SIAM J. SCI. COMPUT.
Vol. 38, No. 5, pp. S750–S771

PERFORMANCE-BASED NUMERICAL SOLVER SELECTION IN THE LIGHTHOUSE FRAMEWORK*

ELIZABETH JESSUP†, PATE MOTTER†, BOYANA NORRIS‡, AND KANIKA SOOD‡

Abstract. Scientific and engineering computing rely heavily on linear algebra for large-scale data analysis, modeling and simulation, machine learning, and other applied problems. 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. In the Lighthouse project, we enable application developers with varied backgrounds to readily discover and effectively apply the best available numerical software for their problems, aiming to maximize both developer productivity and application performance. Lighthouse is a search-based expert system built on a software taxonomy that combines expert knowledge, machine learning–based classification of existing numerical software collections, and automated code generation and optimization. In this paper we present the integration of PETSc and Trilinos iterative solvers for sparse linear systems into the Lighthouse framework. In addition to functional information in the taxonomy, we have created a comprehensive machine learning–based workflow for the automated classification of sparse solvers, which can be generalized to other types of rapidly evolving numerical methods. We present a comparative analysis of the solver classification results for a varied set of input problems and machine learning methods, achieving up to 93% accuracy in identifying the best-performing linear solution methods in PETSc and Trilinos.

Key words. linear algebra, taxonomy, machine learning

AMS subject classifications. 65Y10, 65F50, 15A06, 68N19

DOI. 10.1137/15M1028406

1. Introduction.

Scientists and engineers in a wide variety of disciplines rely extensively on linear algebra algorithms; see, e.g., [19, 46, 63]. The current high-performance implementations of numerical linear algebra software are based on decades of applied mathematics and computer science research. Hence, application developers who cannot rely on simple implementations because of the size or complexity of the problems they are solving *must* use optimized libraries developed by others. However, selecting a suitable library and using it effectively to solve a given problem can require a significant background in numerical analysis, high-performance computing (HPC), software engineering, and domain science. Indeed, discovering the best approach to 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 best-performing solution method is steadily decreasing.

*Received by the editors July 1, 2015; accepted for publication (in revised form) September 21, 2016; published electronically October 27, 2016.

http://www.siam.org/journals/sisc/38-5/M102840.html

Funding: This work was supported by National Science Foundation (NSF) award CCF-1219089. This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357. It also used the Janus supercomputer, which is supported by NSF award CNS-0821794 and by the University of Colorado Boulder.

†Department of Computer Science, University of Colorado, Boulder, CO 80309 (Elizabeth.Jessup@colorado.edu, Pate.Motter@colorado.edu).

‡Department of Computer and Information Science, University of Oregon, Eugene, OR 97403 (norris@cs.uoregon.edu, kanikas@cs.uoregon.edu).

A number of taxonomies exist to aid developers in the translation of linear algebra algorithms to numerical software; see, e.g., [1, 25, 45, 54]. However, these taxonomies do not provide accessible, comprehensive, and usable interfaces, nor do they supply tools for high quality code production. Lighthouse [49] is the first framework that offers an organized taxonomy of software components for linear algebra that enables functionality- and performance-based search and generates code templates and optimized low-level kernels.

In the Lighthouse project, we enable developers with varied backgrounds to readily discover and effectively apply the best available numerical software for their problems. Lighthouse is a search-based expert system that combines expert knowledge recorded in the taxonomy, machine learning (ML)–based performance classification of existing numerical software collections, and automated code generation and optimization. This novel software engineering environment is aimed at maximizing both developer productivity and application performance. It currently offers support for sequential dense linear algebra computations provided by LAPACK [2, 6] and for sequential and parallel sparse linear algebra computations provided by PETSc [7, 8, 9] and SLEPc [3, 40]. We are in the process of adding the Trilinos [61] parallel sparse linear algebra library as well.

Several projects have pursued automated solver selection and configuration, but none of them is easily generalizable. Neither have they produced usable software infrastructure that enables users to apply the methods to their own applications. Hence, our goal is to produce an extensible, general methodology for classifying algorithms and software that can be applied repeatedly as solvers evolve. Lighthouse can be used by computational scientists during the design and optimization stages of application development or as an educational tool for introducing high-performance numerical software to students.

The contributions in this paper can be summarized as follows:

- Integration into Lighthouse of a large number of PETSc and Trilinos preconditioned Krylov methods for parallel solution of sparse linear systems.
- A generalizable ML-based workflow for classifying arbitrary sparse linear systems using different sized feature sets.
- Comparison of several machine learning algorithms' performance for classifying the PETSc and Trilinos solvers.

This rest of the paper is organized as follows. Section 2 presents the background to this work. Section 3 reviews related work. Section 4 describes the approach we took to extending the Lighthouse taxonomy to sparse linear solvers. Section 5 covers our experimental results. Section 6 outlines our conclusions and future work.

2. Background. While, in this paper, we focus on the new Lighthouse support for PETSc and Trilinos [61] preconditioned Krylov methods, we briefly introduce the overall Lighthouse infrastructure in this section. The development of Lighthouse began with the selection of routines in LAPACK, an extensive collection of serial routines for solving a variety of linear algebra problems with dense matrices. LAPACK presented a relatively straightforward target for Lighthouse. The user's answers to a series of questions about a specific problem typically lead to exactly one LAPACK routine, except for those operations, including symmetric (generalized) eigenproblems or SVD, for which multiple relatively robust representation (MRRR) implementations [23, 66] are available. In the latter case, the user's answers may lead to two options: the standard QR algorithm and the MRRR algorithm, which is marked as faster but requiring more memory. The user can then choose based on resources available.

Lighthouse presently supports LAPACK functionality for linear systems, eigenvalue problems, SVD, and Sylvester matrix equations. We are in the process of adding LAPACK orthogonal factorizations and linear least squares problems. This paper describes the extension of Lighthouse to the sparse linear algebra domain (not supported by LAPACK), which presents a substantially more difficult target. As discussed in [28], in iterative sparse linear system solving, there are a large (and growing) number of preconditioners and solution methods. The convergence behavior of a given method on a given system cannot be predicted without prohibitively expensive computation (e.g., computing the eigenvectors of the preconditioned system), making this approach impractical. And while previous efforts described in more detail in section 3 have used various approaches to classify numerical solution methods, none so far has produced an extensible framework that can be used in production applications that rely on continuously evolving numerical software packages.

2.1. Using sparse linear algebra libraries. The first problem of interest is the solution of sparse linear systems with routines from the PETSc package [7, 8, 9]. In structure, PETSc resembles LAPACK as a collection of parallel routines for direct solvers, Krylov iterative methods, and preconditioners that can be used in application codes written in C, C++, Fortran, and Python. While PETSc also addresses other aspects of the scalable solution of systems of PDEs, our current focus is on iterative methods and preconditioners.

The process of getting started with PETSc is reasonably well supported. There are multiple 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 commands to configure and build it in the appropriate PETSc tutorial or other online PETSc documentation. Needed packages are automatically downloaded, configured, built, and installed with PETSc. A number of PETSc examples instruct the user on writing PETSc programs and setting command line options.

Selecting the appropriate PETSc routines, however, presents a substantially more difficult problem compared to choosing a LAPACK routine. A sparse solver is typically paired with a preconditioner. To the uninformed user, the set of parallel Krylov methods and preconditioners contained in PETSc and summarized in Table 1 suggest that there are more than 300 possible pairings. Which iterative solvers and preconditioners from this collection are the best choices for a given linear system depends on properties of its coefficient matrix and may also depend on the physics of the problem. Determining how to choose requires a search of the extensive numerical linear algebra literature and may also require reading in the domain science. The performance of a chosen solver-preconditioner pair, in turn, strongly depends on structural and spectral 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 computed solution. Neither specific system features nor parameter details are typically addressed in the literature. As a result, achieving best performance is generally a matter of experimenting with a variety of options.

All libraries for the solution of sparse matrix algebra problems inherit the same difficulties in selecting the best routines, and some introduce new complications of their own. As we proceed with PETSc, we are also working on extending Lighthouse to include Trilinos. Trilinos is an open source framework designed for creating scientific applications.

Like PETSc, Trilinos consists of parallel routines for direct solvers, Krylov it-

TABLE 1
PETSc parallel Krylov iterative solvers and preconditioners.

