12-2007

Statistical and Machine Learning Techniques Applied to Algorithm Selection for Solving Sparse Linear Systems

Erika Fuentes
University of Tennessee - Knoxville

To the Graduate Council:

I am submitting herewith a dissertation written by Erika Fuentes entitled "Statistical and Machine Learning Techniques Applied to Algorithm Selection for Solving Sparse Linear Systems." I have examined the final electronic copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Computer Science.

Jack Dongarra, Major Professor

We have read this dissertation and recommend its acceptance:

J.D. Birdwell, Victor Eijkhout, Lynne Parker, Tsewei Wang

Accepted for the Council:

Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

To the Graduate Council:
I am submitting herewith a dissertation written by Erika Fuentes entitled "Statistical and Machine Learning Techniques Applied to Algorithm Selection for Solving Sparse Linear Systems". I have examined the final paper copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Computer Science.

Jack Dongarra
Major Professor

We have read this dissertation
and recommend its acceptance:

J.D. Birdwell

Victor Eijkhout

Lynne Parker

Tsewei Wang

Accepted for the Council:

Carolyn R. Hodges
Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

Statistical and Machine Learning Techniques Applied to Algorithm Selection for Solving Sparse Linear Systems

A Dissertation
Presented for the
Doctor of Philosophy
Degree
The University of Tennessee, Knoxville

Erika Fuentes
December 2007

Dedication

This dissertation is dedicated to my parents, who have always believed in me. Their love and support has made me the person I am today and motivated me to achieve my goals.

Acknowledgments

I would like to express my sincere gratitude to my advisor, Dr. Jack Dongarra, for all of his support and guidance throughout my studies, for giving me the opportunity to work and learn under his advisement, and for encouraging me to develop my skills as a scientist and as a professional in general.

My deepest thanks and appreciation are due to Dr. Victor Eijkhout for devoting so much time to sharing his knowledge with me, and for all of his valuable input to this dissertation. His vast expertise has greatly helped me to improve my scientific background and has motivated me to pursue greater goals. His advice, support, and counsel over the years have been extremely valuable to me every day and in every way.

I would also like to thank the rest of my committee: Dr. Lynne Parker, Dr. TseWei Wang, and Dr. Douglas Birdwell. Their support and helpful contributions have been invaluable during my studies, as well as in the development of this dissertation. Their dynamic scientific character has inspired my curiosity and motivated my desire to learn and professional grow.

I would also like to acknowledge the support and help of my friends and colleagues. Many thanks to Paul DeWitt for spending so much time discussing algorithms and theorems, to Wei Jiang for his valuable discussions on pattern recognition and for all of his continued support over the years, to Dr. Hairong Qi for her helpful suggestions in the pattern recognition area and Dr. Julien Langou for his invaluable inputs regarding linear algebra, to Alejandra Fuentes for sharing her time and creativity with me, to Alan Pendleton for his support and help with my writing style, and to Matthew Parsons for all of his valuable discussions, efforts, patience, and understanding. Much gratitude is due to the aforementioned people in addition to many other friends for their emotional support and encouragement (Eva, Liz, Amanda and Alison). My sincere appreciation goes to my colleagues and friends Jelena, Thara, Peng, Alfredo and Stan, for their valuable input and suggestions. Thanks as well to all those who may not be mentioned by name here, but whose knowledge, guidance, and support has played a part in the formation of this dissertation.

Abstract

There are many applications and problems in science and engineering that require large-scale numerical simulations and computations. The issue of choosing an appropriate method to solve these problems is very common, however it is not a trivial one, principally because this decision is most of the times too hard for humans to make, or certain degree of expertise and knowledge in the particular discipline, or in mathematics, are required. Thus, the development of a methodology that can facilitate or automate this process and helps to understand the problem, would be of great interest and help. The proposal is to utilize various statistically based machine-learning and data mining techniques to analyze and automate the process of choosing an appropriate numerical algorithm for solving a specific set of problems (sparse linear systems) based on their individual properties.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Many fundamental tasks in scientific computing can be approached through multiple algorithms; such is the case of the solution of sparse linear systems. The solution process can be improved by adapting certain algorithms to the characteristics of the linear systems. Our research centers on using non-numerical techniques, such as statistics and machine learning, to experimentally analyze and find the most suitable solution method to a given numerical problem based on its numerical properties. These findings can ultimately be used to build a recommendation system that can facilitate the algorithm selection process for users in other disciplines, who lack numerical expertise. The particular set of numerical problems we will address is finding the solution (using iterative methods) for $Ax = b$ where A is a system of linear equations with a sparse coefficient. The main idea of the proposed approach is to use various statistical techniques to understand the behavior of these numerical problems, and then develop a methodology to automate the process of choosing an appropriate numerical algorithm (reliable and optimal among the available) for solving them.

In this chapter, we cover the motivation for this research, related work and contributions.

1.1 Motivation

There are many applications and problems in science and engineering that require large-scale numerical simulations and computations. The issue of choosing an appropriate method to solve these problems is very common but not trivial. Such a decision is typically difficult for humans to make, requiring certain degree of expertise and knowledge in the particular discipline, or mathematics. Furthermore, there is no theoretical knowledge about what type of method is the most suitable for many numerical problems, or about the relationship between the characteristics of the problem and the behavior of the methods. Thus, research and experimentation using a statistical approach, which can help to uncover these relationships, would be of great interest and help. The development of a software tool that implements this knowledge can facilitate and automate the decision making process.

Choosing a suitable numerical algorithm in scientific computing implies searching for optimal solutions, in terms of performance (time for solving the system) or in terms of accuracy, depending on the requirements of the application and the user. Part of the problem is that numerical solvers usually possess a large number of parameters; it is necessary to understand and find a way of tuning these parameters for optimal performance.

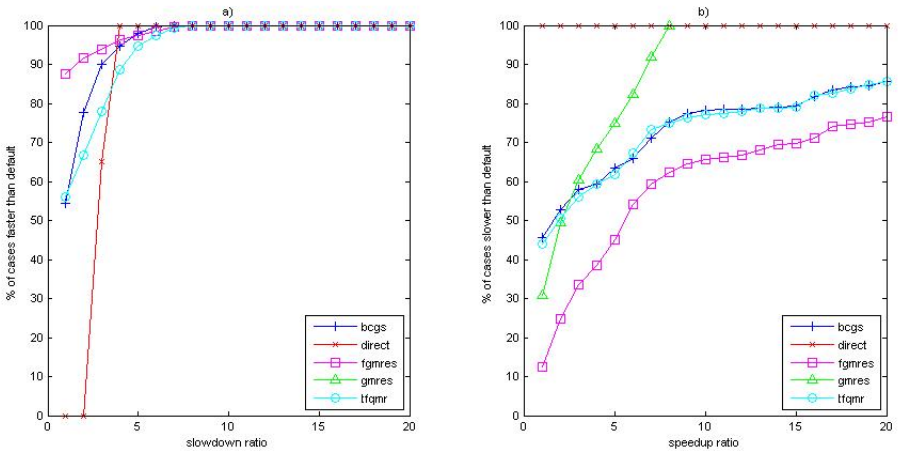


Figure 1.1: Performance profile for iterative methods with respect to default method $gmres20$: (a) Shows the cumulative proportion of cases for each iterative method where the default method was faster for different slowdown ratios. (b) Shows the cumulative proportion of cases for each iterative method where the default method was slower for different speedup ratios.

The group of algorithms for solving numerical problems is composed of direct and iterative approaches. When solving dense matrices using direct solvers the time complexity is known. Iterative methods for solving sparse linear systems may be more efficient, but the time required by these methods is unknown and depends greatly on the type of problem or preprocessing steps involved. In some cases, iterative methods may take "infinite" time (in such a case we say that the iterative method *diverges*).

This research focuses on the task of solving linear systems of sparse coefficients using various iterative methods. The process of choosing an optimal method depends not only on the properties of the problem, but also on the behavior and characteristics of the different applicable methods. The impact of the properties of the matrices on the performance of the solvers has not been considered very often; thus, the problem presents an interesting field for experimentation and research.

Many numerical libraries, applications and engineering packages for solving numerical problems are required to be reliable and work with stability most of the time, even if they are not as efficient as they could be. For this reason, a conservative solver is usually established as default. For users who are not skillful enough to tune various solver parameters or to choose preprocessing and preconditioning options, picking an efficient solver is a difficult task. Although a default solver may be a safe option in most cases, more efficient solvers and options can be used; naturally, there will still be problems for which even a default iterative solver will not work. Figure 1.1 shows performance comparisons of iterative methods applied on an experimental dataset (M3D in Appendix A.2.4) against a default method. These plots are known as "performance profiles" and they show how fast each available method is compared to a default choice ($gmres$ with restart parameter 20 for this particular example), and in how many cases it was faster than the default. From Figure 1.1(a) we can see that in general the default is not much faster than other methods, and other methods can be much faster than the default, in which case there is a lot of room for improvement by picking a different method (Figure 1.1(b)).

In numerical literature we can predominantly find qualitative knowledge about the comparative advantage of different methods on different types of problems, since there is almost no quantitative information that explains these relations. Part of the proposed research studies the performance of various algorithms and how it relates to the particular properties of different problem instances [Demmel et al., 2005]. The discovery of this kind of information could be helpful in the design and development of automated decision-making software tools, which could combine prior knowledge with heuristic models and new problems to perform the algorithmic choice.

1.2 Contributions

The task of choosing an appropriate algorithm to solve linear systems (or other numerical problems), based on some metric combining memory capacity and execution time in a realistic computing environment, is very difficult even when a relatively small class of problems is considered. We propose the use of statistics, machine learning and data mining − non-traditional methodologies in the numerical area − to study the problem and develop a methodology that will serve as a tool to facilitate the process of appropriate algorithm selection. Such a methodology will make it easier to use numerical techniques in an application context without expert knowledge, and to systematize their benefits for a wider community.

The proposed works aims to reduce the cost and increase the efficiency of large-scale computations and simulations in many fields of science and engineering. The most significant impact can be for large-scale computations, where even small improvements achieved by choosing optimal solvers can result in considerable savings of time and computational resources. Smaller-scale computations can also benefit from optimal-algorithm selection, because generally, default methods embedded in many applications often converge slowly or not at all, yet the next better choice is out of reach of the unsophisticated, unexperienced user. In both cases, a leading benefit is the identification of reliable methods (e.g., converging iterative methods); gains in performance are important, but avoiding methods that diverge is extremely valuable.

There are many factors involved in the process of finding an appropriate solver. A few of these are documented in the literature as rules or parameter settings, while others exist only as heuristic knowledge in the mind of experts in numerical analysis. Our research also attempts to identify and study the importance of these factors as we integrate them into an automated decision making process.

Machine learning and statistics have been widely used in many areas, but its application in numerical analysis has not been explored nor taken advantage of. One of the innovative characteristics of this research lies in the integration of numerical analysis with these 'unconventional' techniques. By using some of the tools they provide we aim to understand if/how the aforementioned factors impact the behavior of different numerical algorithms. The proposed research does not focus on numerical solvers themselves, nor on the study of statistical and machine learning peculiarities, but in their combination, aiming to discover trends in numerical problems that will simplify their application and understanding by people in other disciplines. The concepts and strategies here developed can be extended to other problems and applications not addressed here, such as eigenanalysis, parameter tuning and optimization.

Furthermore, by using data mining techniques we look to uncover data relationships, explore different numerical algorithms' behaviors and see whether and how they correlate with the characteristics of different input problems. The identification of currently unknown patterns can create new research paths and motivate future investigation in the numerical area.

Additionally, most of the related research in this area has often overlooked the impact of numerical features of the input problems on the behavior of solvers and other algorithms. Given that the performance of most statistical techniques depends greatly on the input data, it is crucial to study and understand the nature of these properties. We propose the application of some powerful tools in data mining, such as Principal Component Analysis, which have not had an application in the numerical area, to evaluate their efficiency for determining relevant data and its relations. The use of these analysis tools provides valuable knowledge (or possible directions for future research) about currently unknown trends and the significance of specific properties of the problems.

1.3 Related Work

As mentioned in section 1.1, there is no current quantitative knowledge of the effect of the properties of linear systems on the performance of different methods and transformations. However, there exist some efforts that attempt to address this problem. In this section, we briefly describe some of the related research into algorithmic adaptivity and adaptive numerical software. We also explain how the research presented in this dissertation differs from other approaches. Note that our interest is only in systems that work on the algorithmic level, not taking into account those systems for kernel and network optimization levels.

One of the first concepts that addressed the idea of having a computer choose the best available algorithm to solve a problem was the "polyalgorithm" [Rice and Rosen, 1966], which was first formulated in 1966. The polyalgorithm also (intrinsically) provides numerical analysis features for very simple problems.

LINSOL is a package [Häfner et al., 1998] that picks an iterative solver through backtracking using a "poly-algorithm". Different from our study, the linear system in question is not analyzed before the iteration process; the decision making process is done on-line during the iteration, based on tracking the error norm. Furthermore, the backtracking is done only through the space of iterative methods without considering the effects or ordering of preconditioners, and the number (and type) of iterative solvers considered is very small.

Research by Kuroda, *et al* [Kuroda et al., 1999] introduces ideas to perform automatic tuning. In [Kuroda et al., 1999] they concentrate on the analysis of restarted GMRES (definition in Section 2.1.1) and identify related factors for the tuning (although no actual tuning is performed). Some of these factors are numerical parameters that influence the restart length of GMRES, orthogonalization strategies, and software implementation and architectural parameters such as loop unrolling and some MPI communication commands. They identify, study, and experiment with parameters (which are assumed to be independent) for the parallel tridiagonalization problem. The determination of numerical parameters is based on timing results from a set of simple tests (which include the use of preconditioners and GMRES); the determination of architectural parameters are determined by the matrix. The authors have also developed a library for linear systems and eigenvalues called ILIB [Kuroda et al., 2002].

Related work by Katagiri [Katagiri et al., 2004] describes an auto-tuning software architecture framework. The research is based on performing tuning based on user derived knowledge about

different parameters of numerical software and architecture-related factors such as loop-unrolling. This is also an experience-based learning system, but its application area is specifically the solution of eigenvalue problems, and their approach does not stress the use of numerical properties of the problems.

IPRS The University of Kentucky's Intelligent Preconditioner Recommendation System (IPRS) [Xu and Zhang, 2004] focuses on providing a recommendation of a suitable preconditioner and its parameters for solving efficiently sparse linear systems. The research proposed here mainly differs from the IPRS project in that their work is limited to the prediction of preconditioners, while we expect to address a wider range of decisions on iterative methods and other transformations. The behavior and performance of these precondtioners are characterized in terms of the structural properties of matrices. Their experiments are limited to the use of a couple of variants of the ILU preconditioner on a relatively small pool of matrices.

Pythia The Pythia project [Houstis et al., 1995, Houstis et al., 2000], started at Purdue University, is a system that recommends scientific software using methodologies for knowledge discovery in databases based on the processing of performance data. The idea behind Pythia is to build a "recommender system" based on the performance-related information stored in a database. The system PYTHIA II is an example of this idea, which targets the solution of PDEs using PELLPACK [Houstis et al., 1998]. Both the framework and learning methodologies used in this project differs from ours. Pythia uses knowledge discovery in databases, which is a static learning methodology, while we propose a dynamic approach. Furthermore, they focus on a limited amount of linear system solvers, only applicable to elliptic systems.

In consequent research by Ramakrishnan and Ribbens [Ramakrishnan and Ribbens, 2000] parameter tunning is investigated in the context of setting a method parameter depending on the variations of the parameters or characteristics of a problem. Some of the limitations of this work reside in the difficulty that such a system has for generalization. The parameters are grouped in fixed intervals or "bins" and the decision process works by exhaustively testing all possible problems in one group. The problem's parameter is actually a parameter in the PDE that originates the problem, which makes the exhaustive search possible because the parameter space is bounded. This limits the ability of the model to work with parameters beyond these established groups. Although this is a common problem in machine learning, this model seems to be very parametrized (could result in overfitting behavior), and there is no evidence that the model proposed can be extended to use experimental parameters or even other types of parameters.

