Data-driven Performance Modeling of Linear Solvers for Sparse Matrices

Jae-Seung Yeom[†], Jayaraman J. Thiagarajan[†], Abhinav Bhatele[†], Greg Bronevetsky[*], Tzanio Kolev[†]

[†]Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, California 94551 USA
[*]Google, Inc., Mountain View, California 94043 USA
Email: [†]{yeom2, jayaramanthi1, bhatele, kolev1}@llnl.gov, [*]bronevet@google.com

Abstract—**Performance of scientific codes is increasingly dependent on the input problem, its data representation and the underlying hardware with the increase in code and architectural complexity. This makes the task of identifying the fastest algorithm for solving a problem more challenging. In this paper, we focus on modeling the performance of numerical libraries used to solve a sparse linear system. We use machine learning to develop data-driven models of performance of linear solver implementations. These models can be used by a novice user to identify the fastest preconditioner and solver for a given input matrix. We use a variety of features that represent the matrix structure, numerical properties of the matrix and the underlying mesh or input problem as input to the model. We model the performance of nine linear solvers and thirteen preconditioners available in Trilinos using 1240 sparse matrices obtained from two different sources. Our prediction models perform significantly better than a blind classifier and black-box SVM and k-NN classifiers.**

I. Introduction

Linear algebra and specifically, sparse linear systems of equations are used in a variety of computational science and engineering (CSE) codes. The sparse linear solve typically accounts for a majority of the execution time in such codes making it crucial to optimize its performance. In addition, performance of a linear solver and thus, its parent CSE code, is increasingly dependent on the input problem, its matrix representation and the underlying hardware. This makes the task of identifying the fastest algorithm for solving a linear system more challenging. Several preconditioners and solvers have been designed and implemented in many numerical libraries and packages [1], [2], [3], [4], which also adds complexity to the challenge.

In this paper, we use machine learning techniques to help a novice user in identifying the fastest preconditioner and linear solver (PC-LS) for a given input problem or matrix. In order to do this, we create performance models of the implementations of different numerical algorithms for solving sparse linear systems. The models are used to predict the fastest PC-LS combination for a matrix not seen by the model before. Data-driven models refer to the use of matrix properties as input features. These features describe different characteristics of the input problem or formulated matrix: 1. matrix structure, 2. numerical properties of the matrix, and 3. the underlying mesh or problem from which the matrix was generated.

We have developed a three-step approach to understanding sparse matrices and predicting the best PC-LS choice for different matrices. The first two steps are used to train the prediction model. First, we use regression analysis to find a subset of features that are useful for predicting execution time. There are two benefits of identifying a subset of critical features: 1. a reduced feature set will control the model complexity during model learning, and 2. it is computationally efficient to calculate only a subset of features for the test cases. The second step is to build a *performance vector space* which is similar to building dense vectors for words in natural language modeling [5]. This is better than using a black box classifier because of the limited size of training data and the highly unbalanced distribution of labels. In the third step, when testing the model, we use the performance vector space from the second step and project the unseen test sample onto it. Then, we employ a k-Nearest Neighbor (k-NN) classifier for predicting the best PC-LS choice.

The study is performed in the context of the Trilinos framework [1]. However, the approach can be easily applied to other numerical packages and libraries. We model the performance of nine linear solvers and thirteen preconditioners available in Trilinos. Sparse matrices used for training and testing the model are obtained from two sources: a finite element discretization library called MFEM [6] and the University of Florida (UFL) sparse matrix collection [7]. The MFEM dataset consists of 879 sparse matrices that approximate solutions of four arbitrarily chosen physics problems using 31 mesh geometries at various discretization orders and mesh refinement levels. We use 361 sparse matrices from the UFL dataset. In summary, we make the following novel contributions:

- A three-step approach to understand sparse matrices and develop data-driven performance prediction models that can identify the best preconditioner and linear solver for an unseen input matrix.
- Identify and explore 35 input features in the prediction model and use regression analysis to identify those that are useful in predicting the execution time of linear solvers for sparse matrices.
- Combining tools from statistical learning and natural language processing, build a performance vector space and use it with a k-NN classifier to identify the best choice of PC-LS for a test matrix.

- Evaluate the generated model on two datasets of 1240 sparse matrices and show that our prediction model has significantly higher prediction accuracy than a blind classifier and black-box support vector machine (SVM) and k-NN classifiers.

II. Related work

Prior work on modeling the performance of high performance computing (HPC) applications has focused on general modeling techniques and application-specific models. Kerbyson, Hoisie et al. developed application-specific performance models for complex applications, such as the Monte Carlo N-Particle (MCNP) simulation [8] and SAIC's Adaptive Grid Eulerian (SAGE) hydrocode [9], [10]. More recently, Tallent and Hoisie built upon this work to create the Palm model generation tool that helps developers model their own applications [11]. Their work has demonstrated the promise of application-specific modeling and the importance of including basic properties of the input problem (e.g. matrix size or the number of non-zeroes) in the model. Our work is complementary to these efforts and demonstrates the potential of models that capture application behavior and its dependence on input problems more comprehensively.

Gahvari et al. have developed performance models [12], [13], [14] of the algebraic multigrid implementation in the Hypre library [15]. Since their focus is on parallel performance, their models take into account factors such as the number of processes and threads being used, and metrics that capture CPU and data transfer cost (e.g. the number of unknowns in the linear system). Bhatele et al. have developed performance prediction models for several application classes and used them to predict performance on future exascale platforms [16].

In [17], the authors adopt a variety of classification techniques to select the optimal solver for a given linear system and study their generalization behavior. The importance of feature selection in such formulations is analyzed in [18]. In order to reduce the programming burden of developing numerical software, the Lighthouse taxonomy system recommends appropriate methods for users' needs from existing linear algebra libraries, such as LAPACK [2], PETSc [3], and SLEPc [4], and provides usage examples [19]. For instance, it helps users to identify well-performing eigensolver algorithms for their problem types via guided searches using a web-interface or by direct analysis on problems represented as matrix data. In addition, it offers implementation templates in Fortran or C for the identified algorithms. In case of a dense linear problem, it further takes advantage of an existing optimization technique for code generation [20].

To incorporate PETSc and SLEPc into the framework, Norris et al. evaluate the performance of various Krylov subspace methods in PETSc and eigensolvers in SLEPc on a subset of sparse matrices from the University of Florida collection [19], [21]. This work is closest to our research in that they rely on machine learning to build prediction models using support vector machine [22] and decision trees. Jain and Bhatele have also used supervised learning to model communication performance of parallel applications based on network hardware counters data [23], [24].

Existing approaches to select the optimal solver for a given linear system employ blackbox classification algorithms on a predetermined set of features characterizing the properties of the linear system. There are two crucial shortcomings with these approaches. First, the model training does not take into account the relative performance of different solver choices, and hence the problem of distinguishing a large number of classes becomes extremely challenging due to the limited dataset sizes and the highly unbalanced distribution of labels. We address this issue by constructing a performance vector space, which effectively incorporates the relative importances of solver choices, for prediction tasks. Second, the metrics typically used to evaluate the model, such as the Receiver Operating Characteristic (ROC) curves, can be highly misleading since they do not clearly indicate how suboptimal the chosen solver is when the prediction is wrong. In contrast, our metric is the actual runtime of the chosen solver for a particular linear system. As we demonstrate in our results, standard classifiers such as the SVM perform significantly poorly (high runtime) in few of the cases where their predicted label is not the optimal one. Surprisingly, blindly choosing the solver that is optimal for most of the training data samples performs better than the trained classifiers under our new metric.