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

erative methods, and preconditioners. Trilinos also contains functionality for other aspects of scientific computing, such as load balancing and meshing. Each of these computational areas is handled by individual packages within Trilinos. Each package is a self-contained module created by an independent group and designed to solve a specific problem commonly found in scientific computing. These packages interface with each other when appropriate using common matrix and vector objects provided by packages such as Epetra or Tpetra.

Almost all of Trilinos's packages are written in C++ and are designed to be used in other C++ codes. However, there is support for using Trilinos with C, Python, and Fortran codes via included wrappers.

Due to the large number of packages and numerous options for tweaking individual parameters, choosing an efficient set of options in Trilinos can prove difficult. As of the most recent release (12.0.1), Trilinos consists of roughly 60 unique packages, the relevant subset of which is shown in Tables 2 and 3. This broad range of options allows a user to solve a problem effectively but also creates a complex search space for selecting an appropriate combination of these packages. A simple linear algebra problem solved using Trilinos can require as many as five packages, while a complicated problem may take many more. Each package has its own unique collection of C++ classes and functions, with very little if there is any similarity between packages.

When first starting to use Trilinos, a new user is directed to the *Trilinos Hands-on Tutorial* [62]. which features examples and lessons designed to help someone new to the library. Unfortunately, the tutorial covers only a small subset of available packages with varying thoroughness. Being able to use a package requires looking

TABLE 2
Trilinos packages for solving linear system and eigenvalue problems.

Capability		Package(s)
Services		
	Linear algebra objects	Epetra, Jpetra, Tpetra, Kokkos
	C++ utilities, I/O	Teuchos, EpetraExt, Kokkos, Triutils, ThreadPool, Phalanx, Trios
Linear solvers		
	Iterative solvers	AztecOO, Belos, Komplex
Preconditioners		
	ILU-type	AztecOO, Ifpack, Ifpack2
	Multilevel	ML, CLAPS, Muelu
	Block	Meros, Teko

TABLE 3
Trilinos parallel Krylov iterative solvers and preconditioners.

Capability	Algorithm
Ifpack2 preconditioners	ILUT, RILUK, Diagonal, Relaxation, Krylov, Chebyshev
Belos methods	Block GMRES, Hybrid Block GMRES, Pseudoblock GMRES, Recycling GMRES, Block CG, Pseudoblock CG, Pseudoblock Stochastic CG, Recycling CG, MINRES, LSQR, TFQMR, Pseudoblock TFQRM, Preconditioned Conjugate Projected Gradient, Fixed Point

at the documentation for the necessary classes and methods defined within it. The documentation consists of a handful of examples for certain packages and Doxygen documents for every package's source code. In some cases it is necessary to examine the source code itself when the documentation is lacking. Each package's documentation is maintained by the developers of that specific package. Therefore large variation can be seen in the quality of documentation and examples across various packages.

Rarely does a new or even average user need to use all of the available packages, since some can be viewed as being mutually exclusive. For example, Epetra and Tpetra are the two core packages of Trilinos for creating linear algebra objects like matrices, vectors, and graphs. Because both Epetra and Tpetra provide very similar objects, most packages are designed to work with only one or the other, but rarely both. For example, the Trilinos packages Ifpack and Ifpack2 perform matrix preconditioning, but the former provides support only for Epetra while the latter supports only Tpetra. There are a small number of packages that do support both: Amesos2, a direct solver for sparse linear systems, is one such package providing functionality for matrices and vectors defined in either Tpetra or Epetra [61].

There is not currently a prebuilt or easy install version of Trilinos available, so it must be built from source. All packages are included when downloading Trilinos but are built and installed only if the user specifies in the CMake file that they should be. While Lighthouse does not address the challenges in building complex software packages, it can help determine which Trilinos packages are actually needed based on a high-level description of the user's problem and hence simplify the installation process.

2.2. The Lighthouse framework. We have implemented a prototype of the Lighthouse framework [49] for assisting scientists, engineers, and students with the

implementation of the matrix algebra computations that dominate many high-performance applications. Lighthouse is the first framework that combines a matrix algebra software taxonomy with code generation and tuning capabilities. In addition, its user-friendly interfaces can accommodate users with different backgrounds via different search interfaces and code generation options. For example, after the search is performed, Lighthouse generates a complete program template in C or Fortran that correctly uses the search result. The template can be downloaded, compiled, and run immediately, provided the user has installed the corresponding library on their computer. This template can be used as a starting point for an application or as a component to be extended and integrated into an existing application. The Lighthouse interface is a working product for the LAPACK and SLEPc packages. It is a work in progress for linear solvers from PETSc and Trilinos. Next, we give an overview of the design and implementation of the Lighthouse prototype.

2.2.1. Lighthouse for LAPACK. Lighthouse was inspired by the LAPACK Search Engine [45] and two classroom usability studies of later search prototypes. As one of the most widely used serial dense direct solver packages, LAPACK was the logical first choice for inclusion in the Lighthouse taxonomy. As noted earlier, Lighthouse has broad functionality across LAPACK. In this section, we discuss the interface for the solution of systems of linear equations. This part of Lighthouse contains over 800 LAPACK subroutines.

The taxonomic information for LAPACK is stored in a MySQL database in which the subroutines are first categorized by the kinds of tasks they perform. The Lighthouse system then sorts and identifies 11 different matrix types based on five different storage properties. It then categorizes the precision level (single or double) and parameter type (real or complex) of the subroutines.

The database-driven user interface of Lighthouse is implemented as a Django [24] application. Django provides a dynamic database access application programming interface (API) using the Python programming language and supports an automatic administrative interface that makes future data maintenance simple and convenient. Through pairing with Haystack [39], a modular search application for Django that offers powerful database queries and multiple search indices, Lighthouse enables LAPACK subroutine search via three methods: guided search, advanced search, and keyword search. In the guided search interface, users are prompted to answer increasingly detailed questions describing the problems they wish to solve. Portions of the interface are automatically generated based on earlier responses using Django dynamic forms and the Django session framework. After answering the last question, a user sees exactly one subroutine that matches all of the answers. Populated with numerous help buttons, Lighthouse also serves as an educational tool. Information about the library, the routines, and the definition of a word or a phrase can be accessed easily by clicking on the help buttons.

The guided search dialogue for solving a dense linear system with LAPACK is depicted in Figure 1. Unlike the guided search, the advanced search is designed for users who are familiar with LAPACK. The advanced search interface provides users with a form containing checkboxes where users can make multiple selections, enabling the simultaneous search for multiple subroutines in different types of routine categories.

The keyword search interface supports keyword-based search of the taxonomy information. In order to enhance the effectiveness and efficiency of the keyword search, Lighthouse provides automatic completion and spelling correction with words col-

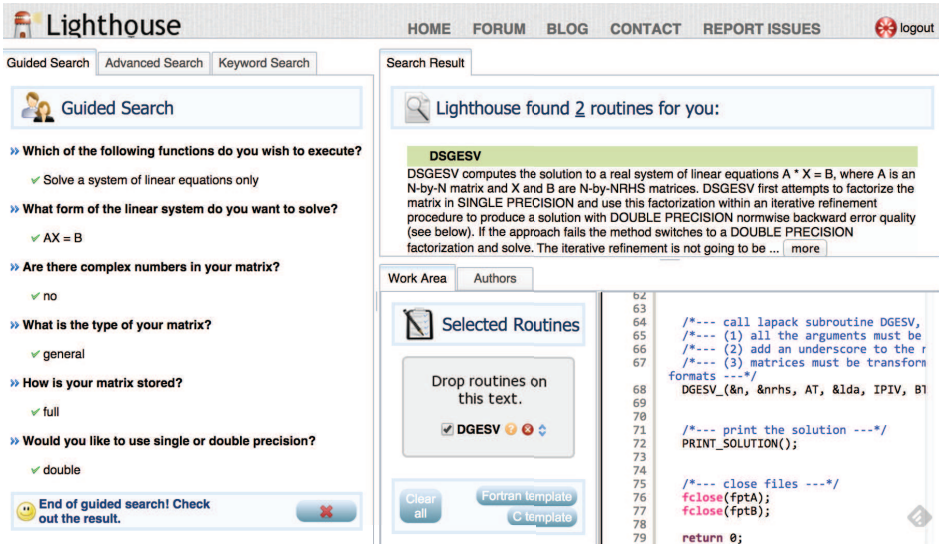


FIG. 1. *Guided search dialogues in the Lighthouse LAPACK interface showing the result and a portion of the generated code template for using a* dense linear *solver.*

lected from linear algebra textbook indices. In addition, Lighthouse uses a list of linear algebra keywords and phrases for Django-Haystack filtering in order to reduce search time.