ITBL The project Test of Iterative Solvers on ITBL [Fukui and Hasegawa, 2005] presents a similar framework to Pythia. The main focus of this work is the solution of sparse linear systems, using the sparsity structure of the problem and the documented performance (in time only) for various solvers in different computing environments.

Recommender for iterative solvers Work done by T. George and V. Sarin [George and Sarin, 2007] describes a generic approach to build a recommender for preconditioned iterative solvers. The main strategy is based on the ranking of a list of solvers for a particular linear system. Each solver is ranked based on its "feasibility" (whether it converges to solution or not), and "goodness" (the best approach based on a performance metric − time to solution or memory usage during

execution). In their work, they try to focus more on the properties of the problems (linear systems) and they acknowledge the difficulty of feature selection (some of the features we used are referenced in their work). Their work is still in early stages and there are no results yet. One of the main differences with our work in that their method space consists only of one method (GMRES) and one preconditioner with different parameters.

MPI Collective Operations This research focuses in the use of statistical learning techniques to perform optimal algorithm selection for MPI collective operations [Pješivac-Grbović et al., 2007b, Pješivac-Grbović et al., 2007a]. The research presents the use of decision trees to choose the optimal algorithm based on several parameters and characteristics of the problems and architectures used. This work differs from ours in that the application area is very different, but also the feature space they explore is considerably smaller than ours, which allows a more thorough exploration of the decision tree structures used to build the decision rules; finally, the only statistical learning methodology used is decision trees.

Combinatorial composite methods Work presented in [McInnes et al., 2003] proposes the use of "multi-methods" for problems where a number of methods are tried in sequence. This sequence is determined experimentally by an overall reliability ranking, independent from particulars of the input problems. In our research, the main criterion to choose (or rank) methods is given by the input data.

Related research [Bhowmick et al., 2005, Bhowmick et al., 2004] presents a combinatorial scheme for combining individual methods into a more reliable composite. Each method is ranked by a utility value $u_i = t_i/r_i$, where t_i is the time to solution (perhaps an upper bound) of method i for a problem and r_i is the reliability of the method. By ordering the methods according to increasing utility, they obtain a method sequence with lowest expected solution time over all permutations of that group of methods.

Again, the characteristics of the input problems are not taken into account. Also, the handling of method divergence seems to pose a problem: even if some method in a sequence diverges for the same set of problems, it is still applied, so this time is wasted, making the definition of t_i doubtful. In our approach, we use properties of the problems to help determine divergence.

Others One of the first and most common applications of algorithmic adaptation is some type of compilers. These use trace data from previous program executions to tune certain parameters and make compile-time decisions for optimal code generation. We extend this idea to algorithmic adaptivity based not only on execution history, but also on specific properties of problems.

Work by Kunkle [Kunkle, 2005] proposed the use of Support Vector Machines (SVM) to do algorithm selection. They suggest the use of SVM as an evolutionary algorithm to determine the most efficient solver. Their problem scope is, like ours, solving linear systems using various methods; however, the work discussed is at a very early stage (data collecting stage) and there are no signs that more results have been obtained. SVM is a powerful technique, however is computationally expensive and it is used for two-class problems only, which means that multiclass problems need to be solved hierarchically as binary problems, increasing even more the computation time.

The work [Lagoudakis and Littman, 2000] presents a methodology to perform algorithm selection system by using Reinforcement Learning. This research focuses on recursive and non-recursive algorithms, where the selection is based on the run-time performance of each algorithm and it is

done on-line. The problem is modeled as a Markov Decision Process (MDP) with reinforcement learning, an approach suited well for algorithms that have long-term and short-term timing trade-offs, such as recursive algorithms. The feature space of the problems used in this work consists only of one feature, the size of the problem. The on-line nature of this approach makes it difficult to extend to problems with large feature spaces, or with a more varied choice of algorithms, because of the increase in its complexity.

Oreizy *et al* [Oreizy et al., 1999] describes an architecture-based model that allows the dynamic modification of the functionality of the software. The type of applications targeted is very different from ours. There is no formal description of their model.

Work by Brewer [Brewer, 1995] and Sussman [Sussman, 1992] consists of modeling the runtime of a small set of algorithms by using a small set of parameters measured from specific tests. This work, like ours, uses the a-priori knowledge notion for their models. However, in our research it is not possible to predict runtime, due to the amount and type of algorithms and their parameters as well as the involvement of actual problem characteristics.

Collaborations Ongoing work with collaborators at Columbia University and UCSD [Bhowmick et al., 2007] is more closely related to the research presented in this dissertation. This project evaluates the performance of various machine-learning techniques on different datasets, concentrating only on a few numerical properties of the problems. The problems are solved with parallel iterative solvers (due to their size and complexity). Our collaborators have focused on the analysis of the Boosting learning algorithm [Freund and Schapire, 1999] for making method predictions. In our research we use other statistical techniques and we continue to stress the importance of characterizing more numerical features of the problems and on the possible impact they have on the behavior of iterative solvers. Additionally, we have introduced the use of data mining techniques such as PCA for a more extensive feature analysis.

1.4 Assumptions and Limitations

The experiments and results presented in this dissertation do not consider parallel implementations of iterative methods and preconditioners. Although some parallel implementations for preconditioners were used in the experiments, these were applied using a single processor. While a preconditioner may behave similiarly in both uniprocessor and parallel implementations, these experiments do not account for the preconditioners as a function of processors. This is a different problem which involves architectural properties instead of the matrices' numerical properties. Although the methodoloy will be similiar for the parallel implementation, this is beyond the scope of this research.

The machine learning methods used in this research do not use validation techniques. For example, the decision trees methodology would handle overfitting or overtraining problems by using pruning techniques, increasing the accuracy of classification. However, based on the descision tree results, this work assumes that the results are sufficiently accurate and further validation would only result in minimal gains. Work in [Elomaa, 1999] discusses the cost of pruning, which in many problems can be too high and does not dramatically improve the behavior of decision trees.

1.5 Outline of the Document

This document is organized as follows. Chapter 2 contains an outline of the numerical problems addressed in this dissertation. Chapter 3 describes various statistical and classification techniques of this research, giving a preview of their application in the numerical context. Chapter 4 explains general concepts of how a classification and statistical learning approach can be used as a framework to address the algorithm selection and recommendation problem. This chapter also describes its different subproblems: reliability and performance. Chapter 5 focuses on the description of the reliability problem, which consists of determining which methods are reliable for certain problems. Chapter 6 concerns itself with the description of the performance problem, which is finding and predicting which method is the optimal for a problem. Both chapters cover definitions and formalization of the elements used to address these problems and describe in detail the particular algorithms developed for classification and recommendation applied to the problem of method selection for linear systems. In Chapter 7, we present various experiments and discuss how to analyze and interpret how their results were used to develop the strategies developed and described in Chapters 5 and 6. It also shows how recommendations can be built based on the experimental results from classification and recommendation. In Chapter 8 we present conclusions and discuss future work.

Chapter 2

Numerical Background and the SALSA Framework

In this chapter we present the numerical scope and concepts considered in this research, and some of the limitations in this context. We also give an outline of the *Self Adaptive Large-scale Solver Architecture* (*SALSA*) framework, which is one of the application ideas that served as motivation for the work presented in this dissertation.

2.1 Numerical Background

When we talk about numerical problems in this research, we refer specifically to linear system solving. There are dense and sparse linear systems. Dense systems can generally be solved efficiently using some implementation of Gaussian elimination like in Lapack [Angerson et al., 1990] or Scalapack [Choi et al., 1992]. The choices that are left to make regarding the solver are whether to run it sequentially or in parallel and if so, how many processors to use. In some cases, iterative methods are also suitable for dense problems. The solvers for sparse matrices are different. They can be direct, iterative or multigrid. For these methods there are different and more choices. This choice of algorithm mainly depends on the how the sparse matrix was originated (e.g., different iterative methods behave differently for problems originating from PDE's than for those from optimization processes), and on the additional data that is generated during the solving process of the matrix. To address the later case, preconditioners are brought into play. The scope of our research focuses on the use of iterative solvers applied on sparse linear systems since they pose an interesting and unexplored ground in terms of method selection.

Direct methods [Duff et al., 1989] can assure convergence to solutions better than any iterative method; they may have excessive memory requirements and exceed accuracy requirements for some problems. Iterative methods [Axelsson, 1987] are less robust but also require less memory. However, it is more challenging to understand their behavior and estimate the time to convergence.

The applicability of machine learning and statistical techniques in the numerical area may go beyond these particular problems; however, these seem to be a good starting point to investigate the potential of such techniques, in particular because of the broad type of areas where they appear, as well the possibility to improve their usage.

Many iterative methods have more than one parameter that governs their performance or that can benefit from data preprocessing or transformations such as preconditioning. Different transformations alter different properties of the linear systems and can affect the performance of iterative methods. Some types of transformations are:

Preconditioner (pc): this is perhaps the most important transformation used in our research. This is not a transformation of the coefficient matrix like the other transformations used in here. A preconditioner is an matrix (Symmetric Positive Definite) M which approximates the matrix A, such that $Mx = b$ is inexpensive to solve as opposed to $Ax = b$ [Saad, 1995].

Scaling: performs point wise left, right or symmetric scalings by the diagonal of the coefficient matrix of a linear system.

Approximation: when a preconditioner is not derived from the coefficient matrix, it can be derived from linear element discretization of the same problem, e.g.,in the cases of higher order finite element matrices.

Distribution: comprise permutations and load balancing. The sensitivity of a linear system to this type of transformation typically comes through the preconditioner, for instance incomplete factorizations are sensitive to permutation while block Jacobi preconditioners are to load distributions.

Preconditioning is a very important technique that affects the performance of iterative methods. A good preconditioner can dramatically improve the convergence time of a solver, and a bad preconditioner can affect its performance and perhaps even its reliability. Thus, the preconditioner is an important part of the method selection. We will also address the problem of picking a suitable preconditioner as part of our prediction and recommendation system.

Next, we give a brief description of the different iterative methods and preconditioners that were used in our research and experiments.

2.1.1 Iterative Methods

In numerical mathematics, an iterative method is a type of solver that uses successive approximations to a solution (parting from an initial guess) to try to solve a problem, $Ax = b$ in our case. Direct methods, on the other hand, solve the problem in a single-step approach. The advantage and application area of iterative methods is for problems that have a large number of variables (perhaps order of 10^6 or more). In such cases, direct methods are computationally very expensive and require excessive amount of storage [Saad, 1995, Barrett et al., 1994].

The type of iterative methods used for the experiments in this dissertation are known as Krylov Subspace methods (ksp). The Krylov Subspace is the linear subspace spanned by the vectors $b, Ab, \ldots, A^{n-1}b$ where n is the number of rows in the matrix A. To solve a linear system, the ksps try to avoid matrix-matrix operations by using instead matrix-vector operations with the Krylov Subspace vectors [Saad, 1995]. The iterative methods we have used in our experiments are ksps and are:

- BCGS: the main idea behind the Bi-Conjugate Gradient Stabilized method is to eliminate the use of A^T that the BCG (Bi-conjugate gradient) method uses. This method is built upon CGS (Conjugate Gradient Squared method), and it also attempts to smooth its convergence [Sonneveld, 1989, van der Vorst, 1992].

- BICG: the Bi-Conjugate Gradient is a modification of the Lanczos algorithm [Lanczos, 1952] as a generalization for solving non-hermitian systems. This method is based on progressively solving a tridiagonal system (with no pivoting) at the same time as the tridiagonalization is performed. This method is sometimes inaccurate because of the lack of pivoting [Fletcher, 1975].

- CGNE: this is the Conjugate Gradient Method when applied on the Normal Equations. This is one of the simplest approaches for non-symmetric or indefinite systems [Wolfram, 2007].

- GMRES: the Generalized Minimal Residual Method approximates and uses the minimal residual of a vector in a Krylov subspace to approximate the solution. [Saad and Schultz, 1986]

- FGMRES: a modification of GMRES that allows variations in the preconditioner at each iteration [Saad, 1993]. With this method, iterative methods themselves can be used as preconditioners.

- TFQMR: the Transpose-Free Quasi-Minimum Residual (QMR) method is a variant of BCG based on a "quasi-minimization" of the residual norm of the current approximation. This method is a QMR implementation of the CGS method that avoids using the transpose of A [Freund and Nachtigal, 1991, Freund, 1993].

The implementations of these iterative methods (and the preconditioners mentioned in next section) are available from PETSc [Balay et al., 2004]. PETSc stands for Portable, Extensible Toolkit for Scientific computation, and it is a numerical toolkit that contains serial and parallel implementations for solving large-scale systems of equations originating from PDEs [Balay et al., 2007].

2.1.2 Preconditioners

Preconditioning is about transforming the system $Ax = b$ into another system, whose properties are more suitable for its solution using an iterative solver. A preconditioner is a matrix that is used to carry out such transformation. The motivation behind using a preconditioner is to try to improve the spectral properties of the coefficient matrix [Benzi, 2002]. A good preconditioner produces a matrix that is easier to solve than the original so that each iteration of a solver converges faster. It is also hoped that the application and construction of such a preconditioner is cheap, so that it minimizes the cost of each iteration. The choice of preconditioner depends on many factors, such as the type of matrix and the type of iterative method it is used with because these affect the behavior and the efficiency of preconditioners.

Suppose we have the system $Ax = b$, and M is a nonsingular matrix that approximates A. We have that the system

$$M^{-1}Ax = M^{-1}b$$

may be easier to solve and has the same solution as $Ax = b$. This is also called *left preconditioning*, where M is the preconditioner [Saad, 1995]. It is also possible to do *right preconditioning*:

$$AM^{-1}y = b, \;\; x = M^{-1}y.$$

The preconditioners considered for our research (also available from PETSC implementations [Balay et al., 2004] are:

- ASM and RASM: Additive Schwarz Method, derives a preconditioner by decomposing the original problem's domain into a number of subdomains (possibly overlapping), and the solution of each subdomain contributes to the global solution [Smith et al., 2004]. RASM is a modification of ASM, and the R stands for Restricted [Cai and Sarkis, 1999].

- Bjacobi: the Block Jacobi preconditioner comes from the Jacobi preconditioner which is the simplest technique for preconditioning, consisting only of the diagonal of a matrix. The blocked version comes from partitioning the variables into mutually disjoint sets, such that the preconditioner is a block-diagonal matrix [Barrett et al., 1994].

- BoomerAMG: a method that can be used as a solver or preconditioner, and it comes from a parallel implementation of algebraic multigrid [Benzi, 2002, Henson and Yang, 2002].

- ILU and SILU: the Incomplete LU factorization is also one of the simplest preconditioning methods. It consists of performing an incomplete factorization of the matrix A ($A = LU - R$) and computing both a lower triangular matrix L and an upper triangular matrix U, such that the residual error of the factorization R satisfies certain conditions [Saad, 1995]. SILU is a modification of ILU, and stands for Shifted ILU.

- PILUT: the Parallel ILU factorization is a parallel preconditioner that produces a nonsymetric preconditioner even from symmetric systems [Falgout and Yang, 2002].

2.2 The SALSA Framework

The Self-Adapting Large-scale Solver Architecture [SALSA, 2007] (SALSA) is a concept that was developed to integrate the continuously evolving numerical software with non-numerical, vanguard techniques, such as data mining and machine learning, to create a software framework that facilitates the use of numerical techniques in an application context for users that lack the expertise in numerics. This project's goal is to simplify the problem of algorithm selection through the implementation of a heuristic decision-making agent, based on experience and integration of new knowledge over time. Overall, the idea behind SALSA is to create a software architecture that enables non-experts from other science and engineering disciplines to use and take advantage of sophisticated numerical software. Figure 2.1 illustrates the system's general structure and flow of information.

