Automated Selection of Numerical Solvers

Kanika Sood
University of Oregon
Eugene, OR 97403-1202
kanikas@cs.uoregon.edu

October 5, 2015

Abstract

Many complex problems rely on scientific and engineering computing for solutions. High-performance computing depends heavily on linear algebra for large scale data analysis, modeling and simulation, and other applied problems. Linear algebra provides the building blocks for a wide variety of scientific and engineering simulation codes. Sparse linear system solution often dominates the execution time of such applications, prompting the ongoing development of highly optimized iterative algorithms and high-performance parallel implementations. We are particularly interested in a scientific toolkit called Parallel Extensible Toolkit for Scientific Computation (PETSc) because of its efficiency, unique features and widespread popularity. In this report we present the algorithm classification results for the preconditioned iterative solvers in PETSc. In addition, we have created a comprehensive machine-learning-based workflow for the automated classification of iterative solvers, which can be generalized to other types of rapidly evolving numerical methods.

1 Introduction

Large sparse systems of linear equations are widely used in many high-performance computing applications. The size and complexity of the new generation of linear systems arising in typical applications are growing; hence, solving large sparse linear systems is a fundamental problem in high-performance scientific and engineering computing. Fundamentally, there are two main types of solution algorithms: direct and iterative. With direct methods, the average number of operations to solve a system of linear equations for these methods is nearly n^3 where n is the order of a square matrix. Considering the size of these systems and the computational complexity of exact solutions ($O(n^3)$), for very large problems, approximate solutions are computed using iterative methods (with typical convergence in $O(n)$ iterations).

Mathematicians and computer scientists have created a number of comprehensive numerical software packages that can solve such linear systems. But because of the overwhelming number of reasonable choices to consider, finding the most suitable solution to a particular problem is a non-trivial task, even for experts in numerical methods. Application developers who deal with complex or huge problems cannot rely on simple implementations because they might not be able to offer enough memory or might be too slow for such problems.

Therefore they must use high-performance computing (HPC) libraries developed by others. The current high-performance implementations of numerical linear algebra software are based on decades of applied mathematics and computer science research. Therefore, selecting a suitable library and using it effectively to solve a given problem can require significant background in numerical analysis, HPC, software engineering, and the researcher's domain science. This makes the application developers face the challenge of predicting which method will converge fastest, or converge at all, for a given linear system. Indeed, discovering the best approach to solving a linear algebra problem typically involves reading documentation (when available) or researching publications outside of the developer's area of expertise as well as experimenting, across software options. While continuous advances in numerical analysis and HPC libraries allow scientists and engineers to solve larger and more complex problems than ever before, the likelihood that a user will identify the most relevant and well-performing solution method is steadily decreasing. We apply several machine-learning techniques to help make this decision based on relatively few, easily computable properties of the input linear system.

This work can provide support for identifying good solution methods for sparse linear systems to existing taxonomies that aid developers in translating linear algebra algorithms to numerical software, which do not have such support for sparse matrices at present. One such taxonomy is Lighthouse [3], which is an open-source Web application that currently serves as a guide to the dense linear system solver methods.We plan to work on expanding the Lighthouse framework for the production of matrix algebra software by adding support for sparse matrix algebra computations. The approach we follow for solver selection is to use machine-learning algorithms to generate functions to map linear systems to suitable solvers. This contains two parts, feature extraction and classification. The feature extractor computes numerical quantities, called features or properties, such as structural and spectral estimates of the given linear system. The set of features is designed to represent the characteristics of the system that are predictive of the performance of solvers. The classifier maps the given feature values to a choice of solver. The task of the learning algorithm is to choose a subset of the features, hereby referred to as feature selection, and associate the values of the selected features to the choice of the solver.

The contributions in this report can be summarized as follows.

- A generalizable machine learning-based workflow for classifying arbitrary sparse linear systems using different-sized feature sets.
- Comparison of several machine-learning algorithms' performance for classifying the Portable, Extensible Toolkit for Scientific Computation (PETSc) [2] solvers.
- Suggestions for good solver-preconditioner configurations for a given linear system.
- A set of solver-preconditioner configurations that are most likely to perform well.

2 Motivation

Several attempts have been made in the past to support automation of solver selection and configuration, but none of them are general enough or have produced a usable software infrastructure that would enable users to apply them for their applications. Hence, our goal is to define an extensible methodology for classifying algorithms and software that supports

reusability and can also be used for the new solvers as they evolve. We apply various machine-learning algorithms to do the classification using Weka [21]. Weka is a tool that contains a collection of algorithms for data analysis and predictive modeling. We classify solvers based on the properties of the linear system and select the solver configuration that has been determined to perform best based on an extensive training set of problems.

In the past, many researchers have used machine-learning to identify "good" solvers in the context of parallel nonlinear PDE solution [8, 10, 28]. Depending on the conditions specified in the simulation code, some matrices are weakly diagonally dominant (difficult to solve), whereas others are strongly diagonally dominant (easy to solve). For matrices that are weakly diagonally dominant, this machine-learning framework was successful in identifying solvers with expensive preconditioners (such as BoomerAMG [34]). It was also successful in identifying that very simple solvers (such as Jacobi [30]) would be fairly good for the strongly diagonally dominant. The success of the limited-domain exploration motivated the work presented in this report, which covers a wider variety of domains and considers a much greater number of machine-learning methods.