Drag-and-drop functionality allows users to select a routine retrieved by Lighthouse by dragging it to the Selected Routines work area for generating a code template. Lighthouse currently offers two code template languages: Fortran90 and C. The template helps users to explicitly declare the arguments and correctly construct routine calls. The template program is split into several subprograms, making it easier for users to modify the code. Moreover, Lighthouse code templates contain some decision logic to enable tailoring of the solution to specific problem characteristics. For example, code templates for equilibration can also determine whether a provided matrix is worth scaling. If it is, a method of matrix scaling (row, column, or row-and-column) is automatically selected and executed.

In addition to the search component, Lighthouse provides a client interface to the Build to Order (BTO) BLAS compiler [12, 44] for generating custom autotuned C matrix algebra computations based on high-level MATLAB-like input. Lighthouse connects to a server running BTO, which generates optimized C implementations of these operations.

The design and implementation of Lighthouse for LAPACK taxonomy are described in more detail in [56].

2.2.2. Lighthouse for SLEPc. SLEPc is a toolkit for solving large sparse eigenvalue problems and is built on top of PETSc. The interface of Lighthouse for SLEPc relies on the same technology just described for Lighthouse for LAPACK, and its use is similar. A user of the SLEPc guided search again answers a series of questions about the problem. The user also must enter some specific information that cannot be encoded with simple questions: the matrix order, the desired number of eigenvalues, and the residual tolerance. Because SLEPc provides parallel solutions, the user must also enter the number of processors to be used.

The main difference between Lighthouse for LAPACK and Lighthouse for SLEPc is in the decision process. Most of the eigensolvers available in SLEPc are compatible with most types of problems. However, not all solvers are equal in terms of efficiency. To find the most efficient eigensolvers, we use an ML approach. We performed experiments on compatible SLEPc solvers with varying problem domains recording such statistics as time to solution, residual, and number of iterations. In this way, we collected more than 10,000 experimental data points using real matrices obtained from Matrix Market [55] and the University of Florida Sparse Matrix Collection [21] to serve as a training set for a decision tree [37] for classifying algorithms. With the information from the classification, Lighthouse delivers not a unique result but rather a set of eigensolvers all predicted to have performance within 10% of the top-performing eigensolver. The user can then select a solver from this set and generate a code template for it.

2.2.3. Lighthouse for system solution. We have in place a preliminary version of Lighthouse for PETSc. As is true for the other packages, the guided search user interface for PETSc is an interactive system that enables users to generate and download PETSc programs for solving sparse linear systems. The style of this interface is different, however. To begin, the user has the option to upload a coefficient matrix.[1] If the user chooses to do so, Lighthouse computes a variety of matrix features and uses those values in an ML classification algorithm to predict a good-performing solver for a system using that matrix. This algorithm is rudimentary and does not yet rely on the detailed analysis presented in this paper. If the user does not upload a matrix, Lighthouse offers a choice of downloading either a PETSc program for computing matrix properties or else a general PETSc program for solving a linear system. In the latter case, the program is generic without a Krylov method or preconditioner specified, so it is a reasonable alternative only for an experienced PETSc user. We will continue work on Lighthouse for PETSc and Trilinos using the work that we describe in sections 4 and 5 of this paper.

3. Related work. We discuss briefly some of the most relevant prior work in numerical software taxonomies and performance-based algorithm classification.

3.1. Other taxonomy efforts. A number of taxonomies exist to aid developers in translation of matrix algebra algorithms to numerical software. Perhaps the oldest one is the Netlib Mathematical Software Repository [52], started in 1985, which contains freely available software, documents, and databases pertaining to numerical computing, including matrix algebra. Contents are provided as lists of packages or routines, with or without some explanatory words. In newer work, the Linear Algebra Software Survey [25] lists more than 100 items categorized as support routines, dense direct solvers, sparse direct solvers, preconditioners, sparse iterative solvers, and sparse eigenvalue solvers together with a checklist specifying problem types for each entry. NIST's Guide to Available Mathematical Software (GAMS) [54] includes even more basic matrix algebra software along with software for a variety of other numerical applications. While the Linear Algebra Software Survey is a linear list, GAMS allows search by problem solved, package name, module name, or text in module abstract. The now discontinued HotGAMS [53] Java-based client allowed an interactive search of the GAMS repository. Both the Survey and GAMS index into Netlib for software

[1] While currently only the PETSc binary format is supported, new matrix formats will be added in the near future.

downloads. Another example is the LAPACK Search Engine [45], which provides a simple way to search the list of LAPACK routines from Netlib.

Existing numerical software taxonomy approaches are general and allow relatively stand-alone algorithms to be found, downloaded, and compiled or used perhaps through a domain-specific Web interface. Operations for which no library implementation exists or more complex software packages, however, cannot be accommodated by this function-level indexing and query capability. For example, the functionality of large toolkits, such as Trilinos and PETSc, is difficult or impossible to represent and maintain in most taxonomies, which at present simply point the user to the toolkits's home pages and do not offer support for selecting solution methods based on both functional and performance requirements.

3.2. Performance-based algorithm classification. Several previous efforts relate to what we describe in this paper, although none exactly matches in function. The goal of Self-Adapting Numerical Algorithms (SANS) [26], for example, is to build a common framework for different tools for the optimization of software. In particular, SANS is an umbrella for ATLAS [65] and generic code optimization (GCO) [34], which was part of the DOE SciDAC PERI project [50]. Both projects use empirical search to create portable high-performance codes. ATLAS is directed at matrix algebra, while GCO produces implementations for adaptive multiresolution methods in multiwavelet bases. ATLAS has been used to create tuned versions of the Basic Linear Algebra Subprograms (BLAS) [1] and some LAPACK routines, and we expect that it will have a place in the Lighthouse taxonomy in the future. Our goal is to allow the integration of such existing performance tools into Lighthouse, not replicate their functionality. Another related taxonomy example not from linear algebra is the decision tree for optimization software at Arizona State University [51]. It refers to Netlib entries if available or points directly to the home page of the software package.

This work builds on results of researchers (including one of the authors) who have previously used ML to identify "good" solvers in the context of parallel nonlinear PDE solution [15, 16, 57]. In the area of sparse linear system solution, several ML approaches are used to classify a limited set of methods [13, 14, 29, 41, 67]. Barrett et al. [10] introduce the idea of algorithmic bombardment, which composes multialgorithm approaches by executing several Krylov methods simultaneously. Bhowmick et al. [16] introduce composite linear solvers, in which several different Krylov methods execute in sequence. Kotthoff, Gent, and Miguel [47] evaluate the performance of several ML algorithms on five datasets of hard algorithm selection problems from the literature. Weerawarana et al. [64] present a knowledge-based system called PYTHIA for selecting methods from Parallel ELLPACK [43], a package of routines for solving elliptic PDEs. PYTHIA matches the features of a given problem with those of PDEs in a known collection and then uses performance information about solvers to match one based on user-supplied error and time bounds. While similar in style to Lighthouse, PYTHIA requires substantially more human participation than does Lighthouse. Eijkhout and Fuentes [28] present a comprehensive approach to the classification and selection of preconditioned sparse iterative solvers, including the development of the Anamod feature extraction library, which we are using in our work. At a high level, our approach is similar to that of [28], and in fact we plan to incorporate some of the hierarchical classification ideas in our future work. Unlike most previous research efforts, we aim to completely automate the classification and recommendation process and ensure that the framework is general enough to enable the integration of many different types of numerical software methods and HPC packages.

The success of these previous explorations into performance-based multisolver methods motivates the work presented in this paper, which is not limited to a particular problem domain and considers a broader range of matrices, solvers, and ML methods.

4. Approach. The most straightforward approach to building a taxonomy of mathematical software is the one that we have followed in constructing Lighthouse for LAPACK. The first step is to enumerate methodically all provided algorithms, as well as their inputs and outputs. For each problem, we then construct a decision tree, where each node corresponds to making a decision related to the problem being solved, e.g., *what form of linear system do you want to solve?* or *is the coefficient matrix symmetric?* The leaves of the decision tree are single algorithms. The root of the tree is the fundamental problem statement of the problem to be solved. The best algorithm is identified by traversing the tree from root to a single leaf by answering the question posed at each node along the way. The design and implementation of the Lighthouse for LAPACK taxonomy are described in more detail in [56].