The structure of this system is Layered:

1. Interface Layer: an application passes problem data into the system (rather than to a numerical library like accustomed). Additional information can also be provided as metadata.

2. Analysis Layer: here, the system extracts numerical characteristics of the input data. These can be derived numerically, heuristically or translated from metadata.

3. Algorithm Selection Layer: based on the extracted characteristics of the data, the system picks a suitable algorithm to solve the original problem, this constitutes a method *recommendation*. The decision making process in this layer is heuristic.

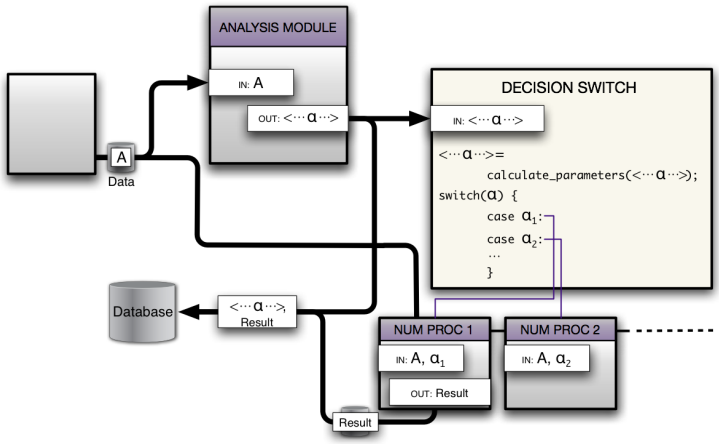


Figure 2.1: The SALSA framework. A science or engineering process produces data (numerical problems), which is then passed to the feature-extracting process implemented as a set of Analysis Modules. The output of this analysis is a vector of numerical features descriptive of the input data. In *learning* mode, these features are stored in the Database, and the system attempts to solve the original problem using each available solver implemented as Numerical Processes Modules. The outcome of each solver is then also stored in the Database. The Decision Switch is a learning process which then uses the information in the Database to model decision rules based on the "knowledge" stored in the database. In *recommendation* mode, the system extracts the features from a problem and, using the decision rules previously derived, predicts a suitable solver, which can then be applied to compute the solution to the input problem.

The adaptivity of the system resides in the way it "accumulates" experience. A database is used to store characteristics of problems, together with the performance measurements (timing, convergence history, error) resulting from the application of different solvers. This information is kept and used later to create (and tune) heuristics for the decision process.

The research presented in this dissertation originated from the idea of developing the Decision Switch module of the SALSA system (see Figure 2.1). SALSA embodies the type of learning process for which supervised machine learning is very appropriate. The existence of a database makes it natural to use data mining techniques to uncover patterns and possible relations among the data. The large amount of data (which is also expected to grow over time) makes the application of statistical techniques also suitable.

Chapter 3

Statistical and Machine Learning Context

Classification is an important task in machine learning that helps to address a variety of problems in different fields and types of applications. In particular, classification can be applied in the numerical context, as it helps us in the prediction and eventual recommendation of suitable methods to solve numerical problems (this concept is embodied by the SALSA framework described in Section 2.2).

From the various classification approaches, we focus on the use and study of statistical and machine learning techniques. These techniques can provide, in many cases, the means to solve these problems in a simple way within a reasonable range of accuracy. The supervised learning approach, in particular, provides a natural and flexible way to approach many classification problems as well as other probability related questions. Another technique is the Bayesian methodology, which is a simple yet powerful approach that can also be used as a building block and starting point for more complex methodologies.

In this chapter, we will describe how we can use this theory to identify patterns in numerical data and use these to classify various types of problems based on certain criteria, such as solving method suitability. We also describe other techniques we use in this dissertation, such as Principal Component Analysis, which can be used to process the input data and facilitate the identification of important data properties that directly affect the classification process. Techniques to process the output data that help to interpret the results from our experiments are also discussed in this chapter (e.g., statistical error analysis).

3.1 Introduction and Problem Statement

Classification is the process of assigning individual items to groups based on one or more characteristics of these items. The classification problem addressed in *machine learning* and is often used in data mining applications and in pattern recognition. It is also known as discrimination and can be seen as a problem of prediction and class assignment. Statistical methods are a common approach for classification because they provide a simple approach for addressing many problems and are relatively easy to implement. However, they should be used with caution since they work on many assumptions. The successful use of these techniques depends both on the size of the data set as well as the previous information regarding the set. Statistics provide methods for data analysis, such as sampling, stochastic modeling, predictions, experimental design, and exploratory data analysis.

Statisics-based methodologies have a wide range of applications in many different fields such as medicine, genetics, business, etc. The simplicity and natural interpretation of the way they output results makes them suitable for many applications despite of some of their disadvantages (e.g., in some cases Bayesian models tend to be naive or too simple for certain types of data sets).

Some of the many applications suitable for statistical methods are related to computer science problems, where the increasing size and complexity of data require new types of models and approaches [Klösgen and Zytkow, 2002]. However, they have not been extensively used in the numerical context, and their advantages have not been exploited for understanding and studying numerical data. Part of our study has been to explore the possibility of their applicability to numerical problems.

In this dissertation we focus on the use of the machine learning approach known as *supervised learning*. This technique involves learning patterns or functions from "experience." This experience is acquired by looking at examples of inputs for which the corresponding correct outputs is provided. From these observations we can then learn to predict outcomes based on certain conditions. A typical example of this approach is the Bayes method, where a class-specific probability model is designed for the random observations, given that a learning sample of representative data vectors is provided for each class (this approach will be discussed in detail in Section 3.3).

The inputs to a supervised learning process are usually a set of variables that are measured and preset (in our case these correspond to numerical properties of matrices). The outputs are variables that are "dependent" on the inputs (these would be the class memberships of different input matrices) [Hastie et al., 2001]. Usually the supervised learning approach consists of two stages:

- Learning: involves the incorporation of the experience-based knowledge into the learning system, e.g., a set of example problems with their corresponding correct membership to a class.

- Testing: this part concerns the evaluation of how accurately a trained system can "predict" the correct answer.

As mentioned in Chapter 1, the type of numerical problem we are concerned with is the solution of sparse linear systems using iterative methods. Using statistical and machine learning techniques, we want to determine if it is possible to choose a suitable iterative method and ultimately build a recommendation system that will give a good prediction of such a method to solve a particular instance of a numerical problem. The problem of choosing a suitable method for solving a matrix has two parts: reliability and performance (see Figure 3.1). For the *Reliability* problem, we want to find whether an iterative method can solve (converge to a solution) or not solve (diverge) certain matrices. In the *Performance* problem, we are more interested in determining, among various iterative methods, which one is the most efficient for solving particular problems.

Based on these criteria for picking a suitable method, we can make predictions to build a recommendation system. Making a recommendation depends on the problem and on the needs of particular users. Consider the case of a user who needs an efficient and robust method for solving a large sparse matrix resulting from some engineering or scientific process modeled using PDEs. Such a user, not being an expert in the field of linear algebra, does not have the mathematical expertise to set specific parameters and options for different methods, or may not even know about preconditioning techniques. On the other hand, consider a user who may be interested only in the

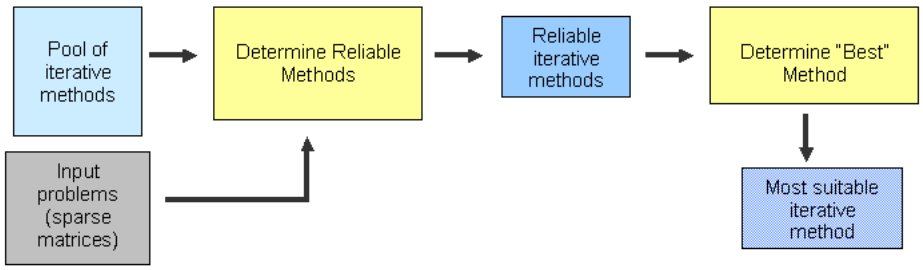


Figure 3.1: The process of selecting the most suitable method. The process of selecting the most suitable method. This consists of two stages: first, find which methods are reliable, and second, find which of those *reliable* methods is the best for solving a particular problem.

study of the behavior of preconditioners on a particular method or vice versa. To address different types of needs, we have considered the following types of recommendations:

- Preconditioner & iterative method: this is probably the most generic type of recommendation but also the most difficult. Given a matrix, choose a reliable and efficient combination of an iterative method and a preconditioner. The difficulty of this problem arises from the fact that it is necessary to compare every possible pair (preconditioner, iterative method) with all the others to decide which one is the best. Also, it is complicated to establish what is the best choice of preconditioner, depending on how its relationship to the iterative method is evaluated.

- Iterative method: for this particular option, consider the case when the matrix provided is already preconditioned, and the only interest is to determine which is the best iterative method to match with the preconditioner. Another case is for the instance when a default or constant preconditioner is used, like in many numerical libraries and applications. Such a recommendation would be interesting for a user with more expertise in the numerics field, or if the user is satisfied with the default preconditioner assigned by the software.

- Preconditioner only: the selection of an appropriate preconditioner can greatly improve the performance of an iterative method. Many software packages and numerical libraries provide a default preconditioner with the application of iterative solvers. However, a better choice may result in a more effective solving processes.

These various options give origin to various strategies for classification and recommendation, which will be discussed in Chapters 5 and 6.

3.2 Statistical Classification Framework

Statistical methods are widely used in discrimination and pattern recognition problems because they provide a simple framework for classification. The statistical approach for classification generally consists of the following tasks [Duda et al., 2000] methods:

- Information gathering: form a data set with examples of the problem of interest. Group these examples (observations) into the different desired classes; no observation should appear in two classes at the same time. Choose variables and measurements that can be used to distinguish between the classes, some data preprocessing may be required at this point. This step may also involve the creation of *training* and *test* data sets, depending on the approach.

- Feature extraction:determine features (or characteristics) of the observations that are representative of the classes and also adequate for the classification method.

- Creation of a methodology for classification: construct a classifier using the information available from the existing classes, e.g., use a training set to choose or design a classifier, then assess and tune its behavior using validation examples.

- Evaluation of the classification results: measure the classifier's performance, e.g., determine class membership for examples provided and check if the answers are correct.

- Generalization: application of the classifier to new problems to test how well it can solve different examples.

In a classification problem, the observations in a data set can be described by a vector x of k features (components). Each of these observations originally belong to one of the classes. This membership is unknown and the classification process is used to determine which class they belong to, which is "guessed" based on the values of x. Each class has certain characteristics or data distribution; by using a function we can predict or estimate of the true but unknown class of an observation using its features x [Klösgen and Zytkow, 2002]. This is feasible when each class is described by a particular data distribution, and the observations have feature values that are distinctive of the class. Unfortunately, in practice, target classes are usually not completely independent and separated from each other; for this reason their data distributions will very likely overlap, leading to misclassifications [Ye, 2003]. One of the objectives when building classifiers is to minimize the overlapping areas; this problem can be easily illustrated using the Bayesian approach, which will be addressed with more detail in Section 3.3.

Classification methods commonly use *probability distribution* functions to discriminate between different groups of data. A probability distribution function is defined (in one dimension) as:

$$F_X(x) = P(X \leq x) \tag{3.1}$$

where $x \in \Re$, and X is a one-dimensional, random variable defined on a sample space S (X is a function whose domain is S and whose range is in the real numbers). For each x there is a subset of S such that $\{s|X(s) \leq x\}$ which is known as an *event*. $P(X \leq x)$ is the probability of the event happening [Apostol and Holbrow, 1963]. In classification, an example of such an event would be that an observation belongs to a particular class.

A *probability density function* is a continuous representation of a probability distribution function, obtained by integrating a function f over an interval. We say that a probability distribution has a density function f, when f is non-negative, real, and satisfies:

$$\int f(x)dx = 1 \tag{3.2}$$

For instance, consider the function in Equation 3.1, we would have the density function as follows:

$$F_X(x) = P(X \leq x) = \int_x^{-\infty} f(t)dt. \tag{3.3}$$

In classification problems, density functions can be used to describe (or model) each of the involved classes; there is one density function per class. This function can be chosen so that it represents or captures certain aspects of the classes such that it will be possible to differentiate between them.

There are two approaches for classification and learning problems based on how the density functions are computed. The density function is what determines how the rules for making decisions will be built. These approaches are known as *parametric* and *non-parametric*; we use both in this research:

- Parametric Classifiers: in this approach we estimate the parameters for a presumed probability distribution function such as the Normal (Gaussian) distribution, then we use this function as a basis for making decisions [Dunham, 2002]. The problem with this approach is that sometimes it is too naive because the data originating from many problems does not follow any particular distribution. Furthermore a few parameters cannot fully describe more complicated problems. In order to choose an appropriate distribution it is necessary to have a great deal of knowledge about the data or the originating problem [Martínez and Martinez, 2002].

- Non-parametric Classifiers: the decision function is built specifically from the collected data instead of assuming the existence of a specific probability distribution. This approach can be better for data mining applications since the problem is data-driven and there are no explicit equations to determine the models; however they require lots of input data to perform better. Examples of these approaches are clustering techniques and decision trees [Dunham, 2002] [Duda et al., 2000].

Another important technique used in this work is the Bayes theorem (which is the heart of the *Bayesian methodology*). This is a statistical concept that can be used as a basis for pattern recognition, classification and problem discrimination [Dunham, 2002]. It can also be used as a starting point for implementing more complex techniques, e.g. data mining for knowledge discovery in databases. Before getting into more detail regarding parametric and non-parametric techniques, we will address the Bayesian approach in the next section. This approach was used as a starting point in the construction and evaluation of classifiers using probability density functions. Parametric and non-parametric techniques will be covered later in Sections 3.4 and 3.5, including a description of how the Bayesian approach can be used in both of them.

3.3 The Bayesian Approach

The Bayesian approach in statistical data analysis is based on a probability model dependent on observed or given information. The Bayesian approach allows us to derive *a-posteriori* knowledge expressed as *posterior probabilities*, based on experience or *a-priori* knowledge given by *prior probabilities*. This model finds the conditional probability distributions for the possible classifications of the data of interest; in other words, it estimates the likelihood of some property given some input data as evidence. The Bayes theorem is used in particular to compute the conditional posterior

probability distributions, with the aid of probability density estimation methods [Ye, 2003]. It puts the problem in terms of probabilities, all of which must be known in order to make a classification decision.

The simplest form of the Bayes Theorem, states the following:

$$P(A|B) = \frac{P(A)P(B|A)}{P(B)} \tag{3.4}$$

where A and B are two events [Douglas and Montgomery, 1999] and P is the probability of an event happening. $P(A|B)$ denotes the *posterior conditional probability* of event A occurring, given that event B was observed. This follows from the definition of *joint probability*, which is the probability of two events occurring together and is also the *prior* probability:

$$P(A \cap B) = P(B|A)P(A) = P(A|B)P(B).$$

When used for classification, the events translate as A and B become "being in a class" (ω) and "problem characterized by properties x" respectively:

$$P(\omega|x) \text{ is the probability of class } \omega \text{ given that the problem has properties } x.$$

In other words, we can determine the probability of a problem x belonging to a class ω, based on the experience of observed problems (with properties similar or equal to x) also belonging to class ω. Assuming that the classes are given, and that by previous experiments we have determined the proper classification of a number of input problems, Equation 3.4 can be written as:

$$P(\omega|x) = \frac{P(\omega)P(x|\omega)}{P(x)}.$$

We can calculate the *posterior probability* $P(\omega|x)$ if we can determine

- $P(\omega)$: the *prior probability* of the class. Usually we will have no analytic knowledge of the prior probability of a class, so we take the fraction of the feature vectors in this class.
- $P(x)$: the *marginal probability* acts as a normalizing constant. We express it as

  $$P(x) = \sum_{\text{classes}\omega} P(x|\omega)P(\omega)$$

  in terms of the final quantity:
- $P(x|\omega)$: the *class-conditional probability* or conditional probability. Computing this requires us to know the probability distribution of the features in each given class.

