Application of Machine Learning in Selecting Sparse Linear Solvers *

S. Bhowmick[1], V. Eijkhout[2], Y. Freund[3], E. Fuentes[4], and D. Keyes[1]

[1] Department of Applied Physics and Applied Mathematics, Columbia University
[2] High Performance Computing Group, Texas Advanced Computing Center,
[3] Department of Computer Science and Engineering, University of California, San Diego
[4] University of Tennessee, Department of Computer Science

* This work was sponsored in part by the U.S. National Science Foundation under award 04-06403 to the University of Tennessee, with subcontracts to Columbia University and the University of California at San Diego.

Sanjukta Bhowmick (Corresponding Author)
Department of Applied Physics and Applied Mathematics, Columbia University,
200 S.W. Mudd Building, 500 W. 120th Street, New York, NY 10027
sb2423@columbia.edu

Victor Eijkhout
High Performance Computing Group, Texas Advanced Computing Center, J.J. Pickle Research Campus, 10100 Burnet Road (R8700), Building 137, Austin, Texas 78758-4497
eijkhout@tacc.utexas.edu

Yoav Freund
Department of Computer Science and Engineering, Room 4126 ,
University of California, San Diego 9500 Gilman Drive,La Jolla, CA 92093-0404
yfreund@ucsd.edu

Erika Fuentes
University of Tennessee, Department of Computer Science
203 Claxton Complex, Knoxville, TN 37996-3450
efuentes@cs.utk.edu

David Keyes
Department of Applied Physics and Applied Mathematics, Columbia University,
200 S.W. Mudd Building, 500 W. 120th Street, New York, NY 10027
kd2112@columbia.edu

Abstract. Many fundamental and resource-intensive tasks in scientific computing, such as solving linear systems, can be approached through multiple algorithms. Using an algorithm well adapted to characteristics of the task can significantly enhance the performance by reducing resource utilization without compromising the quality of the result. Given the numerous parameters governing resource trade-offs, algorithmic choices can explode combinatorially. Even for the same simulation, the "best" solution method can vary across architectures and input data, thereby making the selection a very challenging problem. In this paper, we demonstrate how machine learning techniques can be used in selecting solvers for sparse linear systems and in adapting them to runtime-dependent features of the data and the architecture.

This process includes processing information in the dataset, identification of relevant features, and specifying selection criteria for algorithmic choices. Our future research projects include more ambitious plans for machine learning; this paper is but an initial demonstration of their applicability.

1 Introduction

The solution of large, sparse systems of linear equations, such as those arising from finite discretization of partial differential equations, is a fundamental problem in scientific computing. The number of reasonable choices to consider for their solution is overwhelming. As basic methods are parameterized and nested, the algorithmic options explode combinatorially. It is easily demonstrated, e.g., [25, 38], that there is no uniformly best method, independent of the data, even for modest-sized systems on a serial computer with a perfect memory system. The task of choosing the best method, based on some metric combining memory capacity and execution time in a realistic computing environment, defies theory even within relatively narrow problem classes.

Consider, for instance, the family of typically ill-conditioned linear systems arising at each step of an iterative process to solve a system of nonlinear partial differential equations. Successive systems are related, though the characteristics, and therefore the best method for the inner problem, may evolve over the course of the nonlinear iteration. Furthermore, based on the simulation stage, the accuracy to which the linear system needs to be solved can also vary and the memory constraints can change according to the code implementation or architecture of the machine. A direct method [23] with dynamic pivoting might give a greater guarantee of convergence than any iterative method, but there may not be sufficient memory available and this method may wastefully exceed accuracy requirements for an inner solve in the early stages of a nonlinear problem. Iterative methods [5,44] require less memory but are not as robust. A very conservative robust iterative method might consist of a Krylov solver preconditioned by a form of multigrid, smoothed on each level (except the coarsest) by another Krylov solver preconditioned by some form of relaxation or an incomplete factorization. This is, in fact, a default for the widely used library PETSc [6], where the outer Krylov method is FGMRES, the multigrid is an F-cycle, the smoother is GMRES, its preconditioner a block ILU(0) with one block per processor, and the coarse level solve is performed by a parallel direct method. Such a solver is drastic overkill for some commonly arising problems, such as implicitly time-differenced parabolic systems, but PETSc has to work robustly "almost all of the time" for users who lack the sophistication to set its options, and thus a conservative solver is made the default. However, much simpler and less costly solvers can often be used. For example, if the system is a symmetric positive definite M-matrix, the inner GMRES-ILU smoother can be replaced with a block damped Jacobi iteration, and the outer FGMRES with Conjugate Gradients, or nothing at all.

Examples such as these highlight the importance of selecting a solver to match the problem attributes. Selection of an appropriate solver can lead to benefits such as reduced memory requirement, lower execution time, fewer synchronization points in a parallel computation, and so forth. A challenge remains, however, in formulating algorithms for solver selection. There have been several investigations and software developments related to selecting efficient solvers. The Linear System Analyzer (LSA) [31] is a component-based problem-solving environment that allows the user to specify combination of preconditioner and linear solver without being required to know the details of the implementation. Various approaches to tuning solvers with applications include: i) a *poly-iterative linear solver* [7], which is based on applying several iterative methods simultaneously to the same system; ii) a *composite multi-method solver* [12,13], where the linear system is applied to a sequence of solvers; iii) an *adaptive multi-method solvers* [11,36], where the linear system is selected dynamically by observing linear system properties as they evolve during the simulation process; and iv) a *self adapting large-scale solver architecture* (SALSA) [17–19], which uses statistical techniques such as principal component analysis for solver selection.

Our present approach to solver selection is to use machine learning algorithms to generate functions that map linear systems to suitable solvers. A mapping function consists of two parts: a *feature extractor* and a *classifier*. The feature extractor computes numerical quantities, called *features*, such as rank, norm or spectral estimates of the given linear system. The set of features is designed to capture 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 feature extractor

is designed by a human expert. The role of the learning algorithm is to choose a subset of the features (so-called "feature selection") and a function that maps the values of the selected features to the choice of solver.

To construct the classifier, the learning algorithm receives as input a *training set*. The training set is a set of linear systems and the amount of time (or other resource of prime consideration) required by candidate algorithms to solve these systems. The classifier searches for a mapping that makes good predictions on the training set, with respect to the resource. An implicit assumption is that a solver that performs well on the training set will also perform well on new, as yet unseen systems, in the *test set*. For this assumption to be valid the training set has to be representative of the test set. Formally, we assume that there exists a fixed but unknown distribution $\mathcal{D}$ over systems and that systems in the training set and the test set are drawn independently at random according to $\mathcal{D}$.

Traditional approaches to pattern recognition and machine learning [22] suggest separating feature selection from the construction of the classifier. The reason for the separation is that most algorithms for learning classifiers perform poorly on the test set when the number of features is large compared to the size of the training set. This is the so-called "over-fitting" problem which is a manifestation of the "curse of dimensionality." The traditional approach, which is still the common approach, is to avoid overfitting by carefully selecting a small set of features and then applying the learning algorithm to the selected set. One problem with this approach is that the optimal choice of features depends on the amount of available training data. Features that are useful when the training set is large often have a negative effect on the test set performance when the training set is small. Choosing a good set of features has become the main problem in constructing good classifiers for real world problems and common wisdom is that the choice of learning algorithm is of secondary importance. In our case, we use a tool, AnaMod [24], that has a large repertoire of numerically relevant features, and the problem becomes one of eliminating redundant or irrelevant features.

Two relatively new approaches to machine learning: Support Vector Machines (SVMs, see e.g., [16, 50]) and AdaBoost [29] have proven to be very resilient with respect to over-fitting. These algorithms have been used in many real-world problems in which the number of features is much larger than the size of the training set, with no apparent degradation in test set performance. Some theory has also been developed to explain this surprising behavior [9, 46]. The upshot is that the expert designing the feature set is free to include in the feature set any feature she deems to be informative, without concern of over-fitting the training set.

In this paper we use *Alternating Decision Trees* [27], a learning algorithm based on AdaBoost, to generate the function for selecting an effective solver. In our experiments we study the application of this approach to a practical set of problems, including a model *Driven Cavity Flow* [10, 33] for computational fluid dynamics from the PETSc toolkit and the *M3D* [41] magnetohydrodynamics code.

The problem of selecting sparse linear solvers presents a domain of almost limitless data; even one code running one application, such as wing design or radar cross-section, in production mode over a decade provides enviable opportunities for machine learning. Machine learning techniques can be used to assist users in selecting linear solvers and in adapting them to runtime-dependent features of the data and the architecture. Every step of the process is an issue for further research, including specifying relevant features, evaluating them cost-effectively, collecting and processing the resulting data, and specifying metrics for success of an algorithmic choice.

The remainder of this paper is organized as follows. Section 2 gives an overview of Adaboost and Alternating Decision Trees. In section 3 we describe and justify our method for mapping the solver selection problem into a classification problem and the way in which we quantify the accuracy of the classifier. This section also discusses how ROC (Receiver Operator Characteristics) curves can be used to measure the accuracy of the prediction. Section 4 describes one way in which the machine learning tools of the preceding section can be applied to linear solver selec-

tion. Section 5 briefly describes the two application domains and presents experimental results. Section 6 discusses conclusions and directions of future work.