3 Background

Existing numerical software taxonomy approaches are general and support relatively stand-alone algorithms. However, these approaches do not accommodate tasks for which no library implementation exists, or tasks that involve more complex software packages. For instance, the functionality of large HPC toolkits, such as Trilinos [4] and PETSc, is difficult or rather impossible to represent and maintain in most previous taxonomies, which at present simply direct the user to the toolkit's home pages and do not provide any support for selecting solution methods based on performance requirements. In this report we focus on adding support in our larger project, Lighthouse, in the area of preconditioned iterative solvers with PETSc.

PETSc provides good support for getting started with the toolkit. There are many ways of downloading PETSc: using a Git repository, installing a Debian package, or following a direct web download link. Once PETSc has been successfully installed, it is easy to find the guidelines to configure and build with the help of appropriate PETSc tutorials and other online documentation. Mandatory packages are automatically downloaded, configured, built and installed with PETSc. A number of PETSc examples are also available to instruct the user on writing PETSc programs and setting various command line options.

Selecting the appropriate PETSc algorithms, however, presents substantial difficulties. A sparse linear solver is typically paired with a preconditioner. To the uninformed user, the set of parallel Krylov methods and preconditioners contained in PETSc, summarized in Table 1, can be overwhelming because there are more than 300 possible pairings. Moreover, each solver-preconditioner pair can be configured through a set of algorithm-specific parameters, further expanding the search space. From this collection, the best choices for iterative solvers and preconditioners for a given linear system, depend on properties of its coefficient matrix and may also depend on the physics of the problem. Choosing a solution configuration requires a search of the extensive numerical linear algebra literature and may also require intense document reading or knowledge in the domain. The users may or may not have the expertise, knowledge and time required to do these tasks, therefore making the selection process very challenging.

Capability	Algorithm
Preconditioners	Jacobi point block Jacobi block Jacobi additive Schwarz
Incomplete factorizations	ILU dt
Matrix-free	infrastructure
Multigrid	infrastructure geometric (DMDA for structured grid) geometric/algebraic structured geometric classical algebraic (BoomerAMG/hypre) classical algebraic (ML/Trilinos) unstructured geometric and smoothed aggregation
Physics-based splitting	relaxation and Schur-complement least squares commutator
Approximate inverses	approximate inverses
Substructuring	balancing Neumann-Neumann BDDC
Krylov methods	Richardson, Chebyshev, conjugate gradients, GMRES, Bi-CG-stab, transpose free QMR, conjugate residuals, conjugate gradient squared, bi-conjugate gradient , MINRES, flexible GMRES, LSQR, SYMMLQ, LGMRES, GCR , Conjugate gradient on the normal equations

Figure 1: PETSc Krylov iterative solvers and preconditioners [2].

Next, the performance of a chosen solver-preconditioner pair strongly depends on specific features of the linear system, and the PETSc implementation of each method has several configuration parameters that can affect the accuracy and performance of the solution computed by that method. Neither specific system features nor parameter details are clearly stated in the literature, and as a result, achieving best performance is generally a matter of experimenting with a variety of options, which typically cover only a small fraction of all possible methods. All libraries for the solution of sparse matrix algebra problems inherit the same difficulties in selecting the best methods, and some introduce new complications of their own.

With our technique we classify solvers based on the system features and select the solver configuration that performs best on an extensive training set of problems. We then use the resulting models to classify the solvers and finally make the solver prediction for the new linear systems based on their features. This technique would eliminate the need for the users to make the solver selection choice and automates the process, making it easier and faster.

4 Approach

Our work includes creating a comprehensive machine learning-based workflow for the classification of sparse solvers and then presenting a comparative analysis of the solver classification results for various machine-learning methods and input problems from different domains, achieving up to 87% accuracy in identifying the well-performing linear solution methods in PETSc. The overall structure of our system involves various stages, which are discussed in detail later in this section. The first stage involves taking the linear systems as input and computing the properties of these systems. The second stage involves solving these linear systems with various solver and preconditioner combinations and timing the solution. Next, the solvers need to be categorized as "good" or "bad" based on the time it takes to solve the system. The categorization is followed by the classifier construction process. To construct the classifier, the learning algorithm receives a training set as input. The training set is a set of features of the linear system and the solver with its classification. The classifier searches for a linking that makes good predictions on the training set, with respect to the resource. The final stage involves choosing the best classifier to suggest a solution for an unknown linear system. Figure 2 shows the complete workflow of the process of linear system solver selection.