III. Problem domain

Many CSE problems consist of compute kernels that solve sparse linear systems repeatedly. In this study, we concentrate on understanding the dependence of performance of sparse linear solver kernels on the input matrix data. In particular, our focus is on execution time rather than convergence behavior, which has been studied extensively in the mathematics community. In this section, we describe the sparse matrices used in this paper and the input features that we derive from these matrices for studying and modeling the behavior of preconditioner and linear solver (PC-LS) choices. We also describe the setup for collecting empirical data for this study.

A. Sparse matrix data

We use sparse matrices from two sources: a finite element discretization library called MFEM [6] and the University of Florida sparse matrix collection [7]. We label the former set of matrices as **MFEM** and the latter as **UFL** in the rest of this paper. We produce the MFEM dataset by varying the discretization order and refinement level for each of four arbitrarily chosen problems. These problems correspond to discretizations in the H^1 (nodal), vector H^1, $H(curl)$ (edge) and $H(div)$ (face) finite element spaces, which are the building blocks for a wide variety of PDE-based CSE applications. The specific problems we consider are:

P1 Laplace equation, $\Delta u = f$

P2 the elasticity problem, $-div(\sigma(\mathrm{u})) = f$

P3 electromagnetic diffusion problem, $curl\ curl\ E + E = f$

P4 the flux-based diffusion system, $-grad(divF) + F = f$

The MFEM dataset consists of 879 symmetric positive-definite matrices that exhibit the typical challenges with matrices in PDE-based simulations. The matrices that we consider are representative of practical scenarios that appear in the modeling of heat transfer, structural mechanics, accelerator design, radiation diffusion flow, and other simulations.

In the case of symmetric positive-definite matrices, the condition number is the ratio of the maximal to minimal eigenvalue of the matrix. While for all problems in MFEM this ratio grows at the same rate (e.g. with respect to N), the other eigenvalues in the spectrum of the matrix can also play an important role in determining the performance of the linear solver. From that perspective, P1 is the simplest problem since it is "near-nullspace", the eigenvectors corresponding to almost zero eigenvalues, and consists of only one vector. P2, on the other hand, has four vectors in 2D and six vectors in 3D. The problems P3 and P4 have much larger near-nullspace of dimension comparable to the global size of the problem, which is usually a major challenge to linear solvers.

The UFL dataset consists of various types of matrices collected from diverse fields in science and engineering. We use 361 square non-Hermitian matrices that are small enough to fit in memory of a single compute node for this study. This set includes both symmetric and non-symmetric matrices, and both positive/negative definite and non-definite ones.

B. Candidate input features

Table I shows the candidate features that can be derived from input matrices and used as input to the prediction model. The features are categorized by their generality and by the computational cost to derive them. Features color-coded green are referred to as *Basic* features and computing them has a low to medium time complexity – $O(N_{row})$ for many, $O({N_{row}}^2)$ for computing $||A||_F$, and $O(NNZ)$ for others.

The second group, called *Advanced* features and color-coded red, has features that are significantly expensive to compute and not practical as inputs to the model. However, we still use them in the paper to understand how we can improve the prediction by exposing information that is crucial but potentially missing in simpler features. Discovery of a highly useful but computationally expensive feature can encourage us to investigate methods to obtain a reasonable approximation in less time. For instance, we take advantage of domain-specific knowledge to compute inexpensive alternatives to the condition number for the MFEM dataset.

The third group consists of *Domain-specific* features (color-coded blue) as opposed to the generic Basic and Advanced features. Domain-specific features are derived from mesh properties and the CSE problems being solved. The addition of these features does not cost much more than simply reading the problem-specific parameters which is $O(1)$.

Some of the generic features are only useful or meaningful under certain circumstances. For example, the distinction between lower and upper triangle as well as the symmetry metrics are not meaningful for matrices that are known to be symmetric a priori such as those in MFEM. Similarly, we only define features based on singular values for MFEM in which all the matrices are positive-definite.

	Feature	Meaning
🟩	N_{row}	number of rows
🟩	NNZ	number of non-zeros, $nnz(A)$
🟩	NNZ_L	number of non-zeros in lower triangle
🟩	NNZ_U	number of non-zeros in upper triangle
🟩	NNZ_{max}	maximum number of non-zeros per row
🟩	NNZ_{avg}	average number of non-zeros per row
🟩	DE_{max}	largest element in magnitude along diagonal
🟩	DE_{min}	smallest non-zero element in magnitude along diagonal
🟩	bw_L	bandwidth of lower triangle
🟩	bw_U	bandwidth of upper triangle
🟩	NO	number of ones
🟩	NDD_{row}	number of rows that are strictly diagonally dominant, i.e., satisfying $2 * \lvert a_{j,j}\rvert - \sum_j \lvert a_{i,j}\rvert > 0$
🟩	$spread_L$	normalized sum of distances from diagonal of non-zero elements in lower triangle $\sum(i-j)/S$ where $i > j$, $a_{i,j} \neq 0$, and $S = \sum_{i=1}^{N_{row}-1} i * (N_{row} - i)$
🟩	$spread_U$	normalized sum of distances from diagonal of non-zero elements in upper triangle $\sum(j-i)/S$ where $j > i$, $a_{i,j} \neq 0$, and $S = \sum_{i=1}^{N_{row}-1} i * (N_{row} - i)$
🟩	$symm$	property showing how symmetric the matrix is as $1 - nnz(A - A.^T)/(nnz(A) - nnz_d(A))$ where $A.^T$ is a non-conjugate transpose of A and $nnz_d(A)$ is number of non-zeros on diagonal of A
🟩	$symm_s$	property showing how skew-symmetric the matrix is as $1 - (nnz(A + A.^T) - nnz_d(A))/(nnz(A) - nnz_d(A))$
🟩	$symm_p$	property showing how symmetric the non-zero pattern of matrix is as $1 - nnz(P - P.^T)/(nnz(P) - nnz_d(P))$ where $p_{i,j} = 1$ if $a_{i,j} \neq 0$, 0 otherwise
🟩	$\lvert\lvert A\rvert\rvert_1$	1-norm
🟩	$\lvert\lvert A\rvert\rvert_F$	Frobenius norm
🟥	$\rho(A)$	spectral radius
🟥	λ_2	second largest absolute eigenvalue
🟥	λ_{min}	non-zero smallest absolute eigenvalue
🟥	$\sqrt{\lambda_1/\lambda_2}$	where $\lambda_1 = \rho(A)$
🟥	$\sqrt{peak}$	$peak = \lambda_1/\left(\left(\sum_i \lambda_i\right) - \lambda_1\right)$
🟥	$\sqrt{\kappa}$	where κ is the condition number computed by Matlab
🟥	s_{90}	portion of $\sqrt{\sigma} > 0.9 \times (\sqrt{\sigma_{max}} - \sqrt{\sigma_{min}}) + \sqrt{\sigma_{min}}$, where σ is a singular value
🟥	s_{mid}	$1 - (s_{90} + s_{10})$
🟥	s_{10}	portion of $\sqrt{\sigma} \leq 0.1 \times (\sqrt{\sigma_{max}} - \sqrt{\sigma_{min}}) + \sqrt{\sigma_{min}}$
🟦	N_{dim}	number of spatial dimensions (from the mesh)
🟦	N'_{row}	N_{row}/N_{dim} if P2, N_{row} otherwise
🟦	$Mesh_{do}$	finite element discretization order
🟦	$Mesh_{rl}$	mesh refinement level
🟦	$\sqrt{\kappa'}$	approximation to condition number term $\sqrt{\kappa}$, where $\kappa' = (N'_{row})^{2/N_{dim}}$
🟦	$\sqrt{\kappa'_{hmin}}$	approximation to condition number term $\sqrt{\kappa}$, where $\kappa'_{hmin} = (1/min(h))^2$
🟦	$\sqrt{\kappa'_{hmax}}$	approximation to condition number term $\sqrt{\kappa}$, where $\kappa'_{hmax} = (1/max(h))^2$

TABLE I: List of candidate input features for a prediction model. We organize the features into three groups – readily computable or *Basic*, expensive to compute or *Advanced*, and *Domain-specific* (color-coded by group).