With a posterior probability given, we arrive at the *Bayesian decision rule*:

> Assign a feature vector x to class j if $P(\omega_j|x) \geq P(\omega_i|x)$ for all i.

For multi-dimensional problems, a rudimentary solution is to adopt *naive Bayesian analysis*, which makes the simplifying assumption that all features that describe the input problems are independent, so

$$P(\bar{x}|\omega) = P(x_1|\omega) \cdots P(x_n|\omega). \tag{3.5}$$

The biggest problem with this naive assumption is that in practice, problems with especially high dimensions usually possess more than two properties that are *correlated*, i.e., when two features are inversely or directly porportional to each other. To handle this kind of situation, we can apply certain preprocessing to the data, such as Principal Component Analysis, which transforms the original set of features in to a new set of linearly independent features (this will be covered in detail in Section 3.6).

In Bayesian classification, the learning process estimates the probabilities instead of finding an explicit rule. The advantage of this approach is that the classifier will reach the minimum error when the dataset is large, assuming that the data is representative of the true distribution. If the person who designs the classifier has some prior knowledge about the data that would lead to the belief that some model is a particularly good approximation to the probability distribution, the classifier will have a very good performance. The problem is that the model used in the classification might not be the best estimation to the probability distribution. However, unrealistic models that make naive assumptions are not necessarily bad and very often will lead into relatively good performance compared to other techniques. The Bayesian classification gives the most likely classification, which might not be the best solution, since the cost of the errors could be different (for example, cancer diagnosis or tornado warning [Troyanskaya et al., 2003, Klösgen and Zytkow, 2002]).

3.4 Parametric Approach

Techniques that follow a parametric approach assume that the input data has a particular distribution, such as normal, exponential, Poisson, etc. The probability density function is then defined in terms of the distribution and its particular parameters (hence the name "parametric"). This approach can simplify the computation of probability estimates and it can be very useful and efficient to model problems when we know (from experience) the data exhibits a particular distribution. The most simple and most common distribution used is the Normal or Gaussian, which is described by the mean (or expected value) μ and standard deviation σ parameters:

$$f(x, \mu, \sigma^2) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left\{\frac{(x-\mu)^2}{2\sigma^2}\right\}$$

where σ^2 is the variance. The multi-dimensional version of this (the multivariate normal distribution) is:

$$f(\bar{x}, \bar{\mu}, \Sigma) = \frac{1}{(2\pi)^{d/2}|\Sigma|^{1/2}} \exp\left\{-\frac{1}{2}(\bar{x}-\bar{\mu})^t \Sigma^{-1}(\bar{x}-\bar{\mu})\right\}$$

where $\bar{\mu}$ is a vector of means and Σ is a matrix of covariance.

The parametric method using a single Gaussian distribution is a simple approach that can be used to explain many of the concepts and procedures of statistical-based classification and learning. However, for many problems, like the one addressed in this dissertation (algorithm determination) its practical application is very limited, mainly because data is not known to have any specific probability distribution. The assumption that this data follows a Gaussian distribution can be more harmful than beneficial for determining prior and posterior probabilities. Parameters like a mean and standard deviation can only give very general information about the sample.

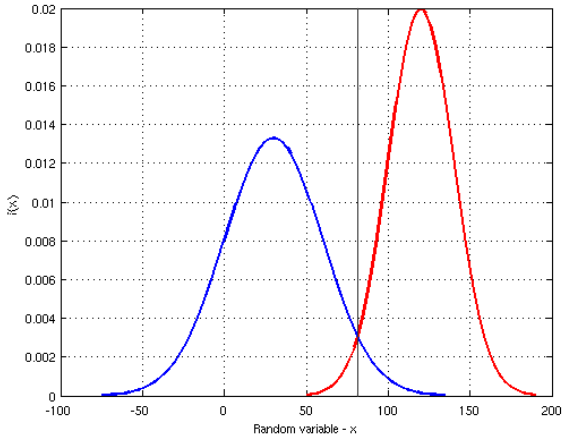


Figure 3.2: Normal distributions corresponding to two classes for a uni-variate problem. The single feature is represented along the x axis as a random variable. The distribution for C_1 appears in blue, and the distribution for C_2 in red. The grey marker line shows the point on the x-axis where decision for classification changes.

Parametric Approach using Bayesian Methodology: an Example

The application of the Bayesian methodology is straight forward for classification problems using a parametric approach. Consider the simplest case: a set of input problems (sparse matrices) that can be characterized with one feature (condition number for instance), and each problem can belong to either of two classes C_1 or C_2. Assume also that these problems have a normal distribution, so we have two normal distributions that describe each class (see Figure 3.2).

We represent the distributions for C_1 and C_2 with their mean and variance parameters respectively: μ_1, σ_1^2 and μ_2, σ_2^2. Now suppose we have an input problem x; to determine which class it belongs to, we compute the posterior probabilities for both classes:

$$P(C_1|x) = P(\mu_1, \sigma_1^2|x) = \frac{P(\mu_1, \sigma_1^2)P(x|\mu_1, \sigma_1^2)}{P(x)}$$
$$P(C_2|x) = P(\mu_2, \sigma_2^2|x) = \frac{P(\mu_2, \sigma_2^2)P(x|\mu_2, \sigma_2^2)}{P(x)}$$

Then we compare the posterior probabilities: if $P(C_1|x) > P(C_2|x)$ then we say that x belongs to C_1, and belongs to C_2 otherwise. Also, we can easily visualize this in Figure 3.2; any $x \leq 80$ (marker line) would be assigned to C_1, because the corresponding posterior probability for C_1 is grater than that of C_2 in this region. When $x > 80$, the class assigned would be C_2.

3.5 Non-Parametric Approach

Non-parametric techniques for probability density estimation are a form of supervised learning (each of the sample data are labeled with a class membership tag). The difference with the parametric approach is that we do not make assumptions about what shape or function models the probability density.

For the Bayesian approach, there are three basic techniques: histograms, finite mixtures, and kernel density estimation. In our experiments, we concentrate on the use of kernel density estimation, which can be considered a special case of finite mixtures. We will be using the simplest case of kernel density estimation (multivariate case) which is called *product kernel*.

There are other non-parametric techniques that do not use the Bayesian approach. One of the most important and effective techniques is the use of *decision trees* for classification [Dunham, 2002]. We will focus on the analysis of non-parametric methodologies using the Bayesian approach and Decision Trees.

3.5.1 Finite Mixtures

The finite mixture term comes from the statistical concept of probability mixture model, which is a convex combination of several probability distributions, meaning that the sum of the components is 1. It can also be seen as a type of model where several independent variables act as fractions of a total [Titterington et al., 1985]. A *finite mixture* is a specific type of mixture model where the number of component distributions is finite; a common example of this is a mixture of Gaussian distributions.

The main idea is that the density function f can be modeled as the sum of c weighted densities, assuming that $c << n$, where n is the number of sample points or events (FM stands for finite mixture):

$$f_{FM}(x) = \sum_{i=1}^{c} w_i G(x) \tag{3.6}$$

where G is any continuous or discrete probability function for modeling the component densities of the mixture (which must be non-negative and sum to 1), and w_i is the weight or mixing coefficient for each component of the mix. For example, if we use normal density functions for G with mean $\hat{\mu}$ and variance $\hat{\sigma^2}$, we can re-write Equation 3.6 as:

$$f_{FM}(x) = \sum_{i=1}^{c} w_i \phi(x|\mu_i, \hat{\sigma_i^2}) \tag{3.7}$$

where we have $c - 1$ independent mixing coefficients, c means and c variances. We can easily extend this model to the multivariate case, as the weighted sum of multivariate component densities:

$$f(\bar{x}) = \sum_{i=1}^{c} w_i G(\bar{x}|\theta_i) \tag{3.8}$$

where $\bar{x}$ is a data point vector in a d-dimensional space (each dimension corresponds to a feature). In this case we would have c d-dimensional vectors of means μ and c covariance matrices σ^2 of size $d \times d$. Similar to the univariate case, Equation 3.8 can be re-written as:

$$f_{FM}(\bar{x}) = \sum_{i=1}^{c} w_i \phi(\bar{x}|\mu_i, \sigma_i^2) \tag{3.9}$$

3.5.2 Kernel Density Estimators

A *kernel density estimator* is a type of *finite mixture* method that requires the use of a "smoothing parameter" h. This parameter determines how accurate the results of the function are. For example, suppose we have a histogram for which the bars define the probability density function, i.e., sum of the area of the bins = 1. The smoothing parameter would be the bin width (or *window*) in the histogram. As h approaches 0 the more exact the solution will be; the choice of h can give very different estimates of the probability density. This parameter can dramatically influence the performance of a classifier but can be difficult to tune. In the *finite mixture* method we do not need this parameter, however it is necessary to determine the number of components c in the mixture, which can also be very hard to tune depending on the problem [Bosq, 1998, Martínez and Martinez, 2002].

To understand the idea behind this approach, consider the example of a probability density function f modeled by histograms. As the number of histograms m approaches infinity, the sum of the areas of the histogram's bins approximates the real value of the area under the curve defined by f . An example of this technique is the averaged shifted histogram, where the idea is to create many histograms of the same width h with different origins and then average the histograms together; this creates a piecewise constant function, which is the average of a set of piecewise constant functions [Bosq, 1998, Martínez and Martinez, 2002].

A kernel estimator for a univariate case is as follows:

$$f(x) = \frac{1}{n} \sum_{i=1}^{n} K_h(x - X_i) \tag{3.10}$$

where the parameter h is the smoothing factor or window width, and K is called the *kernel* function that must satisfy, as mentioned in the previous section, the following conditions:

$$\int K(x)dx = 1 \text{ where } K \text{ is non-negative and real} \tag{3.11}$$

Kernel functions are usually symmetric probability functions, such as the normal distribution, which is the one we used in our experiments. The estimated probability function can be computed by placing a *weighted kernel function* at each data point and then taking the average of them in a similar way as the histogram approach.

To extend to the multivariate case we use the *product kernel* approach, which is the simplest case, as follows. Consider each that sample point is now a d-dimensional vector denoted as $\bar{x}$, and there are n observations in the sample. The density estimator is given by:

$$f_{Kernel}(\bar{x}) = \frac{1}{nh_1 \dots h_d} \sum_{i=1}^{n} \{ \prod_{j=1}^{d} K(\frac{\bar{x}_j - X_{ij}}{h_j}) \} \tag{3.12}$$

where X_{ij} corresponds to the j-th component of the i-th observation.

The smoothing parameter h is computed for each dimension of the d-dimensional data vectors X using Scott's rule [Scott, 1992]:

$$t = \frac{4}{n \times (d+2)}^{\frac{1}{d+4}} \sigma_j,\ j = 1, ..., d \tag{3.13}$$

and h is computed as:

$$h_i = \Sigma_{ii} \times t, i = 1 \dots d \tag{3.14}$$

where $\Sigma = cov(X)$, and X is a $n \times d$ matrix (each row of this matrix corresponds to an observation from the input data set).

This rule is also called *normal reference rule* for multivariate product kernels using the normal function.

Non-Parametric Approach using Bayesian Methodology

Using the Bayesian approach in this type of non-parametric techniques, we do not train a classifier to learn the parameters of a Gaussian distribution like in the parametric example. Instead, we estimate prior probabilites by computing the estimate for the density function using a Kernel Product for each class. Then, similar to the parametric approach, we can compute posterior probabilities based on these estimates and assign the data point to whichever class gives the highest probability.

3.5.3 Classification using Decision Trees

Another approach to non-parametric classification is decision trees. A decision tree is a recursive structure that is constructed to model the classification process. The two basic steps in this technique are: building the tree based on information gathered from experience and applying the tree to the data [Dunham, 2002], which are consistent with the supervised learning approach. Decision trees are particularly useful for problems that have many variables or features involved. For instance, for the type of numerical problems on which this research is centered, we have extracted around 50 different numerical properties (most of these are continuous, some are discrete, and a few are binary).

The construction of the decision tree structure is called *induction* step, and it is done using "training" data. The idea is to classify the members of this set at the leaf nodes, such that the elements in each resulting class have the same associated (actual) class. The classes are constructed by filtering each observation in the dataset through the tree structure.

The decision tree is a tree-structured collection of nodes with the following properties [Quinlan, 1990]:

- There is a single Root node N_0 that has no parent nodes.
- Each internal node (also called non-terminal) corresponds to a feature and may have one or more children nodes.
- Every internal node N_k, $k > 0$, has exactly one parent node.
- A terminal or leaf node has no children and is associated with a class C_j. This association is given by the class C_j-membership of a set (perhaps empty) of input problems.

This tree structure embodies a set of simple rules, e.g., for the type of numerical problems addressed in this dissertation a rule could be "if the condition number κ of a matrix A is greater than a threshold, assign A to class C_1" (see Figure 3.3). These rules do not require any assumptions about the distribution of the measurements of each variable in each group. The variables or features can be categorical, discrete numeric, or continuous. Every non-terminal node has a rule of this kind associated with the feature corresponding to that node. This rule is *queried* every time the node is visited during the evaluation of a problem, by comparing the actual value of the feature to the current threshold in the node.

The features in the internal nodes are called *splitting atributes* or *predictors* [Breiman et al., 1983, Dunham, 2002] and are the places in the tree where the decisions take place. The rules associated to each of the nodes are called *splitting predicates* and are basically a comparison with respect to the aforementioned threshold. There are many different algorithms for constructing decision trees. They differ in the criteria by which they choose the splitting attributes and the splitting predicates and in the order they are assigned on the tree.

The tree structure is created recursively until all the elements of the training data are classified correctly. Some tree algorithms may use a stopping criteria [Dunham, 2002] to avoid the creation of very large trees or overfitting problems.

In our experiments we have focused on using the approach known as CART (Classification and Regression Trees) developed by [Breiman et al., 1983]. This approach generates a binary decision tree, and uses an entropy model similar to the ID3 (Iterative Dichotomiser 3 algorithm) approach [Quinlan, 1986] to select the best splitting attributes and predicates. This approach tries to minimize the number of comparisons by ordering and grouping features that contribute the most information.

An important characteristic of the decision tree methodology is the ability to handle overfitting. This is an aspect that was not covered by the other techniques we have described. The process known as *pruning* is a validation step that allows us to eliminate possible overtraining, in particular for small datasets. This consists of eliminating certain nodes and branches to improve the generalization power of the tree structure and yield even better classification results. There are many different approaches for pruning. In our experiments, we do not carry out any pruning, mainly because the accuracy of classification for decision trees is very high even in the generalization tests.

3.5.4 Support Vector Machines

Support Vector Machines (SVM) is a supervised learning method for classification. It maps the input vector into a multi-dimensional space and builds an optimal hyperplane to separate the data into two categories. The separating hyper-plane is constructed with respect to the feature space in such a way that it maximizes the distance between the plane and the nearest data points from both classes (this is known as margin) [Vapnik, 1998]. The margin of the linear classifier is the width that a boundary around the hyperplane may be increased before hitting any datapoint. The *support vectors* are those points that the margin pushes against. When there are points of one class cross over the hyper-plane and intersect with the other class (these are known as noise), the technique seeks to minimize the distance of these "error" points to their correct location (i.e., on the other side of the dividing hyper-plane).

This method is a linear classifier because it uses a plane to separate two classes; however, it can be used for non-linear classification problem by transforming the original feature space [Aizerman and Braverman, 1964]. Since this methodology is used for binary (2-class) problems, its applicability in our research is only for the Reliability problem.

