most software uses dense eigenvalue solvers, requiring $O(N^3)$ running time and $O(N^2)$ space, where N is the number of observations (samples); (b) parallelization, when offered, is typically performed across different problem instances rather than to allow for larger problem sizes. However, there is strong interest in techniques and software that handle datasets with a million or more samples, which is at least an order of magnitude larger than what off-the-shelf packages like scikit-learn are capable of. Such datasets are both conceptually and practically important. First, many applications

in finance, medicine, and vision have benchmarks with 500K to 3 million samples [52, 53]. For example, the (original) ImageNet has 1.2 million images [12] which, to our knowledge, cannot be studied with current KRR/KPCR solvers. Second, recent investigation on deep learning demands a scalable kernel learning solver to better understand approximation error between NTK and different deep artechiture [2, 38]. To address this scaling problem, a significant stream of research has focused on reducing the time complexity of the kernel matrix through approximations [14, 41, 51, 54, 56]. The design of our software tool is independent of how the user provides the matrix-vector multiplication and therefore it allows for use of kernel approximations. In this paper, however, we address the exact (non-approximated) kernel matrix as it is more general, more computationally challenging, and still needed by practitioners as it obviates the need to bound another source of error.

A software package that scales up the currently feasible problem size by more than an order of magnitude must leverage multiple existing technologies. For example, iterative methods for eigenvalues or linear systems can and have been used to bring the complexity down to $O(N^2)$ [49] but without distributed computing, the target problem sizes would still be infeasible. In addition, the software should employ techniques that build and apply the kernel matrix efficiently, include hyperparameter optimization in the pipeline, and the eigensolver performance should be tuned for KPCR.

Our solution consists of the following components: *(i)* the use of a state-of-the-art distributed-memory eigenvalue iterative solver that computes k selected eigenpairs in $O(kN^2)$, *(ii)* the development of high performance computing matrix-vector multiplication routines that employ both multi-threading and distributed memory parallelism, and, when needed, can work under limited memory by rebuilding the kernel tile-by-tile on the fly, *(iii)* the development of a software pipeline consisting of two interacting Python front-end drivers, one handling the hyperparameter optimization and the other the regression on an HPC back-end, ensuring a fault-tolerant execution. The main contribution of the paper is the design of a software tool using novel algorithmic integration rather than the introduction of new algorithms.

We demonstrate the efficacy and efficiency of the software on two million-scale downstream applications. First, we apply it on 2 million observations from empirical asset pricing, a notoriously difficult ML problem with a low signal-to-noise ratio. The fast execution of our method allows us to implement a boosted KPCR which demonstrates superior performance [19]. Second, we run KPCR on the entire Imagenet dataset, which to our knowledge is the first time a linear kernel model is used to fit Imagenet. We run this as a feasibility and stress test for our HPC software by computing tens of thousands of eigenvectors. Although, because of $O(N^2)$ complexity, we cannot expect scalability to sizes beyond $O(10^7)$, our experiments demonstrate that we can enable the solution of a variety of important problems with datasets of size $O(10^5 - 10^7)$.

2 PRELIMINARIES AND RELATED WORK

Kernel Regression. We consider the problem of fitting a real-valued function using a total number of N data points $\{\mathbf{x}_i, y_i\}_{i\leq N}$, in which $\mathbf{x}_i \in \mathbf{R}^F$ and $y_i \in \mathbf{R}$. Let $k(\mathbf{x}_i, \mathbf{x}_j)$ be a (positive semi-definite) kernel function that intuitively describes (dis)-similarities between $\mathbf{x}_i$ and $\mathbf{x}_j$. By Mercer's theorem [42], there exists a feature map $\phi(\mathbf{x})$ such that $k(\mathbf{x}_i, \mathbf{x}_j) = \langle \phi(\mathbf{x}_i), \phi(\mathbf{x}_j) \rangle$. Kernel regression models aim to find a linear relation $y \sim \langle \beta, \phi(\mathbf{x}) \rangle$. Note that $k(\cdot, \cdot)$ is part of a model specification and ϕ does not need to be computed explicitly. In addition, the dimension of $\phi(\mathbf{x}_i)$ could be infinite so regularization is needed. When we add a regularizer of $\lambda \|\beta\|_2^2$, the model becomes kernel ridge regression (KRR), and when we use a low rank matrix to approximate the Gram matrix $K \in \mathbf{R}^{N\times N}$, where $K_{i,j} = k(x_i, x_j)$, the model becomes kernel principal component regression (KPCR). It was recently shown that KRR and KPCR are mostly equivalent [13] but the latter often is considered more interpretable because it has a smaller number of learnable parameters. Examples of kernel functions include the Gaussian kernel $k(\mathbf{x}_i, \mathbf{x}_j) = \exp(-\gamma \|\mathbf{x}_i - \mathbf{x}_j\|^2)$, inner product kernel $k(\mathbf{x}_i, \mathbf{x}_j) = \langle \mathbf{x}_i, \mathbf{x}_j \rangle$, and polynomial kernel $k(\mathbf{x}_i, \mathbf{x}_j) = (\langle \mathbf{x}_i, \mathbf{x}_j \rangle + \gamma)^d$. Neural tangent kernels (NTK) are a new family of kernel functions discovered recently that approximate neural nets at the width limit [25]. Computing NTK usually is more resource hungry [38].

Kernel regression can also be used to solve classification problems using standard transformations and is found to be effective [4]. A drawback is that c kernel models need to be fitted to solve a classification with c classes.

Computational challenges. There are two major challenges.

1. Memory and computation. It takes $\Theta(N^2)$ space to store the matrix K. Fitting the model requires us to solve a large least squares system in the form $\tilde{K}\alpha = y$ (obtained from the dual of the MSE cost in β). In KRR, $\tilde{K} = K + \lambda I$, whereas in KPCR, $\tilde{K}$ is a low-rank approximation of K. A combination of the two is also possible. The solutions can be obtained by direct methods e.g., inverting K or $\tilde{K}$, or performing the SVD decomposition using the LAPACK library [3]. The cost of any of these computations is $O(N^3)$ which makes it prohibitive for massive datasets.

Iterative methods have been a central tool in large scale scientific computing [5, 7] but until recently had not received much attention in kernel learning. Based on matrix-vector multiplication, methods such as Conjugate Gradient can solve the KRR problem iteratively with complexity $O(N^2)$ [49], even for many different regularization parameters λ [16]. The Lanczos method computes k largest eigenpairs to form a low rank approximation $K = V_k \Sigma_k V_k^T$ with complexity $O(kN^2)$. Besides interpretability, the benefit of the low rank approach is that we can later solve $\alpha = V_k \Sigma_k^{-1} V_k^T y$ for many different y or even combine it with many different λ.

Recently, some packages for kernel regression have included a sequential version of the restarted Lanczos method (through the ARPACK software [32]) or the Randomized SVD method [20]. See for example [1, 10, 17, 31, 34, 39]. However, they cannot address large scale problems that will not fit into the memory of a single server or that may require a large number of compute nodes (or GPUs) to reduce execution time. Moreover, the optimization of the matrix vector multiplication is critical for performance but is typically left to the user. Other approaches avoid the solution of the entire kernel matrix by randomization, partition, gradient descent methods, or fast multipole approximations [14, 41, 51, 54, 56] but they need to bound the distance of the obtained solutions from those of the original problem, and they may still pose great computational

demands. Although, our software tool could be used in this case, we focus on the exact kernel solution.