2 Machine Learning Methods

In this section we give a brief introduction to machine learning in general and to the specific algorithms used in this paper.

2.1 A Short Introduction to PAC Learning

The main task of machine learning algorithms is to generate *classification rules*, i.e., mappings from some *instance space* X to a finite *label set* Y. In our application the instance space is the space of feature vectors, consisting of characteristics of the linear system and parameters of the solvers. The label set is based on the performance of the set of candidate solvers with respect to the linear systems. The most studied problem is that of *binary* classification in which the set Y consists of two elements; in the rest of this section we concentrate on the binary case.[1] We denote a classification rule by $h : X \to Y$.

We call an instance+label pair an *example* and assume that examples are drawn from some fixed but unknown distribution $\mathcal{D}$ over the examples space $X \times Y$. The learning algorithm is presented with a finite set of examples, called the *training set* $T = \{(x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m)\}$, drawn independently at random according to $\mathcal{D}$. The learning algorithm, given the training set as input, generates a classification rule h. The goal is to generate a rule with a small *generalization error*:

$$\mathrm{err}(h) \doteq P\left(h(x) \neq y\right),$$

where the probability is with respect to the distribution $\mathcal{D}$ over the choice of x and y. In practice, one cannot know the generalization error of a hypothesis and has to use an estimate instead. The estimate is computed using a second set of examples drawn IID[2] from $\mathcal{D}$ which is called the *test set*. The fraction of the test set on which $h(x) \neq y$ is call the *test error* of the h and is an unbiased estimator of the generalization error.

The rule generated by the learning algorithm is a function of the training set. The training set generated by IID draws from the distribution $\mathcal{D}$. As a result, there is always some small probability that the training set is unrepresentative and the resulting rule has a very high generalization error. It is therefore unreasonable to require that the classification has small generalization error in *all* cases. We use a definition of learning based on Valiant's *Probably Approximately Correct (PAC) learning* [49].

Definition 1 (PAC learning). *Algorithm* $\mathbf{A}$ *learns a distribution* $\mathcal{D}$ *with accuracy* $\epsilon > 0$ *and reliability* $\delta > 0$ *if there exists some training set size* m *such that if* $\mathbf{A}$ *is given a training set generated by* m *independent random draws according to* $\mathcal{D}$ *it generates a classification rule* h *such that* $err(h) \leq \epsilon$ *with probability at least* $1 - \delta$.

The field of PAC learning theory, started by Valiant's seminal paper, is concerned with the computational complexity of learning algorithms. In particular, it studies learning algorithms for setups in which there is a *functional* relationship between instances x and labels y. In other

[1] As there are usually more than two solvers, the natural formulation of the solver selection problem as a classification problem is not a binary problem. In the next section we describe how we map this non-binary problem into a binary problem and explain the reasons for using this mapping rather than the natural set of labels.

[2] We say that a set of examples is drawn *Independently and Identically Distributed* (IID) according to $\mathcal{D}$ if they can be seen as independent draws from the fixed distribution $\mathcal{D}$. In other words, if they are independent random variables all having distribution $\mathcal{D}$.

words, the conditional distribution of y for any fixed value of x is concentrated on a single label. Furthermore, the function $c : x \to y$ is assumed to be an element from a set of functions called the *concept class* which we denote by C. We say that $\mathcal{D}$ is *consistent* with C if the conditional distribution it implies between x and y is consistent with some $c \in C$.

We now give the two definitions that are at the basis of boosting algorithms. Note the order of the quantifiers in these definitions.

Definition 2 (Strong PAC learning). *A concept class C is* strongly learnable *if there is a learning algorithm* **A** *for it that has the following property.* **For any** $\epsilon > 0$ *and any* $\delta > 0$ *there exists a training set size m such that for any distribution $\mathcal{D}$ that is consistent with C,* **A** *can learn $\mathcal{D}$ with accuracy ϵ and reliability $1 - \delta$ using m training examples.*

Definition 3 (Weak PAC learning). *A concept class C is* weakly learnable *if there is a learning algorithm* **A** *for it that has the following property.* **For some** $1/2 > \epsilon > 0$ *and* $1 > \delta > 0$ *there exists a training set size m such that for any distribution $\mathcal{D}$ that is consistent with C,* **A** *can learn $\mathcal{D}$ with accuracy ϵ and reliability $1 - \delta$ using m training examples.*

2.2 Boosting

Clearly, strong PAC learning places a stronger requirement on the learning algorithm. Basically, all that weak learning requires is that the generated classification rule be slightly correlated with the true function, i.e. its performance be just slightly better than that of a random guess whose error is always 1/2. It was therefore very surprising when Schapire [45] proved that Weak and Strong PAC learning are in fact equivalent. Schapire's proof is constructive; he gave an algorithm, now called a *boosting* algorithm, which transforms any weak learning algorithm into a strong learning algorithm. Moreover, the boosting algorithm is computationally efficient, that is, all of the resources it requires (training examples, running time and storage) scale polynomially with $1/\epsilon$ and $1/\delta$, where ϵ and δ are the accuracy and reliability required for strong learning.

Schapire's algorithm, and all of the other boosting algorithms developed since are based on the following simple idea. While a weak learner is only required to find a hypothesis whose performance is slightly better than chance, it has to do this for *any* distribution that is consistent with C, in other words, for any distribution over the instance space X. The boosting algorithm utilizes this assumed ability by running the weak learning algorithm multiple times, each time using a different distribution over the instances. It then combines the resulting hypotheses, usually by some kind of a majority vote function, to create the final, highly accurate hypothesis. Intuitively, the process changes the distribution over the instances so that is it increasingly concentrated on instances that are harder to classify, thereby forcing the weak learner to focus its efforts on those examples.

2.3 Adaboost

Schapire's original boosting algorithm had some important practical limitations, and so did the boost-by-majority algorithm of Freund [26]. It was only when the Freund and Schapire introduced the Adaboost algorithm [28] that boosting became a useful tool for practical problems. Adaboost is a very simple and parameter-free boosting algorithm, as described in Figure 1. The algorithm operates in iterations, indexed by t. Each iteration produces a *weak classifier* h_t and a *classifier weight* α_t. The *scoring function* F_t is the sum of all weak classifiers produced so far, taken with their corresponding weights. The final *strong classifier* is the rule defined by $sign(F_t(x))$.

On each iteration Adaboost calls the weak learner with a different weighting of the training set and receives back the weak classifier h_t. The positive number $w_i^t = e^{-y_i F_{t-1}(x_i)}$ is the *example weight* associated with the training example (x_i, y_i) at iteration t. Under the weak learning assumption the total weight of the examples on which $h_t(x_i) = y_i$ should be larger than the

total weight of the examples on which $h_t(x_i) = -y_i$. If this is the case then $h_t(x)$ is correlated with y and the classifier weight α_t is positive. In fact, if the relation between the total weights is reversed, things work out just as well, because in this case $h_t(x)$ is *anti-correlated* with y which is equivalent to $-h_t(x)$ being correlated with y and in this case α_t is indeed negative. If the two total weights are equal, there is no correlation, $\alpha_t = 0$ and $F_{t+1} = F_t$. The only case in which the formula for α_t explodes is if $h_t(x) = y$ (or $h_t(x) = -y$) on all examples (note that the example weights are never zero). This explosion is quite justified, since in this case $h_t(x)$ is not a weak rule at all, but rather is a perfect rule (on the training examples), which means that boosting is not needed in the first place.

$F_0(x) \equiv 0$
for $t = 1 \ldots T$
$\forall i;\ w_i^t = e^{-y_i F_{t-1}(x_i)}$
Get h_t from *weak learner*
$\alpha_t = \frac{1}{2} \ln \left(\frac{\sum_{i:h_t(x_i)=y_i} w_i^t}{\sum_{i:h_t(x_i)\neq y_i} w_i^t} \right)$
$F_{t+1} = F_t + \alpha_t h_t$

Fig. 1. The Adaboost algorithm.

The main provable property of the Adaboost algorithm is a bound on the training error of the strong rule as a function of the weighted errors of the weak rules. We define the *edge* γ_t of the classifier h_t as the advantage that h_t has over random guessing with respect to the weights w_i^t. The edge is equal (within a constant factor) to the weighted correlation between the prediction $h_t(x)$ and the label y:

$$\gamma_t = \frac{\sum_{i:h_t(x_i)=y_i} w_i^t}{\sum_{i=1}^m w_i^t} - \frac{1}{2}.$$

Using the definition of the edge, one can easily prove the following bound on the number of mistakes that the the combined rule makes on the training set:

$$\sharp\{i : \mathrm{sign}(F_T(x_i)) \neq y_i\} \leq m e^{-2\sum_{t=1}^T \gamma_t^2}. \quad (1)$$

Clearly, if $|\gamma_t| \geq \gamma > 0$ for all iterations t, the number of mistakes made by the strong rule decreases exponentially with the number of iterations T.