One of the challenges with matrices in the MFEM dataset is that the condition number κ, a measure of how difficult it is to solve a system with the matrix, is proportional to h^{-2} where h is a measure of the size of the mesh elements used. On a uniform grid in N_{dim} dimensions, the number of unknowns in each dimension is on the order of h^{-1}. Thus, the total number of unknowns N in the problem is proportional to $h^{-N_{dim}}$, and κ is proportional to h^{-2} or $N^{\frac{2}{N_{dim}}}$.

For the MFEM dataset, we can put an upper bound on the number of iterations required for a preconditioned conjugate gradient method as follows:

$$N_{iter}^{PCG} \leq \frac{1}{2} \times \sqrt{\kappa} \times \log \frac{2}{\epsilon} + 1$$

where κ is the condition number and proportional to $N^{\frac{2}{N_{dim}}}$, and ϵ is the convergence tolerance. N is the total number of unknowns in the problem and grows as,

$$N \propto Mesh_{do} \times 2^{N_{dim} \times Mesh_{rl}}$$

where $Mesh_{do}$ is the discretization order and $Mesh_{rl}$ is the refinement level. Given a matrix, N is equivalent to the number of rows of the matrix, N_{row} except for the problem P2, which is a vector problem. In P2, there are N_{dim} unknowns at each mesh point. Thus, $N^{P2} = N_{dim} \times N^{P1}$ on the same mesh. Since h is the mesh size, we divide N_{row} by N_{dim} to obtain N, which we label as N'_{row} in Table I. We include a cost-effective approximation, ${N'_{row}}^{1/N_{dim}}$, as an input feature in place of the condition number term $\sqrt{\kappa}$.

C. Experimental setup

As we formulate a linear problem out of a CSE problem, we obtain a matrix $\boldsymbol{A}$. For each matrix $\boldsymbol{A}$ in the MFEM and UFL datasets, we construct a linear problem $\boldsymbol{Ax} = \boldsymbol{b}$ by initializing the solution vector $\boldsymbol{x}$ to $\mathbf{0}$ and randomly assigning an RHS vector $\boldsymbol{b}$. We set each element in the RHS to a randomly chosen number between 0 and $\|A\|_\infty$. We define "convergence" as the case where the relative residual $\|r\|_2/\|b\|_2$ becomes less than 10^{-9} within a specified number of solver iteration steps. We terminate the solver when it reaches 10,000 iterations and consider the case as non-converging. We also mark a case as non-converging if the algorithm terminates prematurely because it determines that the problem is not solvable due to loss of precision.

There exist a number of algorithms for an end user to solve a linear system. While domain experts are likely to be aware of optimal choices for their problems, novice users are left with a large optimization space to explore, especially when dealing with new problems not encountered before. In addition, the theoretical knowledge on convergence behaviors of certain solvers on a certain category of problems may not correlate with memory access costs or the extent of SIMD optimization in a numerical library. Hence, it may not be sufficient to predict the performance of a solver in a software stack for a particular input matrix running on a particular machine platform. We consider nine linear solvers and thirteen preconditioners available in Trilinos as listed in Table II to mimic such an environment. We use Trilinos version 11.12.1 and compile the code using icc version 14.0.3 and MKL version 11.1.3. We perform our experiments on a single core of an Intel Xeon E5-2670 node with 32 GB of memory, and use the high-precision function clock_gettime() in the CLOCK_MONOTONIC_RAW mode for recording runtime.

Preconditioners	Solvers
NONE, JACOBI, NEUMANN, LS, DOM_DECOMP, ILU, ILUT, IC, ICT, CHEBYSHEV, POINT_RELAXATION, BLOCK_RELAXATION, AMESOS, SORA, IHSS	GMRES, GMRES_CONDNUM, GMRESR, CG, CG_CONDNUM, CGS, BICGSTAB, TFQMR, FIXED_PT

TABLE II: Choices of preconditioners and solvers available in Trilinos and experimented with in this study.

Each matrix is solved using each PC-LS combination for ten different RHS vectors and each run is repeated eleven times. We discard the first out of eleven runs to mitigate cold cache effects and this leaves us with 100 execution times for a given matrix and PC-LS choice. We use the median execution time from these 100 executions for the prediction model. Using the median value can suppress the effects of system noise and prevent misleading the instance-based learning. We do not include multiple samples of an identical setup in the same training data subset as it would make the cross-validation less effective. There remains a further challenge with the UFL dataset as matrices collected from the same source tend to exhibit high similarity in terms of both the features and the prediction target.

IV. Approach

We have developed a three-step approach to understanding sparse matrices and predicting the best preconditioner and linear solver (PC-LS) choice for different matrices. Below, we present the algorithmic details of the proposed modeling framework. Further, we illustrate each step of the algorithm using the MFEM dataset as an example, and present discussions on feature influences in predicting runtime performance.

Figure 1 provides an overview of the three-step approach, which consists of a training phase followed by a testing phase. The training phase involves two independent steps: 1. identifying a subset of features that are useful for predicting the runtime, 2. building a *performance vector space* that reveals the relationships between matrices in terms of performance. For the testing phase, we develop a novel technique that projects the features of an unseen test sample onto the performance vector space, and employ a k-Nearest Neighbor (k-NN) classifier for predicting the best preconditioner and linear solver choice.

A. Step I: Feature selection

The first step in the pipeline is to identify a set of features that are crucial in predicting performance. In particular, we aim to select features that are useful to predict the fastest runtime using the best PC-LS for solving each input matrix. The benefit of this feature selection process is two-fold: (a) a reduced feature set will control the model complexity during model learning, and (b) it is computationally efficient to calculate only a subset of features for the test cases. Further, building a regression model enables sensitivity analysis of the response variable (runtime) to the different input variables (features).

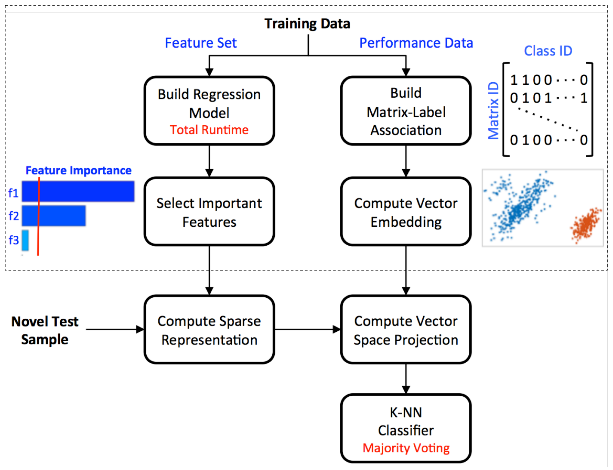


Fig. 1: An overview of the proposed modeling approach to predict the fastest preconditioner and linear solver for a new matrix provided by the end user.