Our work takes a holistic approach to Kernel regression solvers, optimizing both distributed matrix vector multiplication for a given kernel and memory constraints, and the underlying iterative solver for these problems. We employ the use of PRIMME, one of the state-of-the-art parallel eigensolvers [45]. By adjusting block size, basis size, and other parameters, PRIMME methods can be tuned to converge nearly optimally and its MPI implementation can work on distributed memory computers. Other high quality packages include SLEPc [21] and Anasazi [6], but they are much bigger and less agile, while not performing better than PRIMME [45].

2. Hyperparameter search and boosting. In practice, extensive hyperparameter search is crucial, e.g., finding the optimal γ of the Gaussian kernel. Search methods include discrete grid search, Bayesian optimization, and learning curve based optimization [28]. When we perform a search, the data is usually split into training and validation (or testing) sets, in which the training set is used to fit a function with a specific hyperparameter set. The quality of the fitting is determined by using the validation set to compute a score. Typical scoring functions include MSE, classification error, and correlation between validation and predicted outputs. Hereafter, we use T and S to denote quantities related to the training and validation set, respectively (e.g., K_T is the Gram matrix formed from the training set). Hyperparameter search adds an extra dimension of complexity to the computational cost of fitting a kernel model (i.e., thousands of models may need to be fitted to find a reliable hyperparameter). Different models can be fitted independently in parallel, and this task based parallelism can be exploited in some packages such as TensorFlow. However, we are not aware of a software package that combines this with distributed memory, and parallel tasks, and this is the second thrust of this work. In addition, models fitted from different hyperparameters usually extract complementary signals. A boosting method consolidating predictions from multiple models usually further strengthens forecasting power and is also frequently used in practice [29].

In the following, we describe our design and implementation of a Machine Learning software pipeline that integrates hyperparameter optimization with a high performance, parallel regression solver, addressing the entire stack: kernel generation, matrix-vector, and eigensolver optimization, and evaluation.

3 DRIVER WORKFLOW

The regression and the automatic machine learning are performed by separate drivers, the regression and evaluation driver and the hyperparameter optimization driver. Both are written in Python and interact with each other as shown in Fig. 1. The hyperparameter optimization driver applies Bayesian optimization and boosting to find the best hyperparameters for the model. It is an iterative method that considers the history of evaluation scores for past choices of hyperparameters, decides which hyperparameters to evaluate next, and passes them to the regression and evaluation driver. The regression and evaluation driver trains the model for the requested hyperparameters, evaluates their fitness score, and returns those scores to the hyperparameter optimization driver to continue the optimization. We can view the hyperparameter optimization driver as a "consumer" of the regression and evaluation driver's results and as an issuer of requests for more results. The regression and evaluation driver coordinates the work requested by assigning it to one or more parallel jobs. Decoupling the drivers into two interacting but independent processes allows for a more flexible, fault tolerant, and scalable design. Under such an architecture, users can switch between different hyperparameter optimization algorithms or different evaluation methods with little effort.

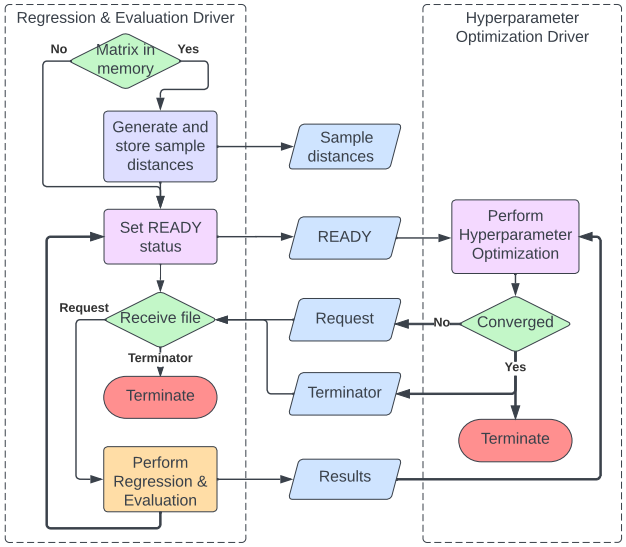

Figure 1: Driver architecture and interaction.

3.1 Hyperparameter optimization driver

The hyperparameter optimization driver is initialized with user-defined configurations including Bayesian optimization parameters (e.g. hyperparameter search space and convergence criteria), and directory paths for storing results needed by the regression and evaluation driver. For each hyperparameter optimization iteration, the hyperparameter optimization driver first identifies the hyperparameters that need to be evaluated with Bayesian optimization based on existing observations (past evaluation results from regression and evaluation driver). Then it creates a request file containing the hyperparameters for which the regression and evaluation driver will compute model solutions and waits for the results. When the regression and evaluation driver finishes the evaluations and writes them to disk, the hyperparameter optimization driver locates the result files based on the request ID, adds the evaluation results to the existing observations with the boosting method and updates the Bayesian optimization.

The hyperparameter optimization driver only submits new requests when a regression and evaluation driver is available, as indicated by the existence of a READY file which contains the process ID of the evaluation driver. After submitting a request, the hyperparameter optimization driver resets the status of the regression and evaluation driver by deleting this file.

3.2 Regression and evaluation driver

The regression and evaluation driver is initialized with the process ID of the hyperparameter optimization driver and the paths to the training and testing data. If the entire kernel matrix can fit in available memory, the sample distances are computed and saved to disk; the regression solver will be generating the kernel matrix from the sample distances. Otherwise, and assuming the sample data can fit in the memory of one node, the regression solver will be performing the matrix-vector operations on the fly by regenerating portions of the matrix from the sample data. At this point, the regression and evaluation driver enters the request loop with the hyperparameter optimization driver, indicating that it is ready to compute a regression model by creating the READY file containing its process ID.

When ready, the hyperparameter optimization driver responds with a request file, containing a list of hyperparameters for each regression model that needs to be solved. For each hyperparameter in the request, the regression and evaluation solver (architecture seen in Fig. 2) computes a model solution for each rank computed by the SVD. It then computes a fitness score for each model solution. The regression and evaluation driver saves to disk the model solution with the highest evaluation score and reports the location of this information back to the hyperparmeter optimization driver in a results file for the current request. Once the regression and evaluation driver has computed the optimal model solutions for all requested hyperparameters, it finalizes the results file, creates a READY file, and awaits a new request from the hyperparmater optimization driver.

3.3 Benefits of a two-driver workflow and fault tolerance

It is natural to address a consumer-producer workflow with a design of separate drivers. The Bayesian optimization function can be changed to different software in any language without affecting the code of the numerical solvers and vice versa. For example, Bayesian optimization can work with partial information and update it as new results arrive. On the other hand, the regression and evaluation driver can decide to launch separate parallel jobs to evaluate different hyperparameters if the resources are available.

The hyperparameter optimization and regression and evaluation drivers implement a checkpoint system for fault tolerance. While a driver is waiting, it can periodically poll the OS with the process ID of the other driver program. If it determines that the other driver has terminated, it will save its current state and terminate as well. If the hyperparameter optimization driver has converged, or if some other error prevents it from continuing, it saves its state, creates a terminator file and terminates itself. The regression and evaluation driver will detect this file and follow its own shutdown procedure. Any running regression and evaluation solvers will finalize their results which can be consumed by the hyperparameter optimization driver when the drivers are restarted at a later time. During a restart, the regression and evaluation driver will check if there is a request that was incompletely processed. If it finds an incomplete results file for the last received request, it will solve the regressions for any outstanding hyperparameters and finalize the results file before creating the READY file.