In the case of serial or parallel sparse linear solvers, a taxonomy of the LAPACK sort that contains only functional descriptions of methods is not sufficient. The performance of a given method is strongly dependent on the problem features. The 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 nsolver for a very large system should ideally be guided to an HPC implementation such as those available in PETSc. In this section, we describe our efforts to incorporate performance awareness into the taxonomies for PETSc and Trilinos.

To classify solvers based on their performance for inputs with given characteristics (features), we employed several supervised ML techniques. Supervised learning involves designing a classification function based on a set of already classified data [38]. The training set is used to build the classifier, and the testing set is used to verify the accuracy of the classifier. This process is repeated k times (k-fold cross validation), each time with a different subset as the testing set. The results are combined to produce the final classifier that is applied to testing sets. A binary classifier determines in which of two groups to classify an unknown entry. A tertiary classifier determines in which of the three groups to classify an unknown entry. In this case, each entry is a combination of a linear solver and preconditioner method with certain configuration parameters. The prediction accuracy of a binary classifier is measured by the sensitivity and specificity. If the labels are "good" and "bad" or "good", "fair", and "bad", the sensitivity is the probability that the classifier will predict a "good" entry as "good" and the specificity is the probability that a "bad" entry will be predicted as "bad". Our primary software for this work was Weka [35], which is a collection of ML algorithms for data mining tasks. The sensitivity and specificity information is produced by Weka in the form of a confusion matrix [31].

To enable performance-aware algorithm discovery for sparse linear solvers in the proposed taxonomy, we developed a multistep approach, which we describe in the remainder of this section.

4.1. Creating the training dataset. For PETSc, we used a total of 154 solver and preconditioner configurations to solve a set of sparse linear systems derived from 1,015 input matrices in the University of Florida Sparse Matrix Collection [4, 21] with

all right-hand-side elements set to one. We shuffled the resulting large set of data points and then selected the first 10,000 points to use for further testing. For those systems, we captured the time taken to solve the system, the number of iterations, and solver and preconditioning options like number of blocks and overlap. For Trilinos, we considered a total of 67 parallel solver and preconditioner configurations, solving a set of 1,429 matrices from the same collection.

4.2. Training data preparation. The next step was to convert the data in the training set into a form that is usable by Weka. In particular, the input to the learning process was 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 configurations. We refer to each such combination as a method $S_i, i \in \{1, N\}$, for N possible solvers. Given M input matrices, an exhaustive training dataset consists of $M \times N$ data points. Because this number can be prohibitively large, we constructed the training set by computing a smaller number of randomly selected points, $P_{i,j}$, $j \in \{1, M\}$. For binary labeling, we labeled each point $P_{i,j}$ as "good" or "bad" or for tertiary labeling as "good", "fair", or "bad"based on the performance of the solver S_i on matrix M_j based on a threshold parameter b in the range $\{0, 1\}$ specifying how close S_i's performance is to the known best-performing method. For tertiary labeling a threshold parameter r in the range $\{\mathrm{b}, 1\}$ was considered in addition to the parameter b for labeling solvers as "fair". For example, for binary labeling, when $b = 25$, solvers whose performance for a given problem is within 25% of the best were labeled "good", while all other solvers were labeled as "bad".

4.3. Feature computation. For both PETSc and Trilinos, we used Anamod [27] to extract 68 features of the coefficient matrices. In particular, we computed linear system attributes in several categories, including simple (norm-like quantities), variance (heuristics estimating how different matrix elements are), normality (estimates of the departure from normality), structure (nonzero structure properties), and spectrum (eigenvalue and singular value estimates produced using SLEPc). To provide an easy way to compute features for Trilinos applications without requiring users to install PETSc and Anamod, we also computed a smaller, 38-feature set with Trilinos, which mostly overlaps with some of the Anamod features but does not require PETSc.

4.4. Feature set reduction. The cost of computing features varies widely from milliseconds to minutes or hours depending on the time out parameter settings for interrupting nonconvergent feature computations (e.g., iterative eigenvalue algorithms). Hence, in order to reduce the overall cost of the process, we performed analysis to remove features that do not contribute significantly to the accuracy of the classification.

First, we reduced the number of features by using Weka's RemoveUseless filter. This filter removes the data points corresponding to features whose values either remain constant or else vary too much (over 99% variance). Using this simple filter reduced our number of features from 68 to 54 and typically improved 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 worth 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 do not contribute much to the classification. The evaluators we used

are Gain Ratio, ChiSquared, CfsSubset, Information Gain, and Principle Component Analysis [35]. The search methods we chose were GreedyStepwise and Ranker [36]. Tables 7, 9, and 10 show reduced feature sets for PETSc and Trilinos solvers, which are those determined to be the best by all or a majority of the aforementioned evaluators for maximizing the classifier's true positive rate or sensitivity ("good" as "good" predictions).

The best features are expected to vary if new feature sets are evaluated and reranked. Computing the smallest eigenvalue with either Trilinos or SLEPc can take on the order of 10^{-2} seconds for relatively small matrices ($< 1,000,000$ nonzeros), while a bandwidth computation requires 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 minimal runtime overhead of selecting a good linear solver configuration.

4.5. Solver classification. The next step was the classification of the data. We used Weka [35] to compare the performance of several classification algorithms. Weka allows us to choose different classifiers. In this paper, we examine Bayes Net [17], k-nearest neighbor [5], Alternate Decision Trees (ADT) [33], multiclass extension of Alternating Decision Trees (LADT) [42], Random Forests [58], J48 [59], Voting Feature Interval (VFI) [22], and Support Vector Machines (SVM) [20]. We also tested bagging [18], which is a technique used to improve accuracy of models by generating multiple versions of a predictor and then using them to get an aggregated predictor. We used Decision Stump [32] and LADT bagging techniques for our experiments. Figure 2 shows the Weka knowledge flow components we defined and used to generate the results described in section 5.

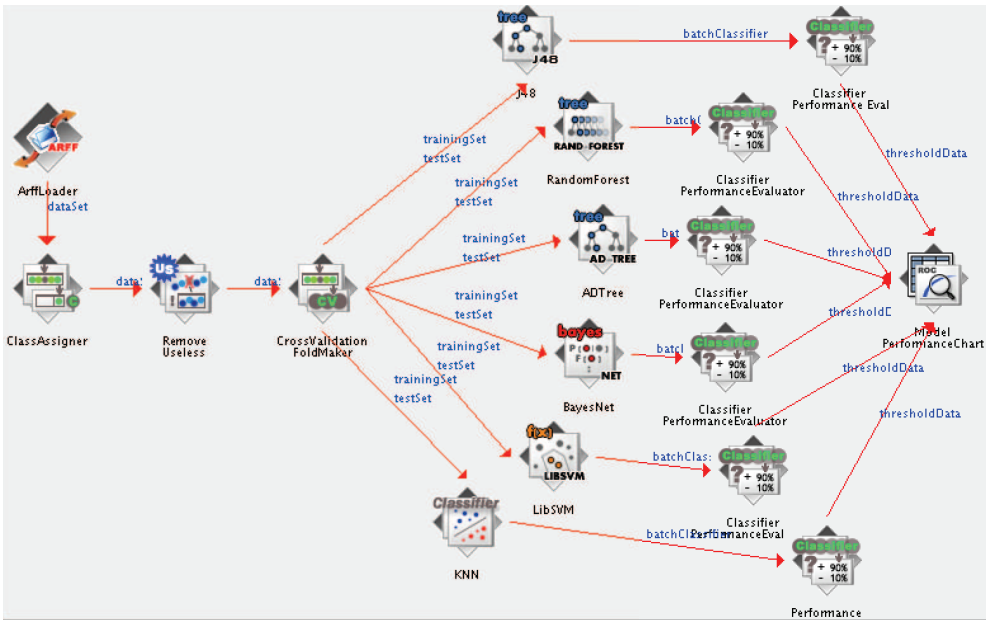


FIG. 2. *Weka workflow showing a subset of the classifiers. The same workflow was used for the full and reduced feature sets.*

4.6. Performance evaluation of the classification. We measured the prediction accuracy by using the confusion matrix produced by Weka, which enabled us to compute the sensitivity (percent of accurate "good as good" predictions) and specificity (percent of accurate "bad as bad" predictions) of each classifier. In this stage, we compared the performance of the ML methods in terms of their sensitivity and the cost of building these classifiers. We used 10-fold cross validation on each dataset. Tables 8 and 12 are the confusion matrices for PETSc and Trilinos for a 66%-34% train-test data split. Tables 6 and 11 are the confusion matrices for PETSc and Trilinos for 10-cross validation. These tables show the overall accuracies for the good and bad solvers identified by the classifiers for PETSc and Trilinos.