2.4 Adaboost Resistance to Overfitting

While the last result is interesting, it is insufficient for showing that boosting is an effective learning method. After all, Equation 1 bounds the number of mistakes that the strong rule makes on the *training set*, while we are interested in the number of mistakes the rule is likely to make on an independent *test set*. Fortunately, using tools from uniform convergence theory, one can prove that the performance on the test set would not differ too much from the performance on the training set given that the number of boosting iterations T is small and that the weak of weak rules has finite Vapnik-Chervonenkis dimension (see [50] for details.)

However, as it turns out, this analysis is overly pessimistic in analyzing the performance of Adaboost in practical applications. Cortes and Drucker [21], Quinlan [42] and Breiman [14] observed that the behavior of the test error is usually significantly better than would be expected from the uniform convergence bounds. In fact, it is often the case that the test error of the strong rule continues to decrease long after Adaboost drove the training error to **zero**!

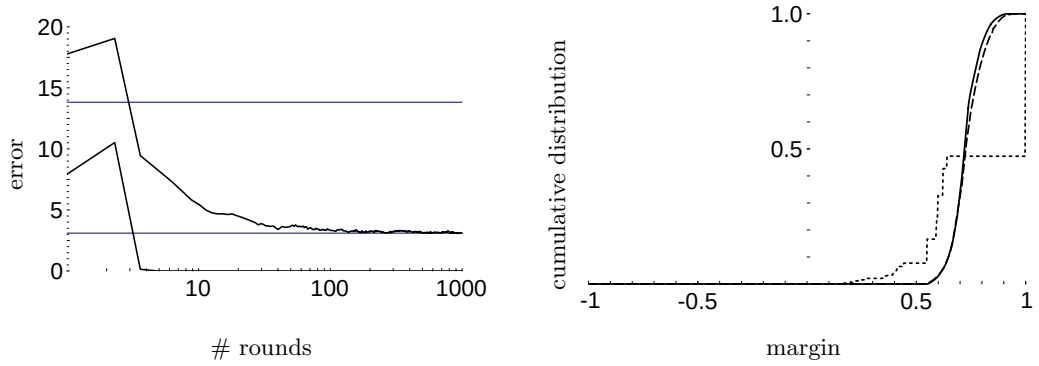

Fig. 2. Error curves and the margin distribution graph for boosting C4.5 [43] on the letter dataset as reported by Schapire et al. [46]. *Left*: the training and test error curves (lower and upper curves, respectively) of the combined classifier as a function of the number of rounds of boosting. The horizontal lines indicate the test error rate of the base classifier as well as the test error of the final combined classifier. *Right*: The cumulative distribution of margins of the training examples after 5, 100 and 1000 iterations, indicated by short-dashed, long-dashed (mostly hidden) and solid curves, respectively.

Figure 2 summarizes an example of this behavior [3]. The left graph in Figure 2 shows the training error and test error of the strong classification rule as a function of the number of boosting iterations. The training error decreases to zero within five iterations, this is to be expected as the weak learner that is used (C4.5) is a powerful learning algorithm and the rules that it generates are quite accurate. The test error decreases from 18% to 10% in the first five iterations, this is also consistent with the standard theory. The surprising fact is that the test error *continues to decrease after the fifth iteration*. Moreover, the decrease is both significant: from 10% down to about 3% and persistent: the error does not increase significantly even after one thousand iterations! To appreciate how surprising this is, consider that each of the decision trees has more than a hundred nodes, which means that the combined hypothesis has more than 100,000 parameters, while the training set contains only 16,000 examples. Building a model with more parameters than data-points is usually a nonsensical proposition, but apparently not in this case!

There is still no completely satisfactory explanation of this surprising phenomenon. A partial explanation was given by Schapire et al [46]. The explanation is based on the concept of a *classification margin*. Recall the Scoring function $F_T(x) = \sum_{t=1} \alpha_t h_t(x)$ described earlier. The normalized margin for examples (x, y) is defined as

$$M(x, y) = y \frac{\sum_{t=1} \alpha_t h_t(x)}{\sum_{t=1} |\alpha_t|}$$

It is easy to verify that the normalized margin is in the range $[-1, +1]$ and that it is positive if and only if $\text{sign}(F_T(x)) = y$, i.e. when the strong hypothesis is correct. The main idea of the margins is that while $M(x, y) > 0$ corresponds to a correct prediction, $M(x, y) > \theta > 0$ corresponds to a *confident* correct prediction. The implication is that if most of the training examples have a large positive margin then the test error is likely to be small. This explains how boosting improves the

[3] This is one of several examples presented and discussed in detail in [46]. The particular dataset discussed here is the "letter" database from the UC-Irvine repository [39]. There are 16,000 training examples and 4,000 test examples, each example consists of 16 numerical features representing an image of a hand-written English letter and the label is the identity of the letter. The weak learning algorithm used in this example is Quinlan's C4.5 [43].

test error of the strong hypothesis even after the iteration in which the training error decreased to zero. On that iteration the minimal margin over the training set becomes positive; beyond that point the minimal margin continues to increase, implying that the predictions are more confident, and corresponding to the fact that the test error continues to decrease. This is demonstrated in the right graph in Figure 2, in which we see that the margins on the training set increase from iteration 5 to iteration 100 after which they change very little. This corresponds to the significant decrease in the test error between iterations 5 and 100 which is followed by almost no change up to iteration 1000. Schapire et al. [46] also prove an upper bound on the generalization error which depends on the distribution of the classification margins on the training set but not on the number of boosting iterations.

2.5 Alternating Decision Trees

Adaboost is a very general algorithm. It can work with any weak learning algorithm. In order to apply it to a specific problem one needs to choose a weak learning algorithm. One choice that has enjoyed much popularity and success is boosting decision trees [14,30,42]. In this combination the weak rules are generated by an a standard learning algorithm such as CART [15] or C4.5 [43] and Adaboost is run on top of the learning algorithm to produce a weighted average of classification trees.

Freund and Mason [27] developed a learning algorithm for rules called "alternating decision trees" that uses only boosting. Alternating decision trees are a generalization of weighted sums of decision trees. An example of an alternating decision tree is given in Figure 3. In this case, instances consist of two real-valued features, a and b. There are two types of nodes in the tree: *prediction nodes*, denoted by ellipses, which contain a real value, and *decision nodes* denoted by rectangles, which contain an inequality between one of the features and a constant. The name "alternating trees" comes from the fact that the two types of nodes are organized in alternating layers.

The alternating decision tree associates a real-valued score with every instance. The sign of that score is the predicted label for the instance. To calculate a score, one starts at the root and proceeds along multiple paths down the tree according to the following rules.

1. If the node is a prediction node (ellipse) proceed along all of the dotted edges emanating from it.
2. If the node is a decision node (rectangle) proceed along the edge marked Y if the condition in the decision node holds. Otherwise proceed along the edge marked N.

The score that is associated with an instance is the sum of the values in all of the prediction nodes that are reached according to these rules. For example, consider the instance $a = 1, b = 1$. For this instance the conditions $a < 4.5$ and $b > 0$ hold while $a > 2$ and $b > 1$ do not. Thus the prediction nodes that are reached are those whose values are $+0.5, -0.7, +0.3, -0.2, +0.1$ and the total score is 0.0.

Learning an alternating decision tree can be done by considering the addition of each decision node and the two prediction nodes emanating from it as an addition of two weak classifiers, one for each of the prediction nodes. For details, see Freund and Mason [27]. The MLJava package is an implementation of this algorithm that was used in this work.

3 Solver Selection as a Classification Problem

In this section we discuss how solver selection can be represented as a binary classification problem

Consider L to be the set of linear systems, defined by a matrix and right-hand side pair and S to be the candidate set of solvers. Let solution of each linear system $l \in L$ by each solver $s \in S$ be measured by a performance parameter $\pi(l, s)$.

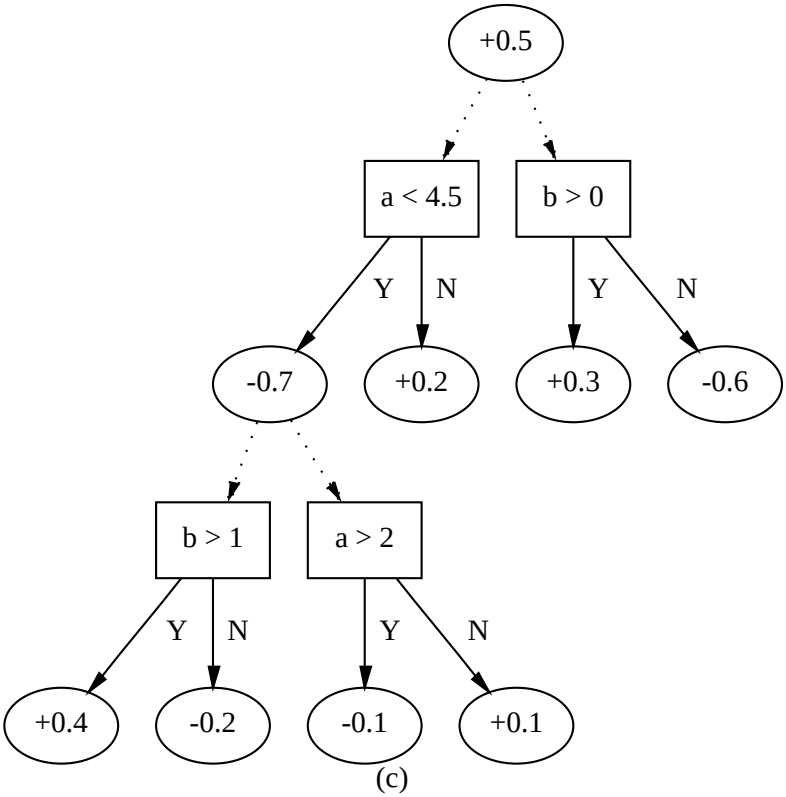

Fig. 3. An alternating decision tree.