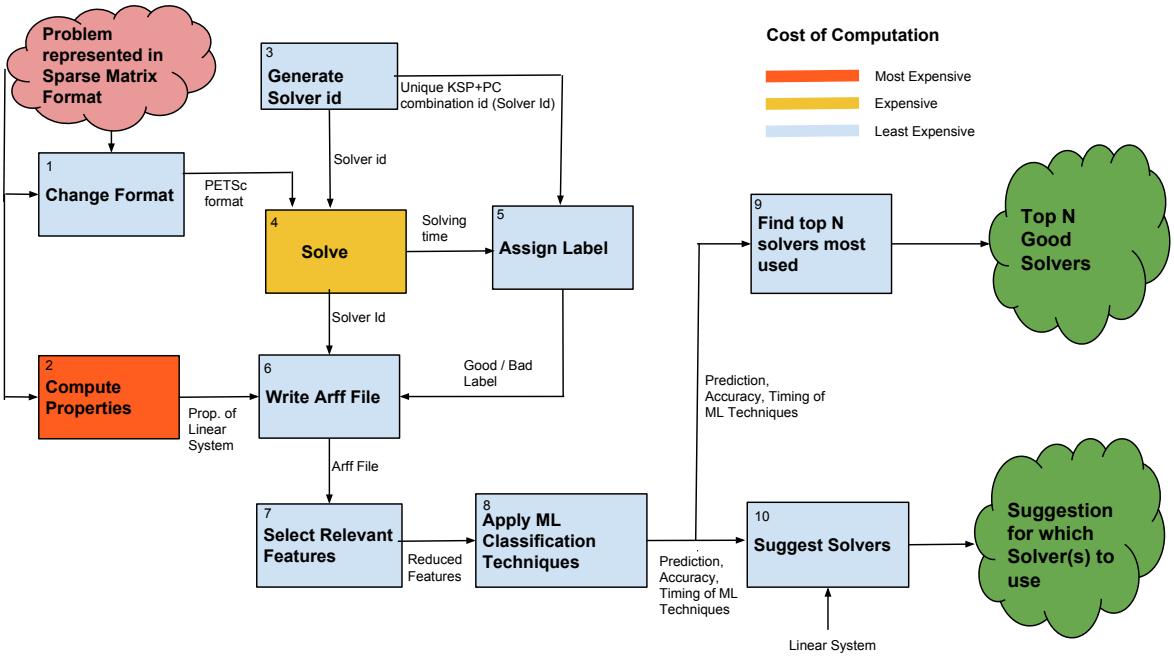


Figure 2: The overall workflow of the linear system solver selection process.

4.1 Creating the Dataset

The sparse linear systems used for our experiments are obtained from the University of Florida sparse matrix collection [5]. Our dataset comprises 1,015 sparse matrices. The size of these matrices varies from 14 to 43,460 rows and columns. These matrices cover a wide spectrum of domains and include those arising from problems with underlying 2D or 3D geometry (computational fluid dynamics, acoustics and other discretization) and the ones

which typically do not have such geometry (optimization, circuit simulation and networks and graphs). Below are some of the kinds of the problems that are covered by our dataset:

- Electromagnetics problems
- Linear programming problems
- Acoustics problems
- Computational fluid dynamics problems
- Circuit simulation problems
- Undirected weighted graph problems
- Optimization problems
- Power network problems
- Undirected random graph problems
- Chemical process simulation problems

When using supervised machine-learning methods for building models, a common problem is overfitting. Overfitting occurs when a model begins to "memorize" training data rather than "learning" to generalize from the trend. The chances of overfitting are reduced by the large variety of problem domains that our dataset covers.

The input matrices are in Matrix Market format [1]. For reading in parallel, Matrix Market is not very suitable; hence, for all cases, serial and parallel, we pre-process and convert the Matrix Market format to PETSc binary format.

4.2 Preparing Data for Training

Once we have the dataset ready, the next step is to convert the data in the training set into a form that is usable by Weka. The input to the learning process is Weka's Attribute-Relation File Format (ARFF) ASCII text file, which describes a list of instances that share some attributes. Each data point includes a list of feature values, the solver identifier, and a label. The solver identifier is unique to each pairing of specific Krylov method and preconditioner. So for M matrices, we have $M * N$ data points, N being the number of possible solvers. Because this number can be prohibitively large, we constructed the training set by computing a smaller number of randomly selected points. We labeled each data point as "good" or "bad" based on the performance of the solver on a matrix based on a threshold parameter b in the range 0,1 specifying how close the solver's performance is to the known best performing method. For example, when $b = 0.25$, solvers whose performance for a given problem is within 25% of the best were labeled as "good", while all other solvers were labeled as "bad". The threshold used for labeling the dataset in this case was b = 0.35, which means that methods within the top 35% of the best solver time were labeled as good. This value of b was chosen as the best among several sampled values between 0.01 and 0.45. We considered binary labels for classification, because our experiments with three-class labels showed that, with the given number of data points collected, they are not enough to be distributed in three classes, as we need enough data points in each class. We do not consider ranking either, because binary classification has the greatest number of machine-learning methods available for classification.

4.3 Solving the linear systems

PETSc has a collection of parallel algorithms for direct solvers, Krylov iterative methods, and preconditioners that can be used in application codes written in C, C++, Fortran and Python. Our focus is on iterative Krylov methods and preconditioners. To solve the sparse linear systems derived from the input matrices, we used 154 preconditioner-solver configurations chosen from more than 300 valid options available in PETSc, with all right-hand side elements set to one. These 300 options can be expanded further as they are configurable with additional parameters, for both, solvers and preconditioners. We consider the following preconditioners and solvers chosen from PETSc.

4.3.1 Preconditioners

Preconditioning refers to the process of applying a transformation on the original problem and brings it into a form that is more suitable for the solving methods. The main idea behind applying a preconditioner is that, instead of solving $Ax = b$, solve $M^{-1}Ax = M^{-1}b$ using a nonsingular m x m preconditioner M, which has the same solution x.

Here we mention the various preconditioners we considered and the subset of parameters for them. The preconditioners are as follows:

1. **Incomplete factorization preconditioners (ILU)**: ILU is an approximation of the LU (Lower Upper) factorization. LU factorization factors a matrix as the product of the lower and the upper triangular matrix.

   Parameters: Factor levels which are the number of levels of fill for ILU.

   Parameter values: 0, 1, 2 and 3.

2. **Additive Schwarz method (ASM)**: Solves an equation approximately by splitting it into boundary value problems and adds the results.