We use gradient boosted trees to build the regression models, taking advantage of their ability to approximate complex functions using an ensemble of simple models [25]. Gradient boosted machine (GBM) is a machine learning paradigm where the key idea is to assume that the unknown function f is a linear combination of several *base learners*. The base learners are greedily trained by setting their target response to be the negative gradient of the loss with respect to the current prediction. The base learner can be imagined to be the "basis function" for the negative gradient. Concretely, let us assume the function of interest,

$$y \approx f(\mathbf{x}) = \sum_{j=1}^{m} \beta_j \psi_j(\mathbf{x}|\mathbf{z}_j) \quad (1)$$

where $\psi_j(\mathbf{x}|\mathbf{z}_j)$ is the base learner parameterized by $\mathbf{z}_j$ and m is the number of learners (or iterations). The GBM proceeds by performing a stage-wise greedy fit,

$$(\beta_j, \mathbf{z}_j) = arg \min_{\beta,\mathbf{z}} \sum_{i=1}^{n} L(y_i, f_{j-1}(\mathbf{x}_i) + \beta\psi_i(\mathbf{x}_i|\mathbf{z})) \quad (2)$$

where L is the loss function, n is the number of data samples and f_{j-1} is the estimate of the function obtained at the previous iteration,

$$f_{j-1}(\mathbf{x}) = \sum_{t=1}^{j-1} \beta_t \psi_t(\mathbf{x}|\mathbf{z}_t) \quad (3)$$

The estimate of base learner parameters at iteration j is obtained by setting its response $\widetilde{\mathbf{y}}$ to be the negative gradient

$$\widetilde{y}_i = -\left[\frac{\partial L(y_i, f(\mathbf{x}_i))}{\partial f(\mathbf{x}_i)}\right]_{f(\mathbf{x})=f_{j-1}(\mathbf{x})} \quad (4)$$

for all $i = \{1, \ldots, n\}$. The parameter $\mathbf{z}_j$ is updated using

$$\mathbf{z}_j = arg \min_{\gamma,\mathbf{z}} \sum_{i=1}^{n} (\widetilde{y}_i - \gamma\psi_i(\mathbf{x}_i|\mathbf{z}))^2 \quad (5)$$

The base learner coefficient β_j is updated using Equation 2. In our implementation, we consider the Huber loss function, and use simple decision trees as the base learner. Since GBM sequentially adds models by directly optimizing for the negative gradient of the loss function, the final approximation is often superior to several other strategies. Further, it is typical to compute relative feature importances in regression methods such as decision trees, and it is straightforward to extend that idea with GBM [25].

Illustration with the MFEM dataset: Our assumption is that the performance of PC-LS in serial execution can be attributed to two main components. The numerical properties of the input problem determine the number of iterations while the floating point operations and memory accesses determine the time per iteration. To understand the importance of features as well as the behavior of the prediction framework under various scenarios of feature availability, we consider the following sets of features for our analysis based on the Basic, Advanced and Domain-specific categories described in Table I:

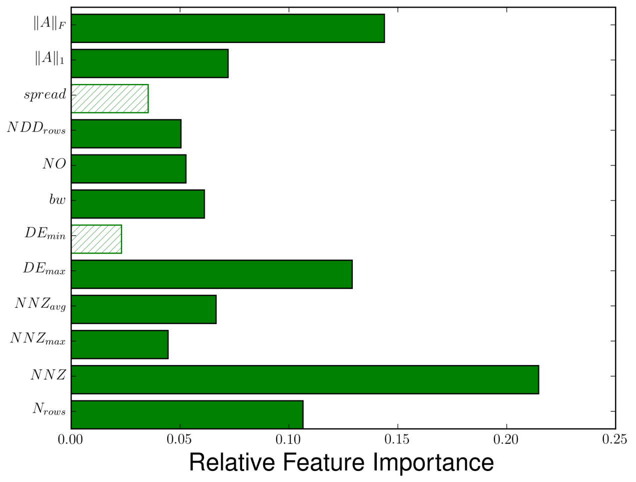
- **F1:** Basic only
- **F2:** Basic and Advanced
- **F3:** Basic and Domain-specific
- **F4:** Basic, Advanced and Domain-specific

We build regression models using GBM (Huber loss function), with the number of estimators fixed at 1000, and the learning rate fixed at 0.05. Figure 2 shows the relative feature importances for the four scenarios. We observe that a few Basic features are influential in all cases, such as NNZ, $||A||_F$, DE_{max}, and NNZ_{avg}. NNZ is directly relevant to the number of floating point operations, while NNZ_{avg} is likely to be associated with memory access costs. Advanced features such as $\sqrt{\kappa}$, λ_{min}, and s_{mid}, are important in both cases in which they are included. These features capture the numerical properties of the problem and hence can be crucial in predicting the number of iterations for convergence.

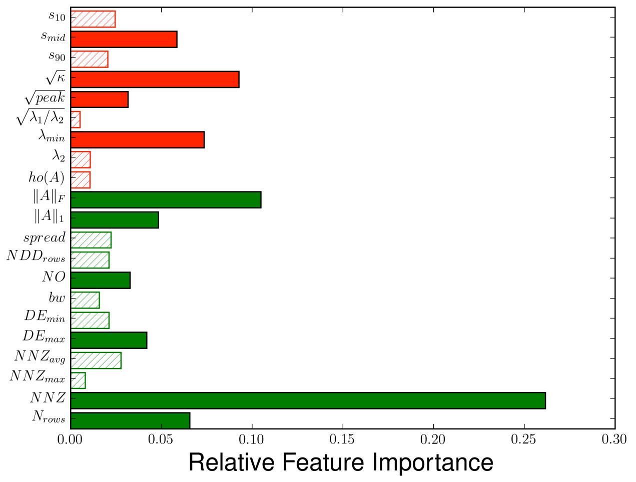
Domain-specific features, such as $\sqrt{\kappa'}$, and N'_{rows}, can potentially provide inexpensive alternatives to the Advanced features based on full eigen decomposition of the matrix, which is evidenced by the high prediction power of the feature set F3. The R^2 measures shown in Figure 2 increase as we add more features to the Basic set. Interestingly, R^2 of F2 is lower than that of F3, indicating that domain-specific knowledge can provide better insights into the convergence behavior of linear systems. We only choose features whose relative importance score is higher than 0.03 for subsequent analysis. Note that in all cases, the set of dominantly important features is small and hence it reduces the computational burden of extracting all the features for unseen test examples.

B. Step II: Building a performance vector space

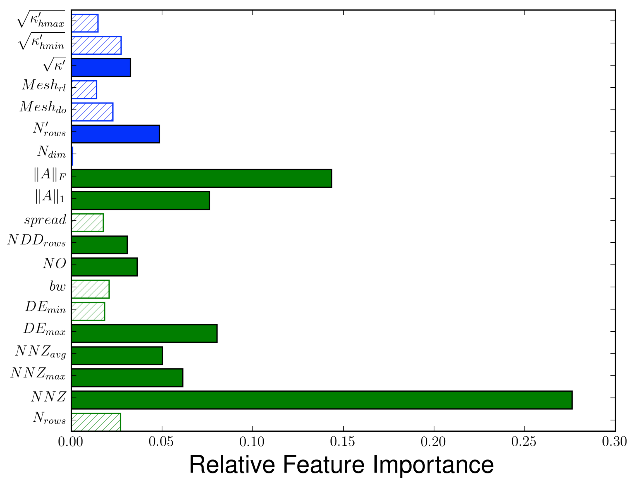
Existing approaches in the literature [19], [21] build a blackbox classifier with the features directly by using the best performing PC-LS choice as the label and build a classifier for determining class evidences. In these approaches, the number of classes is $N_C = N_P \times N_S$, where N_P is the total number