We define $\hat{s}$ to be the optimal solver for a linear system l, if

$$\pi(l, \hat{s}) = min(\pi(l, s))_{\forall s \in S}.$$

As discussed in section 1, the cardinality of the set of solvers may be quite large and an exhaustive search would be expensive. In addition, there might be solvers for which the values of the associated performance parameters are close enough to be approximately equal, and ordering these solvers would yield no further advantage. Thus it is generally more realistic to define *a set* of nearly optimal methods $\hat{S}$, as opposed to a *single* method. We can thus partition S into two sets $\hat{S}$ and $\bar{S}$. In practice, division of the methods into set $\hat{S}$ and $\bar{S}$ is determined by the following user-specified variables: i) the *improvement factor*, ρ, $\rho > 1$ and ii) a default method $\tilde{s}$.

The set of nearly optimal methods is now defined in terms of these two variables as;

If $\rho(\pi(l, s)) < \pi(l, \tilde{s}), s \in \hat{S}$; Otherwise $s \in \bar{S}$.

Therefore, finding a set of suitable solvers for a linear system l can be expressed as defining a function

$F_\pi : L \rightarrow S$, such that $F_\pi(l) = \hat{S}$.

For each linear system l and associated performance parameter π, the solvers can be divided in two groups: i) solvers that are elements of $\hat{S}$ and ii) solvers that are elements of $\bar{S}$. This is equivalent to a binary classification problem where the instance space is $L \times S$, the labels represent whether the solver is an element of $\hat{S}$ or $\bar{S}$ and the ideal classification rule would be

$$h_\pi : (L \times S) \rightarrow \{\text{“good”, “bad”}\}, \text{ such that:}$$
$$h_\pi(l, s) = \text{“good” if } s \in F_\pi(l); \text{ and } h_\pi(l, s) = \text{“bad” otherwise.}$$

Of course, it is generally impractical to form h_π to exactly satisfy the above equations, and we use machine learning algorithms to approximate h_π.

3.1 Accuracy of the Prediction

Our goal is to generate a rule with low prediction error. The accuracy of prediction can be measured by the *sensitivity* and *specificity*. Let the binary labels be "good" 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". The *receiver operator characteristic* (ROC) curve [51] is commonly used for assessing the tradeoff between sensitivity and specificity.

Figure 4 shows a binary classification where the entries are arranged according to a test value, for example, the performance parameter. The two Gaussian curves represent the distribution of the "good" and "bad" classes. There is an area of overlap between the curves where it is not possible to distinguish between the two classes. The dotted line represents a cutpoint. Any value on the left of the line is "bad" and any value on the right is "good". According to the position of the line the entries can be

- True Positive (TP): actual label is "good" and predicted label is "good".
- False Negative (FN): actual label is "good" and predicted label is "bad".
- False Positive (FP): actual label is "bad" and predicted label is "good".
- True Negative (TN): actual label is "bad" and predicted label is "bad".

Sensitivity is calculated as $\frac{TP}{(TP+FN)}$ and specificity is calculated as $\frac{TN}{(TN+FP)}$.

ROC curves plot the true positive rate (sensitivity) by the false positive rate (unity minus specificity) as the cutpoint discriminating between "good" and "bad" is shifted. Suppose the cutpoint starts from the rightmost point, then there are no entries predicted as "good". Therefore both the true positive rate and the false positive rate is zero. As the threshold is moved towards left, the number of true positive entries increase. In most cases, initially the number of false positives remain zero, till the overlapping region is reached. For an ideal classification, where there is no overlapping region, the true positive rate would reach its maximum value 1.0, while the false positive rate remains at zero, and then, when the threshold is shifted further left, the true positive rate would remain constant at 1.0 while the false positive rate increases. Such as an ROC curve will closely follow the lefthand border and then the top border. In contrast, if the two distributions exactly overlap, then the increase in true positive rate would be exactly equal to the increase in the false positive rate. The ROC curve is now the $x = y$ line, and represents a random guess. It the curve falls below this line, then the classification is worse than a random guess and is of no use at all.

From these observations it is easy to see that the *area under the ROC curve* is an important measurement of the accuracy of the classification.

3.2 Issues in Implementing Solver Selection as a Classification Problem

Solver selection problem can easily be expressed as a binary classification once the linear system, solver set and the selection criteria are fixed. However, determining and representing the elements in these groups is a nontrivial exercise. The efficiency of the classification is affected by how closely the input to the classification represents the original problem. Given below are some of the issues involved in defining the classification inputs;

- **Linear System Set:** Most linear systems to be solved are generated in course of a simulation process and the characteristics of the matrices are influenced by the linear solvers used earlier

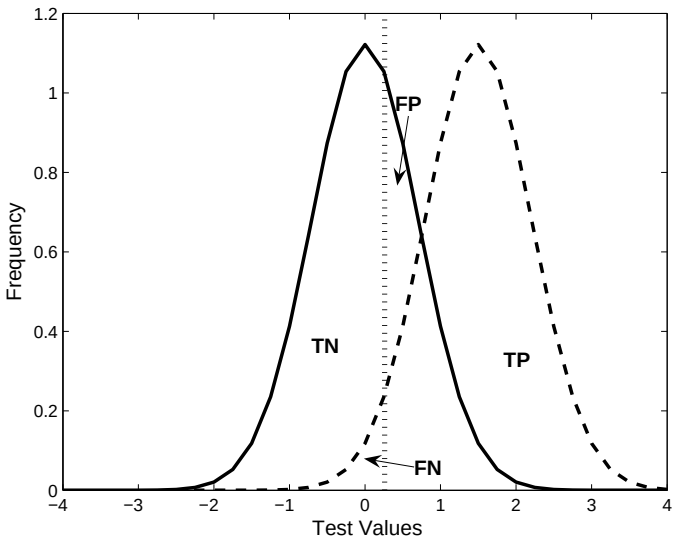

Fig. 4. Distribution of elements into a binary classification. The bold line represents the distribution of class 1 (the "bad" elements) and the dashed line represents the distribution of class 2 (the "good' elements).

during the simulation. In order to have a consistent set of linear systems, we designate one solver as the *default solver* $\tilde{s}$. The simulation is executed using the default solver to solve the linear systems. The matrices and corresponding right hand sides generated in course of the simulation, are stored from the elements of the linear system set L. The binary classification problem, with respect to a performance parameter π is posed as follows: $h_\pi(l, s) =$ "good" if $\rho(\pi(l, s)) < \pi(l, \tilde{s}) : \rho > 1$ and $h_\pi(l, s) =$ "bad" otherwise.

– **Multiple Selection Criteria:** We derived the binary classification representation for examples where the solver selection depends on only a *single* performance parameter. However, generally the selection criteria might be defined by multiple parameters, $\pi_1, \pi_2, \ldots \pi_n$. For example, the accuracy of the solution, represented by the norm of the residual vector might be as important as execution time in determining the solver. The multiple criteria would be expressed as the union and/or intersection of the individual performance parameters. Similarly, the set of suitable solvers would be formed by the union and/or intersection of the individual solver subsets obtained from $F_{\pi_1}, F_{\pi_2}, \ldots F_{\pi_n}$, as appropriate.

3.3 Feature selection and computation

The selection of appropriate features is an important step in the setup of any classification process. We use the AnaMod software [24] developed by two of the authors. In this section we will expound on various issues that come up in computing problem features.

In our problem domain of solving systems of linear equations, we have a fairly large number of features to choose from, which we can sort in a number of categories.

structural Properties that describe the sparsity structure of the matrix, such as bandwidth, average or max/min number of nonzeros per row, and the number of structurally unsymmetric elements ($a_{ij} = 0$ while $a_{ji} \neq 0$).

normlike The 1-, infinity-, and Frobenius- norms of the matrix, as well as these norms taken of the symmetric and nonsymmetric part of the matrix. These quantities, like those of the

previous category, can all be computed in time proportional to the number of nonzeros of the matrix.

spectral Properties that describe the spectrum or field of values of the coefficient matrix. These properties can not be computed exactly, but estimation is feasible. For the features in this category, we run a modest number of iterations of GMRES with the coefficient matrix, and analyzing the resulting Hessenberg matrix.

normality Various bounds on the departure from normality. While these bounds can be computed exactly, the accuracy of the bounds as such may be questionable.

variance Various heuristic measures of how 'wild' a matrix is, such as standard deviation from the average value of the matrix diagonal, or variability inside rows or columns.