Predicting what the best solver is for any given sparse linear system is impossible by using a purely analytical approach, i.e., without any empirical performance analysis [30]. Hence, we adopted the empirical approach described here. 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 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: $TPR = TP/P = TP/(TP + FN)$, where P is the actual number of positive instances, i.e., solvers labeled as "good".

5. Experimental results. We performed the steps described in section 4 to construct classifiers that label a solver and preconditioner combination S_i as either "good", "fair", or "bad" based on the features of the input linear system. The performance data for Trilinos was collected using Ifpack2 for preconditioning, Belos for iterative solvers, and Tpetra for the common data structures. We collected performance data for different S_i chosen from PETSc and Trilinos on two architectures; the results are described in sections 5.1 and 5.2, respectively. We also performed a preliminary combined analysis using solver timings from both toolkits together, but because we do not yet have complete results for both toolkits on the same architectures and for the same set of matrices, we do not discuss it here. More comprehensive training data collection is planned as future work as the software infrastructure we are developing automates more of the process. All training data and Weka workflows used in this paper are available at the Lighthouse web site [49].

Table 5 summarizes the solver configurations that were most likely to perform well among all the configurations we tested and lists the most-frequently used solver-preconditioner combination for our experiments (described in more detail in the remainder of this section). For the PETSc solvers, the current analysis considers only sequential runs, while the Trilinos results reflect small-scale (12 MPI tasks) parallel runs.

While each solver has its own unique parameters, each library does contain some default parameters that are similar between solvers. More information can be found in the documentation of Trilinos's Belos package [11] and PETSc's Krylov subspace component [48]. Default solver parameters are shown in Table 4.

5.1. PETSc solver analysis. We collected solver performance information with PETSc version 3.5.3 on two supercomputers: a Blue Gene/Q at Argonne National Laboratory and the Aciss cluster at the University of Oregon consisting of nodes containing two hex-core 2.66GHz Intel Westmere (X5650) processors and 72 GB of RAM. Because the University of Florida matrices are not very large, each experiment used a single node. We performed binary and tertiary labeling for the PETSc solvers, and the results indicate that tertiary labeling outperformed binary labeling. There-

TABLE 4
Default solver parameters in PETSc and Trilinos's Belos.

	PETSc	Trilinos
Maximum iterations	10000	1000
Residual tolerance	$1.0e^{-5}$	$1.0e^{-8}$
GMRES restart size	30	20

TABLE 5
Top 10 *solvers that were labeled as "good" as a percentage of all "good" solvers for PETSc and Trilinos.*

PETSc		Trilinos	
31.6%	LSQR, ASM(1)	17.2%	Hybrid Block GMRES, Chebyshev
16.3%	GMRES, ILU(0)	9.4%	Hybrid Block GMRES, Diagonal
14.3%	LGMRES, ILU(3)	8.8%	Hybrid Block GMRES, Jacobi
12.5%	FGMRES, ASM(0)	3.9%	TFQMR, Chebyshev
8.0%	TFQMR, ILU(2)	3.7%	Hybrid Block GMRES, ILUT
7.7%	GMRES, ILU(2)	2.8%	Hybrid Block GMRES, RILUK
5.6%	BiCG, ASM(0)	2.8%	Block CG, Jacobi
1.4%	TCQMR, ILU(3)	2.7%	Block CG, Diagonal
1.3%	CG, ILU(2)	2.4%	Pseudoblock CG, Jacobi
0.2%	BCGS, ICC(3)	2.4%	TFQMR, Jacobi

fore we present the tertiary labeling results in this section. Some of the solvers had substantially more data points than others because of the simple random selection method we used. In order to balance the amount of data for different solvers in the complete dataset, some of the data points (i.e., the solvers for which fewer than 10 timing results are available) are removed. This operation leaves us with a total of 16,861 data points on the Blue Gene/Q and 5,437 data points on the Xeon cluster. These datasets are split into training and test subsets in the two types of validation described next.

For 10-fold cross-validation of the classification on the Blue Gene/Q, using all 68 features computed by Anamod, the VFI [22] classifier, a very fast classification method based on voting feature intervals (the same algorithm that performed best for the Trilinos solver classification described in section 5.2) had the best true positive rate (TPR) of 88.4% (Table 6). The best TPR of 82.9% was delivered by Random Forest when we redid the classification with the eight features in Reduced Feature Set 1 ($RS1$) shown in the second column of Table 7. The best accuracy for Reduced Feature Set 2 ($RS2$) was 82.2 % and was achieved by Random Forest, as shown in the third column of Table 7. We created $RS2$ by removing size-dependent features to evaluate the sensitivity of the classification accuracy to problem size. We observe that removing size-based features has minimal impact on the accuracy of the best classification.

TABLE 6
10*-fold cross validation for PETSc (Blue Gene/Q).*

Class labels	All features			Reduced Feature Set 1			Reduced Feature Set 2		
Predicted label	good	fair	bad	good	fair	bad	good	fair	bad
True label (good)	1206	33	126	1131	180	54	1122	187	56
True label (fair)	5435	8680	984	114	14957	28	127	14935	37
True label (bad)	327	9	61	120	128	149	120	135	142

On the Aciss cluster, we achieved TPR of 84.3% ($RS1$) and 83.7% ($RS2$) with Random Forest. At the time of writing, we have comprehensive data for a relatively small number of solvers on this architecture and hence the success of a relatively simple classification method. We expect that as our training dataset grows, we may find a different best classification algorithm for Aciss.

For the 66%-34% train-test data split of the Blue Gene/Q dataset, when using all 68 features computed by Anamod, VFI performed the best with an accuracy of 86.5% (Table 8). With reduced feature sets $RS1$ and $RS2$, accuracies of 77.9% and 78.1% were achieved, respectively, by Random Forest. For the Aciss dataset, the 66%-34% train-test data split resulted in over 82% accuracy for the reduced feature sets. The full feature set TPR accuracy was at 78.3% (using J48). These results suggest that future classifiers can be created without having to include data points for a wide range of matrix sizes.

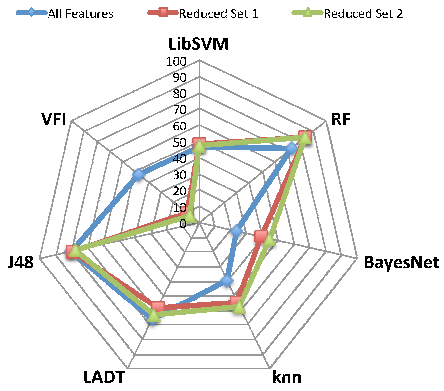
TABLE 7
Reduced feature set for PETSc.

Feature name	Reduced Set 1 ($RS1$)	Reduced Set 2 ($RS2$)
Average distance of nonzero diagonal to main diagonal	X	X
Total number of nonzeros	X	
1-norm, maximum column sum of absolute element sizes	X	X
Column variability: $\max_j \log_{10} \frac{\max_i \lvert a_{ij} \rvert}{\min_i \lvert a_{ij} \rvert}$	X	X
Minimum number of nonzeros per row	X	X
Row variability: $\max_i \log_{10} \frac{\max_j \lvert a_{ij} \rvert}{\min_j \lvert a_{ij} \rvert}$	X	X
Number of diagonals that have any nonzero element	X	
Estimated condition number	X	X

TABLE 8
Validation with 66%-34% train-test data split for PETSc (Blue Gene/Q).

Class labels	All features			Reduced Feature Set 1			Reduced Feature Set 2		
Predicted label	good	fair	bad	good	fair	bad	good	fair	bad
True label (good)	411	12	52	370	83	22	371	86	18
True label (fair)	1742	2973	399	41	5069	4	41	5066	7
True label (bad)	113	5	26	52	46	46	48	52	44

Figure 3(left side) is a radar chart that shows the performance of the six ML methods we tested for the full and reduced feature sets (invoked with default Weka parameters). The radar chart contains 10 radial axes, with values ranging between 0% and 100% from the center to the perimeter in increments of 10%. An accuracy of 100% indicates a classifier that predicts the good classifiers correctly each time. An accuracy of 0% indicates a classifier that fails to predict good classifiers in each case. We tested more methods than are included in the figure, but none performed better than the best method shown in the figure. The threshold used for labeling the dataset in this case was $b = 40\%$ and $r = 60\%$, which means that methods within the top 40% of the best solver time were labeled as "good" and labeled as "fair" for all the cases with times greater than the "good" criterion but within 60% of the minimum. This value of b was chosen as the best among several sampled values between 1 and 45 for the BG/Q dataset. The total number of actual "good" class instances was 1,365. For the Aciss dataset, we were able to choose $b = 40\%$, for which the actual number of "good" class instances was 1,135. The classification was performed on an Intel Core



Method	T_{All}	T_{RS1}	T_{RS2}
LibSVM	1.21	0.32	0.31
RF	4.25	2.84	2.88
BayesNet	0.09	0.02	0.02
knn	0.001	0.001	0.001
LADT	3.3	0.5	0.4
J48	0.56	0.11	0.11

FIG. 3. *ML 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 7.*

i5 MacBook Pro.