   Parameter considered: The amount of overlap between sub-domains.

   Parameter values: 0, 1, 2 and 3.

3. **Jacobi or diagonal**: One of the simplest forms of preconditioning in which the preconditioner is the diagonal of the matrix as shown below.

   $M = diag(A)$ for $M^{-1}Ax = M^{-1}$ b.

4. **Block Jacobi**: It is similar to Jacobi, except that in this case, instead of the diagonal, the block-diagonal is chosen as the preconditioner (M).

5. **Incomplete Cholesky factorization (ICC)**: It is a sparse approximation of the Cholesky factorization. The Cholesky factorization A is $A = LL^*$ where L is a lower triangular matrix. An incomplete Cholesky factorization is given by a sparse lower triangular matrix K that is very close to L. The corresponding preconditioner is KK^*.

   Parameter considered: Factor levels which are the number of levels of fill for ICC.

   Parameter values: 0, 1, 2 and 3.

These and other Krylov methods are described in more detail in [31].

4.3.2 Solvers

Similar to preconditioners, solvers also have their own parameters. For this research we are considering only default parameters. We considered the following iterative solvers.

1. **Generalized minimal residual method (GMRES)**: This method approximates the solution by the vector in a Krylov subspace with minimal residual. The Arnoldi iteration is used to find this vector.
2. **The Flexible Generalized minimal residual method (FGMRES)**: It is a generalization of GMRES that allows larger flexibility in the choice of solution subspace than GMRES.
3. **LGMRES**: It augments the standard GMRES approximation space with approximations to the error from previous restart cycles.
4. **Conjugate gradient method (CG)**: This method starts with an initial guess of the solution, with an initial residual and with an initial search direction.
5. **Biconjugate Gradient Method (BICG)**: Implements the Biconjugate gradient method, similar to running the conjugate gradient on the normal equations.
6. **Biconjugate gradient stabilized method (BCGStab)**: It is a stabilized version of BiConjugate Gradient Squared method.
7. **Improved Stabilized version of BiConjugate Gradient Squared (IBCGS)**: It is an improved stabilized version of BiConjugate Gradient Squared method.
8. **Transpose-Free Quasi-Minimal Residual Method (TFQMR)**: It is a quasi-minimal residual version of CGS. It retains the desirable convergence features of CGS and corrects its erratic behavior.
9. **TCQMR**: It is a variant of quasi-minimal residual provided by Tony Chan.
10. **LSQR**: This is an algorithm for sparse linear equations and sparse least squares.
11. **Chebyshev**: This method requires enough knowledge about the spectrum of the matrix, which is an upper estimate for the upper eigenvalue and lower estimate for the lower eigenvalue. Chebyshev iteration method avoids the computation of inner products as is necessary for the other methods.

We capture the time taken to solve the system, the number of iterations, and solver and preconditioning options, such as number of blocks and overlap.

4.4 Feature Computation

We used Anamod [20], which is a library of modules that uses PETSc functions, to compute various properties of a system and extracted sixty-eight features shown in Figure 1 of the coefficient matrices. These features include several categories as mentioned below:

1. **Simple (norm-like quantities)**: Properties which are estimates of the departure from normality such as 1-norm, infinity-norm and Frobenius-norms of the matrix, as well as these norms taken of the symmetric and non-symmetric part of the matrix.

Feature names	
avgnnzprow	right-bandwidth
avgdistfromdiag	symmetry
n-dummy-rows	blocksize
max-nnzeros-per-row	diag-definite
lambda-max-by-magnitude-im	lambda-max-by-magnitude-re
ellipse-cy	nnzup
ruhe75-bound	avg-diag-dist
nnz	left-bandwidth
lambda-min-by-magnitude-im	lambda-min-by-magnitude-re
norm1	sigma-min
upband	n-struct-unsymm
colours	diagonal-average
diagonal-dominance	dummy-rows
ritz-values-r	symmetry-snorm
symmetry-fanorm	symmetry-fsnorm
lambda-max-by-real-part-im	lambda-max-by-real-part-re
lambda-max-by-im-part-re	lambda-max-by-im-part-im
col-variability	trace-abs
ritz-values-c	nnzeros
diag-zerostart	loband
positive-fraction	trace
min-nnzeros-per-row	diagonal-sign
row-variability	nrows
colour-offsets	n-colours
relsymm	diagonal-variance
departure	nnzlow
n-nonzero-diags	sigma-max
dummy-rows-kind	kappa
n-ritz-values	colour-set-sizes
sigma-diag-dist	symmetry-anorm
ellipse-ax	ellipse-ay
ellipse-cx	lee95-bound
normInf	normF
nnzdia	trace-asquared

Table 1: Full feature set (68 features) [20].

2. **Variability**: Properties that are various heuristic measurements of how far the matrix is from a model problem, such as diagonal-variance.

3. **Structure**: Properties that describe the sparsity structure of the matrix, such as bandwidth, average number of non-zeros per row or maximum or minimum number of non-zeros per row, etc.

4. **Spectrum**: Properties that describe the spectrum or field of values of the coefficient matrix. These properties can not be computed exactly, but estimation is feasible. Examples of such properties include eigenvalue and singular value estimates.

5. **JPL**: These features compute a Jones-Plassmann multicolouring of a matrix such as colours, colour offsets, etc.