Relevance of features Some of these features (especially the spectral and normal categories) have a direct, and theoretically (qualitatively, though not necessarily quantitatively) known relationship with the method performance that we want to optimize. Others may have an unknown relation, or may be indirect correlation to relevant features. For instance, among different discretizations of one partial differential equation, using higher order elements tends to a higher matrix condition number, making the system harder to solve. At the same time, it increases the number of nonzeros per row, so that latter feature, while not mathematically relevant in itself, becomes correlated to solver behaviour.

Computation of features In the tests reported in this paper, we have computed all available features without regard for the cost. In this section we will outline some of the issues relating to the cost of feature extraction.

First of all, for the training of our classifier system, since it will happen only once, it makes sense to compute all features, regardless of the cost. The training of the system is expected to be amortized over many future applications of the classifier. However, in using the classification system, the feature calculation is a preprocessing step for a single system solution, or a low number of them, and its cost therefore should not overwhelm the expected cost of computing the solution of the system.

We have already alluded to the fact that certain features are more expensive to compute than others. The cost of computing the spectral features, for instance, is theoretically much higher than that of solving a system if we would compute to full numerical precision. Even an approximate compution of these features is essentially that of partway solving a system with the given coefficient matrix. Only if the solution of the system is expected to take a large number of iterations, or if many systems with this coefficient matrix are going to be solved, is such an expenditure defensible.

Missing features Bounds on the departure from normality are potentially very useful, but are even harder to compute than the spectral features. Most such bounds involve the norm of the commutator $AA^t - A^tA$, which can be seen to involve the inner products of all pairs of rows and all pairs of columns. Such a calculation is only feasible for fairly narrow banded, or unrealistically small, matrices.

Thus, ideally we have to take into account in our system that certain features may not always be available, and we have to gauge the accuracy of classifiers when using a strict subset of the whole collection of features. Interestingly, Boosting can use features with occasionally missing values. In assessing the usefulness of a feature, Boosting will measure how much information can be extracted from the feature, taking into account that with some probability the value of the feature is not known. It is important that the fraction of the training set on which the feature calculation fails would be representative of the eventual application, which underscores, once more, the importance of having large and representative training sets.

4 Applying Machine Learning

Application of machine learning to solver selection consists of the following three stages; a schematic diagram of the process is given in Figure 5.

- *Database Construction:* The machine learning software uses the information stored in a database to build a classifier that maps the matrices to their corresponding solvers. The database is formed of entries representing the efficiency of the solution of a set of linear systems, L, with respect to a candidate set of solvers S. Each linear system, obtained from user input or generated from an application code, is (in principle) solved by all the solvers from the candidate set. An entry in the database, associated with the solution of $l \in L$ by a solver $s \in S$ consists of: i) the feature vector representing the numerical characteristics of l, ii) the parameters of the solver s, and iii) labels, representing the ranking of the solution with respect to a performance parameter such as execution time, terminal residual norm, etc. The total number of entries in the database is *number of systems* times *number of solvers*. As discussed in section 1, feature selection is by itself an important challenge. Since we are using Adaboost as the machine learning tool, the problem of "over-fitting" is alleviated. However, the software should be able to apply other machine learning methods if needed. In such a generalized scenario, feature selection may be of prime importance. Furthermore, one of our future research projects is to compare the efficiency of different sets of features. Keeping this in mind, we have added an option for identifying important features from each entry in the database.
- *Classifier Creation:* The database is divided into *training* and *testing* sets based on a three-fold cross validation (three different divisions of training and testing set) of the database. The machine learning algorithm (MLJava) is applied on each of the training and their corresponding testing sets. The learning algorithm builds the classifier using the training set and predicts its accuracy with the testing set. The accuracy of the classifiers is measured by the area under *ROC curves* [51]. We choose classifiers with high accuracy.
- *Solver Prediction:* The choice of solver depends primarily on the matrix characteristics. Therefore it is likely that the solvers that perform well for systems recorded in the database, would also be applicable for any other systems that have similar characteristics. The feature set of such a new matrix and the set of solvers S, *without labels*, are given as input to the classifier. The output is a prediction which labels each solver in S as "good" or "bad" according to its suitability for the new matrix.

We use the suite of linear solvers from PETSc for solving the linear systems. The system characteristics are obtained employing AnaMod, as described in section 3.3 [1], a library of modules that uses PETSc functions to compute various properties of a system. PETSc, the Portable Extensible Toolkit for Scientific Computing (PETSc) [6], was developed at the Argonne National Laboratory, and is in use in thousands of scientific software projects around the world and has run on essentially all of the major scientific computing architectures. PETSc software has powered three Gordon Bell prizes for "special achievement" [2–4]. This toolkit consists of a suite of distributed data structures and modules for the scalable solution of various scientific applications expressed at different levels from linear systems to method-of-lines integration of partial differential equations. PETSc applications are composed hierarchically. Users may employ PETSc's own structured array or unstructured low-level data objects for representing meshes, vector fields, and linear operators, or they may register their own.

We use MLJava [37] to create the classifiers. The input to MLJava consists of three files: the training data, the test data, and the data specification. The data specification file lists the names and types of each example attribute (in this case we list the linear system properties that were evaluated and the solver parameters) and the possible set of labels (e.g., "good" or "bad") for each example. There are currently three attribute datatypes: *number* (represented as a floating

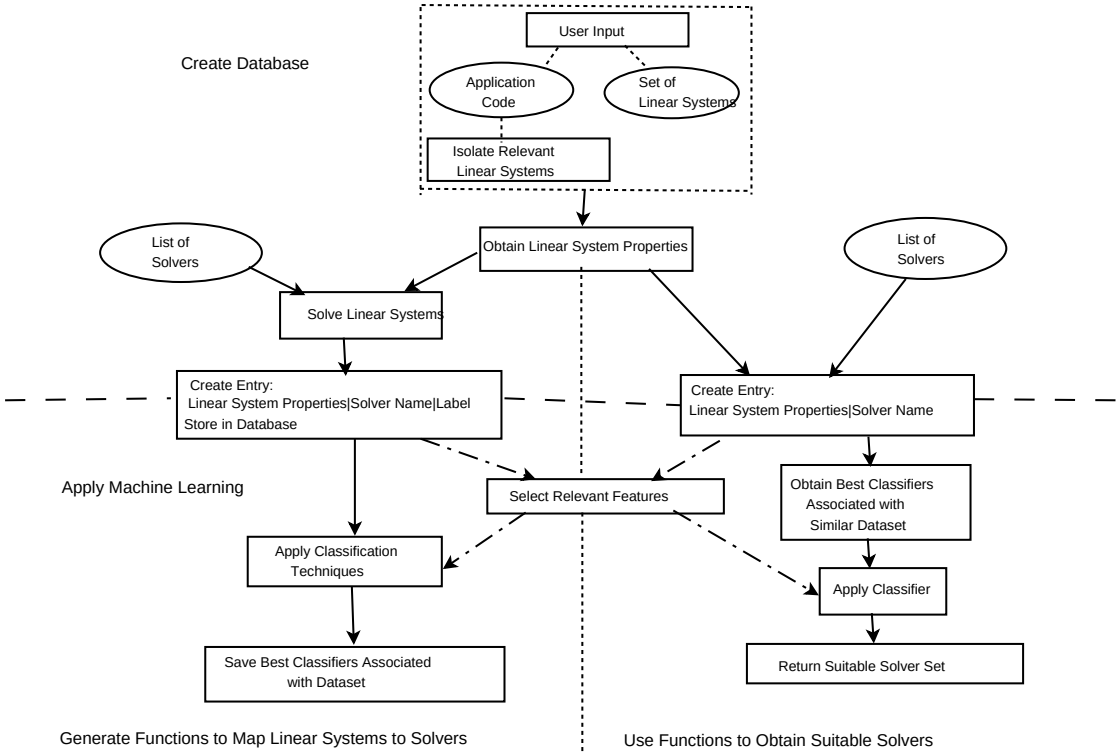

Fig. 5. Flow Chart of the implementation of machine learning for predicting suitable solvers.

point number), *text* (a sequence of words), and *finite* (composed of a defined set of strings and used for specifying attributes which have a fixed set of values, e.g. the list of solvers). MLJava implements boosting [29] and produces classifiers in the form of Alternating Decision Trees [27].