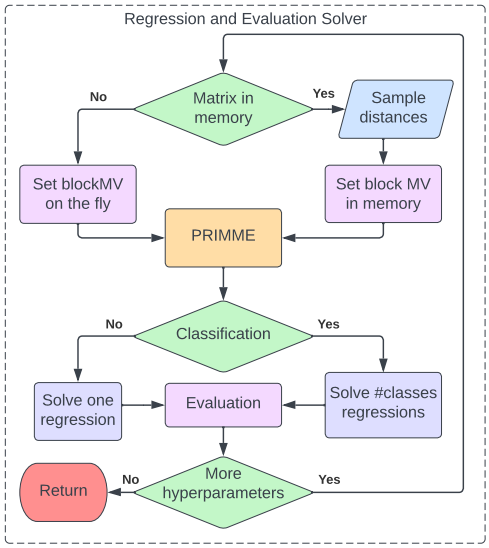

Figure 2: Regression and evaluation solver. Pre-processing sets up the matrix for PRIMME which then performs the truncated SVD. Evaluation depends on whether the problem is regression or classification.

4 THE DESIGN OF THE KERNEL LEARNING SOLVER

The software architecture we described earlier is generic and can work with a variety of hyperparameter optimization and regression software, both sequential and parallel. In this section we describe what makes our package unique; the development of a high performance, parallel code that solves the regression problem and forms and evaluates its predictions. To perform at the extreme scale required for large data sets, the code must support distributed memory parallelism. This is because a single high-end server (a) does not have the memory to store dense matrix kernels of size more than a million, and (b) even if the memory is available, the execution time required on one node would be prohibitive. In addition, the code should support multi-threading on each distributed node so that many-core or GPU environments are utilized efficiently.

Our implementation uses the MPI+X paradigm, with heavy use of optimized LAPACK and BLAS multithreaded libraries. This allows easy transition from many-core to GPU architectures, especially since our iterative eigensolver PRIMME provides a GPU interface through MAGMA [47]. For this paper, OpenMP optimizations and tuning have been performed for the target KNL architecture on the Stampede2 supercomputer at TACC [44], funded by XSEDE [48]. This choice was prompted at the start of this project when KNLs provided much larger memory than the corresponding GPU architectures. With KNLs being phased out and GPUs having significantly larger memory, our work will soon be ported to newer architectures. Nevertheless, the high computation/communication ratio achievable on KNLs provides a stress test scenario for our

implementation. The code is written to support both single precision and double precision arithmetic. Mixed precision support is planned for the future.

There are three computational stages. The pre-processing stage is where the training and testing data is read to create the training and testing kernel matrices. The Truncated SVD stage uses the eigenvalue package PRIMME to compute a low-rank eigenspace that sufficiently approximates and regularizes the training kernel. The third stage solves the regression problems with the low-rank space and evaluates the score of its predictions.

4.1 Pre-processing

4.1.1 Matrix generated on-the-fly. With large scale data, the discriminating factor is the number of features F. If the number of features is small (say $F \leq 10$), the training data, $\mathbf{x}_T$, has low memory demands, $O(NF)$ numbers, and can remain in the memory of each node. The training kernel matrix, however, requires storage of $O(N^2/p)$ numbers, where p is the number of nodes in the parallel program. The matrix is used by the iterative solver in a matrix times a block of vectors multiplication kernel (hereafter called block matvec) with time complexity $O(2bN^2/p)$, where b is the size of the block. Therefore, when the number of samples N is too large to allow for storage of the entire matrix, or to do so we would need an excessive number of nodes, we can instead keep the training set in memory and recompute the matrix on the fly at every block matvec. This increases the block matvec complexity to $O(N^2F/p + 2bN^2/p)$ which for $F \ll b$ is a negligible increase. With this flexibility, the code can run in much smaller processor allocations, e.g., in smaller clusters that users may have available, and achieve higher utilization. Moreover, it allows for multiple hyperparameter spaces to be explored in parallel on different partitions of supercomputers.

For this case of on-the-fly generation of K, the pre-processing is straightforward. Depending on the file system, all nodes read the training data, or one node reads it to avoid disk contention and broadcasts it to all other nodes.

4.1.2 Matrix generated and stored. When the number of features is large (say 10^4 or 10^5), regenerating the matrix on the fly at every iteration becomes prohibitive. In this case the matrix must be created in the pre-processing stage. We follow the usual distribution, where node j is assigned the task to generate a set of matrix rows with indices denoted I_j. To compute its local part of the matrix, node j would need to compute $k(\mathbf{x}_T(I_j), \mathbf{x}_T)$. However, because of its large size, $\mathbf{x}_T$ may not be stored in local memory. We perform this in parallel with each node storing only two sections of the data; its own $\mathbf{x}_T(I_j)$ and a communicated $\mathbf{x}_T(I_m)$. Specifically, a pipelined ring communication starts by every node j sending its data block to node $j+1$ and receiving the data block from node $j-1$ in a non-blocking fashion. At the same time it computes its local $k(\mathbf{x}_T(I_j), \mathbf{x}_T(I_j)$ part of the matrix. When the new data block arrives, the node forwards it to $j+1$, posts another receive, and computes $k(\mathbf{x}_T(I_j), \mathbf{x}_T(I_{j-1})$. The algorithm continues for p steps, at which point all data blocks have been seen and recorded. Since communication time $O(NF)$ is far smaller than computation time $O(N^2F/p)$, we expect it to be completely overlapped.

There are two additional computational considerations. First, for very large F, the matrix generation cost is much larger than the time to solve the problem with the eigensolver. Thus, it would be useful if many more nodes were used to generate the matrix than to solve it. However, this would create different row distributions between the two stages. Second, the regression driver issues requests for the solution of multiple regression problems corresponding to different hyperparameters (often in the order of 100s). For each hyperparameter, a different kernel needs to be generated, which becomes computationally infeasible. One solution would be to transform the $K_{i,j}$ element of the current hyperparameter to the matrix element of the new hyperparameter, e.g., for the Gaussian kernel $K_{ij}^{\gamma_2} = \exp(\gamma_2/\gamma_1 \log K_{i,j}^{\gamma_1})$. We have found this to introduce too much floating point error, especially when storing the kernel in single precision arithmetic. A second alternative is the pre-processing step to store a copy of all distances $\|\mathbf{x}_{Ti} - \mathbf{x}_{Tj}\|$ in memory from which we can compute a kernel for any hyperparameter. This, however, doubles the memory requirements and does not address the problem of different number of nodes between matrix generation and eigensolver.

Our approach is to let the pre-processing stage write all the pairwise distances $\|\mathbf{x}_{Ti} - \mathbf{x}_{Tj}\|^2$ to a file and not compute the kernel matrix explicitly. A node can then read the file and broadcast the rows to the all the nodes in a new partition so that they can compute locally their kernel matrix. The approach has several benefits; (a) it allows different node distributions and partitions between preprocessing and solver; (b) does not introduce extra floating point error in matrix creation; (c) the computation of the matrix from distances takes less time than a matrix vector multiplication.

4.2 Truncated SVD

Applying a high performance iterative eigenvalue solver to compute the Truncated SVD (TSVD) of the kernel is not a ground breaking idea. Yet, most current ML packages use either a complete SVD decomposition through LAPACK or, the most advanced ones, have been using sequential Lanczos or Randomized SVD methods [35]. PRIMME, as one of the state-of-the-art eigensolvers, has been shown to scale well to massively parallel platforms and has been used to find eigenpairs of (sparse) matrices of dimension more than a billion [45]. It is therefore a natural choice to include as the TSVD solver in our package.

Among the several methods available in PRIMME, we choose to work with block GD+K. This method provides near optimal convergence rate, much faster than randomized SVD and at least as fast as LOBPCG, depending on basis size. Because our kernel matrices are dense, any reduction in the number of iterations translates directly to less computing time. The second key issue is the implementation of an efficient and scalable block matvec. The presence of a block matvec is critical to improve the per-node computational intensity and to reduce communication latency during matvec. At the same time, if the block is too large then convergence of the eigensolver deteriorates. We study the choice of block size in section 5. Denote the block matvec as KX where X is of size $N \times b$.