4.5 Feature Reduction

The cost of computing features varies widely from milliseconds to minutes or even hours depending on the timeout parameter chosen for interrupting non-convergent feature computations. For instance, spectrum properties such as eigenvalues take a very long time to compute as compared to some other properties. Therefore, in order to reduce the overall cost of the process, we performed analysis to remove features that do not make a significant contribution to the classification process. This was achieved in two ways. First, we reduced the number of features by using Weka's RemoveUseless filter. This filter removes the data points corresponding to those features whose values either remain constant or vary too much (over 99% variance). Using this simple filter brought down our number of features from 68 to 54 with 0% drop in accuracy, in fact typically improving the accuracy of subsequent classifications. We completed the selection with Weka by combining five attribute evaluators with two search methods. The evaluator determines a method to assign a weight to each subset of features. The search method determines what style of search is performed. These evaluators rank the features, allowing us to discard those that are not very significant. The evaluators we used are Gain Ratio, ChiSquared, CfsSubset, Information Gain, and Principle Component Analysis [25]. The search methods we chose were Greedy Stepwise and Ranker [22]. These evaluators may or may not have the same set of features; therefore, we chose the subset of features ranked highly by all or the majority of these evaluators for maximizing the classifier's true positive rate or sensitivity ("good" as "good" predictions). Sensitivity is the probability that the classifier will label a "good" entry as "good". We generated two reduced feature sets for PETSc solvers, one of which is a subset of the other set. These best features are expected to vary if new feature sets are evaluated and re-ranked, but since we started with a very comprehensive set, it is not likely that they will vary.

Computing the smallest eigenvalue can take time on the order of 10^{-2} seconds for relatively small matrices (<1,000,000 non-zeros), while a bandwidth computation requires time on the order of 10^{-5} seconds. Our experiments show that the expensive features do not contribute significantly to the performance of the classification, and hence, they can be safely removed. Removing expensive features ensures that this approach is feasible, incurring minimal runtime overhead of selecting a good linear solver configuration.

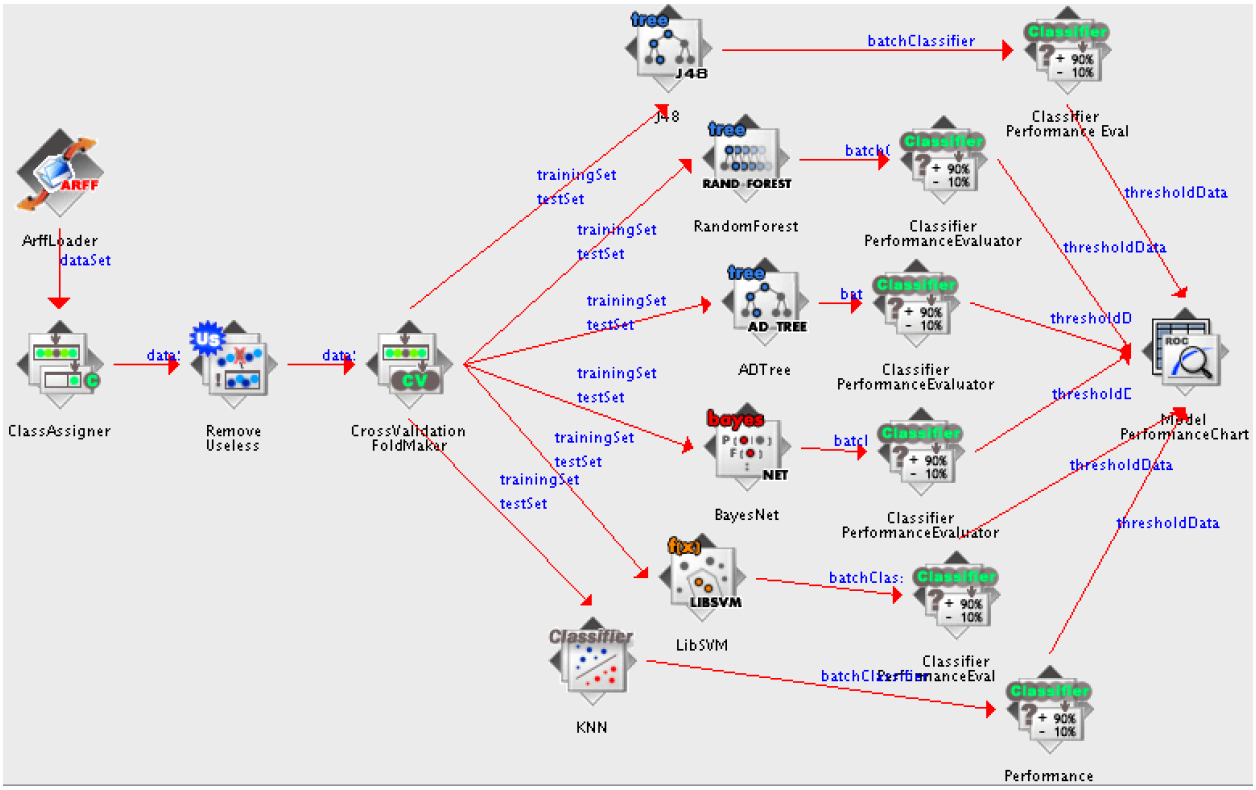


Figure 3: Weka workflow snapshot showing a subset of the classifiers. The same workflow was used for the full and reduced feature sets.