5 Experimental Results

For our experiments we use linear systems from two different applications: i) the lid- and thermally-driven cavity flow in PETSc's scalable nonlinear equations solvers (SNES) demonstration example `ex27.c` and ii) the M3D extended MHD simulation code. We use MLJava to predict the outcome of solvers for the linear systems arising from parameterized instances of these problem sets. The performance parameters are measured by running the applications on the Jazz cluster at Argonne National Laboratory [34], which has a Myrinet 2000 network and 2.4 GHz Pentium Xeon processors with 1-2 GB of RAM. Our results include ROC graphs illustrating the accuracy of the prediction and a comparison of the execution times of the default and predicted solvers. They demonstrate that machine learning techniques can indeed predict "good" solvers. The results also show that the set of suitable solvers can change between different applications, as well as between different instances of the same application, that is, that the performance prediction problem is nontrivial.

5.1 Driven Cavity Flow with Pseudo-transient Continuation

The driven cavity flow model [10, 33] is an example of incompressible flow due to the combined effects of a frictional driven lid and buoyancy in a two-dimensional rectangular cavity. The lid

velocity is steady and spatially uniform and generates a principal vortex and subsidiary corner vortices. The principal vortex is opposed by the buoyancy vortex, which is induced by differentially heated lateral cavity walls. The governing differential equations are the incompressible Boussinesq Navier-Stokes equations in velocity-vorticity form and an internal energy equation. These four linear or quasi-linear scalar elliptic equations are discretized on a uniform mesh using upwind differencing on a nine-point stencil. The resulting system of nonlinear algebraic equations is solved using an inexact Newton method [32, 40]. The linearized Newton system is solved approximately by an iterative Krylov solver. For a fixed grid size, the nonlinearity of the system is determined by the values of the Grashof number and the lid velocity (a Reynolds number). The Prandtl number is set to unity in our tests. It has been observed that Newton's method often struggles at higher values of these parameters when started from a "cold" initial condition. This problem can be overcome by using a globalization technique known as pseudo-transient continuation [33]. We used the driven cavity implementation from the PETSc example [20].

We store the linear systems formed in course of the simulation, and construct a database based on these stored matrices. We use the GMRES Krylov subspace method with PETSc's default restart interval of 30 iterations and block incomplete LU (ILU) with level of fill 0 as the linear solver during the simulation of the driven cavity code, and this is considered as the default solver. We label a solver as "good" if its execution time improves upon the default by a specified factor, such as 1.5 here. The set of solvers is built from the following eight Krylov accelerators: BCGS, TFQMR, FGMRES(5,30,60) and GMRES(5,30,60), where the integers in parentheses denote the restart sizes, and each size variant is considered a different method. The problem was consistently distributed over 4 processors (a reasonable choice for overall execution turnaround) using Block Jacobi as the distributed memory domain-decomposed preconditioner. The preconditioners for individual subdomains include point-block ILU with levels of fill 0,1, and 2; a full-elimination LU sparse solver, point-block Jacobi, and a solver instance where no subdomain preconditioner is used. We will consider a "solver" as being the combination of the Krylov method, the domain decomposition preconditioner, and the subdomain preconditioner. Therefore, for this set of experiments the cardinality of the solver set is 48.

We run simulations with the following values of the physical parameters: Reynolds number based on the lid velocity of (5,15,25) and Grashof number based on the lateral wall temperature differential of (100,500,1000). The dataset has about 15, 000 entries composed by solving linear systems involving the Jacobian matrix and the residual vector as the right-hand side, written out over many pseudotransient timesteps from these 9 different simulations. The classifier is tested on a new set of simulations with Lid Velocities 10 and 20 and Grashof number (100,500,1000). Figure 6, shows the ROC curves and simulation time of the entire driven cavity application with the predicted solvers.

The results show that the BiCGS Krylov solver combined with subdomain preconditioner ILU with levels of fill 0 and 1 performs much better than the default. We see that for this application, the set of suitable solvers remain unchanged over problem instances. The driven cavity problem therefore provides an example in which the time spent in solver selection pays off in a trivial way, by reducing the simulation time over the default, without any need to be dynamic in the selection.

5.2 Linear Systems arising from M3D

Our second problem set is composed of linear systems created during the execution of the M3D code [35, 41]. M3D is a parallel three-dimensional plasma simulation code developed by a multi-institution collaboration in the U.S. Department of Energy. It is suitable for performing linear and nonlinear calculations of plasma in toroidal topologies, including tokamaks (which are toroidally symmetric) and more general stellarators. Ideal and resistive MHD models, as well as a two-fluid model, are implemented. M3D employs finite differences in the toroidal direction and unstructured linear finite elements on triangles in each poloidal crossplane. Only the crossplane problems

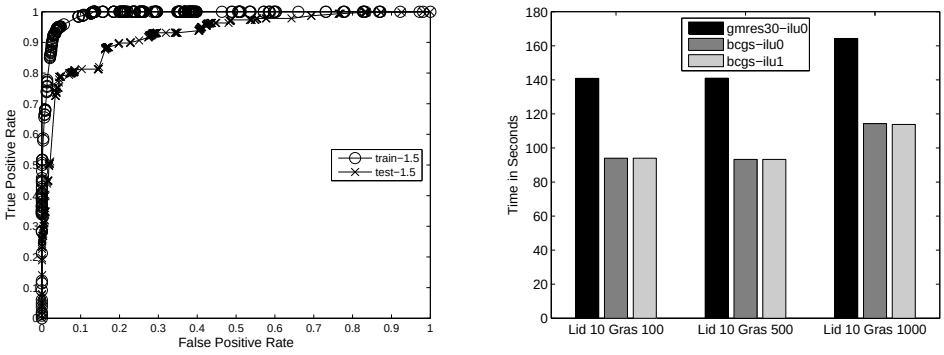

Fig. 6. *Left:* ROC curves for linear systems arising in the Driven Cavity Flow Problem. *Right:*Execution Time for solving Driven Cavity Flow Problem with different solvers

are handled fully implicitly; within each timestep each of the crossplane problems is solved independently as two-dimensional scalar systems. Some of the systems are parabolic in time, some are purely elliptic with Dirichlet boundary conditions, and some are purely elliptic with Neumann boundary conditions. The latter problems possess a nontrivial nullspace of constant functions. Our examples are drawn from a resistive, single-fluid model.

The matrices are segregated into three groups, according to their degree of diagonal dominance: weak, moderate, or strong. The diagonal dominance provides a traditional indicator of the difficulty of finding a solution, weakly dominant matrices being more difficult to solve. In this case, not all solvers are able to solve all the linear systems, therefore we apply two criteria for solver selection: i) guarantee of convergence (to a relative tolerance of 1e-05, within 600 iterations of the Krylov method) and ii) low execution time. The execution time was compared to the time taken by the default solver either to converge or to determine that convergence is not possible.

We used MLJava to build two classifiers, one for each of the above criteria. The classifier corresponding to the first criterion, predicted a solver as "good" if it converged to a solution, without taking in to account the execution time. and the classifier corresponding to the second criterion, predicted a solver as "good" if the execution time was better than the default method by a factor of 1.5, regardless whether it converged or not. It is easy to see that the set of solvers that satisfy both the criteria, can be obtained by taking the intersection of the "good" solvers from the two predictions.

The default solver was GMRES with restart of 30 and ILU with level of fill 0. The solver set comprised of two separate direct solvers, SuperLU [48] and Spooles [47], and six Krylov iterative methods; BCGS, TFQMR, FGMRES(5,30,60), and GMRES(5,30,60). The problem was distributed over 4 processors. The top level preconditioner was ASM with either 0, 1, or 2 degrees of overlap, or an algebraic multigrid method from the Hypre package such as BoomerAMG, ParaSails, and Euclid. When an ASM method was used we also employed subdomain preconditioners including ILU and Incomplete Cholesky (ICC) with 0, 1, and 2 levels of fill, LU, Jacobi, SOR and also an option where no subdomain preconditioner is used. The total number of solvers is 242.

Three groups of linear systems, based on three different two-dimensional finite element meshes with approximately 7,000, 13,000 and 18,000 nonzeros, respectively, were used for building the dataset. The classifier was applied on additional matrices with approximately 12,000 and 24,000 nonzeros. For weakly and moderately diagonal matrices the challenge lies in finding a solver that solves the linear system at all (the default method fails to converge) and for the strongly diagonally dominant matrices the goal is to find a solver that is faster by a multiplicative margin. The results showing the ROC curves and solution times of the matrices are given in Figures 7-

9. Both the weakly diagonally dominant and moderately diagonally dominant matrices require memory-expensive preconditioners like LU and BoomerAMG.

The prediction yields some false positives (it predicted convergence that was contradicted by the results) in the following two cases: i) GMRES(30) with BoomerAMG as a solver for the weakly diagonally dominant matrix with 24,000 nonzeros (failure to converge) and ii) GMRES(60) with ASM overlap 0 and ICC with fill 2 as a solver for the moderately diagonally dominant matrix with 12,000 nonzeros (overflow of available memory). Compared to the number of choices of solvers, the number of false predictions is deemed miniscule. In contrast, there were no false positives in the predictions of the strongly diagonally dominant matrices and though all solvers converged, the predicted "good" solvers were the ones using the Jacobi preconditioner, a preconditioner requiring considerably less memory.