4.2.1 Matrix generated on-the-fly. As discussed in the previous paragraph the block sizes that are optimal for our problem are very small (often much smaller than the number of eigenvalues required). This implies that the block matvec cannot benefit from state-of-the-art algorithms for matrix-matrix multiplication [11, 30]. For

matrix-vector multiplication, the algorithm found in ScaLAPACK [8] uses a 2D cyclic partitioning of the matrix elements and achieves optimal weak scalability asymptotically. While such a choice of matvec should be provided in the software tool, it is not sufficient for a couple of reasons.

First, the last step of the 2D algorithm involves a global reduction which cannot be fully overlapped with computation. This may not affect the asymptotic weak scalability of the method but for smaller number of processors (a more practical regime for most users) a fully overlapped communication might give better results. Second, when the matrix is generated on-the-fly there is additional computation allowing even a simple 1D algorithm to fully overlap the communication and scale perfectly to larger numbers of processors. Therefore, we provide the alternative of a 1D approach that overlaps communication with both generation of matrix elements and their multiplication.

Each node is responsible for about N/p rows of K and stores the corresponding rows of X. In this case, we have the flexibility to generate the local matrix tile by tile, so that the tile fits in local memory and its size depends on the performance of SGEMM (the BLAS sequential matrix-matrix multiplication function).

Based on this 1D row distribution, a simple pipelined algorithm multiplies the parts of X that have arrived to the node while communicating in a non-blocking fashion the parts of X that will be needed next. We can perform the ring communication of the blocks $X(I_m)$ while overlapping the computation with only the first tile in the local set of rows. After the first tile has been computed, X is resident in its entirety on each node, and the rest of the tiles can be generated and multiplied without communication. With large number of features F this is sufficient.

A more scalable approach is to perform the multiplication of the local I_j rows in p groups of $I_1, \ldots, I_p$ columns, also through a pipelined ring communication of the block vectors $X(I_1), \ldots, X(I_p)$. The benefit is that we can overlap the entire generation of the matrix section $K(I_j, I_m)$ with communication.

It is relatively simple to model the performance of this algorithm, say for single precision arithmetic. At each step, every processor will eventually generate, regardless of tile size, a $N/p \times N/p$ section, multiply it with the Nb/p block of vectors, while communicating Nb/p words. Generating the section takes $O((N/p)^2(3F + 15))$ operations, where F is the number of features based on which the distance is calculated and 15 is an average number of flops required for computing the exponential. Assuming that this computation can achieve a rate of g_0 GFLOPs, the time to do this computation on a node is $O((N/p)^2(3F + 15)/g_0)$. Assuming SGEMM achieves a rate of g GFLOPs, the time to compute the block matvec of this section on a node is $O(2(N/p)^2 b/g)$. In our experiments we observed $g \approx 2g_0$, so we will use this relation in the model.

At the same time, we send $O(bN/p)$ single precision numbers, or $O(32bN/p)$ bits, to the next numbered processor and receive an equal amount from the previous one. Assuming that the network allows this ring communication to proceed without contention (as is the case on the fat tree topology of the Omni-Path network of Stampede2) then the communication time required for this step is $O(32bN/(pw))$, where w Gbps is the network bandwidth. The ratio of compute time over communication time is then greater than one, i.e., communication fully overlapped, if

$$p < \frac{Nw}{16g}\left(\frac{3F + 15}{b} + 1\right). \tag{1}$$

Notice that increasing the block size b beyond the one that achieves peak SGEMM performance and hides messaging overheads is reducing the scalability of the algorithm. On the other hand, keeping the block size too small reduces both SGEMM performance and achieved network bandwidth. The situation is further complicated by the increase in the actual number of matrix vector multiplications of the iterative solver. The SGEMM performance is studied in section 5. As a rule of thumb, block size should not be much larger than the smallest b for which communication is fully overlapped. With respect to scalability for $N = 10^6$, for the worst case where $F = 0$ and the ratio $w/g < 10^{-3}$, full overlapping of communication should be achieved up to 62 nodes.

4.2.2 Matrix in memory. For smaller number of nodes, the 1D method above continues to be competitive. For larger number of processors, a 2D algorithms as the one in ScaLAPACK can be used. The stored distances can be redistributed to the nodes in any desired distribution, including the required 2D block cyclical partitioning.

4.2.3 Initial guesses across hyperparameters. A significant advantage of PRIMME is that it can use multiple initial guesses to converge faster to the required eigenspace [45]. Since PRIMME is called repeatedly for a sequence of hyperparameters, the eigenvectors computed from a previous K^{γ_i} can be passed as initial guesses to the $K^{\gamma_{i+1}}$. The quality of these guesses depends on the closeness of γ_i to γ_{i+1}. Therefore, hyperparameters should be processed in a sorted order. However, also the rank k_i depends on each γ_i, with smaller hyperparameters requiring a computation of much smaller ranks. For this reason, we solve the corresponding eigenproblems such that larger hyperparameters that require a larger rank k are executed first. Thus, two successive kernel matrices will have the smallest distance and the computed rank of the previous matrix will always provide enough initial guesses for the following problem. We have followed this technique with the Gaussian kernel, where solving the kernels in order of decreasing γ yields between 30-40% speedup.

4.2.4 Choosing the rank of TSVD. Choosing the optimal rank k of the eigenspace is a central problem in statistics and machine learning. The role of the eigenspace is to act as a regularizer to the noise inherent in the data, which is typically unknown. Often, experts would have some idea on a lower bound of the smallest eigenvalue to be included in the eigenspace, e.g., $\sigma_k \geq \delta$. Other times, they would relate the σ_k to the norm of $\|K\|$, e.g., $\delta = \|K\|10^{-4}$. PRIMME has an option for a user provided function that implements any desired stopping criterion [45]. Our software implements the above criteria based on a user provided δ. This functionality also allows us to combine stopping PRIMME with the evaluation of a current low rank space. If at a certain point during the iteration, k eigenpairs have converged, we can evaluate them against a set of testing data (see next subsection) and decide whether more eigenpairs need to be computed. This functionality is currently under development but it has been provisioned in the design of the software.

4.3 Computing predictions and evaluations

Having obtained a low rank approximation of the training kernel $K_T \approx V\Sigma V^T$, we can solve the linear regression problems $\alpha = V\Sigma^{-1}V^TY$ where Y is the single vector of training responses in regression, while in classification Y has a number of columns equal to the number of classes. As V and Y are distributed by rows, the V^TY involves one global reduction, while other computations are performed in parallel. The predictions are formed as a matrix vector multiplication with the testing kernel matrix $y_{predict} = K_S\alpha$. The algorithm for this multiplication depends on the size of K_S. If the number of rows of K_S is small, it can be distributed by columns and the matvec performed as a set of inner products with a global reduction. If the number of rows is large, we can use the same algorithm as the block matvec with K_T.

Usually, the rank yielding the optimal evaluation is not known and therefore we would like to check the predictions for many or all ranks $i = 1, \ldots, k$. To avoid recomputing the low rank regression for each i, we can update $\alpha^{(i)}$ from the solution at the lower rank $\alpha^{(i)} = \alpha^{(i-1)} + \sigma_i^{-1}V_iV_i^TY$. The computation involves only level-1 BLAS for regression and level-2 BLAS for classification, as well as one synchronization point, but the time is constant for any $i = 1, \ldots, k$.