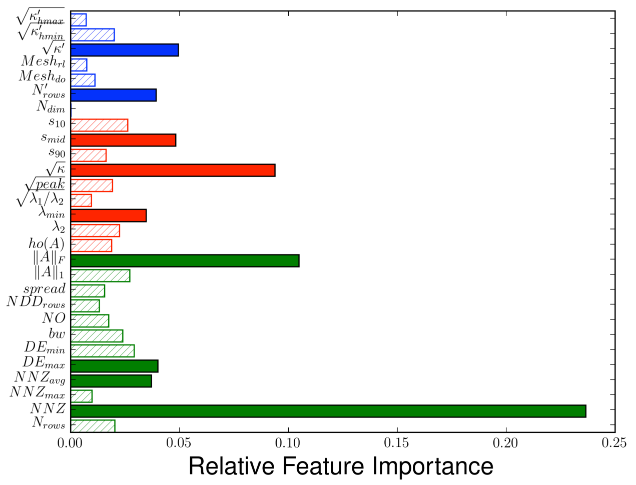
(a) F1: Basic only ($R^2 = 0.71$)



(b) F2: Basic and Advanced ($R^2 = 0.79$)



(c) F3: Basic and Domain-specific ($R^2 = 0.86$)

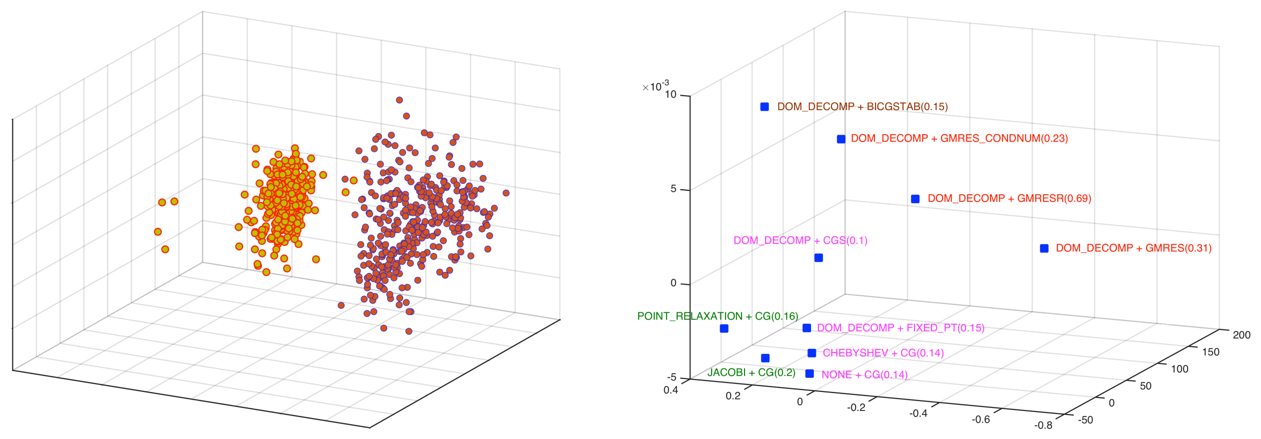


(d) F4: Basic, Advanced and Domain-specific ($R^2 = 0.88$)

Fig. 2: Feature selection for the MFEM dataset using the gradient boosted tree regressor under various feature availability scenarios. In each case, the subset of influential features, which are used for subsequent analysis, are marked in solid color. In all cases, only a small subset of features are picked, thereby reducing the burden on feature extraction for the test matrices.

of preconditioner choices and N_S is the total number of linear solvers considered. However, without considering the relative performance of other PC-LS choices, the problem of distinguishing a large number of classes becomes extremely challenging due to the limited size of training datasets and the highly unbalanced distribution of labels, leading to severe model overfitting. In contrast, we project sample features onto a new vector space in which spatial proximity of samples (matrices) will directly depend on their similarity in the set of appropriate PC-LS choices. This enables us to make more robust decisions. We refer to it as a *performance vector space* (PVS). In particular, we adopt ideas from recent advances in natural language modeling that attempt to represent words as dense vectors using neural networks [5]. The PVS approach radically differs from the traditional dimensionality reduction methods such as the singular vector decomposition (SVD) in that samples are treated as atomic symbols and thus the relationship is not identified based on the input's property. Note that we build a PVS utilizing the *word2vec* algorithm independent of matrix feature representation [26]. Each matrix is merely treated as a symbol or a word.

word2vec offers two modes of applications: continuous bag of words (CBOW) and skip-gram. Our approach is similar to the latter as we predict the multiple PC-LS choices (context words) with good performance for a given matrix (an input word). We produce the training samples (pairs of the input and the output of the network) based on the matrix-label association encoded in the co-occurrence matrix. We associate a label c to a training matrix m, if the runtime for that particular PC-LS setting, $R_c^m \leq \tau R_o^m$, where R_o^m is the lowest observed execution time (also referred to as the *oracle*), and τ is a user-defined threshold for quantifying the performance



(a) (b)

Fig. 3: 3D visualization of the vector spaces for the MFEM dataset. The axes of this visualization correspond to the three embedding dimensions from the *word2vec* algorithm. (a) The performance vector space obtained by applying *word2vec*, a neural vector embedding technique, on the matrix-label association, (b) Applying *word2vec* on the transpose of the matrix-label association enables us to determine groups of PC-LS that are effective for the dataset.

to be acceptable. In our experiments, we set $\tau = 1.05$, i.e., within 5% of the execution time for the oracle case.

word2vec relies on a neural network with only one hidden layer. An input to the network is encoded as a one-hot vector. As both matrices and PC-LS choices belong to the same vocabulary space, the size of a one-hot vector N_I equals to the sum of the number of matrices (N_M) and the number of PC-LS choices (N_C). The activation function at the hidden layer is simply the linear identity function while the output layer uses the softmax function. A training results in two weight matrices learned, W and W'. W transforms the input layer to the hidden layer, and W' transforms the hidden layer to the output layer. We use the former for mapping a matrix onto the PVS. The dimension of W is $N_I \times N_V$ and that of W' is $N_V \times N_I$, where we empirically set N_V, the dimension of the vector space, to be far smaller than N_I, the dimension of the input. A row of W is a vector corresponding to a particular matrix, and a column of W' to a particular label.

This problem can be solved more efficiently using the following formulation. Let us consider a matrix-label pair, (m, c), and we denote the probability $P(D = 1|m, c)$ that this pair came from the observed data D, and equivalently define $P(D = 0|m, c)$. The distribution is modeled as

$$P(D = 1|m, c) = \sigma(\mathbf{x}_m.\mathbf{x}_c) = \frac{1}{1 + \exp(-\mathbf{x}_m.\mathbf{x}_c)} \tag{6}$$

where $\mathbf{x}_m, \mathbf{x}_c$ are the vector representations for the matrix and the class label to be learned, . denotes the dot product, and σ is the sigmoid function. The *word2vec* algorithm in [26] attempts to maximize $P(D = 1|m, c)$ for observed samples, while maximizing $P(D = 0|m, c)$ for randomly sampled "negative" examples, as given by:

$$\log(\sigma(\mathbf{x}_m.\mathbf{x}_c)) + kE_{c_N}[\log(\sigma(-\mathbf{x}_m.\mathbf{x}_{c_N}))] \tag{7}$$

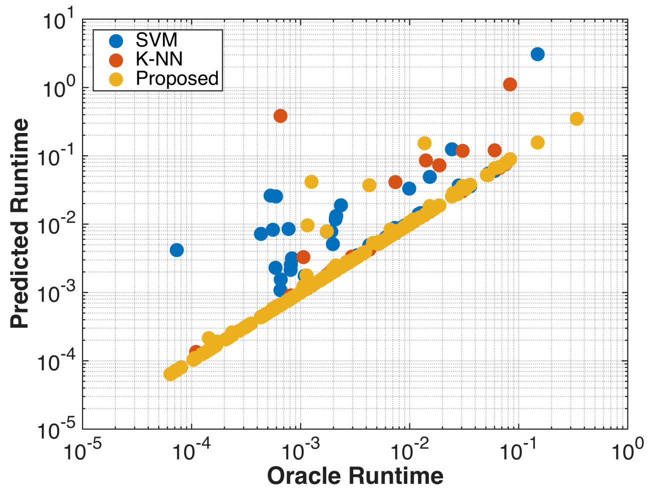
where k is the number of negative examples, and c_N indicates a randomly sampled class for the matrix m. The objective is optimized in an online fashion using stochastic gradient descent over the observed data, and we obtain vector representations for all training matrices.