As the results illustrate, the M3D matrices have more varied characteristics than the driven cavity problem because the set of "suitable" solvers change across the matrix groups and even across problem instances in the same group. While weakly and moderately diagonally dominant linear systems require expensive preconditioners such as BoomerAMG, strongly diagonally dominant systems can be solved efficiently with much cheaper preconditioners like Jacobi. Machine learning techniques serve an important role by matching appropriate solvers to these characteristics.

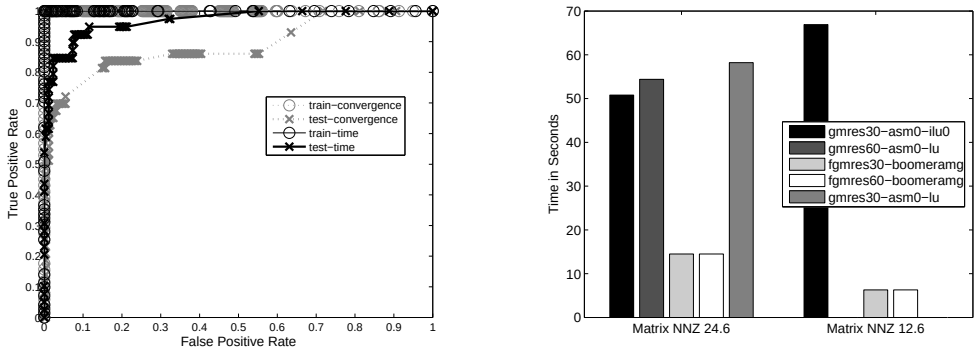

Fig. 7. *Left:* ROC curves for weakly diagonally dominant M3D matrices. *Right:* Time Taken to solve (or detect failure of convergence of) weakly diagonally dominant M3D matrices; GMRES30-ILU0,the default method fails to converge.

6 Conclusions and Future Plans

This paper presents an initial demonstration of the applicability of machine learning to the selection of solution algorithms for large sparse linear systems drawn from computational fluid dynamics and magnetohydrodynamics applications, which are among the leading consumers of resources at major computational user facilities. Boosting, the first of several methods to be evaluated for this purpose, can indeed identify efficient solvers for previously unseen linear systems drawn from a class on which the learner has previously been trained. Given that the solution of linear systems is often the subtask of greatest computational resource demand in an overall scientific or engineering simulation, there is enormous potential in equipping solver libraries with machine learning capabilities. However, machine learning has large data needs and a training cost that are best amortized over applications that have long lifetimes.

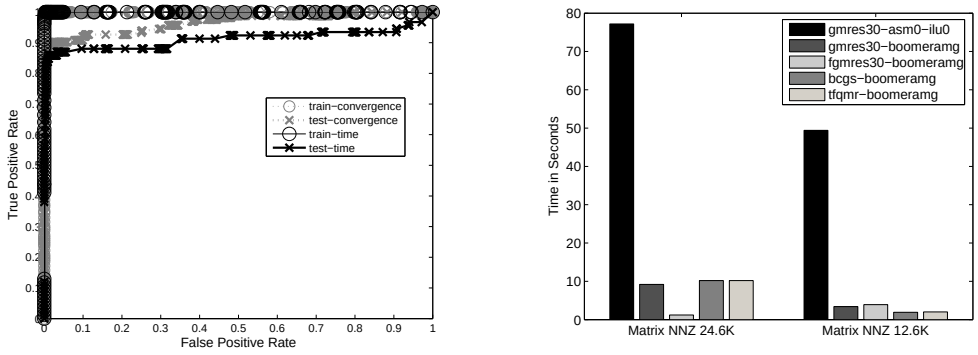

Fig. 8. *Left:* ROC curves for moderately diagonally dominant M3D matrices. *Right:* Time Taken to solve (or detect failure of convergence of) moderately diagonally dominant M3D matrices; GMRES30-ILU0,the default method fails to converge.

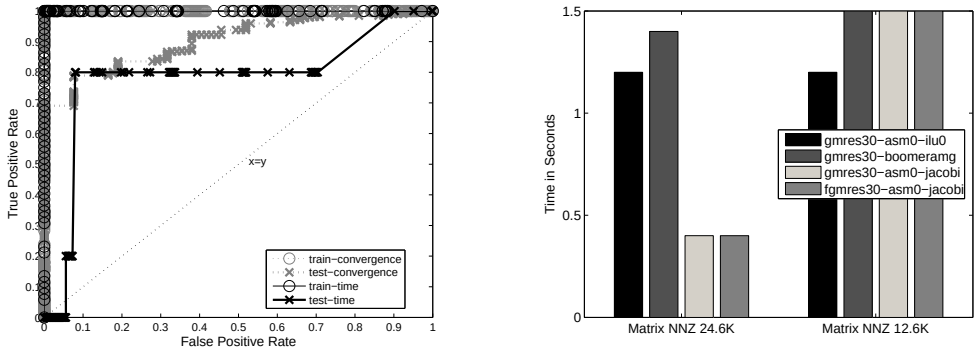

Fig. 9. *Left:* ROC curves for strongly diagonally dominant M3D matrices. *Right:* Time Taken to solve strongly diagonally dominant M3D matrices; GMRES30-ILU0 is the default method; GMRES30-BoomerAMG was not predicted for matrix with nonzeros 24.6K, it is included as comparison between the predicted methods with Jacobi.

We observe that we can order quantitative and descriptive properties of matrices according to their relative importance by counting the number of times each of them appear in the classifier. The correlation between matrix properties and linear solvers is a topic of extensive theoretical and experimental study among numerical analysts (see, e.g., flowchart in [8]) and studying the structure of the classifiers may enable contributions in this area.

Another interesting avenue of research is the comparison of the results of different machine learning methods for solver selection. Just as there are many solution algorithms, so is there a multitude of machine learning algorithms to be employed to select among them, which suggests a meta-machine learning task beyond the first-level task. We plan to extend our experiments to a larger set of application domains and learning methods.

7 Acknowledgments

We would like to thank Jin Chen of the Princeton Plasma Physics Lab for providing us with the M3D matrices. We are also grateful to Raphael Pelossof of Columbia University for his package to render ROC curves from the MLJava output files.

8 Author Biographies

Sanjukta Bhowmick: Sanjukta Bhowmick is a Post Doctoral Research Associate with a joint appointment at at the Department of Applied Physics and Applied Mathematics, Columbia University and the Mathematics and Computer Science Department, Argonne National Laboratory. She obtained her Ph.D. in Computer Science from the Department of Computer Science and Engineering at the Pennsylvania State University. Her current research interests include developing fast and reliable solvers of large scale sparse linear systems, use of machine learning algorithms for detecting linear solvers and combinatorial problems in automatic differentiation.

Victor Eijkhout: Victor Eijkhout is a Research Scientist at the Texas Advanced Computing Center of The University of Texas at Austin. He obtained his Ph.D. in mathematics from the University of Nijmegen in the Netherlands under Owe Axelsson, and subsequently held a postdoc position at the University of Illinois, Urbana-Champaign, and research faculty positions in mathematics at the University of California, Los Angeles and computer science at the University of Tennessee. His current interests are in parallel numerical linear algebra, performance optimization of sparse kernels, and self-adaptive numerical software.

Yoav Freund: Yoav Freund is a professor of computer science in the University of California, San Diego. His primary interest is in machine learning, statistics and their applications. His focus is on algorithms for making statistical inference, especially using large and high-dimensional datasets. Freund and Schapire invented the Adaboost learning algorithm for which they won the 2003 ACM-SIGACT Godel award and the 2004 ACM Kanellakis award.

Erika Fuentes: Erika Fuentes is a PhD student at the University of Tennessee, Knoxville. She obtained her BS. degree in Computer Engineering from the Institute of Technology and Superior Studies of Monterrey (ITESM), Mexico City. Obtained an MS. degree in Computer Science from the University of Tennessee, Knoxville in 2002. Currently working as a graduate research assistant for the Innovative Computing Laboratory in the SALSA project, with main research area in Self-Adaptive Numerical Software and applied statistical methods in numerical problems.

David Keyes: David Keyes is a computational mathematician with primary interests in parallel numerical algorithms and large-scale simulations of transport phenomena – fluids, combustion, and radiation. He is the currently the Vice President-at-Large of the Society of Industrial and Applied Mathematics (SIAM), the Fu Foundation Professor of Applied Mathematics at Columbia University, and the Acting Director of the Institute for Scientific Computing Research (ISCR) at Lawrence Livermore National Laboratory. He is active in the domain decomposition, parallel CFD, and petaflops architecture communities. His academic background includes degrees in mechanical engineering (B.S.E., Princeton, 1978) and applied mathematics (Ph.D., Harvard, 1984). He shared in the 1999 Gordon Bell "Special" Prize.