Finally, $y_{predict}$ needs to be evaluated against the user provided known responses for the testing data y_S. Such responses could be reserved at the start or could be part of a cross validation scheme. We currently provide functions that compute the MSE and correlation of $y_{predict}$ with y_S. For classification we provide a metric that discretizes the real values in $y_{predict}$ to identify the class it corresponds to and compares this with the classes in y_s. We plan to also implement the t-statistic and Sharpe metrics. Ultimately the evaluation function depends on the specific problem and therefore a user-defined function can be provided to do the evaluation.

5 EVALUATING PERFORMANCE

5.1 Matrix generated on the fly

When the matrix is generated on the fly during each matvec, we have to decide on the size of the tile and on the block size which determines not only the single node performance but also the performance of the pipelined matvec algorithm.

Figure 3 studies performance trade offs on a single KNL node. The top two subplots show the execution time and the TFLOPs achieved for a variety of block and tile sizes. Different blocks are plotted with different lines, and the block size is annotated at the ends of the line. The red vertical bars depict the time or TFLOPs of the tile generation, which is part of the matvec. Clearly, for small block sizes, most of the time is spent generating the tile, while large block sizes take more time but multiply more vectors and thus amortize the tile generation. The subplot on the right shows that the tile generation gets better performance for smaller tile sizes, while the SGEMM performance prefers tile sizes around 512 to 1280. Performance peaks with block size 128. The third subplot reports the matvec time per block vector multiplied. The best overall performance occurs with tile sizes 768 to 1280 and block sizes 128 and 256 respectively. Given similar matvec times, a smaller block should be preferred in order to reduce the overall runtime of the eigensolver.

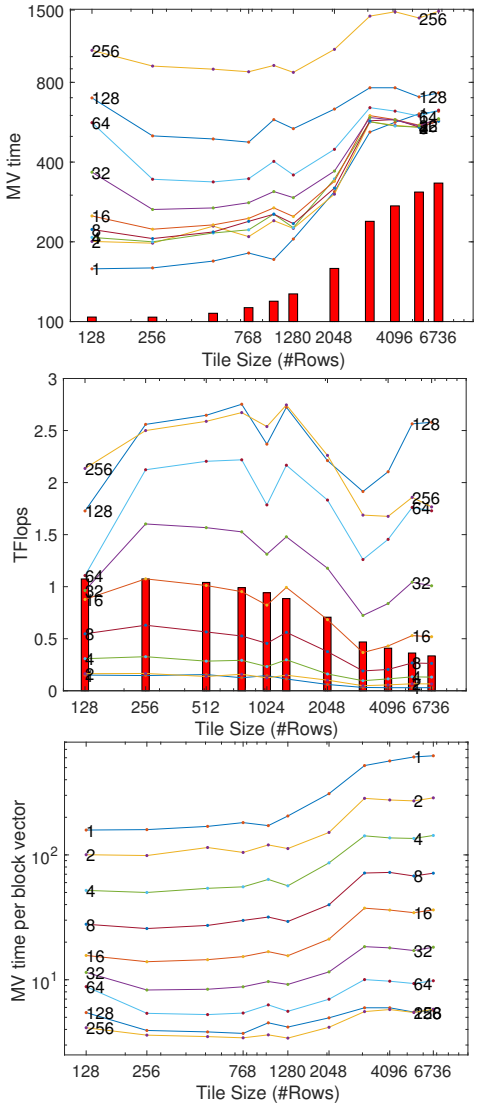

Figure 3: Matvec efficiency as a function of tile size and block size. The overlayed numerals indicate the block size, the red bars indicate the overhead of the tile generation.

Using the previous information on what tile size to use, we study next the parallel scalability of the pipelined on-the-fly algorithm for a data set from the stock data with matrix size 1996975. There are two features per data point ($F = 2$) and we test several block sizes. We use 58 threads on each node of Stampede 2, because more threads (up to the 68 cores per node) perform very inconsistently, increasing the overall runtime. Figure 4 shows only speedups ($T(1)/T(p)$) from 16 to 128 nodes of KNL. Smaller numbers of nodes show linear or superlinear speedups. The results show the best times out of 10 runs, as there were still considerable fluctuations of the single-node SGEMM performance between runs. As suggested by the model (1), we have linear scaling up to 64 nodes for all block sizes, and it's only for block sizes 16 and 32 that we

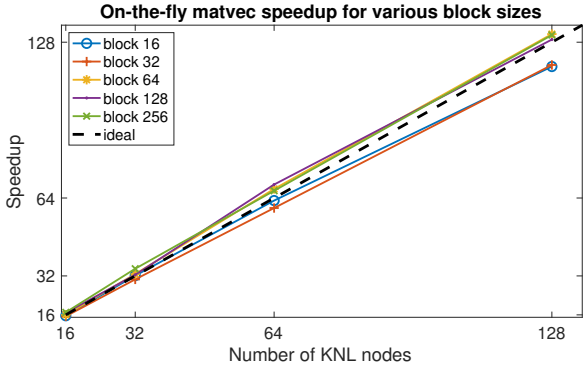

Figure 4: Matvec speedup with N=1996975.

see a small reduction in efficiency at 128 nodes. We believe that for many practical ML applications, 128 such powerful nodes is a reasonable number of resources.

Based on the above, we report timings of the regression and evaluation solver on the large stock data set in Table 1. For this data we compute 150 eigenvalues of a matrix close to 2 million. We use a large block size of 150, which reduces as eigenvalues converge. We see that matvec takes the vast majority of the time, but the total execution time is reasonable, allowing the solution of regression problems for a series of hyperparameters γ.

Stage		Time
pre-processing	Read data once	negligible
SVD Computation	# sing. values computed	150
	# outer iterations	15
	# single vector MVs	1304
	Total time	204.2 sec
	Matvec time	189.1 sec
	Tile generation	31.8 sec
	Ortho time	1.4 sec
	AllReduce time	10.3 sec
Low rank regression and evaluation	Regress $x = V_k \Sigma_k^{-1} V_k^T y$ Predict and evaluate x Total:	36.1 sec

Table 1: PRIMME execution statistics on 64 KNL nodes (4352 cores) for one of the data sets of the stock data with dimension 1996975, with $\gamma = 190$, which yields a more difficult eigenvalue problem, and with (variable) block size 150. The matrix is generated on the fly with a tile size of 1280. Low rank regression is performed incrementally per rank.

5.2 Matrix in memory

For the ImageNet dataset, the number of features (150528) and the size of the matrix (1281167) make the regeneration of the matrix from the training data prohibitive. Therefore, we use the methods in the software that distributes the entire matrix over the nodes. A minimum number of 128 Stampede2 nodes were necessary to store the matrix. For each new hyperparameter in the optimization, the inter-distances between points are read from disc and the kernel is recomputed. Next, we study the timings for each stage of this process as shown in Table 2.

In the pre-processing stage, we first read the training and testing data and distribute it over the nodes. Then, we run the pipelined tile generation algorithm to compute the local part of the matrix, and finally, write the local matrix of each node to the parallel file system. Clearly, the time is dominated by the generation of the matrix, while the corresponding communication is completely overlapped. Reading the training data and writing the resulting matrix tiles takes less than 10% of the pre-processing time. Because one node reads from disk and redistributes, this I/O time is not scalable as it is not reduced by using more nodes, but it is a cost that any ML method must occur on this machine. However, this operation trivially scales on hardware platforms where each node has a local disk that can hold the desired partition for I/O.