4.6 Solver Classification

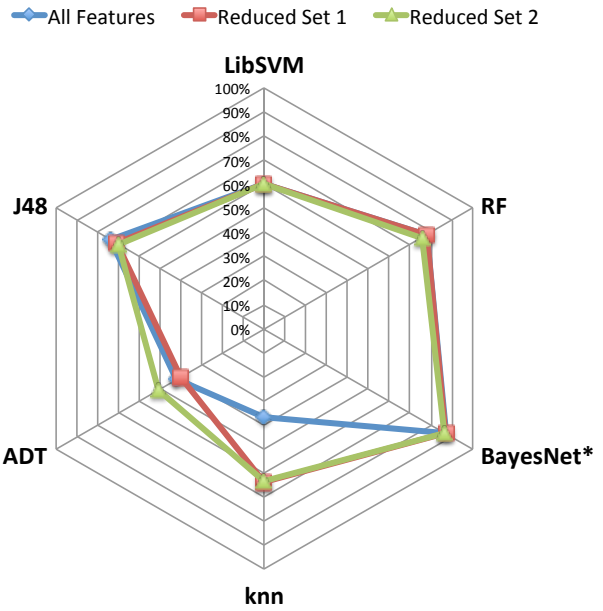
We classify solvers based on linear system features and select the solver configuration that performs best on an extensive training set of problems. In machine learning, classification is the problem of predicting to which category from a set of categories does a new instance belong. This prediction is made on the basis of a training set of data that is used to train the classifier. The training set contains instances with known categories. We used Weka to compare the performance of several classification algorithms. Weka allows us to choose different classifiers. In this report, we examine Bayesian networks [13], Alternating Decision Trees [7], K-nearest neighbor [18], Random Forests [16], J48 [29] and Support Vector Machines (SVM) [17]. We also tested bagging [15], which is a technique used to improve accuracy of models by generating multiple versions of a predictor and then using these to get an aggregated predictor. We used Decision Stump [24] and LADtree [23] bagging techniques for our experiments. Figure 3 shows the Weka knowledge flow components we defined and used to generate the results described in the next section.

4.7 Performance Evaluation

We measured the prediction accuracy by using the confusion matrix produced by Weka, which enabled us to compute the sensitivity and specificity of each classifier. Specificity is the probability that the classifier will label a "bad" entry as "bad". In this stage, we compared the performance of the machine-learning methods in terms of their sensitivity and the cost of building these classifiers. We used 10-fold cross validation on each dataset.

Feature name	Reduced Feature Set 1 (RS1)	Reduced Feature Set 2 (RS2)
avg-diagonal-dist	X	X
nnz	X	
norm1	X	X
min-nnzeros-per-row	X	X
norm1	X	X
row-variability	X	X
n-nonzero-diags	X	
kappa	X	X

Table 2: Reduced feature sets RS1 and RS2.



Method	T_{All}	T_{RS1}	T_{RS2}
LibSVM	6.88	1.94	1.79
RF	2.67	2.40	1.43
BayesNet	0.10	0.02	0.01
knn	0.001	0.001	0.001
ADTree	0.74	0.18	0.09
J48	0.24	0.19	0.06

Figure 4: Machine-learning algorithm comparison for PETSc linear solvers using full and reduced Anamod-based feature sets for training and prediction: "good as good" prediction accuracy (left) and the time (in seconds) for constructing the classifier with each method (right) using all features (T_{All}) and two different reduced feature sets ($T_{RS1}$and T_{RS2}) shown in Table 2.

For any given sparse linear system, predicting the best solver is impossible by using a purely analytical approach, i.e., without any empirical performance analysis [35]. Hence, we adopted the approach described in this work. Because we are using a binary labeling scheme ("good" and "bad"), the accuracy of the classifier is determined by measuring true positives (TP) and false negatives (FN). We focus on the true positive rate (TPR) because the main goal is to identify solution methods that are likely to perform well. Hence, the accuracy measures presented in Section 5 are computed using the usual true positive rate formula shown below:

$TPR = TP/P = TP/(TP + FN)$

where P is the actual number of positive instances, i.e., solvers labeled as "good".

The performance of a given method is strongly dependent on the problem features. The

Occurrence	Krylov Method	Preconditioner
3.8%	LSQR	Jacobi
3.6%	BiCG	ASM (0)
3.4%	BCGS	ILU (0)
3.2%	GMRES	ASM (2)
3.0%	BCGS	ASM (0)
2.8%	BiCG	BJacobi
2.3%	Chebyshev	Jacobi
2.2%	BCGS	ASM (3)
2.1%	iBCGS	ASM (2)
2.0%	CG	ASM (0)

Table 3: Top 10 good solvers for PETSc.

most appropriate solutions also depend on the specific input, the scale of the problem, and the available computing resources. The best choice of method depends on its application, as well. If a student in a linear algebra class is working on homework involving small matrices, a simple and easy-to-use sequential method will satisfy the particular need. On the other hand, a climate scientist seeking an efficient parallel solver for a very large system should ideally be guided to an HPC implementation such as those available in PETSc.

4.8 Solver Prediction

The choice of solver depends mainly on the features of the linear system in consideration. Therefore, it is likely that the solvers that perform well for systems recorded in our training set would also be applicable for any other systems that have similar characteristics. The feature set of such a new system and the set of solvers, without labels, are given as input to the classifier. The output is a prediction which labels each solver as "good" or "bad" according to its suitability for the new system. This information is further used to provide suitable solver(s) for a linear system given as input. This can be better explained with an instance. For example, a user has the input file, which has the linear system to be solved, and the user wants the list of suggestions for the solvers that are good and a list of solvers that are bad for that system. The flow in this case would be as follows. First, the user provides the input file which has the linear system. Next the user chooses to get the suggestions for solving the system. Finally, our infrastructure provides the suggestions for solvers. We also maintain the information about the solvers that are likely to be bad for the input in consideration.