References

1. Anamod online documentation. `http://www.tacc.utexas.edu/~eijkhout/doc/anamod/html/`.
2. M. F. Adams, Harun H. Bayraktar, Tony M. Keaveny, and Panayiotis Papadopoulos. Ultrascalable implicit finite element analyses in solid mechanics with over a half a billion degrees of freedom. In *Proceedings of SC'04*. Winner of Gordon Bell Special Prize at SC2004: Large scale trabecular bone finite element modeling:`http://www.sc-conference.org/sc2004/schedule/pdfs/pap111.pdf`.
3. Volkan Akcelik, Jacobo Bielak, George Biros, Ioannis Epanomeritakis, Antonio Fernandez Omar Ghattas, Eui Joong Kim, David O'Hallaron, and Tiankai Tu. High resolution forward and inverse earthquake modeling on terascale computers. In *Proceedings of SC'03*, 2003.
4. W. K. Anderson, W. D. Gropp, D. K. Kaushik, D. E. Keyes, and B. F. Smith. Achieving high sustained performance in an unstructured mesh cfd application. In *Proceedings of SC'99*, 1999.
5. O. Axelsson. A survey of preconditioned iterative methods for linear systems of equations. *BIT*, 1987.
6. S. Balay, K. Buschelman, W. Gropp, D. Kaushik, M. Knepley, L. McInnes, Barry F. Smith, and H. Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 2.2.1, Argonne National Laboratory, 2004. `http://www.mcs.anl.gov/petsc`.
7. R. Barrett, M. Berry, J. Dongarra, V. Eijkhout, and C. Romine. Algorithmic bombardment for the iterative solution of linear systems: A polyiterative approach. *Journal of Computational and applied Mathematics*, 74:91–110, 1996.
8. Richard Barrett, Michael Berry, Tony Chan, James Demmel, June Donato, Jack Dongarra, Victor Eijkhout, Roidan Pozo, Charles Romine, and Henk van der Vorst. *Templates for the Solution of Linear Systems:Building Blocks for Iterative Methods*. SIAM, 1994.
9. Peter L. Bartlett. The sample complexity of pattern classification with neural networks: the size of the weights is more important than the size of the network. *IEEE Transactions on Information Theory*, 44(2):525–536, March 1998.
10. B. A. V. Bennett and M. D. Smooke. Local rectangular refinement with application to nonreacting and reacting fluid flow problems. *Journal of Computational Physics*, 151:648–727, 1999.
11. S. Bhowmick, L. C. McInnes, B. Norris, and P. Raghavan. The role of multi-method linear solvers in pde-based simulations. In P. M. A. Sloot, C.J. K. Tan, J. J. Dongarra, and A. G. Hoekstra, editors, *Lecture Notes in Computer Science, Computational Science and its Applications-ICCSA 2003*, volume 2667, pages 828–839. Springer Verlag, 2003.
12. S. Bhowmick, P. Raghavan, L. McInnes, and B. Norris. Faster PDE-based simulations using robust composite linear solvers. *Future Generation Computer Systems*, 20:373–386, 2004.
13. S. Bhowmick, P. Raghavan, and K. Teranishi. A combinatorial scheme for developing efficient composite solvers. In P. M. A. Sloot, C.J. K. Tan, J. J. Dongarra, and A. G. Hoekstra, editors, *Lecture Notes in Computer Science, Computational Science- ICCS 2002*, volume 2330, pages 325–334. Springer Verlag, 2002.
14. Leo Breiman. Arcing classifiers. *The Annals of Statistics*, 26(3):801–849, 1998.
15. Leo Breiman, Jerome H. Friedman, Richard A. Olshen, and Charles J. Stone. *Classification and Regression Trees*. Wadsworth & Brooks, 1984.
16. Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, September 1995.

17. Jim Demmel, Jack Dongarra, Victor Eijkhout, Erika Fuentes, Antoine Petitet, Rich Vuduc, R. Clint Whaley, and Katherine Yelick. Self adapting linear algebra algorithms and software. *IEEE Proceedings*, 2004.
18. J. Dongarra and V. Eijkhout. Self adapting numerical algorithm for next generation applications. *International Journal of High Performance Computing Applications*, 17(2):125–132, 2003.
19. Jack Dongarra and Victor Eijkhout. Self-adapting numerical software and automatic tuning of heuristics. In *Proceedings of the International Conference on Computational Science, June 2–4 2003, St. Petersburg (Russia) and Melbourne (Australia), Lecture Notes in Computer Science 2660*, pages 759–770. Springer Verlag, 2003.
20. Driven-Cavity. Nonlinear driven cavity and pseudotransient timestepping in 2d. `http://www-unix.mcs.anl.gov/petsc/petsc-as/snapshots/petsc-current/src/snes/examples/tutorials/ex27.c.html`.
21. Harris Drucker and Corinna Cortes. Boosting decision trees. In *NIPS8*, pages 479–485, 1996.
22. Richard O. Duda and Peter E. Hart. *Pattern Classification and Scene Analysis*. Wiley, 1973.
23. I.S. Duff, A.M. Erisman, and J.K. Rei. *Direct Methods for Sparse Matrices*. Clarendon Press, Oxford, 1986.
24. Victor Eijkhout and Erika Fuentes. A proposed standard for numerical metadata. submitted to ACM Trans. Math. Software.
25. 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, No 3:681–703, 1994.
26. Yoav Freund. Boosting a weak learning algorithm by majority. *Information and Computation*, 121(2):256–285, 1995.
27. Yoav Freund and Llew Mason. The alternating decision tree learning algorithm. *In Proceedings of the Sixteenth International Conference on Machine Learning*, pages 124–133, 1999.
28. Yoav Freund and Robert E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55(1):119–139, August 1997.
29. Yoav Freund and Robert E. Schapire. A short introduction to boosting. *Journal of Japanese Society for Artificial Intelligence*, 14(5):771–780, 1999.
30. Jerome Friedman, Trevor Hastie, and Robert Tibshirani. Additive logistic regression: A statistical view of boosting. *The Annals of Statistics*, 38(2):337–374, April 2000.
31. D. Gannon, R. Bramley, T. Stuckey, J. Balasubramanian J. Villacis, E. Akman, F. Berg, S. Diwan, and M. Govindaraju. The linear system analyzer. In E.N. Houstis, J.R. Rice, E. Gallopoulos, and R. Bramley, editors, *Enabling Technologies for Computational Science*. Kluwer:Dordrecht, 2000.
32. W.D. Gropp, D.E. Keyes, L.C. McInnes, and M.D. Tidriri. Globalized Newton-Krylov-Schwarz Algorithms and Software for Parallel Implicit cfd. *International Journal of High Performance Computing Applications*, 14:102–136, 2000.
33. C. T. Kelley and D. E. Keyes. Convergence analysis of pseudo-transient continuation. *SIAM Journal on Numerical Analysis*, 35:508–523, 1998.
34. LCRC. Argonne National Laboratory Computing Project. `http://www.lcrc.anl.gov/jazz/index.php`.
35. M3D-Home. `http://w3.pppl.gov/$\sim$jchen/index.html`.
36. L. McInnes, B. Norris, S. Bhowmick, and P. Raghavan. Adaptive sparse linear solvers for implicit cfd using Newton-Krylov algorithms. *Proceedings of the Second MIT Conference on Computational Fluid and Solid Mechanics*, June 17-20, 2003.
37. MLJava. `http://seed.ucsd.edu/twiki/bin/view/Softtools/MLJavaPage`.
38. N. M. Nachtigal, S. C. Reddy, and L. N. Trefethen. How fast are nonsymmetric matrix iterations? *SIAM J. Matrix Anal. Appl.*, 13, No 3:778–795, 1992.
39. D.J. Newman, S. Hettich, C.L. Blake, and C.J. Merz. UCI repository of machine learning databases, 1998.
40. J. Nocedal and S. J. Wright. *Numerical Optimization*. Springer-Verlag, New York, 1999.
41. W. Park, E.V. Belova, G.Y. Fu, X. Z. Tang, H. R. Strauss, and L. E. Sugiyama. Plasma simulation studies using multilevel physics models. *Physics of Plasmas*, 6(5):1796–1803, May 1999.
42. J. R. Quinlan. Bagging, boosting, and C4.5. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 725–730, 1996.
43. J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.
44. Yousef Saad. *Iterative Methods for Sparse Linear Systems*. PWS Publishing Company,, 1995.
45. Robert E. Schapire. The strength of weak learnability. *Machine Learning*, 5(2):197–227, 1990.

46. Robert E. Schapire, Yoav Freund, Peter Bartlett, and Wee Sun Lee. Boosting the margin: A new explanation for the effectiveness of voting methods. *The Annals of Statistics*, 26(5):1651–1686, October 1998.
47. SPOOLES. Sparse direct solver. `www.netlib.org/linalg/spooles/spooles.2.2.html`.
48. SuperLU. Sparse direct solver. `crd.lbl.gov/~xiaoye/SuperLU`.
49. L. G. Valiant. A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142, November 1984.
50. Vladimir N. Vapnik. *Statistical Learning Theory*. Wiley, 1998.
51. I.H. Witten and E.Frank. *Data Mining:Practical Machine Learning Tools and Techniques (second edition)*. Morgan Kaufmann, 2005.