Stage		Time
Pre-processing	Read and distribute data	1439 sec
	Pipelined tile generation	27050 sec
	Communication overhead	<3 sec
	Write matrix tiles to PFS	410 sec
	Total:	29305 sec
SVD computation	# sing. values computed	19000
	# outer iterations	336
	# single vector MVs	73863
	Read matrix from PFS	404.4 sec
	Total PRIMME time	2665.7 sec
	Matvec time	1686.9 sec
	Ortho time	652.8 sec
	AllReduce time	416.8 sec
Low rank regression and evaluation	Regress $V_k \Sigma_k^{-1} V_k^T Y$ and evaluate per rank (size(Y,2)=1000)	9.25 sec

Table 2: PRIMME execution statistics on 128 KNL nodes (8704 cores) for the entire ImageNet of dimension 1281167, with $\gamma = 10^{-5}$ and block size = 256. The number of classes is 1000 so low rank regression solves Y with 1000 columns. This is performed incrementally per rank. The evaluation was stopped before rank 19000 was reached.

Once the kernel matrix has been generated, our optimization pipeline executes a sequence of calls to PRIMME. The cost of the hyperparameter optimization part of the program is negligible. Using the Gaussian kernel for the ImageNet data set gave the best results for hyperparameters $\gamma \approx 10^{-5}$. However, the very small spectral decay of this kernel required the computation of a very large number of eigenpairs, as shown in Table 2. This number of eigenpairs is one of the largest reported in the iterative eigenmethods literature, and the only one we are aware of that works with a dense matrix of size of more than a million. For comparison, the EigenExa code is a Petascale dense eigensolver and was recently reported to have diagonalized the first million size dense matrix on the K supercomputer in 3464 seconds using 663,552 cores and achieving 1.7 PFLOPs [24]. Instead, by using an iterative method we can compute 1.5%

of the spectrum in 2666 seconds, on 8704 cores of Stampede 2[44], attaining around 100 TFLOPs for the entire PRIMME run.

More importantly, we do not expect regression problems to require the computation of so many eigenvalues. Machine learning users would want to use a kernel that displays a fast spectral decay for their problem, requiring just a small number of eigenvalues that can be solved far more efficiently (as in Table 1). What the experiment shows, however, is that our software enables this investigation even in extreme cases in an efficient and robust way.

Looking closer at the timings of Table 2, we see that for every eigenvalue problem 404.4 seconds are spent reading the distances to create the kernel locally. Although this is a small part of this problem, it would dominate the execution time for kernels that only require 10-100 eigenpairs. In that case, it is advisable to increase the number of nodes to allow a copy of the distances to be stored in memory so that the matrix is recomputed for every new hyperparameter. The smaller memory footprint of the eigenvectors in that case will moderate the increase in the number of nodes.

PRIMME timings show the computation of a single eigenproblem to take less than an hour. This in itself is remarkable. Orthogonalization takes about 25% of the total execution time. This is expected as its complexity grows as $O(k^2N)$ where k is the number of eigenvalues. To contrast, orthogonalization for our stock data took less than 0.5% of total time. Randomized methods that avoid orthogonalization have been proposed [37] but their robust implementation is still under investigation and the potential benefits are limited. The AllReduce time, which is needed in orthogonalization and in PRIMME inner products, took a 416.8 seconds which is reasonable considering the 19,000 eigenvectors.

We also studied the weak scalability of our pipelined matvec algorithm for this case where the matrix is not computed on the fly. Since this involves less operations to overlap communication and since the problem size is smaller than the stock data, scalability on this problem is more challenging. We have tested weak scalability in a way that extends also to the rectangular matvec used in the evaluation step. We keep the local matrix stored on one of the 128 nodes constant (i.e., of size $N/128 \times N$), replicate it on p nodes, and multiply a matrix of size $pN/128 \times N$ by a vector of size N distributed over p nodes. Figure 5 shows relatively flat timings with the number of nodes. As before we notice fluctuations in the performance of SGEMM. When computing the achieved SGEMM TFLOPs for each run and normalizing the times for the same SGEMM performance, the weak scalability becomes even flatter. This implies that any performance issues are not due to the algorithm but to the multithreaded MKL library on KNL.

The last stage is the low rank regression and evaluation for each wanted rank. Notice that because there are 1000 classes in ImageNet, regression has to solve 1000 linear systems per rank. This is performed with level-2 BLAS and we update for each rank. If all 19000 ranks were needed to be evaluated, this stage would take more than 175000 seconds. However, by monitoring the accuracy that each rank produces, we can stop the evaluations early when evaluation scores cease to increase.

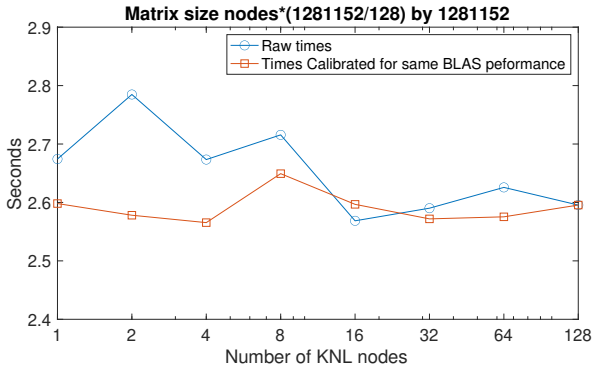

Figure 5: Pipelined matvec weak scalability with N=1281167.

6 MODEL ACCURACY

To show the merit of developing software that combines a high performance, large scale kernel regression optimization with a boosting optimizer, we apply it on the equity return forecasting problem and validate its effectiveness by comparing the obtained accuracy with some widely used machine learning baselines. The goal is not to compare computing performance but to show that making kernel methods computationally feasible adds a competitive tool to the arsenal of machine learning methods.

Universe	50	1000	3000
Train	35108	700253	2004030
Test	2848	56808	163652
Kernel	4.6GB	1.78TB	14.6TB

Table 3: The training/testing datasets sample sizes and estimated kernel memory footprint for different universes.

6.1 Methodology

In the equity return problem, we are usually interested in stocks within a specific universe (e.g., SP500 or Russell3000) and denote the number of stocks in the universe as n. We assume the equity market proceeds in periods. Let $\mathbf{y}_{i,t} \in \mathbf{R}$ be the return of stock i at the t-th period, and $\mathbf{y}_t = (y_{1,t}, \ldots, y_{n,t}) \in \mathbf{R}^d$. Our goal is to forecast $\mathbf{y}_t$ based on all information available up to period $t-1$.

In our experiments, we use seven years of equity data from the US market. We use three consecutive years of data for training and the following three months for testing. For each test year, we examine three different universes. Each universe consists of Top N stocks in trading volume (during the training period) with $n \in \{50, 1000, 3000\}$. Table 3 shows the sample sizes and estimated kernel sizes of different universes. We retrain the model for each test year and each universe. We use past 1-day returns and past 3-months dollar volume as features and next 1-day returns as the response (i.e. each equity market period consists of one day). All returns are the "log-transform" of all open-to-open returns. Our baselines include linear regression ("Lasso" [46] and "Ridge" [23]), GBRT [15], MLP [40] and LSTM [22]. We apply correlation as the metric because it is more suitable than MSE in our setting.

Universe	50				1000				3000			
Year	2015	2016	2017	2018	2015	2016	2017	2018	2015	2016	2017	2018
Lasso	0.0168	0.0471	0.0307	0.0171	0.0161	0.0160	0.0336	0.0136	0.0186	0.0213	0.0133	0.0140
Ridge	0.0104	0.0471	0.0065	0.0301	0.0044	0.0156	0.0301	0.0006	0.0152	0.0152	0.0115	0.0111
GBRT	0.0576	0.0703	0.0476	0.0588	0.0261	0.0180	**0.0432**	0.0285	0.0194	**0.0362**	0.0162	0.0135
MLP	0.0311	0.0267	0.0185	0.0356	0.0190	0.0067	0.0179	0.0117	0.0175	0.0309	0.0154	0.0176
LSTM	0.0258	0.0371	0.0236	0.0493	0.0138	0.0037	0.0104	0.0155	0.0162	0.0147	0.0143	0.0152
KPCR	0.0563	0.0602	0.0565	0.0673	0.0344	0.0183	0.0055	0.0253	0.0169	0.0274	0.0175	0.0183
B-KPCR	**0.0929**	**0.0944**	**0.1035**	**0.1001**	**0.0356**	**0.0249**	0.0069	**0.0296**	**0.0231**	0.0310	**0.0218**	**0.0220**

Table 4: A summary of accuracy results for equity return forecasting problem. Results are presented in testing correlations. Bold numbers denote the highest accuracy for each year. B-KPCR denotes the KPCR with boosting method.