Illustration with the MFEM dataset: Applying the proposed embedding technique with τ fixed at 1.05 yields the vector representation shown in Figure 3(a). Note that though the dimension of the vector space was fixed at 20, for visualization purposes, we apply PCA to create 3D embeddings. Though the axes of this visualization are not directly interpretable since they are learned factors of the embedding, they can reveal interesting insights about the relationships between different matrices in terms of performance.

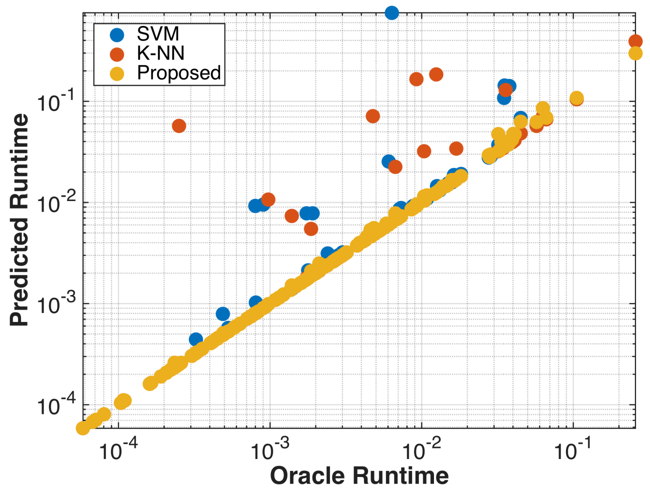
Surprisingly, we observe two reasonably tight clusters of matrices in the resulting vector space, indicating that a classifier based only on the best PC-LS choice can lead to highly sub-optimal predictions. Hence, we propose to perform classification in this performance vector space. Alternately, we can visualize the embedding using the vectors for the class labels ($\mathbf{x}_c$) to identify similarity between PC-LS settings, with respect to the types of matrices for which they are effective (Figure 3(b)). As expected, we find that a set of PC-LS choices (DOM_DECOMP + GMRES_CONDNUM, DOM_DECOMP + GMRESR, DOM_DECOMP + GMRES) lies along a linear plane, indicating that they are similarly effective in solving the MFEM matrices. In addition there is another cluster of methods (colored in magenta) that is found to be highly sub-optimal for the MFEM matrices.

C. Step III: Classification in the PVS

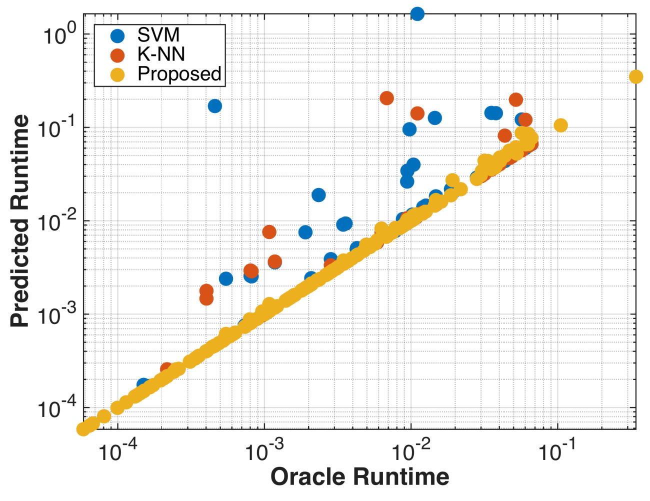
In the testing phase, we predict the most effective PC-LS combination for an unseen/test sample. Our approach is to project the unseen matrix into the PVS, to identify the nearest neighbors in the space, and to find out which PC-LS



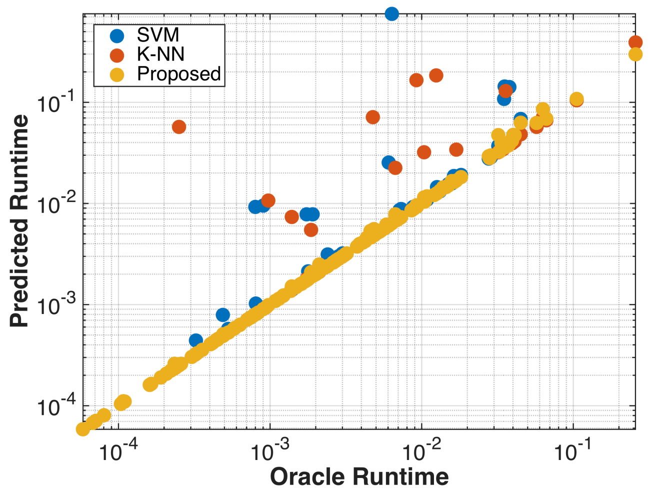
(a) F1: Basic only



(b) F2: Basic and Advanced



(c) F3: Basic and Domain-specific



(d) F4: Basic, Advanced and Domain-specific

Fig. 4: PC-LS prediction accuracy for the MFEM dataset: (a)–(d) Comparison of the execution time obtained using the predicted PC-LS setting with respect to the oracle case. (shown in log scale)

choice is the most useful among the neighbors. Unfortunately, we cannot directly map an unseen matrix onto the PVS as the arbitrary one-hot representation of the matrix cannot be constructed using the vocabulary. We propose a novel out-of-sample extension scheme based on sparse linear modeling. We begin by establishing the representation of the matrix in terms of the matrices used in training. For this, we rely on the matrix features determined during the training phase. Given the test feature vector, $\mathbf{f}_{te}$, and a collection of T training features, $\mathbf{F}_{tr}$, the coefficients, $\boldsymbol{\alpha}$, for sparse linear modeling can be obtained using the equation:

$$\min_{\boldsymbol{\alpha}} \|\mathbf{f}_{te} - \mathbf{F}_{tr}\boldsymbol{\alpha}\|_2^2 \text{ s.t. } \|\boldsymbol{\alpha}\|_1 \leq \delta, \forall i, \alpha_i \geq 0 \tag{8}$$

where $\|.\|_1$ denotes the ℓ_1 norm that measures the sparsity of a vector, $\|.\|_2$ is the ℓ_2 norm, and δ is the desired level of sparsity. The second constraint imposes non-negativity on the solution and results in an additive model that is easy to interpret. In the proposed approach, we assume that the feature space and the performance vector space are consistent, i.e., if two matrices have similar properties they can be solved efficiently by similar PC-LS choices, and hence use a common linear model to represent a test sample in both the spaces. In other words, we compute the vector representation for the test sample in the performance vector space as $\mathbf{g}_{te} = \mathbf{G}_{tr}\boldsymbol{\alpha}$, where $\mathbf{G}_{tr}$ is the collection of training samples in that space. The final step in the pipeline is to perform classification in the performance vector space and determine the suitable PC-LS setting for the linear system. We propose to use the k-NN classifier with majority voting to determine the class label.