We analyzed the impact of our PETSc solver selection strategy on the execution time on Aciss by comparing the execution time of automatically selected good solvers (using the approach described in section 4) with that of the default PETSc solver configuration. We used a speedup value s computed as default solver time divided by recommended solver time (higher is better).

When solvers classified as good were actually good (true positives), which occurs in 82% of the testing dataset, the maximum improvement was $s = 10,331$, the average improvement was $s = 467$, median $s = 1.39$, and minimum improvement was $s = 0.71$ (which is a slowdown). Similarly, in the 18% of cases where we mispredicted a solver as good or fair, the maximum improvement was $s = 5,824$, average $s = 257$, median $s = 1$, and minimum (slowdown) $s = 0.63$. As these results show, while in some cases the recommended solver can be slower than the default, on average we deliver significant (orders of magnitude) improvement in execution time. Figure 4 shows the speedup s of all test dataset solvers classified as good by using the approach described in section 4. A speedup of 1 means that there was no speedup or slowdown, values less than 1 indicate a slowdown, and values over 1 indicate speedup. Another way to represent the loss or gain of performance is by looking at the area under the curve with relationship to the base case (line at $y = 1$). The area corresponding to speedup is 99.96% of the total area and is 2,397 times larger than the area corresponding to slowdown, which is 0.04% of the total area.

5.2. Trilinos solver analysis. The first set of Trilinos experiments was performed on more than 14,000 data points on 66 Belos solvers/preconditioner combinations using the Janus supercomputer at the University of Colorado. Each Janus node contains two hex-core 2.8GHz Intel Westmere processors (X5660) and 24 GB of RAM. We used Weka to build the classifiers on an Intel Core i5 MacBook Pro following the steps outlined in section 4.

First, we used the Anamod-based features to perform the classification on the same matrices as the PETSc solver classification. With 10-fold cross-validation, the best "good as good" (TPR) solver accuracy with all 68 features was 71.8% and was produced by J48 [59]. With Reduced Feature Set 1 ($RS1$) the best TPR was 75%

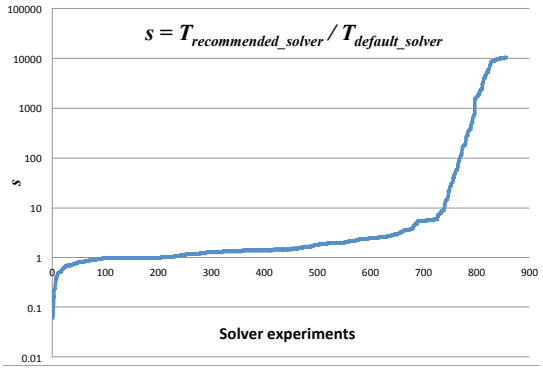


FIG. 4. *Speedup s of the* 851 *test solvers classified as "good". The x-axis represents the individual solver tests, sorted in increasing order of s.*

TABLE 9
Reduced feature sets for Trilinos using Anamod-based matrix properties.

Feature name	Reduced Set 1 ($RS1$)	Reduced Set 2 ($RS2$)
Average distance of nonzeros to the diagonal	X	X
Integer size of blocks that comprise matrix block structure, 1 in the general case	X	X
Left bandwidth: $\max_i\{i-j\colon a_{ij}\neq 0\}$	X	X
Column variability: $\max_j \log_{10} \frac{\max_i \lvert a_{ij}\rvert}{\min_i \lvert a_{ij}\rvert}$	X	X
Diagonal dominance	X	X
Row variability: $\max_i \log_{10} \frac{\max_j \lvert a_{ij}\rvert}{\min_j \lvert a_{ij}\rvert}$	X	X
Number of nonzero elements in the matrix	X	

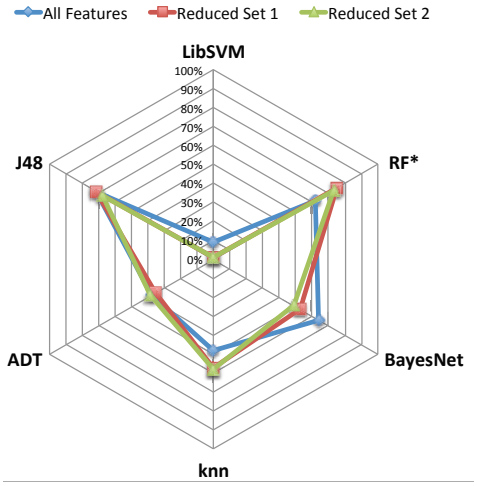
with Random Forest (100 trees), as shown in the second column of Table 9.

With Reduced Feature Set 2 ($RS2$) shown in the third column of Table 9, the best accuracy of 73.1% resulted from Random Forest (100 trees). The performance of other ML methods is shown as a radar chart in the left part of Figure 5.

With a 66%-34% train-test data split, the best accuracy of 72.2% was achieved by J48 for the full feature set. Accuracies of 73.5% and 70% were achieved with reduced sets $RS1$ and $RS2$ with Random Forest (100 trees), respectively. We labeled solutions as "good" if they were within $b = 35\%$ of the best known solution time, which was chosen as the best among values of b between 1 and 45.

Next, we performed the same steps, but this time with Trilinos-computed features because applications using Trilinos need to compute matrix features at runtime and hence most conveniently use available Trilinos functionality. Starting with the full 38-feature set, for 10-fold cross-validation, we identified the features in Table 10[2] using the same set of Weka attribute selection methods listed in section 4.4. We did not implement the equivalent of all 68 Anamod features; hence, the reduced features for Trilinos are somewhat different. The total number of data points used for training and 10-fold cross validation (Table 11) was 39,388, of which 2,986 were "good" solvers for $b = 15$, which is a more stringent labeling of "good" solvers than the best one for PETSc solvers. Again, we selected b based on sampling values between 1 and 45.

[2]Total DNE %: structural property indicating that the fraction of matrix elements for which there is no nonzero in the symmetrical element.



Method	T_{All}	T_{RS1}	T_{RS2}
LibSVM	62.88	23.13	23.17
RF	7.90	3.16	2.63
BayesNet	0.22	0.07	0.05
knn	0.001	0.001	0.001
ADT	4.31	0.43	0.32
J48	0.64	0.09	0.09

FIG. 5. *ML algorithm comparison for Trilinos 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 9.*

TABLE 10
Reduced feature set for Trilinos using Trilinos-computed matrix properties.

Feature name	Reduced Feature Set 1 ($RS1$)	Reduced Feature Set 2 ($RS2$)
Dummy rows	X	X
Trace	X	X
Column diagonal dominance	X	X
Row diagonal dominance	X	X
Diagonal nonzeros	X	
Diagonal mean	X	X
Total DNE %	X	X

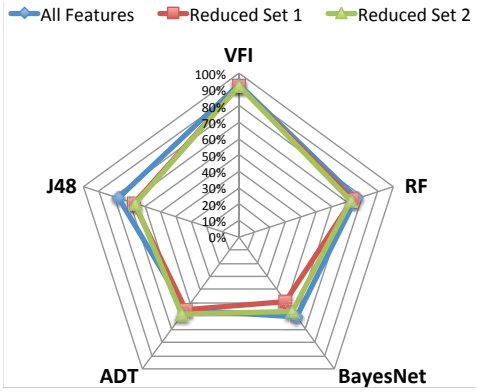
TABLE 11
10-*fold cross-validation for Trilinos.*

Class labels	All features		Reduced Feature Set 1		Reduced Feature Set 2	
Predicted label	good	bad	good	bad	good	bad
True label (good)	2764	222	2690	296	2693	293
True label (bad)	6245	30157	5602	30800	5636	30766

As shown in Figure 6, the best TPR of 93% was delivered by VFI [22]. With the reduced feature sets shown in the last two columns of Table 10, VFI delivered the best TPR of 92% and 92.5%, respectively. We did not consider LibSVM because it is prohibitively slow on this larger dataset (over 300 seconds to build the model).