5 Analysis and Results

For each linear system the features were computed, and then the system was solved using a specific solver configuration. A total of 4648 data points were considered in the dataset, and various machine-learning classification techniques were applied to classify the solvers. The classification has binary labels, "good" and "bad". The results were evaluated using 10-fold cross-validation [32]. The experiments for measuring the performance of the various

solver and preconditioner combinations in PETSc version 3.5.3 were performed on the Blue Gene/Q supercomputer. The classification was performed on an Intel Core i5 MacBook Pro.

In our work, we focus mainly on true positives (i.e., best-performing solvers predicted as best-performing). Our experiments include performance for the full feature set and two reduced feature sets. The first reduced set (RS1) is obtained by removing the features that do not contribute much towards the classification using Weka. The features in RS1 are:

1. **avg-diag-dist**: This is the average diagonal distance.
2. **nnz**: The total number of non-zeros in the matrix.
3. **norm1**: The maximum absolute column sum of A. It is given by the following formula. One norm of A is $\max_{1 \leq j \leq m} (\sum_{i=1}^{m} |a_{i,j}|)$
4. **col-variability**: The element variability in columns.
5. **min-nnzeros-per-row**: The minimum number of non-zeros per rows.
6. **row-variability**: The element variability in rows.
7. **n-nonzero-diags**: The number of diagonals that have any nonzero element.
8. **kappa**: The estimated condition number of the matrix.

We further reduced RS1 to create Reduced Set 2 (RS2). This was achieved by removing size-dependent features to evaluate the sensitivity of the classification process to the problem size. Our observations reveal that removing size-based features has minimal impact on the accuracy of the best classification. This interesting finding suggests that future classifiers can be created, without including data points for a wide range of matrix sizes. Among all machine-learning methods we applied, BayesNet showed the best performance in all cases: full feature set as well as both the reduced feature sets (RS1 and RS2). The accuracy was 87.60% with the full feature set, which has 68 input features. Using only eight computationally inexpensive features in RS1, the BayesNet-based classifier predicted good solvers correctly 86.91% of the time and 86.40% with RS2. RS1 and RS2 are shown in Figure 2. Figure 4 shows the true positive prediction accuracy of several of the machine-learning methods we tested for the full and reduced feature sets. We tested more methods than are included in the figure, but none of them performed better than the best method shown in the figure. We tested a subset of the configurations possible for the solvers and preconditioners and observed the occurrence of each of them to conclude which methods and preconditioners were the most successful. Table 5 summarizes these configurations that were most likely to perform well among all the configurations we tested.

6 Related Work

There have been several attempts and software developments for selecting efficient solvers for linear systems. One such attempt is the Linear System Analyzer (LSA) [14] which is a component-based environment that allows the user to specify combinations of preconditioner and solver, without the need to know the details of the implementation. Other similar

approaches include: a poly-iterative linear solver [6], which applies several iterative methods simultaneously to the same system; a composite multi-method solver [11, 12], where a sequence of solvers is applied to the linear system; an adaptive multi-method solvers approach [9, 26], where linear solvers are selected dynamically to match closely the evolving numeric properties of the linear systems; and a self-adapting large scale solver architecture (SALSA) [19], which uses statistical techniques such as principal component analysis for solver selection.

7 Challenges

There were many challenges involved in the entire process of providing solver suggestions for sparse linear systems. In this section we summarize those challenges. Given a linear system, we derived the features of those systems. The time required to compute the features varied depending on the category of the features. For instance, the cost of computation of the spectral features, is theoretically much higher than even solving the system. Running the solvers is yet another expensive step, and some of the methods do not even converge. Another challenging issue arose during the format conversion from Matrix Market to PETSc binary format. Matrix Market format does not store all diagonal entries, but PETSc format requires all diagonal entries to be stored, even when they are zero. This resulted in errors for many of the matrices.

8 Conclusion

This work demonstrates how machine-learning can be applied to select solution algorithms for large sparse linear systems. Given that solutions of linear systems are involved in many scientific and engineering problems, there is immense scope in automating the process of solver selection. Our study shows that we can rank the properties of matrices according to their share of contribution towards the classification process. The correlation between matrix properties and linear solvers is a recent topic of study among researchers, and understanding the structure of the classifiers can result in strong contributions in this field.

There can be many solutions for a linear system, and there also can be many machine-learning algorithms that can be applied to choose among these solutions. The more options available to choose from, the more confounding it becomes. We present [27, 33] a comparative analysis of the solver classification results for a variety of input problems belonging to different domains and various machine-learning methods, achieving up to 87% accuracy in identifying the well-performing linear solution methods in PETSc.

Acknowledgment

This work is supported by the U.S. Department of Energy Office of Science (Contract No. DE-SC0013869) and by the National Science Foundation (NSF) awards CCF-1219089, CCF-155063 and CCF-1550202.

References

[1] Matrix Market. http://math.nist.gov/MatrixMarket/, 2015.

[2] Portable, Extensible Toolkit for Scientific Computation (PETSc). http://www.mcs.anl.gov/petsc/, 2015.

[3] The Lighthouse Project. http://lighthousehpc.github.io/lighthouse/, 2015.