V. Results

In this section, we evaluate the proposed prediction framework on the MFEM and UFL datasets, which comprise of 879 and 361 matrices respectively. We randomly select 75% matrices for training and the rest for testing. We discard matrices

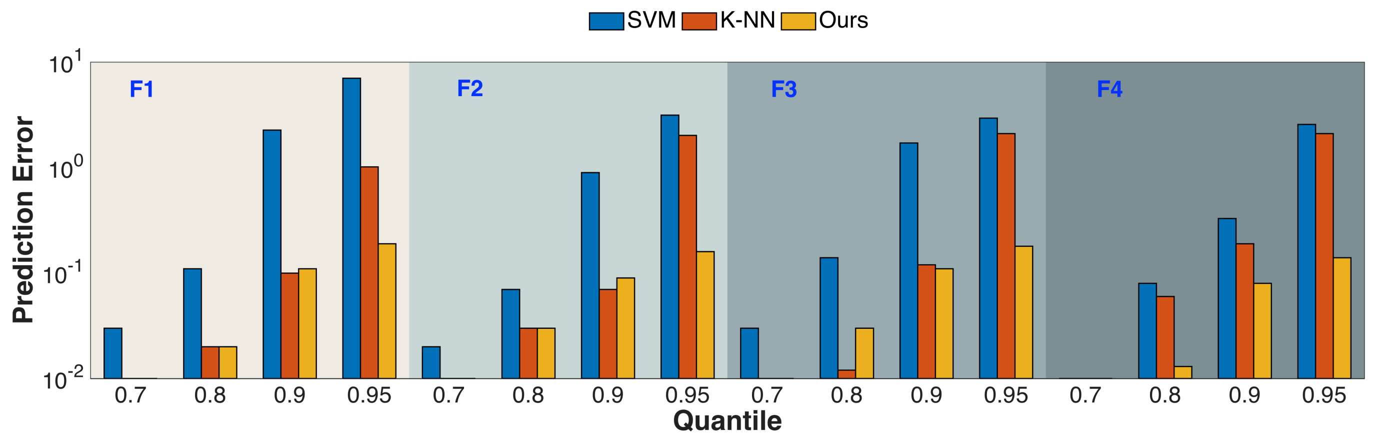


Fig. 5: MFEM dataset: Prediction error, $\frac{|y_p - y_o|}{y_o}$, computed at different quantiles of the test set (ordered by the error $|y_p - y_o|$) using different classifiers. The results reported are averaged over 20 different random train-test split choices for cross-validation.

that do not converge for at least 30% of the runs using any of the PC-LS choices. To explore the behavioral differences with existing classification techniques that cover a significant spectrum of behaviors, we apply two well-understood but very different methods, SVM and k-NN, directly to the matrix features. We report the results averaged over twenty independent train-test split (cross-validation) choices. For evaluating prediction quality, we define the metric *Absolute Relative Error* (ARE) as

$$ARE = \frac{|y_p - y_o|}{y_o} \tag{9}$$

where y_p is the execution time for the predicted PC-LS choice, and y_o is the *oracle* execution time for a test matrix. The oracle value is the median execution time for the best performing PC-LS choice for the particular matrix of a test sample.This metric is user-oriented since it captures the impact of inaccurate prediction rather than the prediction accuracy itself.

For the SVM classifier, we employ the linear kernel with the regularization parameter $C = 0.1$, and assign uniform cost weight for all classes. For the baseline k-NN classifier, the number of neighbors, k, is set to 8. We also rely on k-NN for the classification on the performance vector space in our framework, but with a smaller $k = 5$. We have not put thorough effort into optimizing these parameters as our goal is primarily to understand the impact by behavioral differences. We suspect that SVM with an RBF kernel would have performed better than with a linear kernel, behaving closer toward k-NN.

We evaluate the prediction accuracy of each of the classifiers using all four feature sets (F1, F2, F3 and F4). Finally, we include a naive baseline approach for comparison, wherein the the most effective PC-LS choice for the entire training data is blindly assigned for any test sample. We refer to this scheme as *Blind Selection*.

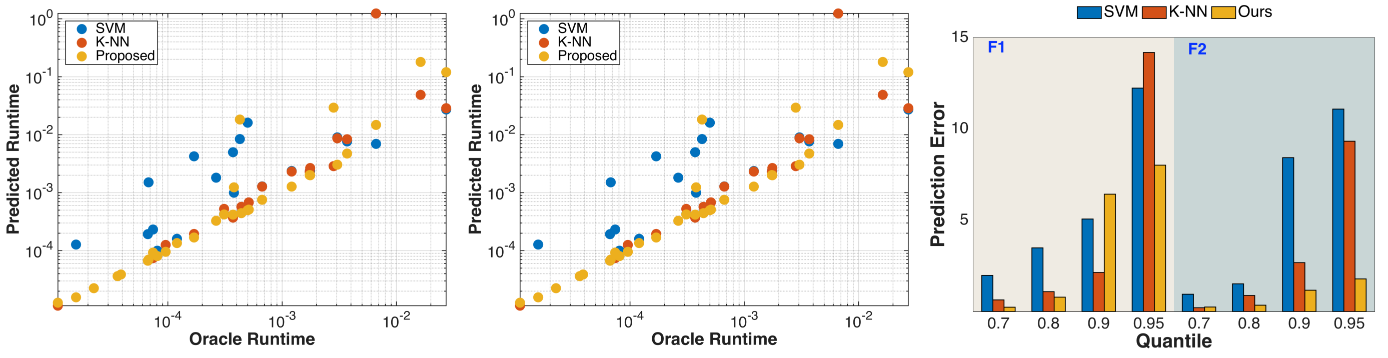
Figures 4 (a)–(d) illustrate the prediction using the three classifiers, for the four feature sets respectively, on the MFEM dataset. We plot the predicted runtime for all test samples against the oracle runtime values. Note that when the classification technique succeeds, all predictions would lie along the diagonal of the plot. The results show that the proposed approach is far more robust than the other blackbox classifiers, under all feature set scenarios. *Domain-specific* features are useful to improve predictions in the absence of the computationally intensive *Advanced* features.

Further, we evaluate the prediction error (ARE) for each of the test samples, and collect statistics including the mean and the error values at $\{0.7, 0.8, 0.9, 0.95\}$ quantiles. Figure 5 compares the prediction statistics for the different classification approaches. Though all approaches produce reasonably low errors at the 0.8 quantile, the SVM and k-NN classifiers produce highly inaccurate predictions on the remaining 20% of the worst samples in terms of ARE, thereby resulting in a high mean error. Especially, the linear kernel of SVM is not effective in learning the complex pattern in multi-dimensional data. In comparison, the proposed approach consistently produces high quality predictions, especially for challenging samples. The worst case ARE of the method (at 0.95 quantile) is less than 0.2 while it does not perform as good as k-NN at 0.90 quantile using F1 and F2 and at 0.80 using F3.

Method\Data	**F1**	**F2**	**F3**	**F4**	**Blind**
SVM	1.58	2.08	3.42	2.06	
k-NN	3.66	1.68	0.39	1.76	0.14
Ours	0.18	0.025	0.03	0.025	

TABLE III: MFEM dataset: Prediction error in ARE using three different classification techniques on the four feature sets. In addition, we report the performance obtained by blindly applying the most commonly identified optimal PC-LS setting.

Table III shows the mean ARE values. Similar to the regression experiments, the feature set F1 works poorly in comparison to the other three feature sets. This suggests that the information conveyed in the set of Basic features is not sufficient. An important challenge with the MFEM dataset is that the label distribution is highly skewed. Interestingly, the Blind Selection strategy outperforms the traditional classifier-



(a) F1: Basic only (b) F2: Basic and Advanced (c) Prediction Error

Fig. 6: PC-LS prediction accuracy for the UFL dataset.

based methods in terms of ARE. This is due to the fact that the classifiers optimize the classification error rather than the impact of misprediction, which is the increased execution time by choosing non-optimal PC-LS in our case.