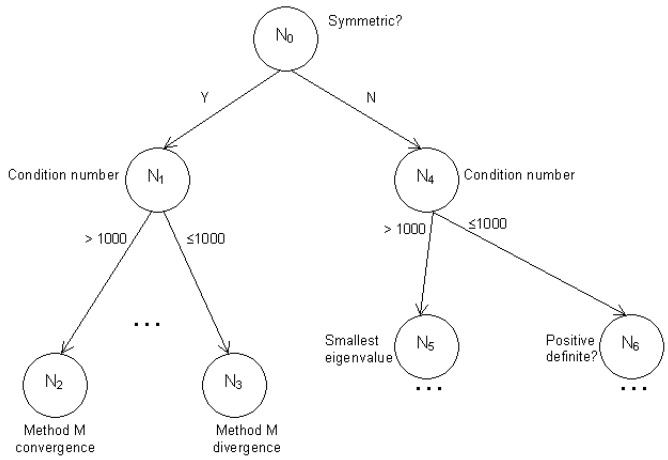


Figure 3.3: Example of a decision tree structure. This could be used to classify the reliability of an iterative solver M, based on a few numerical properties of a matrix: symmetry, condition number, eigenvalues, etc.

The core of the methodology is a kernel function that transforms a non-linear feature space into a linear one to find optimal distances between observations, groupings and boundaries between classes. Some examples of SVM kernel functions are: linear, polynomial, radial basis function, sigmoid (a description of these can be found in [Burges, 1998,Hsu et al., 2003]).

SVM is a relatively new technique for data classification; it is considered an optimization problem which makes it computationally very expensive. Large datasets usually require long and extensive computations to generate results, especially with higher-dimension feature spaces. For this reason we had to limit our study to the use of the bivariate version of the technique for the Reliability problem with a very reduced set of features. This technique was very useful in the early stages of our research since it helped us to understand the effect of certain features in the classification process for the Reliability problem; however its cost and non-applicability to multi-class problems makes it a hard choice as classifier technique.

3.5.5 K-means: an Unsupervised Learning Technique

Although we have centered our research on the application of supervised learning techniques, K-means is a basic approach that can be very useful for initial exploration of the data sample. This is a clustering technique that groups data based on the distance between each of the observations in the sample, minimizing the sum of the variance in each cluster. It is unsupervised learning because we do not know prior to the clustering to which class each observation belongs.

The K-means un-supervised learning algorithm aims to classify data into k clusters. It measures the distance between each data and the corresponding cluster centroid and tries to minimize the sum of the square error function

$$V = \sum_{i=1}^{k} \sum_{x_j \in S_i} |x_j - \mu_i|^2,$$

where there are k clusters $S_i, i = 1, 2, \ldots, k$ and μ_i is the centroid of the ith cluster (geometric mean of the observations). An initial parameter is the the number of classes and the initial center, then the algorithm clusters each of the observations in the input data by using an iterative method until a solution is found.

Although the experimental results for classification using this method have not been very accurate, K-means is useful because it gives a "geometrical" notion of the data and features; e.g., if the data is well clustered in space we would expect other classifiers to differentiate between the classes more easily.

The accuracy evaluation of this method is similar to the other approaches we have discussed. Once the clusters are formed, a test set can be used to see how well it can classify new data based on the computed centroids and number of clusters.

3.6 Principal Component Analysis

Principal Component Analysis (PCA) is a linear projection method based on *singular value decomposition* (SVD). PCA is a linear dimensionality reduction method that identifies the orthogonal directions of maximum variance in the original data and projects the data into a lower dimension space formed by a subset of the highest variance components [Jackson, 2003]. PCA also facilitates data clustering and its analysis by first dimensionality reduction via projections onto principal component axes. The main idea of using PCA is to represent the data with fewer number of dimensions while retaining the main properties of the original data. This methodology is an unsupervised linear projection method, which means it is not required to use the output information as feedback.

The feature space of the set of input problems $\mathcal{A} \subset \mathbb{A}$ (set of sparse matrices in our case) forms a linear space, where each of the features corresponds to a dimension in the original space. By using SVD, it is possible to use a transformed set of orthogonal basis vectors that captures approximately the same degree of information but using less dimensions in the transformed subspace. This transformation linearly combines the original original features into a new transformed set of variables, which are known as "principal components" [Jackson, 2003]. In this way, PCA provides a good way to preprocess the input data of the problem. The transformation process inherently provides information that facilitates the identification of relevant properties of the data and possible correlations between them. Section 3.6.1 describes how this can be achieved by the means of singular value decomposition of the original data matrix.

The motivation for using PCA as part of our analysis comes from two main ideas behind this methodology. First, it allows us to represent multidimensional data with fewer numbers of variables (dimensions) while retaining the main properties of the original data, due to the high degree of correlation among the original variables. Thich is very useful for the implementation of certain types of statistical classifiers. Second, because the elements originated from each step of the PCA algorithm provide a useful and natural way to understand associated properties and correlations in the data, it is particularly useful for characterizing and understanding those variables (features) which are critical for our classification goals. The data can be preprocessed using PCA before carrying

out classification. Some dimensionality reduction can also be achieved by *feature extraction* which implies the elimination of redundant features.

Consider a space of d dimensions (features). With feature extraction, we try to find a new set of $k < d$ dimensions that captures the most variation from the data set and we discard the remaining $(d-k)$ dimensions. The idea is to find a mapping from the original space with d dimensions to a new space having k dimensions, without sacrificing real data. In the new reduced space, the variance is maximized and the difference between sample points becomes more apparent, hence the process of discrimination between classes becomes easier.

Next, we will go through the description of the essential PCA elements and their significance in our experiment. With this method we can get the truly independent variation embedded in the d-dimension space by removing correlated data, which facilitates clusterability and discrimination of the data. In this study, each sparse matrix constitutes an observation and its associated features or properties are the original variables (or dimensions).

3.6.1 Singular Value Decomposition

Singular Value Decomposition (SVD) is the workhorse for implementing PCA. SVD is a decomposition of a matrix into three matrices, each with distinct properties. It factors a matrix X into

$$X = USV^T$$

where U and V are unitary matrices known as the left and right singular vectors respectively, two matrices with mutually orthonormal columns, and S is a diagonal matrix [**?**] containing the singular values of X, which are all positive and arranged in descending order.

In our experiments, the matrix X is composed of vectors with the properties (features) of the input problems. Suppose there are m observations from a particular data set or experiment, and from each of these observations we have n associated features (variables). This data can be arranged into a data matrix $X_{m\times n}$, i.e. each row represents an observation and each column represents a feature variable:

$$X = \begin{bmatrix} x_{11} & \cdots & x_{1n} \\ \vdots & \ddots & \vdots \\ x_{m1} & \cdots & x_{mn} \end{bmatrix}$$

Figure 3.4 depicts the SVD factorization. X can be decomposed into the product of three matrices U, S, and V as follows:

$$X_{m\times n} = U_{m\times r}\, S_{r\times r}\, V^T_{r\times n}, \tag{3.15}$$

where r is the rank of X. This factorization in particular is a *reduced* singular value decomposition, and $m \geq n$ is assumed [Trefethen and Bau, 1997]. While $m >> n$ is desirable, is not a necessary condition (if there are more observations than variables, the method will be more robust).

Each of the matrices resulting from the SVD factorization has particular properties that are important for the analysis and understanding of the data:

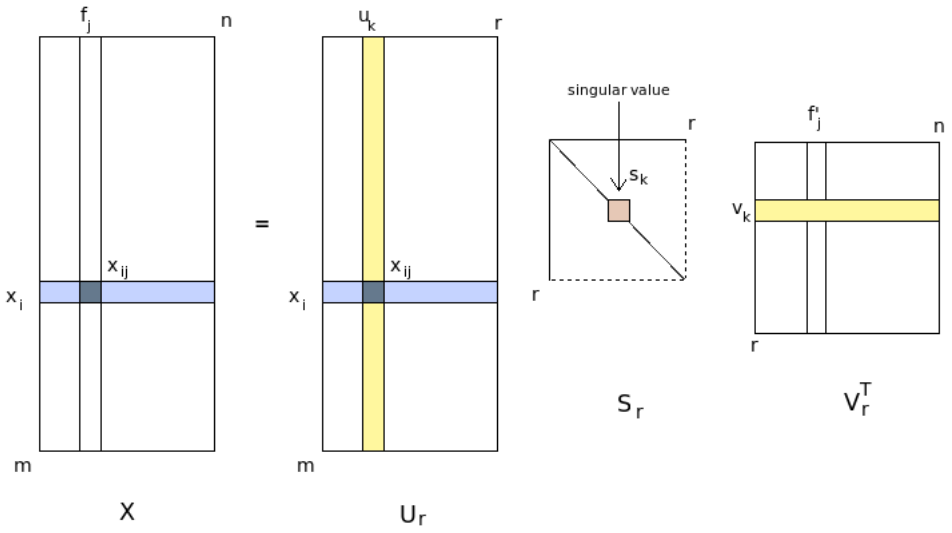


Figure 3.4: Depiction of the Singular Value Decomposition (reduced version).

- U: known also as the *normalized Scores Matrix*. Each row of this matrix corresponds to a normalized observation with respect to the basis vectors of V. Its columns u_k, also referred to as the *left singular vectors* of X, form an orthonormal basis for the scores space.
- S: or the *Singular Value Matrix*. This is a diagonal matrix with all its elements positive and by convention [Wall et al., 2001] are arranged in descending order. These elements are the singular values of X denoted as $\sigma_i \ : \ i = 1 \dots n$.
- V: also known as the *Loadings Matrix*. Its columns are called *right singular vectors* of X or *principal components* (PCs) and form an orthonormal basis for the feature space. Each principal component is a linear combination of the original features. The set of $\{v_i\}_{i=1}^{r}$ vectors span the reduced r-dimensional principal component subspace.

The product of U and S yield another matrix T known as the *Regular Scores*

$$T = X \times V = U \times S \tag{3.16}$$

Each row of T contains the so called *PC-scores* of a sample, and they correspond to the projection of the sample onto each of the PCs. The columns of T are orthogonal to each other and their lengths are the singular values squared:

$$T'T = SU^T US = S^2$$

The mathematical interpretation of the matrices in a SVD factorization pieces provide relevant information that can be used for PCA. Some of the most significant information is about the characteristics of the variables or features interpretation. One of the most important consequences of SVD is that it allows for dimensionality reduction in PCA. To understand this, it is necessary to first emphasize the following series of facts that are associated with the decomposition:

- Let r be effective $rank(X)$, and n be the number of columns of X (number of features), then $n \geq r$.
- Let $i \in [1, n]$, and let σ_i represent a singular value of X. The condition number of X is

$$\kappa \triangleq \frac{\sigma_1}{\sigma_r} \geq 1$$

  and the bigger κ is, the more *near singular* the matrix X is at its *apparent* or *nominal* rank.
- The original SVD factorization of X can be expressed as follows:

$$X_{m \times n} = U_{m \times r} \, S_{r \times r} \, V^T_{n \times r} \tag{3.17}$$

  where
  - $U_{m \times r}$ or U_r, and its vectors span the r-dimension column-space of X.
  - $S_{r \times r}$ or S_r

$$S_r = \begin{bmatrix} \sigma_1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \sigma_r \end{bmatrix} \tag{3.18}$$

  - $V^T_{n \times r}$ or V^T_r, the vectors of V_r span the r-dimension row-space of X.
- By viewing the decomposition this way, we can break X into r outer products (i.e., $X = \sum_{j=1}^{r} u_j \sigma_j v'_j$), where the contribution of the ith outer product to the overall variance represented by the original data set is given by:

$$\frac{\sigma_i^2}{\sigma_1^2 + \sigma_2^2 + \ldots + \sigma_r^2} \tag{3.19}$$

Equation 3.16 states a very important fact about SVD: each row of XV is the representation of the rows of X with respect to the (orthogonal) basis vectors of V, and V is mapped by X into U but scaled by S. In other words, the ith vector of V is mapped by X into the ith vector of U scaled by the ith singular value of X σ_i. This is a one-to-one mapping between $\{v_i\}_{i=1}^r$ and $\{u_i\}_{i=1}^r$

$$\begin{aligned} v_1 &\xrightarrow{X} \sigma_1 u_1 \\ v_2 &\xrightarrow{X} \sigma_2 u_2 \\ &\cdots \\ v_r &\xrightarrow{X} \sigma_r u_r \end{aligned}$$

which also defines the relationship between the matrices U and V: the ijth element of U is the projection of the ith row of X (ith observation) onto the jth column of V (jth principal component) normalized by the jth singular value.

Furthermore, the singular values for X appear in S in decreasing order (by convention) making σ_1 the largest singular value, and σ_n the smallest singular value, which makes the first vectors

of U_r "heavier" than the last ones. After a certain index (e.g., r), the singular values become so small compared to σ_1 that the ratio $\frac{\sigma_k}{\sigma_1}$ is close to 0, so it is admissible to discard the $(n-r)$ last singular values (e.g., making them 0) in reconstructing the data matrix with a reduced effective dimensionality. In this way v_1 will be scaled the most by $X\sigma_1$ in being mapped to u_1, while v_r will be scaled by σ_r (the "smallest" singular value considered by the mapping). The last $n-r$ vectors from V can then be discarded given that they are multiplied by zeroed singular values. Since the vectors in V are the principal components forming an orthonormal basis in the reduced transformed space (each of them being a "new" variable), and the last $(n-r)$ V vectors are eliminated, we have reduced the original dimensionality of n to $(n-r)$ dimensions or variables by projecting the original data onto the reduced r-dimensional principal component subspace.

In practice, it depends entirely on the application or the experiment, the number of singular values that will be considered to represent the transformed data matrix. Depending on the behavior of the data, the desired accuracy, and other factors, one may decide to include more or less dimension.

3.6.2 Scree Plots

Deciding on the number of principal components to use is helped by two types of plots known as *Scree* and *cumulative Scree*. These graphs plot the variance explained as a function of the number of principal components used. They are a useful part of the *dimensionality reduction* process. The Scree plot shows the number of principal components on the x-axis, and on the y-axis:

$$\frac{\sigma_i^2}{\sum_{j=1}^{n} \sigma_j^2}.$$

This measurement is also an indication of how much information is retained by the ith principal component. The cumulative Scree plot shows the number of principal components on the x-axis, and the following on the y-axis, which represents the cummulative fraction of information

$$\frac{\sum_{i=1}^{k} \sigma_i^2}{\sum_{j=1}^{n} \sigma_j^2}, k = 1, \ldots, n. \tag{3.20}$$

These values together with a visual analysis of the plots can help one to decide how many (and which) k principal components (dimensions) to keep. By analyzing the *cumulative Scree* plot we can also decide when adding another principal component does not significantly increase the variance in the data (i.e. when adding more components will not add any significant amount of information). In our experiments in particular, it was enough to keep as many principal components to reach a cumulative total of 0.9 ± 0.05. So if the ratio from Equation 3.20 for the ith singular value is close to 0.9 we use only the first i principal components to represent the effective dimensions.

The use of the scree plots will be described with more detail using the experiments presented in Section 7.2.

3.6.3 Loadings Vectors

The loadings vectors from the loadings matrix V^T are used to find which original features are correlated along each of these principal components. By plotting as bars the magnitudes of each component of each column vector of V it is possible to determine the importance of that feature in the makeup of that principal component. Also, by comparing the magnitudes of each feature with the others, we can look for existence of correlations between features. Usually, we are interested on examining the first k principal components, and how the different features behave in these components (k is obtained from the scree plots as described in Section 3.6.2). Nevertheless, the very last principal components, which are scaled by the smallest singular values, can also provide valuable information about redundant variables.

By studyng the loadings vectors we can assess the importance of each feature and decide whether it can be disregarded or not, e.g., we can eliminate features that do not significantly contribute information in the first principal components. To visually analyze these vectors, we generate a bar chart per principal component in which each original feature is represented as a bar, the length of the bar is the magnitude of that feature as a component of that particular loadings vector, and the direction of the bar (above zero or below zero) corresponds to the sign. In general, the following criteria can give important information regarding the features, based on the analysis of the loadings vectors:

- For any particular principal component, if a feature has a high loading with respect to others, it means that that feature has more weight or importance in that principal component. Furthermore, features that generally score high in the first k principal components are considered as the most *relevant* features, that best characterize the data in X in having the most variation from sample to sample.