In addition, the equity return forecasting problem usually has very small signal-to-noise. For example, a model for predicting the next 1-day return can start profiting when its r^2 score is only 2×10^{-4} (i.e., 2 basis points). This means a boosting approach that aggregates forecasts from multiple models is needed.

6.2 Results

Table 4 shows the accuracy results of three universes. Firstly, linear regressions perform worst among benchmarks as expected. Secondly, deep learning models also do not achieve the best accuracy despite being more powerful than linear models. This is because, compared to other methods, deep learning models have complex structures with many more hyperparameters to tune (e.g. layer number of the model, the type and parameters of each layer, etc), while each hyperparameter is expensive to evaluate (re-train the model and test the accuracy on the dataset). Finally, KPCR consistently outperforms all other models on most datasets, and the boosting method is able to dramatically improve the accuracy of KPCR. This highlights the importance of our efficient KPCR solver because many KPCR instances with different hyperparameters need to be solved for the boosting method.

7 CONCLUSION AND FUTURE DIRECTIONS

There is an important set of downstream applications that depend on the solution of the Kernel Principal Component Regression for datasets of size more than a million observations. Yet, current software packages cannot scale to this size.

In this work, we designed and implemented a software solution that includes a front-end pipeline that handles the hyperparameter optimization and check-pointing, and a high performance backbone based on an efficient block matrix-vector multiplication and a state-of-the-art eigensolver. All algorithms are implemented to achieve high single-node performance and to overlap as much communication as possible. The software can run even when the matrix cannot be stored in its entirety.

As a feasibility study we were able to apply KPCR for the first time on the entire ImageNet dataset with an adverse kernel that stress-tested the eigensolver capabilities. Experiments on an even larger asset pricing dataset showed that with proper hyperparameter optimization KPCR outperforms other models.

Our short term plans are to port the code on a variety of modern multi-core and GPU architectures and to optimize for performance. Based on these, functionality will be extended to include Kernel Ridge regression and to automate the selection of parameters such as rank and the Ridge parameter. Finally, a general user-interface needs to be developed before our GitHub code is released to the community.

8 ACKNOWLEDGMENTS

This work was supported by National Science Foundation grants IIS-2008557 and NSCI-1835821. This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant numbers ACI-1540931 and ACI-1663578.

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/ Software available from tensorflow.org.

[2] Sina Alemohammad, Zichao Wang, Randall Balestriero, and Richard Baraniuk. 2020. The recurrent neural tangent kernel. *arXiv preprint arXiv:2006.10246* (2020).

[3] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Du Croz, A. Greenbaum, S. Hammarling, A McKenney, S. Ostrouchov, and D. Sorensen. 1992. *LAPACK User's Guide.* SIAM.

[4] Haim Avron, Kenneth L Clarkson, and David P Woodruff. 2017. Faster kernel ridge regression using sketching and preconditioning. *SIAM J. Matrix Anal. Appl.* 38, 4 (2017), 1116–1138.

[5] Z. Bai, J. Demmel, J. Dongarra, A. Ruhe, and H. van der Vorst (Eds.). 2000. *Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide.* Software Environ. Tools 11, SIAM, Philadelphia.

[6] C. G. Baker, U. L. Hetmaniuk, R. B. Lehoucq, and H. K. Thornquist. 2009. Anasazi Software for the Numerical Solution of Large-Scale Eigenvalue Problems. *ACM Trans. Math. Software* 36, 3 (2009). http://trilinos.sandia.gov/packages/anasazi.

[7] R. Barrett, M. Berry, T.F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst. 1994. *Templates for the solution of linear systems: Building blocks for iterative methods.* SIAM, Philadelphia, PA.

[8] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. 1997. *ScaLAPACK User's Guide.* SIAM.

[9] Serhat S Bucak, Rong Jin, and Anil K Jain. 2013. Multiple kernel learning for visual object recognition: A review. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 36, 7 (2013), 1354–1369.

[10] Benjamin Charlier, Jean Feydy, Joan Alexis Glaunès, François-David Collin, and Ghislain Durif. 2021. Kernel Operations on the GPU, with Autodiff, without Memory Overflows. *Journal of Machine Learning Research* 22, 74 (2021), 1–6. http://jmlr.org/papers/v22/20-275.html

[11] James Demmel, David Eliahu, Armando Fox, Shoaib Kamil, Benjamin Lipshitz, Oded Schwartz, and Omer Spillinger. 2013. Communication-Optimal Parallel Recursive Rectangular Matrix Multiplication. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. 261–272. https://doi.org/10.1109/IPDPS.2013.80

[12] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 248–255.

[13] Lee H Dicker, Dean P Foster, and Daniel Hsu. 2017. Kernel ridge vs. principal component regression: Minimax bounds and the qualification of regularization operators. *Electronic Journal of Statistics* 11, 1 (2017), 1022–1047.

[14] Petros Drineas, Michael W Mahoney, and Nello Cristianini. 2005. On the Nyström Method for Approximating a Gram Matrix for Improved Kernel-Based Learning. *journal of machine learning research* 6, 12 (2005).

[15] Jerome H Friedman. 2001. Greedy function approximation: a gradient boosting machine. *Annals of statistics* (2001), 1189–1232.

[16] Andreas Frommer and Peter Maass. 1999. Fast CG-based methods for Tikhonov–Phillips regularization. *SIAM Journal on Scientific Computing* 20, 5 (1999), 1831–1850.

[17] Jacob R Gardner, Geoff Pleiss, David Bindel, Kilian Q Weinberger, and Andrew Gordon Wilson. 2018. GPyTorch: Blackbox Matrix-Matrix Gaussian Process Inference with GPU Acceleration. In *Advances in Neural Information Processing Systems*.

[18] Mustansar Ali Ghazanfar, Adam Prügel-Bennett, and Sandor Szedmak. 2012. Kernel-mapping recommender system algorithms. *Information Sciences* 208 (2012), 81–104.

[19] Shihao Gu, Bryan Kelly, and Dacheng Xiu. 2020. Empirical asset pricing via machine learning. *The Review of Financial Studies* 33, 5 (2020), 2223–2273.

[20] Nathan Halko, Per-Gunnar Martinsson, and Joel A Tropp. 2011. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM review* 53, 2 (2011), 217–288.

[21] Vicente Hernandez, Jose E. Roman, and Vicente Vidal. 2005. SLEPc: A scalable and flexible toolkit for the solution of eigenvalue problems. *ACM Trans. Math. Software* 31, 3 (2005), 351–362.

[22] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[23] Arthur E Hoerl and Robert W Kennard. 1970. Ridge regression: Biased estimation for nonorthogonal problems. *Technometrics* 12, 1 (1970), 55–67.

[24] Toshiyuki Imamura, Susumu Yamada, and Masahiko Machida. 2011. Development of a high performance eigensolver on the petascale next generation supercomputer system. *Progress in Nuclear Science and Technology* 2 (2011), 643–650.

[25] Arthur Jacot, Franck Gabriel, and Clément Hongler. 2018. Neural tangent kernel: Convergence and generalization in neural networks. *Advances in neural information processing systems* 31 (2018).