With a 66%-34% train-test data-split, again VFI performed the best with an accuracy of 93.0% for full-feature set. For both of the reduced sets, an accuracy of 90.3% was achieved by VFI (Table 12). Because of the significantly larger dataset used in this case, the time required for building the classifiers using different ML techniques was also longer than for those shown in Figure 5.

6. Conclusions and future work. The Lighthouse framework contains searchable taxonomies for a growing number of numerical libraries, presently including



Method	T_{All}	T_{RS1}	T_{RS2}
VFI	0.09	0.02	0.02
RF	14.33	6.83	6.09
BayesNet	0.30	0.08	0.07
ADT	5.83	1.41	0.77
J48	1.05	0.26	0.18

FIG. 6. *ML algorithm comparison for Trilinos linear solvers using full and reduced Trilinos-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* 10.

TABLE 12
Validation with 66%-34% *train-test data split for Trilinos.*

Class labels	All features		Reduced Feature Set 1		Reduced Feature Set 2	
Predicted label	good	bad	good	bad	good	bad
True label (good)	963	73	935	100	936	100
True label (bad)	2180	10176	1935	10421	1935	10421

LAPACK, PETSc, and SLEPc. We have successfully integrated performance-based decision support for selecting sparse linear solvers and eigensolvers for portions of PETSc, Trilinos, and SLEPc. Results to date indicate that the ML-based classification can produce up to 93% accurate predictions of well-performing sparse linear system solution methods. As we continue to expand the set of input problems and solution methods, we expect this accuracy to improve further.

We will continue to expand the Lighthouse taxonomy with more HPC software routines and libraries. The greater coverage will allow us to fill out the training dataset in order to improve prediction accuracy. We will also extend and fully automate the prototype infrastructure for generating performance models of selected algorithms through ML-based methods. For example, determining the best threshold b for labeling solvers as "good" is currently a manual task, which can be automated fairly easily. While Weka allows quick algorithmic exploration of relatively small datasets, we plan to switch to a more scalable and efficient ML infrastructure, such as scikit-learn [60]. Additional tuning of ML algorithm parameters will also be investigated. Our long-term plan is to integrate the classification-based solver selection capabilities into PETSc and Trilinos, both for initial solver configuration and also to enable some runtime adaptivity. In addition, a new Lighthouse Web page will be created to provide an up-to-date summary of the type of results described in this paper as they are automatically updated with new classification results.

Because both PETSc and Trilinos are designed for parallel computation, we will work on adding another dimension, scalability, when recommending results to a user. While some options may be preferred for single-threaded or small workstation sized problems, other solver-preconditioner combinations may prove to be better for the same problem if using multiple cluster nodes. In further support of parallel computing,

we will consider custom hierarchical ML approaches to provide scalable support for all possible solvers and levels of parallelism. Lighthouse will thus provide support to an ever broader class of scientific users.

Acknowledgments. The authors would like to thank Branden Romero for his participation in and contributions to the project. We thank Prof. Dejing Dou and Prof. Daniel Lowd of University of Oregon for their valuable feedback on the ML aspects of our approach. The Janus supercomputer is a joint effort of the University of Colorado Boulder, the University of Colorado Denver, and the National Center for Atmospheric Research. Janus is operated by the University of Colorado Boulder.

REFERENCES

[1] *Basic Linear Algebra Subprograms (BLAS)*, http://www.netlib.org/blas, 2015.
[2] *LAPACK - Linear Algebra PACKage*, http://www.netlib.org/lapack/, 2015.
[3] *Scalable Library for Eigenvalue Problem Computations (SLEPc)*, http://www.grycap.upv.es/slepc/, 2015.
[4] *The University of Florida Sparse Matrix Collection*, http://www.cise.ufl.edu/research/sparse/matrices/, 2015.
[5] D. W. Aha, D. Kibler, and M. K. Albert, *Instance-based learning algorithms*, Mach. Learn., 6 (1991), pp. 37–66.
[6] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide*, 3rd ed., SIAM, Philadelphia, 1995, https://doi.org/10.1137/1.9780898719604.
[7] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, K. Rupp, B. F. Smith, S. Zampini, H. Zhang, and H. Zhang, *PETSc Users Manual*, Tech. Report ANL-95/11 - Revision 3.7, Argonne National Laboratory, 2016, http://www.mcs.anl.gov/petsc.
[8] S. Balay, J. Brown, K. Buschelman, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang, *PETSc Web Page*, http://www.mcs.anl.gov/petsc, 2016.
[9] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith, *Efficient management of parallelism in object oriented numerical software libraries*, in Modern Software Tools in Scientific Computing, E. Arge, A. M. Bruaset, and H. P. Langtangen, eds., Birkhäuser Press, 1997, pp. 163–202.
[10] R. Barrett, M. Berry, J. Dongarra, V. Eijkhout, and C. Romine, *Algorithmic bombardment for the iterative solution of linear systems: A polyiterative approach*, J. Comput. Appl. Math., 74 (1996), pp. 91–110.
[11] *Belos documentation*, https://trilinos.org/docs/dev/packages/belos/browser/doc/html/namespaceBelos.html, 2015.
[12] G. Belter, E. R. Jessup, I. Karlin, and J. G. Siek, *Automating the generation of composed linear algebra kernels*, in SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, ACM, New York, 2009, pp. 1–12, https://doi.org/10.1145/1654059.1654119.
[13] S. Bhowmick, V. Eijkhout, Y. Freund, E. Fuentes, and D. Keyes, *Application of machine learning to selecting solvers for sparse linear systems*, in Proceedings of the 2006 SIAM Conference on Parallel Processing, San Francisco, CA, 2006.
[14] S. Bhowmick, V. Eijkhout, Y. Freund, E. Fuentes, and D. Keyes, *Application of alternating decision trees in selecting sparse linear solvers*, in Software Automatic Tuning: From Concepts to the State-of-the-Art Results, K. Naono, K. Teranishi, J. Cavazos, and R. Suda, eds., Springer, 2010, pp. 153–173.
[15] 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 Computational Fluid Dynamics, University of Maryland, College Park, MD, 2005, http://www.mcs.anl.gov/papers/P1279.pdf; preprint, ANL/MCS-P1279-0805.
[16] S. Bhowmick, P. Raghavan, L. C. McInnes, and B. Norris, *Faster PDE-based simulations using robust composite linear solvers*, Future Generation Computer Systems, 20 (2004), pp. 373–387, https://doi.org/10.1016/j.future.2003.07.012.