- Features that always score low in the first k chosen principal components can be characterized as not exhibiting much variation in the original data; therefore discarded as being relevant to the data set (we will call these *useless*).

- A feature that has the same scores as another feature in these k components can be considered as *correlated*.

- Features that score high in the last $n - k$ principal components can also be disregarded – these features can be considered as *redundant*. The last few principal components retain very few information because they are scaled by singular values that are close to zero compared to the first singulare value. Since most of the information have been captured by previous dimensions, if a feature is important in any of these components, it means its information can be "represented" by some other feature(s).

Algorithm 1 shows the steps to find relevant features based on these criteria. To facilitate the identification of relevant features and correlations between features using the loadings vectors, we square the entries (preserving the sign) $V_{ij}^2,\ i, j = 1, \ldots, n$. The number of important loadings vectors k and the threshold $h \in (0, 1]$ are determined depending on the application (these may be arbitrary). The threshold is used to determine if a feature is considered as relevant based on its associated magnitude along a loadings vector, for instance, if v_{max} indicates the highest magnitude then other magnitudes that fall between v_{max} and $v_{max} \times h$ are also considered relevant.

Algorithm 1. *Identify Relevant Features*

Input: *loadings matrix V, the number of loadings vectors to analyze k, the number of original features n, threshold h, indexed set of original features F.*

Output: *the set of relevant original features, and a list of original indexes $\mathcal{L}_{relevant}$ for these features.*

initialize $\mathcal{L} = \{\}$

for each column vector V_j *where* $1 \leq j \leq k$

determine the most relevant feature along that vector

$$v_{max} = max_{1 \leq j \leq n}\{(V_{ij})^2\}$$

find other relevant features using h*, for each feature* V_{ij} *where* $1 \leq i \leq n$

if $(v_{max} \times h) \leq V_{ij} \leq v_{max}$ *then*

$\mathcal{L} = \mathcal{L} \cup i$

ouput $\mathcal{L}_{relevant}$ *and* F_i *where* $i \in \mathcal{L}_{relevant}$

To find possible *positive* correlations between features we follow Algorithm 2. The number of vectors to analyze k does not have to be the same as in Algorithm 1. The threshold $h \in (0, 1]$ is used to determine how close are the magnitudes of two features, we want h to be as big as possible, e.g., $h = 0.01$ means that two magnitudes are within 1% of each other. Optionally, we can remove all but one correlated features to eliminate more dimensions.

Algorithm 2. *Identify Positive Correlations*

Input: *loadings matrix V, the number of loadings vectors to analyze k, the number of original features n, threshold h, indexed set of original features F, and $\mathcal{L}_{relevant}$.*

Output: *the set of correlated original features, and a list of original indexes $\mathcal{L}$ for these features.*

initialize $\mathcal{L}_l = \{\}$*, a list of positively correlated features for each relevant feature in* $\mathcal{L}_{relevant}$

for each column vector V_j *where* $1 \leq j \leq k$

for each feature V_{ij} *where* $1 \leq i \leq n$

for each $l \in \mathcal{L}_{relevant}$

initialize $\mathcal{L}_{lj} = \{\}$*, a list of correlated features*

if $(V_{lj} \times (1 - h)) \leq V_{ij} \leq V_{lj}$ *or* $V_{lj} \leq V_{ij} \leq (V_{lj} \times (1 + h))$ *then*

$\mathcal{L}_{lj} = \mathcal{L}_{lj} \cup i$

for each $l \in \mathcal{L}_{relevant}$

if feature i *is in every* $\mathcal{L}_{lj}$ *where* $1 \leq j \leq k$

$\mathcal{L}_l = \mathcal{L}_l \cup i$

for each relevant feature $l \in \mathcal{L}_{relevant}$,

ouput $\mathcal{L}_l$ *and*

for every $\mathcal{L}_l$

output F_i *where* $i \in \mathcal{L}_l$

optinally, in every $\mathcal{L}_l$ *remove all but one feature index*

If we also want to include negative correlations cases, we can use a modification of Algorithm 2, simply use absolute value of V_{lj} and V_{ij} in the comparisons step.

3.6.4 Scores Vectors and Plots

The scores vectors and plots are PCA's main tool to identify data clustering. They are used to plot the projections of each point from the original space of X, onto the space of the principal components. While the loads vector plots give information regarding the features, the scores data provide information about the relationships between the observations in the input data [Jackson, 2003]. The scores vectors are extracted from the matrix U, which has one row per observation. For the scores plots, we actually use the *regular scores* corresponding to T, whose entries are scaled by the singular values.

The scores plots are used to determine which observations are similar to each other in their make-up of attributes, those observations with similar attributes (e.g., spectral properties, structural, simple) will probably have similar effects on the behavior (and consequently the slection) of the iterative solver. As with the scree plots, the detailed application and usage of these vectors will be addressed in terms of the experiments from Section 7.2.

To visually analyze the data represented in the new feature space (principal components), the transformed data is plotted in a 2 or 3 dimensional space using the first 2 or 3 heaviest principal components respectively as axes. This type of mapping usually reveals the correlation structure of the data, which is harder to identify by plotting in the original space [Faloutsos et al., 1997]. The scores plots are also helpful to detect *outliers* that deviate from the common patterns. The outliers are those few points far away from the clusters that have an important effect on the variance calculation from the original data matrix.

In general, these plots give a better pictorial description of the data sample than if we plotted using two or three of the original variables [Russell and Norvig, 1995].

3.6.5 Feature Preprocessing for PCA and Classification

In many cases, due to the variables having different physical units, the variances of the original features may vary considerably, affecting the directions of the principal components, more than the correlations themselves; to correct this problem we can preprocess the original data so that each feature column is mean centered (has mean 0), and it's variance is normalized (unit variance), then we can apply PCA over the preprocessed data.

PCA is based on a variance maximization criteria, which makes the method sensitive to outliers. By using certain methods (e.g. calculating Mahalanobis distance of the data points and isolating the outliers) it is possible to acquire a good estimation and performance of classification [Russell and Norvig, 1995]. So for practical purposes, sometimes it is desirable to eliminate such outlying observations.

Initial reduction of features

There are certain types of features that do not vary in some datasets. Those features that remain constant along the experimental datasets are not of interest because they don't provide any information about the dataset, neither can they be used for statistical purposes. All these features are suitably eliminated "automatically". The criteria used to cancel out these features that are constant (or mostly invariant) is by obtaining the standard deviation of the column corresponding to each feature, the column is eliminated when this value is zero or close to it. For example, suppose that the ith column of A corresponds to some $feature_i$ that is constant, if any of the following holds the feature will be eliminated:

$$stdev(feature_i) \approx 0 \quad \text{or} \quad \frac{stdev(feature_i)}{mean(feature_i)} \approx 0$$

Characterization of Relevant Features

There are three main tasks in feature characterization: identification and elimination of useless features, identification and understanding of relevant features, and discovery and analysis of correlations among features.

For the first task, we want to eliminate features that are statistically useless (e.g., constant values, NaN, etc.), the criteria of elimination of these features is by measuring the standard deviation of the column, when it is 0 or below a certain limit (machine epsilon) the column is considered to be constant (as it was explained in Section 3.6.5).

The next step is to eliminate noisy data. This corresponds to the case when the standard deviation σ of a feature along the dataset is too large specially with respect to its mean μ, which indicates that the scores are widely dispersed and the mean is not very informative about the average score.

As a final step, we want to contemplate the elimination of certain features that may be redundant by being highly correlated to other features. This step is more complex since we have to make at least one pass of the analysis on the data to identify correlations, and then decide upon this what are the features that we want to eliminate (note that this is another way to further achieve dimensionality reduction). This step also requires a closer analysis of the data. In Section 3.6.3, we discussed the loadings-matrix V^T. By analyzing the loadings vectors we can determine which features are the main players along the main principal components and their behavior. We can also find which additional features can possibly be eliminated. In Section 7.2 we will illustrate the process of feature characterization in various experiments and datasets.

PCA and Classification Techniques

PCA can be used as a preprocessing step for the input data. The transformed data and feature space can be used for classification instead of the original ones. This can be advantageous for certain types of classifiers.

On the other hand, there are some classifiers for which the use of PCA-transformed data does not necessarily improve the accuracy of classification. An example of such classifier, found through preliminary experiments, is decision trees. This is a very effective classifier and had overall the best performance in our tests, so it was used extensively to obtain most of the results presented in the experiments chapter. It is important to point out some of the reasons for which using PCA to

preprocess the data may not be completely favorable for decision trees. These are related to inherent characteristics of this classifier; decision trees [Dunham, 2002, Quinlan, 1993]:

- Need more effort to handle continuous data: the numerical nature of our problems (matrices) imply that many of the features used for classification are continuous. Nevertheless, from experiments we have identified several discrete features that are crucial properties and affect the performance of iterative solvers and preconditioners (Section 7.2.3)
  - diagonal sign
  - symmetry
  - maximum number of non-zeros per row
  - diagonal definite
  - number of ritz values
- Can handle well high-dimensional data: part of the idea of using PCA is to reduce dimensionality. Even if we keep 90% of the information in a PCA-transformed space, we could be disregarding small contributions from some features. A decision tree may actually use this little information to improve a classification *.
- Ignore correlations among attributes: as we will see in the experiments presented in Section 7.2.3 there are several features that consistently show correlation with others. With PCA we can identify and eliminate correlated features, but decision trees does not necessarily benefit from this, neither from the use of completely uncorrelated features.

3.7 Statistical Error Analysis

Statistical errors occur when the estimated values from an experiment are different from the true ones, which can be caused by unpredictable randomness in the data and/or from the loss of precision introduced by implementations of algorithms. A statistical error emerges when we make a wrong assertion with respect to the *null hypothesis*. A *null hypothesis*, denoted as H_0, is a statement to be tested to see if in fact it is true. In classification problems like ours, H_0 is the assumption that a particular class is the source of the observations. The data collected from the experiments is used to decide if H_0 should be *rejected*, in order to support instead an *alternative hypothesis* denoted as H_1 [Duda et al., 2000, Alpaydin, 2004]

The simplest structure to analyze statistical errors, is a classification problem with two classes and hence, the two hypotheses. The problem of recommending a reliable problem fits well in this structure, so we will cover this case first (Section 3.7.1). Statistical errors arising from multiclass problems are more difficult to examine and interpret; such is the problem of recommending the fastest method, which is multiclass in the sense that we need to have one class per method.

*By this, we are not saying that the results from PCA should be completely disregarded for decision trees. We only stress that it is not necessary to transform the feature space because a decision tree could make better use of the original non-transformed features.

3.7.1 Statistical Error Analysis for a Two-Class Problem

Since classification problems are based on probability distributions, the problem of testing the null hypothesis becomes that of testing whether the probability of obtaining data given H_0 is smaller than a threshold, in which case we would favor H_1. For instance, consider the two (mutually exclusive) classes in the reliability problem:

- $class_1$: set of problems for which method M converges
- $class_2$: set of problems for which method M diverges

For a single method M and any problem A, we can then define the hypotheses as follows:

H_0: $A \notin class_1$

H_1: $A \in class_1$ (or $A \notin class_2$)

The statistical errors emerge from erroneously accepting or rejecting H_0:

Type I error: also known as an α error, which is the error of rejecting a null hypothesis when it is actually true. This is also known as "false positive".

Type II error: also known as a β error, which is the error of accepting a null hypothesis when the alternative hypothesis is the true state. This is also known as a "false negative".

The number of false positives (FP) and false negatives (FN), and their rates (FP_{ratio} and FN_{ratio}) with respect to the sample are useful measurements when we assign certain type of penalties for making a wrong decision. The overall error rate is measured in terms of both types of errors:

$$Error_{ratio} = \frac{FP + FN}{|TestingSet|} \tag{3.21}$$

The individual error rates FP_{ratio} and FN_{ratio}, which measured respectively, the total count of negative and positive instances, are:

$$FP_{ratio} = \frac{FP}{FP + TN} \tag{3.22}$$

$$FN_{ratio} = \frac{FN}{FN + TP} \tag{3.23}$$

When we have only two classes, the FP of $class_1$ is the same as the TN of $class_2$ and vice versa; and FN for $class_1$ is the same as TP for $class_2$, i.e. whether the statement is positive or negative depends on which class we are analyzing (positive in one class is negative on the other). The kind of penalties assigned depends on the problem and on how important it is to distinguish between FP and FN, but in general, we want to minimize the cases for which we would pay a high price for making such a a mistake to make the best choice (recommendation). Table 3.1 illustrates how these errors originate based on the result of the prediction and the actual result of an experiment.

Table 3.1: Possible predictions for a two-class classification problem.

		Test result: Positive	Test result: Negative	
Actual Condition	True	TP	FN (error)	↗ Wrong Prediction
	False	FP (error)	TN	↘ Correct Prediction

Table 3.2: Confusion matrix for a 3-class classification problem.

		Test result: ω_1	Test result: ω_2	Test result: ω_3
Actual Condition	ω_1	CM_{11}	CM_{12} (error)	CM_{13} (error)
	ω_2	CM_{21} (error)	CM_{22}	CM_{23} (error)
	ω_3	CM_{31} (error)	CM_{23} (error)	CM_{33}

3.7.2 Statistical Error Analysis for a Multi-Class Problem

Finding the fastest method (performance problem) can be viewed as a multi-class classification problem. In such cases it is not trivial to analyze the classification errors to rate the accuracy of a classification algorithm.

This problem usually involves the comparison, side by side, of more than two methods. In such cases, a more appropriate way of analyzing the statistical error is using a *confusion matrix* approach. A confusion matrix [Provost et al., 1998] contains results from a classification process with respect to actual class memberships and misclassifications. This matrix is very similar to Table 3.1, except that it has one row (and one column) per class. Along the diagonal are the "true positives" for each class, or the number of cases that are correctly classified. Every other entry in a row is a misclassification error, that of assigning to a different class an observation that belongs to the class of that corresponding row. For example (see Table 3.2), suppose there are 3 classes so the confusion matrix CM is of size 3×3, the first row corresponds to class ω_1 so in CM_{11} we have all those observations correctly assigned to ω_1; in CM_{12} we have those observations belonging to ω_1 but were classified as ω_2; and in CM_{13} are the observations from ω_1 erroneously assigned to ω_3.

This type of matrix is very useful to see where the classification errors come from, which are the classes a classifier gets "confused" with. The more diagonal the confusion matrix is, the more accurate a classifier for these classes would be. In Section 7.4 we demonstrate the applicability of this concept in our research and experiments for the Performance problem (classification and prediction of most efficient method).

Chapter 4

Concepts for Algorithm Classification and Recommendation

The task of finding an optimal method can be viewed as a classification problem. The set of matrices (or numerical problems in general) constitutes the set of *observations* in the classification problem. Each class is composed of a set of observations (matrices) for which a particular method is the *best*. When a new matrix is presented, the problem of finding the most suitable method becomes that of assigning this matrix to a class.

In general, like many other machine learning problems, the process of finding an optimal method is divided in two: learning and recommending. The learning part consists of training classifiers to learn certain functions (or parameters) and derive decision rules that determine the membership of a matrix in a class. The recommendation part uses these rules to assign a new matrix to one of the classes based on its properties. Obtaining the membership for the new problem is in fact a way of determining the best method for solving it.

In this chapter we describe the learning and recommending stages in our problem's context, as well as the defipnitions of general elements necessary for the setup of specific strategies for the Perormance and Reliability problems.

4.1 Classification and Recommendation Process

In this dissertation, we will focus on the use of *supervised learning* classification approach for the learning process. Supervised learning is a type of classification in which, during the process of modeling the classifier, we know the class to which each observation belongs to (hence the name supervised). The set of observations is divided in two subsets: *training* set and *test* set. This type of classification has two main stages: *training* and *testing*, where the aforementioned data subsets are used respectively.