[26] V Nisha Jenipher and S Radhika. 2021. SVM Kernel Methods with Data Normalization for Lung Cancer Survivability Prediction Application. In *2021 Third International Conference on Intelligent Communication Technologies and Virtual Mobile Networks (ICICV)*. IEEE, 1294–1299.

[27] Taichi Joutou and Keiji Yanai. 2009. A food image recognition system with multiple kernel learning. In *2009 16th IEEE International Conference on Image Processing (ICIP)*. IEEE, 285–288.

[28] Shubhra Kanti Karmaker, Md Mahadi Hassan, Micah J Smith, Lei Xu, Chengxiang Zhai, and Kalyan Veeramachaneni. 2021. Automl to date and beyond: Challenges and opportunities. *ACM Computing Surveys (CSUR)* 54, 8 (2021), 1–36.

[29] Marius Kloft, Ulf Brefeld, Sören Sonnenburg, and Alexander Zien. 2011. Lp-norm multiple kernel learning. *The Journal of Machine Learning Research* 12 (2011), 953–997.

[30] Grzegorz Kwasniewski, Marko Kabić, Maciej Besta, Joost VandeVondele, Raffaele Solcà, and Torsten Hoefler. 2019. Red-Blue Pebbling Revisited: Near Optimal Parallel Matrix-Matrix Multiplication *(SC '19)*. Association for Computing Machinery, New York, NY, USA, Article 24, 22 pages. https://doi.org/10.1145/3295500.3356181

[31] Ivano Lauriola and Fabio Aiolli. 2020. MKLpy: a python-based framework for Multiple Kernel Learning. *arXiv preprint arXiv:2007.09982* (2020).

[32] R. B. Lehoucq, D. C. Sorensen, and C. Yang. 1998. *ARPACK User's guide: Solution of Large Scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods*. SIAM, Philadelphia, PA.

[33] Xin Li, Mengyue Wang, and T-P Liang. 2014. A multi-theoretical kernel-based approach to social network-based recommendation. *Decision Support Systems* 65 (2014), 95–104.

[34] William B. March, Bo Xiao, Chenhan D. Yu, and George Biros. 2016. ASKIT: An Efficient, Parallel Library for High-Dimensional Kernel Summations. *SIAM Journal on Scientific Computing* 38, 5 (2016), S720–S749. https://doi.org/10.1137/15M1026468 arXiv:https://doi.org/10.1137/15M1026468

[35] Per-Gunnar Martinsson. 2019. Randomized methods for matrix computations. *The Mathematics of Data* 25, 4 (2019), 187–231.

[36] Ashin Mukherjee and Ji Zhu. 2011. Reduced rank ridge regression and its kernel extensions. *Statistical analysis and data mining: the ASA data science journal* 4, 6 (2011), 612–622.

[37] Yuji Nakatsukasa and Joel A. Tropp. 2021. Fast & Accurate Randomized Algorithms for Linear Systems and Eigenvalue Problems. https://doi.org/10.48550/ARXIV.2111.00113

[38] Roman Novak, Lechao Xiao, Jiri Hron, Jaehoon Lee, Alexander A. Alemi, Jascha Sohl-Dickstein, and Samuel S. Schoenholz. 2020. Neural Tangents: Fast and Easy Infinite Neural Networks in Python. In *International Conference on Learning Representations*. https://github.com/google/neural-tangents

[39] 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. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[40] Marius-Constantin Popescu, Valentina E Balas, Liliana Perescu-Popescu, and Nikos Mastorakis. 2009. Multilayer perceptron and neural networks. *WSEAS Transactions on Circuits and Systems* 8, 7 (2009), 579–588.

[41] Ali Rahimi and Benjamin Recht. 2007. Random features for large-scale kernel machines. *Advances in neural information processing systems* 20 (2007).

[42] Bernhard Schölkopf, Alexander J Smola, Francis Bach, et al. 2002. *Learning with kernels: support vector machines, regularization, optimization, and beyond*. MIT press.

[43] Hyunseok Seo, Masoud Badiei Khuzani, Varun Vasudevan, Charles Huang, Hongyi Ren, Ruoxiu Xiao, Xiao Jia, and Lei Xing. 2020. Machine learning techniques for biomedical image segmentation: an overview of technical aspects and introduction to state-of-art applications. *Medical physics* 47, 5 (2020), e148–e167.

[44] Dan Stanzione, Bill Barth, Niall Gaffney, Kelly Gaither, Chris Hempel, Tommy Minyard, Susan Mehringer, Eric Wernert, H Tufo, D Panda, et al. 2017. Stampede 2: The evolution of an xsede supercomputer. In *Proceedings of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact*. 1–8.

[45] Andreas Stathopoulos and James R. McCombs. 2010. PRIMME: Preconditioned Iterative Multimethod Eigensolver—Methods and Software Description. *ACM Trans. Math. Softw.* 37, 2, Article 21 (apr 2010), 30 pages. https://doi.org/10.1145/1731022.1731031

[46] Robert Tibshirani. 1996. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B (Methodological)* 58, 1 (1996), 267–288.

[47] Stanimire Tomov, Rajib Nath, Peng Du, and Jack Dongarra. 2011. MAGMA Users' Guide. *ICL, UTK (November 2009)* (2011).

[48] John Towns, Timothy Cockerill, Maytal Dahan, Ian Foster, Kelly Gaither, Andrew Grimshaw, Victor Hazlewood, Scott Lathrop, Dave Lifka, Gregory D Peterson, et al. 2014. XSEDE: accelerating scientific discovery. *Computing in science & engineering* 16, 5 (2014), 62–74.

[49] Ke Alexander Wang, Geoff Pleiss, Jacob R Gardner, Stephen Tyree, Kilian Q. Weinberger, and Andre Gordon Wilson. 2019. Exact Gaussian Processes on a Million Data Points. *33rd Conference on Neural Information Processing Systems, NeurIPS* (2019).

[50] Li Wang and Ji Zhu. 2010. Financial market forecasting using a two-step kernel learning method for the support vector regression. *Annals of Operations Research* 174, 1 (2010), 103–120.

[51] Christopher Williams and Matthias Seeger. 2000. Using the Nyström method to speed up kernel machines. *Advances in neural information processing systems* 13 (2000).

[52] Qiong Wu, Christopher G Brinton, Zheng Zhang, Andrea Pizzoferrato, Zhenming Liu, and Mihai Cucuringu. 2021. Equity2vec: End-to-end deep learning framework for cross-sectional asset pricing. In *Proceedings of the Second ACM International Conference on AI in Finance*. 1–9.

[53] Qiong Wu, Felix M Wong, Yanhua Li, Zhenming Liu, and Varun Kanade. 2020. Adaptive reduced rank regression. *Advances in Neural Information Processing Systems* 33 (2020), 4103–4114.

[54] Yuan Yao, Lorenzo Rosasco, and Andrea Caponnetto. 2007. On early stopping in gradient descent learning. *Constructive Approximation* 26, 2 (2007), 289–315.

[55] Chenyun Yu and Ka Chi Lam. 2014. Applying multiple kernel learning and support vector machine for solving the multicriteria and nonlinearity problems of traffic flow prediction. *Journal of Advanced Transportation* 48, 3 (2014), 250–271.

[56] Yuchen Zhang, John Duchi, and Martin Wainwright. 2013. Divide and conquer kernel ridge regression. In *Conference on learning theory*. PMLR, 592–617.

[57] Chunlin Zhao, Chongxun Zheng, Min Zhao, Yaling Tu, and Jianping Liu. 2011. Multivariate autoregressive models and kernel learning algorithms for classifying driving mental fatigue based on electroencephalographic. *Expert Systems with Applications* 38, 3 (2011), 1859–1865.