Table IV shows that the mean classification accuracies of the classifiers are still higher for all of the feature sets than those of the blind selection, and also higher than those of the proposed method except for k-NN on F3. Surprisingly, the classification accuracies of SVM with the linear kernel are better than those of k-NN while it is mostly the opposite for ARE. Our approach effectively handles the skewness in the label distribution by classifying in a robust performance vector space. Another interesting observation is that the different baseline classifiers make inaccurate predictions on different matrices, and qualitative analysis of this can provide insights into the dataset, which we reserve for future work.

Method\Data	F1	F2	F3	F4	Blind
SVM	73.4	75.8	76.0	80.1	69
k-NN	72.8	75.1	74.6	79.6	
Proposed	70.1	74.2	75.2	77.9	

TABLE IV: MFEM dataset: Classification accuracy (%) using three different classification techniques on the four feature sets.

We carried out a similar evaluation on the UFL dataset and Figure 6 presents the prediction performance obtained with feature sets F1 and F2 respectively (there are no Domain-specific features for the UFL dataset and hence no F3 and F4 feature sets). Table V shows the mean ARE performance. When compared to the MFEM dataset, the label distribution is more diverse, though the sample size is much smaller. Consequently, this dataset is more challenging and the naive Blind Selection strategy does not perform as good as with the MFEM case. Further, the improvement in performance by employing the Advanced features is more apparent. Similar to the previous case, the proposed approach outperforms all other strategies.

Method\Data	F1	F2	Blind
SVM	3.52	2.93	2.19
k-NN	3.41	2.69	
Ours	2.13	0.8	

TABLE V: UFL dataset: Prediction error in ARE using three different classification techniques on the two feature sets.

VI. Conclusion

The use of machine learning and statistical modeling techniques for predictive analysis of high dimensional data has had a profound impact in a variety of scientific applications. The recent surge in the interest to adopt such tools within the HPC community presents a huge potential to solve several challenging problems. In this work, we address the problem of identifying the best performing pair of preconditioner and linear solver for sparse linear systems.

The effectiveness of a particular solver for a problem is often dependent on several numerical aspects of the linear system. Hence, we employ a data-driven approach, wherein we exploit the underlying characteristics of the linear system for predicting the best solver. The problem presents an extensive set of challenges including limited training data sizes, skewed label distributions, feature availabilities, noise and outliers. By proposing a novel combination of tools from statistical learning and natural language processing, we demonstrate that our approach can be highly effective especially in minimizing the impact of misprediction beyond optimizing the classification performance. In summary, our proposed data-driven approach provides new insights into these complex datasets and enables us to provide end-users with enhanced guidance for solving linear systems efficiently.

Acknowledgment

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-704087). This research used computer time on Livermore Computing's high performance computing resources, provided under the ASC Program.

References

[1] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, and K. S. Stanley, "An overview of the Trilinos project," *ACM Trans. Math. Softw.*, vol. 31, no. 3, pp. 397–423, 2005.

[2] "LAPACK - Linear Algebra PACKage," http://www.netlib.org/lapack.

[3] "Portable, Extensible Toolkit for Scientific Computation (PETSc)," http://www.mcs.anl.gov/petsc.

[4] "Scalable Library for Eigenvalue Problem Computations (SLEPc)," http://www.grycap.upv.es/slepc.

[5] O. Levy and Y. Goldberg, "Neural word embedding as implicit matrix factorization," in *Advances in Neural Information Processing Systems 27*, 2014, pp. 2177–2185.

[6] "MFEM: Modular finite element methods," mfem.org.

[7] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1:1–1:25, Dec. 2011.

[8] M. M. Mathis, D. J. Kerbyson, and A. Hoisie, "A performance model of non-deterministic particle transport on large-scale systems," *Future Generation Comp. Syst.*, vol. 22, no. 3, pp. 324–335, 2006.

[9] D. J. Kerbyson, H. J. Alme, A. Hoisie, F. Petrini, H. J. Wasserman, and M. Gittings, "Predictive performance and scalability modeling of a large-scale application," in *Proceedings of the 2001 ACM/IEEE conference on Supercomputing*, November 2001, p. 37.

[10] D. J. Kerbyson, A. Hoisie, and H. J. Wasserman, "Use of predictive performance modeling during large-scale system installation," *Parallel Processing Letters*, vol. 15, no. 4, pp. 387–396, 2005.

[11] N. R. Tallent and A. Hoisie, "Palm: easing the burden of analytical performance modeling," in *2014 International Conference on Supercomputing*, June 2014, pp. 221–230.

[12] H. Gahvari, A. H. Baker, M. Schulz, U. M. Yang, K. E. Jordan, and W. Gropp, "Modeling the performance of an algebraic multigrid cycle on HPC platforms," in *Proceedings of the 25th International Conference on Supercomputing*, June 2011, pp. 172–181.

[13] H. Gahvari, W. Gropp, K. E. Jordan, M. Schulz, and U. M. Yang, "Modeling the performance of an algebraic multigrid cycle using hybrid mpi/openmp," in *41st International Conference on Parallel Processing, ICPP 2012*, September 2012, pp. 128–137.

[14] H. Gahvari and W. Gropp, "An introductory exascale feasibility study for ffts and multigrid," in *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, April 2010, pp. 1–9.

[15] R. D. Falgout and U. M. Yang, "hypre: A library of high performance preconditioners," in *International Conference on Computational Science*, April 2002, pp. 632–641.

[16] A. Bhatele, P. Jetley, H. Gahvari, L. Wesolowski, W. D. Gropp, and L. Kale, "Architectural constraints to attain 1 Exaflop/s for three scientific application classes," in *Proceedings of the IEEE International Parallel & Distributed Processing Symposium*, ser. IPDPS '11. IEEE Computer Society, May 2011. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/IPDPS.2011.18

[17] S. Bhowmick, V. Eijkhouta, Y. Freund, E. Fuentes, and D. Keye, "Application of machine learning to the selection of sparse linear solvers," *Int. J. High Perf. Comput. Appl*, 2006.

[18] S. Bhowmick, B. Toth, and P. Raghavan, "Towards low-cost, high-accuracy classifiers for linear solver selection," in *Computational Science–ICCS 2009*, 2009, pp. 463–472.

[19] R. Nair, S. Bernstein, E. R. Jessup, and B. Norris, "Generating customized sparse eigenvalue solutions with lighthouse," in *Proceedings of the Ninth International Multi-Conference on Computing in the Global Information Technology*, June 2014.

[20] G. Belter, E. R. Jessup, I. Karlin, and J. G. Siek, "Automating the generation of composed linear algebra kernels," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC '09, Nov 2009.

[21] B. Norris, S. Bernstein, R. Nair, and E. R. Jessup, "Lighthouse: A user-centered web service for linear algebra software," *CoRR*, vol. abs/1408.1363, 2014. [Online]. Available: http://arxiv.org/abs/1408.1363

[22] C. Cortes and V. Vapnik, "Support-vector networks," *Mach. Learn.*, vol. 20, no. 3, pp. 273–297, September 1995.

[23] N. Jain, A. Bhatele, M. P. Robson, T. Gamblin, and L. V. Kale, "Predicting application performance using supervised learning on communication features," in *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. IEEE Computer Society, Nov. 2013, LLNL-CONF-635857.

[24] A. Bhatele, A. R. Titus, J. J. Thiagarajan, N. Jain, T. Gamblin, P.-T. Bremer, M. Schulz, and L. V. Kale, "Identifying the culprits behind network congestion," in *Proceedings of the IEEE International Parallel & Distributed Processing Symposium*, ser. IPDPS '15. IEEE Computer Society, May 2015, LLNL-CONF-663150.

[25] J. H. Friedman, "Greedy function approximation: A gradient boosting machine," *The Annals of Statistics*, vol. 29, no. 5, pp. pp. 1189–1232, 2001.

[26] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in Neural Information Processing Systems 26*, 2013, pp. 3111–3119.