[17] R. R. Bouckaert, *Bayesian network classifiers in Weka for version* 3-5-7, Artificial Intelligence Tools, 11 (2008), pp. 369–387.
[18] L. Breiman, *Bagging predictors*, Mach. Learn., 24 (1996), pp. 123–140.
[19] K. Bryan and T. Leise, *The* $25,000,000,000 *eigenvector: The linear algebra behind Google*, SIAM Rev., 48 (2006), pp. 569–581, https://doi.org/10.1137/050623280.
[20] C. Cortes and V. Vapnik, *Support-vector networks*, Mach. Learn., 20 (1995), pp. 273–297.
[21] T. A. Davis and Y. Hu, *The University of Florida sparse matrix collection*, ACM Trans. Math. Softw., 38 (2011), pp. 1:1–1:25, https://doi.org/10.1145/2049662.2049663.
[22] G. Demiröz and H. A. Güvenir, *Classification by voting feature intervals*, in Proceedings of the 9th European Conference on Machine Learning, ECML '97, London, UK, 1997, Springer-Verlag, pp. 85–92, http://dl.acm.org/citation.cfm?id=645325.649678.
[23] J. W. Demmel, O. A. Marques, B. N. Parlett, and C. Vömel, *Performance and accuracy of LAPACK's symmetric tridiagonal eigensolvers*, SIAM J. Sci.Comput., 30 (2008), pp. 1508–1526, https://doi.org/10.1137/070688778.
[24] *Django: The Web Framework for Perfectionists with Deadlines*, http://www.djangoproject.com, 2015.
[25] J. Dongarra, *Freely Available Software for Linear Algebra on the Web*, http://www.netlib.org/utk/people/JackDongarra/la-sw.html, 2015.
[26] J. Dongarra, G. Bosilca, Z. Chen, V. Eijkhout, G. E. Fagg, E. Fuentes, J. Langou, P. Luszczek, J. Pjesivac-Grbovic, K. Seymour, H. You, and S. S. Vadhiyar, *Self-adapting numerical software (SANS) effort*, IBM J. Res. Dev., 50 (2006), pp. 223–238, https://doi.org/10.1147/rd.502.0223.
[27] V. Eijkhout and E. Fuentes, *A standard and software for numerical metadata*, ACM Trans. Math. Softw., 35 (2009), pp. 1–20, https://doi.org/10.1145/1462173.1462174.
[28] V. Eijkhout and E. Fuentes, *Machine learning for multi-stage selection of numerical methods*, in New Advances in Machine Learning, Y. Zhang, ed., In-Teh, Olajnica, Croatia, 2010, pp. 117–136, http://www.intechopen.com/books/new-advances-in-machine-learning.
[29] P. R. Eller, J.-R. C. Cheng, and R. S. Maier, *Dynamic linear solver selection for transient simulations using machine learning on distributed systems*, in IPDPS Workshops, 2012, pp. 1915–1924.
[30] A. Ern, V. Giovangigli, D. E. Keyes, and M. D. Smooke, *Towards polyalgorithmic linear system solvers for nonlinear elliptic problems*, SIAM J. Sci. Comput., 15 (1994), pp. 681–703, https://doi.org/10.1137/0915044.
[31] T. Fawcett, *An introduction to ROC analysis*, Pattern Recognition Lett., 27 (2006), pp. 861–874, https://doi.org/10.1016/j.patrec.2005.10.010.
[32] E. Frank, M. Hall, G. Holmes, R. Kirkby, B. Pfahringer, I. H. Witten, and L. Trigg, *Weka*, in Data Mining and Knowledge Discovery Handbook, Springer, 2005, pp. 1305–1314.
[33] Y. Freund and L. Mason, *The alternating decision tree learning algorithm*, in ICML '99: Proceedings of the Sixteenth International Conference on Machine Learning, Vol. 99, Morgan Kaufmann, 1999, pp. 124–133.
[34] *Generic code optimization*, http://icl.eecs.utk.edu/gco/, 2015.
[35] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, *The Weka data mining software: An update*, ACM SIGKDD explorations newsletter, 11 (2009), pp. 10–18.
[36] M. A. Hall, *Correlation-Based Feature Subset Selection for Machine Learning*, Ph.D. thesis, The University of Waikato, 1999.
[37] J. Han, M. Kamber, and J. Pei, *Data Mining Concepts and Techniques*, 3rd ed., Morgan Kaufman, 2012.
[38] T. Hastie, R. Tibshirani, and J. H. Friedman, *The Elements of Statistical Learning*, Springer, 2001.
[39] *Haystack*, http://haystacksearch.org/, 2015.
[40] V. Hernandez, J. E. Roman, and V. Vidal, *SLEPc: A scalable and flexible toolkit for the solution of eigenvalue problems*, ACM Trans. Math. Softw., 31 (2005), pp. 351–362.
[41] A. Holloway and T.-Y. Chen, *Neural networks for predicting the behavior of preconditioned iterative solvers*, in Computational Science: ICCS 2007, Y. Shi, G. D. van Albada, J. Dongarra, and P. M. A. Sloot, eds., Lecture Notes in Comput. Sci. 4487, Springer, Berlin, Heidelberg, 2007, pp. 302–309.
[42] G. Holmes, B. Pfahringer, R. Kirkby, E. Frank, and M. Hall, *Multiclass alternating decision trees*, in Machine learning: ECML 2002, Springer, 2002, pp. 161–172.
[43] E. N. Houstis, J. R. Rice, N. P. Chrisochoides, H. C. Karathanasis, P. N. Papachiou, M. K. Samartzis, E. A. Vavalis, K. Y. Wang, and S. Weerawarana, *//ellpack: A numerical simulation programming environment for parallel mimd machines*, SIGARCH

Comput. Archit. News, 18 (1990), pp. 96–107, https://doi.org/10.1145/255129.255144.
[44] I. K. Jeremy G. Siek and E. R. Jessup, *Build to order linear algebra kernels*, in Workshop on Performance Optimization for High-Level Languages and Libraries (POHLL 2008), Miami, FL, 2008, pp. 1–8.
[45] E. Jessup, B. Bolton, B. Enosh, F. Ma, and T. Nguyen, *LAPACK Internet Interface and Search Engine*, http://www.cs.colorado.edu/~lapack, 2008.
[46] R. Katz, M. Knepley, B. Smith, M. Spiegelman, and E. Coon, *Numerical simulation of geodynamic processes with the Portable Extensible Toolkit for Scientific Computation*, Physics of the Earth and Planetary Interiors, 163 (2007), pp. 52–68.
[47] L. Kotthoff, I. P. Gent, and I. Miguel, *An evaluation of machine learning in algorithm selection for search problems*, AI Commun., 25 (2012), pp. 257–270, http://dl.acm.org/citation.cfm?id=2350296.2350300.
[48] *Krylov methods - KSP*, http://www.mcs.anl.gov/petsc/petsc-current/docs/manualpages/KSP/index.html, 2015.
[49] Lighthouse Project, http://lighthousehpc.github.io/lighthouse/, 2015.
[50] R. Lucas, *PERI: Performance Engineering Research Institute*, http://peri-scidac.org, 2007.
[51] H. D. Mittelmann, *Decision Tree for Optimization Software*, http://plato.asu.edu/guide.html, 2008.
[52] Netlib, http://www.netlib.org/, 2015.
[53] NIST, *HotGAMS: Guide to Available Mathematical Software*; formerly available from http://gams.nist.gov/HotGAMS/HotGAMS.html, 2008.
[54] NIST, *Guide to Available Mathematical Software*, http://gams.nist.gov, 2015.
[55] NIST, *Matrix Market*, http://math.nist.gov/MatrixMarket/, 2015.
[56] B. Norris, S.-L. Bernstein, R. Nair, and E. Jessup, *Lighthouse: A user-centered Web system for linear algebra software*, Elsevier Journal of Systems and Software (JSS): Special Issue on Software Engineering for Parallel Systems, (2015), to appear; arXiv preprint arXiv:1408.1363.
[57] B. Norris, L. C. McInnes, S. Bhowmick, and L. Li, *Adaptive numerical components for PDE-based simulations*, PAMM: Special Issue: Sixth International Congress on Industrial Applied Mathematics (ICIAM07) and GAMM Annual Meeting, Zurich 2007, 7 (2007), pp. 1140509–1140510, https://doi.org/10.1002/pamm.200700687.
[58] N. Pater, *Enhancing random forest implementation in Weka*, in Machine Learning Conference paper for ECE591Q, 2005.
[59] R. Quinlan, *C4.5: Programs for Machine Learning*, Morgan Kaufmann, San Mateo, CA, 1993.
[60] *scikit-learn: Machine learning in python*, http://scikit-learn.org/stable/, 2015.
[61] *Trilinos*, https://trilinos.org, 2015.
[62] *Trilinos Hands-on Tutorial*, https://github.com/trilinos/Trilinos_tutorial/wiki/TrilinosHandsOnTutorial, 2015.
[63] G. Vidal, *Efficient simulation of one dimensional quantum many-body systems*, Phys. Rev. Lett., 93 (2004), 040502.
[64] S. Weerawarana, E. N. Houstis, J. R. Rice, A. Joshi, and C. E. Houstis, *Pythia: A knowledge-based system to select scientific algorithms*, ACM Trans. Math. Softw., 22 (1996), pp. 447–468, https://doi.org/10.1145/235815.235820.
[65] R. C. Whaley, A. Petitet, and J. J. Dongarra, *Automated empirical optimizations of software and the ATLAS project*, Parallel Comput., 27 (2001), pp. 3–35, citeseer.ist.psu.edu/whaley00automated.html.
[66] P. R. Willems, B. Lang, and C. Vömel, *Computing the bidiagonal SVD using multiple relatively robust representations*, SIAM J. Matrix Anal. Appl., 28 (2006), pp. 907–926, https://doi.org/10.1137/050628301.
[67] S. Xu and J. Zhang, *SVM Classification for Predicting Sparse Matrix Solvability with Parameterized Matrix Preconditioners*, Tech. Report 458-06, Department of Computer Science, University of Kentucky, 2006.