[4] The Trilinos Project. http://trilinos.sandia.gov/, 2015.

[5] The University of Florida Sparse Matrix Collection. http://www.cise.ufl.edu/research/sparse/matrices/, 2015.

[6] R. Barrett, M. Berry, J. Dongarra, V. Eijkhout, and C. Romine. Algorithmic bombardment for the iterative solution of linear systems: a poly-iterative approach. *Journal of Computational and applied Mathematics*, 74(1):91–109, 1996.

[7] S. Bhowmick, V. Eijkhout, Y. Freund, E. Fuentes, and D. Keyes. Application of alternating decision trees in selecting sparse linear solvers. 2010.

[8] S. Bhowmick, D. Kaushik, L. McInnes, B. Norris, and P. Raghavan. Parallel adaptive solvers in compressible PETSc-FUN3D simulations. In *Proceedings of the 17th International Conference on Parallel CFD*, 2005.

[9] S. Bhowmick, L. McInnes, B. Norris, and P. Raghavan. The role of multi-method linear solvers in PDE-based simulations. In *Computational Science and Its ApplicationsICCSA 2003*, pages 828–839. Springer, 2003.

[10] S. Bhowmick, P. Raghavan, L. McInnes, and B. Norris. Faster PDE-based simulations using robust composite linear solvers. *Future Generation Computer Systems*, 20(3):373–387, 2004.

[11] S. Bhowmick, P. Raghavan, L. McInnes, and B. Norris. Faster PDE-based simulations using robust composite linear solvers. *Future Generation Computer Systems*, 20(3):373–387, 2004.

[12] S. Bhowmick, P. Raghavan, and K. Teranishi. A combinatorial scheme for developing efficient composite solvers. In *Computational ScienceICCS 2002*, pages 325–334. Springer, 2002.

[13] C. Bielza and P. Larrañaga. Discrete Bayesian Network classifiers: A survey. *ACM Comput. Surv.*, 47(1):5:1–5:43, July 2014.

[14] R. Bramley, D. Gannon, T. Stuckey, J. Villacis, E. Akman, J. Balasubramanian, F. Breg, S. Diwan, and M. Govindaraju. The linear system analyzer. Technical report, Technical Report TR511, Indiana University, 1998.

[15] L. Breiman. Bagging predictors. *Machine Learning*, 24(2):123–140, 1996.

[16] L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, Oct. 2001.

[17] C. Cortes and V. Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, 1995.

[18] P. Cunningham and S. J. Delany. k-nearest neighbour classifiers. *Multiple Classifier Systems*, pages 1–17, 2007.

[19] J. Demmel, J. Dongarra, V. Eijkhout, E. Fuentes, A. Petitet, R. Vuduc, R. C. Whaley, and K. Yelick. Self-adapting linear algebra algorithms and software. *Proceedings of the IEEE*, 93(2):293–312, 2005.

[20] V. Eijkhout and E. Fuentes. A proposed standard for matrix metadata. Technical Report ICL-UT 03-02, University of Tennessee, 2003.

[21] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The WEKA data mining software: An update. *SIGKDD Explorations*, 11, 2009.

[22] N. Z. Hamilton. Correlation-based feature subset selection for machine learning. 1998.

[23] G. Holmes, B. Pfahringer, R. Kirkby, E. Frank, and M. Hall. Multiclass alternating decision trees. In *ECML*, pages 161–172. Springer, 2001.

[24] W. Iba and P. Langley. Induction of one-level decision trees. In *Proceedings of the ninth international conference on machine learning*, pages 233–240, 1992.

[25] R. Kirkby, E. Frank, and P. Reutemann. Weka explorer user guide for version 3-5-6. 2007.

[26] L. McInnes, B. Norris, S. Bhowmick, and P. Raghavan. Adaptive sparse linear solvers for implicit cfd using newton-krylov algorithms. In *Proceedings of the Second MIT Conference on Computational Fluid and Solid Mechanics*, volume 2, pages 1024–1028, 2003.

[27] P. Motter, K. Sood, E. Jessup, and B. Norris. Lighthouse : An automated solver selection tool. In *Software Engineering for High Performance Computing in Computational Science and Engineering*, Austin, Texas, 2015.

[28] B. Norris, L. C. McInnes, S. Bhowmick, and L. Li. Adaptive numerical components for pde-based simulations. *PAMM*, 7(1):1140509–1140510, 2007.

[29] R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers, San Mateo, CA, 1993.

[30] H. Rutishauser. The Jacobi method for real symmetric matrices. *Numerische Mathematik*, 9(1):1–10, 1966.

[31] Y. Saad. *Iterative methods for sparse linear systems*. Siam, 2003.

[32] J. Shao. Linear model selection by cross-validation. *Journal of the American Statistical Association*, 88(422):486–494, 1993.

[33] K. Sood, B. Norris, and E. Jessup. Lighthouse: A taxonomy-based solver selection tool. In *Software Engineering for Parallel Systems*, Pittsburgh, Pennsylvania, 2015.

[34] U. M. Yang et al. BoomerAMG: a parallel algebraic multigrid solver and preconditioner. *Applied Numerical Mathematics*, 41(1):155–177, 2002.

[35] W. Zhao, R. Chellappa, and N. Nandhakumar. Empirical performance analysis of linear discriminant classifiers. In *Computer Vision and Pattern Recognition, 1998. Proceedings. 1998 IEEE Computer Society Conference*, pages 164–169. IEEE, 1998.