Like in other classification problems, the task in supervised learning is to learn an optimal mapping from an input X (the set of observations) to an output Y (the set of classes). For instance, in the problem of finding the optimal iterative method X would be a set of vectors of features that describe a set of sparse matrices, and Y would be a set of classes, one per iterative method; the mapping would be given by a function that assigns a problem in X to the best-method's class. From a machine learning point of view, such a mapping function, which we will call f, is a model defined in terms of a set of parameters θ, and is known as *discriminant* function. The main goal

is to derive this discriminant function, which separates the instances of different classes and a best set of parameters θ, such that the estimated results are as close as possible to the actual values (minimization of the approximation error) [Alpaydin, 2004]. In the example problem, θ could be the ranges of values that the vectors from X have in each class.

The discriminant f can be viewed as an *approximation* to an unknown ideal function $\mathcal{F}$ that can assign perfectly every observation to the correct class. For us f represents a classifier that decide which method class a problem in X should be assigned to, and $\mathcal{F}$ would be a trivial function that uses the known membership (since we are using supervised learning) to assign each observation to its correct class.

Definition 1. *Let $\mathcal{F} : X \mapsto Y$ be an ideal function that correctly classifies every input $x \in X$. The learning process is used to derive a discriminant function $f : X \mapsto Y$ such that*

$$max_X(\mathcal{F}(x) = f(x))$$

In other words, suppose that the optimal mapping is given by an unknown function $\mathcal{F} : X \mapsto Y$; the learning process derives an *approximation* function $f : X \mapsto Y$ such that for as many cases as possible $\mathcal{F}(x) = f(x)$. Let $x \in X$ and $y \in Y$; then the discriminant function $f : X \mapsto Y$ is

$$y = f(x|\theta) \tag{4.1}$$

The actual values used to model the discriminant function are obtained from a set of observations called *training set*; the training set $X_{tr} \subset X$ are those observations for which $\mathcal{F}(X_{tr})$ is known. The estimates of how well f approximates $\mathcal{F}$ come from testing the discriminant function f on a different set of observations called the *test set* $X_{te} \subset X$, for which $F(X_{te})$ is also known. The best approximation is the one that maximizes the number of cases where: $f(X_{te}) = \mathcal{F}(X_{te})$.

During the *training* stage, the objective is to train a classifier to derive a decision rule. In many cases this means to obtain the parameters θ that model the discriminant function. The discriminant function then allows us to make the decision of which observations belong to which class using the *knowledge* that the training set provides.

In the *testing* stage, we use the test set to evaluate the rule obtained in the training stage. Using the discriminant function, each observation in the test set is assigned to a class. To evaluate how well a classifier works, we count the number of correct assignments made using the decision rule during the testing stage. This is a way of measuring the performance of a classifier and will be referred to as *accuracy of prediction* or *accuracy of classification*. The actual process of obtaining these accuracies is carried out by a function which we will call *accuracy evaluation module*, to differentiate it from the actual classification process (which outputs a membership instead of an accuracy value). In this research, each classifier will have a accuracy evaluation module associated with it.

Figure 4.1 illustrates the training and testing steps in a classification process, and their relation with the input dataset, and the accuracy measurements. To implement adaptivity, the outcome of the recommendation process is used as feedback for the learning process (similar to the way in which the testing process is used with the training process).

In summary, for the experiments presented here, the learning part comprises the following actions:

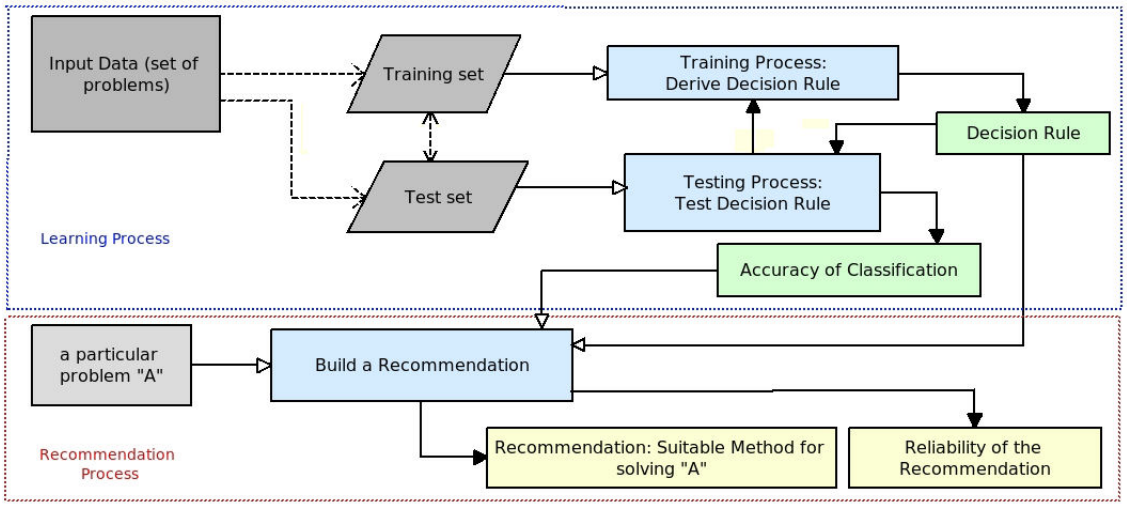


Figure 4.1: General data flow for supervised learning classification. The input of the classifier is a dataset that is partitioned into training and test sets. The classifier function is used to derive and test a decision rule with these datasets respectively. The decision rule is then used to measure the performance of the classifier, which is accuracy of predicting a membership in each class. The testing and training processes are repeated over several iterations (the results of the testing process can be used as a feedback for further training). In each iteration, different combinations of observations from the original dataset are used to build the training and test sets. The resulting accuracy integrates the results from all the iterations (e.g., average).

1. Feature extraction and collection: from various types of numerical problems obtain and/or compute different categories of properties.
2. Collection of convergence and performance measurements: this involves running every available solver (in our software implementation) for on each numerical problem.
3. Feature Processing: this constitutes the identification of relevant features, elimination of useless features, investigation of the possible correlation among features and dimensionality reduction (some of these can be achieved using Principal Component Analysis).
4. Training the classifier: using the features and performance information to derive a decision rule.

Next we give a formalization of the problem of selecting the best method for solving a given numerical problem, and the methodology proposed.

4.2 Problem Formalization

In this section we define the different elements and functions in the method selection problem. Such elements are the set of numerical problems and their features, numerical solvers, performance measurements. The functions are, for instance, feature extraction, transformations, method selection and accuracy evaluation.

4.2.1 Basic Elements

Definition 2. *Let $\mathbb{A}$ be the set of numerical problems.*

In the research presented in this dissertation, $\mathbb{A}$ is the set of *all* linear systems of sparse coefficient (sparse matrices) mainly resulting from PDE problems. We define $\mathcal{A} \subset \mathbb{A}$ as the finite set of those particular matrices available in our research. These matrices have been collected from different sources (e.g., the Matrix Market [Market, 2007]) and originate from different problems.

We define the solution space for numerical problems as a set of solutions $\mathbb{S}$ for the numerical problems with the associated performance measurement $\mathbb{T}$ for each of them:

Definition 3. $\mathbb{R} = \mathbb{S} \times \mathbb{T}$ *is the results space, the space of solutions plus performance measurements,*

where

Definition 4. $\mathbb{T}$ *is the set of performance measurements.*

Now, we can formalize the definition of a method

Definition 5. $\mathbb{M} : \mathbb{A} \mapsto \mathbb{R}$ *is the set of methods that potentially solve the class of numerical problems* $\mathbb{A}$. $M \in \mathbb{M}$ *may converge or diverge when trying to solve* $A \in \mathbb{A}$.

In general, the method space is an unordered, finite set of methods. Here in particular, $\mathbb{S}$ refers to the solutions to $Ax = b$, and $\mathbb{M}$ is the set of iterative solvers. A method M can be an considered an iterative method, or a combination of a preconditioner (or other transformation) and an iterative solver which we call *composite* or *combination method*. We define the set of iterative methods as $\mathbb{K} \subseteq \mathbb{M}$, where $\mathbb{K} : \mathbb{A} \mapsto \mathbb{R}$. The set of available methods in our research is denoted by $\mathcal{M} \subset \mathbb{M}$.

Transformations or preprocessors include scalings, approximations and preconditioners among others, and will be described in detail in Section 4.2.3. The set of transformations will be denoted as $\mathbb{D}$. In the research here presented, we will focus only on the use of the preconditioner transformations, i.e., the set of transformations $\mathbb{D}$ contains only preconditioners. The set of available iterative methods in our research is denoted as $\mathcal{K} \subset \mathbb{K}$ and the set of available preconditioners as $\mathcal{D} \subset \mathbb{D}$.

The set of composite methods is defined as

Definition 6. *Let* $\mathbb{K} \subset \mathbb{M}$. *The set of composite methods is* $\mathbb{D} \times \mathbb{K} \subset \mathbb{M}$.

Suppose for instance we have $ksp \in \mathbb{K}$ and $pc \in \mathbb{D}$. A composite method would be $pc, ksp \in \mathbb{M}$, which means we first transform a problem A with pc and then attempt to solve $pc(A)$ using ksp. The set of available composite methods is given by $\mathcal{D} \times \mathcal{K}$. Our classification problem then consists of finding a suitable $ksp \in \mathcal{K}$, $pc \in \mathcal{D}$, or a combination $pc, ksp\rangle \in \mathcal{M}$. Strategies for composite methods are of particular importance and will be discussed in Section 4.3.

Any method M applied to A takes some time to converge to a solution, or to stop by the criteria of the *stopping test* (meaning that it doesn't converge). Time to solution is one way of evaluating how well a method can solve a problem. There are other types of measurements to evaluate the behavior of iterative methods, such as the accuracy of the solution or the number of iterations to convergence. Here, we will focus on the use of time to solution as a performance metric.

A timing function $T(A, M)$ denotes the time that it takes for method M to solve problem A. For the cases when M diverges we set $T(A, M) = \infty$ ($T(A, M) \neq \infty$ otherwise). By $T(A)$ we denote a vector of timings of all methods on problem A.

Note that if the time t to solve a problem is defined, then there is a corresponding solution $\sigma \in \mathbb{R}$ for every problem $A \in \mathbb{A}$.

The problem of finding most suitable method $M \in \mathbb{M}$ to solve a problem in $\mathbb{A}$ could be in terms of reliability, performance, and/or a combination of both.

4.2.2 Numerical Properties of the Feature Space

A matrix as a numerical problem can have many different properties. In order to generalize the way the numerical problem is taken and analyzed by a classification method, we will describe a matrix in terms of its numerical properties or *features*. The set of features presented in here consider most of the properties and issues that are widely used for studying numerical problems. The features have been grouped in categories according to the type of numerical property they describe. Once they are extracted and/or computed for a problem, they are organized and presented as a vector.

Definition 7. *Let $\mathbb{F}$ be the set of feature vectors that numerically describe the problems in $\mathbb{A}$. Let $\bar{x} \in \mathbb{F}$ be a feature vector that describes a numerical problem (sparse matrix) $A \in \mathbb{A}$. $\Phi : \mathbb{A} \mapsto \mathbb{F}$ is a function that extracts the features from the sparse matrix and arranges them as the components of the feature vector:*

$$\Phi(A) = \bar{x} \tag{4.2}$$

The components of $\bar{x}$ are very unhomogeneous and they can be real numbers, integers, fractions, or elements of a finite set of choices (e.g.,binary values).

The components of a vector $\bar{x} \in \mathbb{F}$, in the particular example of linear system solving, correspond to various features in the following categories:

Simple: norm-like quantities that in general can be calculated in time proportional to the number of nonzeros of a matrix.

Structure: quantities that are strictly a function of the nonzero structure, which stay invariant during a nonlinear solve process or while time-stepping a system of equations.

Spectrum: probably the most informative properties of a matrix. In here are included eigenvalues and singular values, as well as various estimates obtained by running multiple GMRES iterations on a system.

Normal: estimates of departure from normality of a system (usually very expensive to compute).

Variance: heuristic measures not related to any known mathematical theory. These describe how different the elements in a matrix are.

Given these categories, it makes sense to subdivide the feature space in separate dimensions, one for each category

$$\mathbb{F} = \mathbb{F}_1 \times \cdots \times \mathbb{F}_k$$

Accordingly, we can also split up the feature extraction function as follows

$$\Phi = \langle \Phi, \ldots, \Phi_k \rangle$$

where $\Phi_i \colon \mathbb{A} \mapsto \mathbb{F}_i$ represent the individual feature calculations.

4.2.3 Method Selection

Like in other classification problems, for method selection, we need to define a classification function like Equation 4.1. This function should discriminate between the available methods to choose the best given the properties of a matrix. Time to solution is the performance metric upon which we base the decision, so we can define the function for the Reliability and the Performance problems as follows:

Definition 8. *The problem of selecting a suitable method to solve a given problem A, can be viewed as that of constructing a function $\Pi : \mathbb{A} \mapsto \mathbb{M}$ that chooses the most suitable method given a problem A, by mapping the problem space into the method space. Stated formally:*

$$\Pi(A) = M \quad \textit{where} \quad T(A, M) \neq \infty \quad \textit{and} \quad M \in \mathbb{M} \quad \textit{for the Reliability problem}$$

$$\Pi(A) = M \quad \equiv \quad \forall M' \in \mathbb{M} : T(A, M) \leq T(A, M') \quad \textit{for the Performance problem}$$

Since it is not possible to define Π on the problem space $\mathbb{A}$, we reduce and represent a problem A as a vector $\bar{x} \in \mathbb{F}$ of m dimensions, each corresponding to a numerical property (feature) of A.

Definition 9. *The method selection problem, based on Π and Φ (defined in 4.5 and 4.2 respectively) is given by:*

$$\Pi(\bar{x}) = M \;\; \textit{if} \; \exists A \in \mathbb{A} \; \textit{s.t.} \; \Phi(A) = \bar{x}, \;\; \textit{and} \; M \; \textit{s.t.} \; \forall M' \in \mathbb{M} : T(A, M) \leq T(A, M') \tag{4.3}$$

In Equation 4.3 the definition was dependent on finding a problem A with the desired features. This raises the question: what if there are two problems with the same features. Such a situation indeed occurs in practice, and it is the reason that we can not directly hope to predict runtimes, and base our function Π on such runtime prediction.

Theorem 1. *It is not possible to predict a runtime $t \in \mathbb{T}$ or base our function Π on such a prediction.*

Proof. For consistency, a prediction should be done based on optimal method rather than runtime. Suppose for instance that problems A_1 and A_2 have approximately the same features, that is, $\Pi(A_1) \approx \Pi(A_2) \approx \bar{x}$. Let $\Pi(\bar{x}) = M$; then for consistency

$$T(A_1, M) = \min_{M'} T(A_1, M') \quad \text{and} \quad T(A_2, M) = \min_{M'} T(A_2, M')$$

which means that the same method should be optimal, as opposed to having the same runtime $T(A_1, M) = T(A_2, M)$, which does not hold. □

Furthermore, equation 4.3 depends on the features $\bar{x}$ of a problem A. If there are two problems that have the same $\bar{x}$ but scaled differently, it is not realistic to make different predictions for both problems; for this reason, it is necessary to normalize $\bar{x}$ to make the problems *scale-invariant*.

The scale-invariant property for the selected features is partially attained by *normalizing* each scale-dependent feature by another feature that varies, proportionally, depending on the scale of the problem. For example, the number of non-zeros nnz is normalized using the square of number of rows $nrows^2$, which makes the feature $\frac{nnz}{nrows^2}$ scale-invariant.

Additionally, the first steps of PCA also scale and normalize the features, helping to achieve further scale-independence (the details of this process were addressed in detail in Section 3.6). We will define the process associated with PCA as a function that transforms the original feature space $\mathbb{F}$ into a new one

Definition 10. *Let* $\Gamma : \mathbb{F} \mapsto \mathbb{F}'$ *be the function from PCA that transforms the original feature space into a new orthonormal space:*

$$\Gamma(\bar{x}) = \bar{x}'$$

where $\bar{x} \in \mathbb{F}$, $\bar{x}'$, *and* $\mathbb{F}'$ *is an affine subset of* $\mathbb{F}$. *The original vector* $\bar{x}$ *is first mean-centered and scaled, then transformed using singular value decomposition.*

As we saw in Section 3.6, the vectors in this new feature space may have the same or fewer number of features than the original vectors.

We also define

$$\Psi : \mathbb{A} \mapsto \mathbb{M} \tag{4.4}$$

as the ideal, unknown function that indicates the objectively optimal method $M \in \mathbb{M}$ to solve $A \in \mathbb{A}$; we will use these function to define the class spaces. This function would correspond to the $\mathcal{F}$ function described in Definition 1; analogously, the function Π corresponds to the function f from Equation 4.1.

Preprocessor Transformations

There are various preprocessors that can be applied to a problem A; they map one numerical problem into another which may be more simple. These transformations can have positive or negative effects on the performance of the iterative methods. In general, these preprocessors can be viewed as a series of *transformations* that applied a coefficient matrix A produce a new one, say A'. Formally:

Definition 11. $\mathbb{D} : \mathbb{A} \mapsto \mathbb{A}$ *is the set of transformation functions that can be applied to* $\mathbb{A}$.

The function Φ in Definition 7 makes it possible to reflect on $\mathbb{F}$ any changes made on $\mathbb{A}$: Let $A \in \mathbb{A}$ such that $\Phi(A) = F$, and let $D \in \mathbb{D}$ be transformation function such that $D(A) = A'$. It follows that $\Phi(A') = F'$ where $A' \in \mathbb{A}$ and $F' \in \mathbb{F}$. Based on this, we could redefine the function Π to select the best solution method $M \in \mathbb{M}$ as $\Pi : \mathbb{F} \mapsto \mathbb{M}$

$$\Pi(F) = M. \tag{4.5}$$

As mentioned before, in our experiments we will focus on the study of the use of the *preconditioner* transformation only. The classification and recommendation of iterative methods paired with preconditioner is a very important task; there are several ways to approach this problem. These strategies will be introduced and discussed in Section 4.3.

4.2.4 Elements of the Classification Process

A classifier is a function that is used to map from an set of observations (or problem instances) to a set of classes, where a class is generically defined as a set of *observations*. In our case, an *observation* is a matrix $A \in \mathbb{A}$. For each observation there are several timings, each of them corresponding to the time that each method takes to solve it.

Definition 12. *The* class space *denoted as* $\mathbb{C}$, *is a* partition *of the observation space, where each subset is a class.*

The configuration of the class space and its constituent classes depends on the *classification problem*. In our case the *classification problem* is to determine the suitability of numerical methods

in terms of either *reliability* or *performance*. For instance, when we want to find how reliable a method M is, then $\mathbb{C}$ consists of only two classes: the set of matrices for which M converges, and the set of matrices for which it diverges. On the other hand, if we want to determine which method performs the best, we have to delimit sets of matrices for which each converging method was the fastest. In this case there is a class for each method.

As we will see in Section 5.1 and Section 6.1, the *class space* definition for the Reliability (convergence) problem is quite different from the Performance problem. The definition of a class in $\mathbb{C}$, however, can be generalized as set of all matrices for which a particular method is the most *suitable* solver:

Definition 13. *Let M be a method in $\mathbb{M}$. A class is defined as the set of observations $Class_M$, such that $\forall A \in Class_M$, $M = \Psi(A)$ and $Class_M \subset \mathbb{A}$.*

The classifier's purpose is essentially to output a class membership for each input observation, where each of these can belong to *only one* class at a time. The number of classes depends on the type of problem, e.g., the reliability problem has only 2 classes, while the performance problem has as many classes as methods we want to compare. In general, a classifier function can be defined as follows:

Definition 14. *Let $\mathbb{F}$ be the feature space as defined in Section 4.2 and $\mathbb{C}$ a finite set of performance or convergence classes. A classifier is defined as a function $\Upsilon : \mathbb{F} \mapsto \mathbb{C}$.*

A classifier uses different probability distribution functions (see Equation 3.1) to model the behavior of the elements in each class. These function are derived during the *training* stage described in Section 4.1.

Observe that the function Υ takes as input a feature vector F, which implies that we have to extract the features from a problem A, i.e., $F = \Phi(A)$.

The effectiveness of a classifier is obtained after the *test stage* is completed, and can be evaluated as the *accuracy* with which it can predict each of the classes. The process of computing the accuracies is actually one of the stages in *supervised learning* (described in Section 3.2). An *accuracy evaluating function* computes the *accuracy of prediction* (classification) for each class. Generally speaking the accuracy of classification is the probability of correctly classifying observations for a class, i.e., it is the ratio of the number of test observations classified correctly with respect to the number of observations actually in that class. Formally:

Definition 15. *Let Υ be a classifier with class space $\mathbb{C}$, and I be the index set of classes in $\mathbb{C}$. Let $i \in I$, let $C_i \in \mathbb{C}$ be the set of all numerical problems in class i, and let $C_{te_i} \subset C_i$ be the set of numerical problems correctly assigned to class i by Υ. The accuracy of classification of Υ is:*

$$\alpha = \frac{|C_{te_i}|}{|C_i|}$$

where α ranges between $[0, 1]$ and 1 is perfect accuracy.

In any classification problem $|\mathbb{C}| \geq n$ and $n \geq 2$, so there is an α for each class in $\mathbb{C}$. For convenience we can represent the associated set of accuracies for $\mathbb{C}$ as a vector $\bar{Y}$ of n components, where each component corresponds to the accuracy of each class:

$$\bar{Y} = \langle \alpha_1, \alpha_2, \ldots, \alpha_n \rangle \tag{4.6}$$

and $\bar{Y} \in [0,1]^n$, where $n = |\mathbb{C}|$.

Since the accuracy of a classifier is a critical computation for prediction and recommendation purposes, we define an additional function associated with each classifier which calculates the accuracy of prediction for each class. We will call this function *accuracy evaluating function*, and it takes as input a set of feature vectors (observations) and outputs a vector of accuracies:

Definition 16. *For a given classifier* $\Upsilon : \mathbb{F} \mapsto \mathbb{C}$ *the accuracy evaluating function is defined as*

$$\Omega : \mathbb{F} \mapsto [0,1]^n, \text{ where } n = |\mathbb{C}|.$$

4.3 Strategies for Combination Methods

In Chapter 2 we discussed iterative methods and preconditioners. There is a very important interaction between these two, the behavior of one is "conditional" on the application of the other, for example, a preconditioner can improve the performance of an iterative method or make it worse. The effects of this interaction is one of those facts (as mentioned in Section 1.2) that are not very well documented in literature, making this subject a very interesting research question. This interaction, of course, affects the method selection problem as well.

Out of the different types of transformations, we will focus on preconditioners because of the significant relationship they have with iterative methods (the study and strategies here presented can later be extended to other transformations such as scalings or approximations). The set of preconditioners is a subset of $\mathbb{D}$, but since in this research we only concern with the study of preconditioners, for convenience we will use $\mathbb{D}$ to represent the set of preconditioners, and $\mathcal{D} \subseteq \mathbb{D}$ to represent the set of available preconditioners. In a similar way $\mathcal{K} \subseteq \mathbb{K}$ is the set of available iterative methods.

The construction of recommendation strategies for combined methods depend on how strong we consider the dependency to be. To recommend a combination we can either choose the iterative method and the preconditioner as independent events, or as joint events. The goal is to classify a combination of a preconditioner transformation and an iterative method, whether they are dependent or not. Such a pair is represented as (pc, ksp) where $pc \in \mathbb{D}$ and $ksp \in \mathbb{K}$.

To represent this dependency we will use the concept of *conditional* combination*, which refers to a method as the composition of a transformation with an iterative method, e.g. applying the preconditioner ilu on a matrix A and then solving it with the iterative method $gmres$ is the conditional combination method $ilu, gmres$. Conditional combinations are in fact methods, and we will use the notation pc, ksp to represent them, so we have $pc, ksp \in \mathbb{M}$. Conditional classification is about performing classification on pc, ksp where $pc \in \mathbb{D}$ and $ksp \in \mathbb{K}$. These combinations are also "composite methods".

Definition 17. *Let* $pc \in \mathbb{D}$ *and* $ksp \in \mathbb{K}$. *The conditional combination* pc, ksp *is defined* $\forall A \in \mathbb{A}$ *as*

$$pc, ksp \equiv ksp(pc(A))$$

where $pc, ksp \in \mathbb{M}$.

Note the difference between the pair (pc, ksp) and the pair pc, ksp used in *conditional classification*. The combination $pc, ksp \in \mathbb{M}$ indicates that that the dependency is explicitly assumed,

*In here, the term conditional is not to be associated with the conditionality concept of probability and statistics, part of the Bayesian theory.

while (pc, ksp) only denotes a combination for which a recommendation is requested (notation for recommendation algorithms), where the pc and the ksp may or may not be dependent (i.e., pc, ksp is a case of (pc, ksp)).

The purpose is to construct a function Π that chooses the best transformation and the optimal iterative method, find the accuracy of the prediction for such a function and build suitable recommendations. This function can be constructed using different approaches, considering the selection of pc and ksp as independent events or not [†]. The names of the approaches we have developed are: orthogonal, sequential and conditional.

Next, we will describe each of these approaches in general and in Chapters 5 and 6, we will describe their specific use in the Reliability and Performance problems respectively. Later, in Chapter 8 we will see that these approaches perofm differently for those two problems.

4.3.1 Orthogonal Approach

In this approach we perform separately the classification for the iterative method and the transformation. We assume that the event of choosing a preconditioner pc is independent from the event of choosing an iterative method ksp, so we construct separate classifiers for choosing pc based on the features of the linear systems in $\mathcal{A}$, and ksp based on preconditioned features. For example, to select a combination for a given $A \in \mathcal{A}$ we use

$$\Pi^{orthogonal}(\Phi(A)) = pc, ksp, \text{ such that } \Upsilon_{\mathcal{D}}(\Phi(A)) = pc \text{ and } \Upsilon_{\mathcal{K}}(\Phi(D(A))) = ksp,$$

where D is some preconditioner in $\mathcal{D}$, $\Upsilon_{\mathcal{D}}$ is the classifier function for preconditioners and $\Upsilon_{\mathcal{K}}$ is the one for iterative methods.

In other words, separately perform the classification for pc using $\Upsilon_{\mathcal{D}}$ on the features of the unpreconditioned matrix, and the classification for ksp using $\Upsilon_{\mathcal{K}}$ on the features of preconditioned systems (all preconditioners considered); then we put together the results for pc, ksp. The accuracies of classification for the pc and the ksp are computed separately then we just multiply the independent accuracies of classification: $\alpha_{(pc,ksp)} = \alpha_{pc} \times \alpha_{ksp}$.

This approach is straightforward, but ignores interaction between the preconditioner and the iterative method.

4.3.2 Sequential Approach

In this approach, we assume that there is a dependency between the transformation and the iterative method that affects the classification of the transformation. On the other hand, we still consider that finding a reliable ksp, regardless of the transformation is an separate event. We first choose a preconditioner pc based on the features of A and use pc to transform A (i.e., $A' = pc(A)$), and then choose an iterative method ksp based on the features of A'. Using a similar example as in the orthogonal approach, to choose the composite method in this approach we have:

$$\Pi^{sequential}(\Phi(A)) = pc, ksp, \text{ such that } \Upsilon_{\mathcal{D}}(\Phi(A)) = pc \text{ and then} \Upsilon_{\mathcal{K}}(\Phi(pc(A))) = ksp,$$

[†]This is why it is important to note the difference in notation pc, ksp which implies that the behavior of the pc depends on the ksp (or vice versa), but the notation (pc, ksp) does not imply that in the recommendation we will consider that one affects the behavior of the other.

that is, first use the preconditioner classifier $\Upsilon_{\mathcal{D}}$ to choose pc and use it to transform A, then use the classifier $\Upsilon_{\mathcal{K}}$ to choose ksp based on the transformed features of A.

This approach is more accurate and takes on account the interaction between preconditioners and iterative methods, but it is expensive since we need to compute preconditioned features.

4.3.3 Conditional Approach

In the conditional approach we explicitly construct a classifier for a combination of a preconditioner and an iterative method pc, ksp as a unit, based on the features of A. In this case, to choose the composite method we have:

$$\Pi^{conditional}(\Phi(A)) = pc, ksp, \text{ such that } \Upsilon_{\mathcal{M}}(\Phi(A)) = pc, ksp,$$

where $\Upsilon_{\mathcal{M}}$ is the classifier function for composite methods.

It is not completely necessary to consider the selection of the transformation and the iterative method as independent events. Depending on the statement of the recommendation request it may or not be necessary to acknowledge the classification of an iterative method as an independent and additional event.

This approach is convenient because we do not compute preconditioned features required in the sequential approach or the classification of iterative methods alone. It is also reasonably accurate in many cases but could also lead to unreliable classifications.

4.4 Recommendation Issues in the Method Selection Problem

To make a recommendation we need to address the Reliability and Performance problems. The reliability problem concerns itself with the binary choice of classifying a method as converging or non-converging. The performance question is that of recommending the method that will lead to the fastest solution among a set of methods. (Other continuous measures than speed to solution are also possible.) In the performance problem we weigh the following issues in our recommendation of best method:

- What is the gain of choosing the best performing method (in terms of speed)?
- For every two methods that converge, which one is faster in more cases?
- By how much is a method faster compared to other methods (speedup factor)?
- From a numerical point of view, what is a good recommendation? For instance, compared to a default method choice (e.g., a direct method), can we recommend a faster method?

In our experiments, we define the suitability method as the time it takes to converge. In many cases, the time to apply a preconditioner ($setup_{time}$) is longer than the time to actually solve the system ($solve_{time}$). In order to prevent the setup time from obscuring the solution time we $solve_{time} * 10 + setup_{time}$ as the total performance measure. One of the reasons we do not use number of iterations is that we also intend to compare with a direct method which converges in "one single iteration" (however this iteration takes a long time). Similarly, in practice there are some methods which converge in many less iterations than others for the same type of problem,

however it is possible that those few iterations are very expensive. In such case, the total time to solution may actually be much longer than a method that uses more (but cheaper) iterations.

In general, the recommendation strategy for the performance or reliability of a method may be based on the following criteria, depending on the sensitivity of the problem or the desired result:

- Reliability: select a method that will converge even if it is slow, in this case the main goal is to recommend a method that will guarantee a solution.
- Performance: choose a method that is the best depending on the performance measure (e.g., smallest solving time, best accuracy, least number of iterations, etc.). For this type of recommendation, we might consider cases when it is previously known that the problem is well-behaved and we only require a fast solution, or the choice of a set of methods that are very likely to converge is the most suitable recommendation.
- Reliability and Performance: this is perhaps the hardest recommendation to make; it is a combination of the two previous types and consists of choosing a method that is both reliable and efficient.

Based on this we can decide what constitutes a better recommendation: choose a slower method that is more likely to converge, or choose a faster method even if there is a chance that it diverges. The best choice would be, of course, a reliable method that is also the fastest.

In the introduction to classification (Section 4.1) we mentioned that the input dataset is split into two subsets: training and test. However, it is common to split instead the dataset into three parts: training set, test set and what we will call *unseen* set. The *unseen* is a set that is excluded from the learning process and is used for further testing the constructed classifiers to see how well a classifier can make recommendations on new data, this process is called *generalization*. The name "unseen" comes from the idea that the classifiers have never seen this data before, it has not been part of the modeling of the discriminant functions, nor has it taken part in the derivation of the accuracies of prediction. The results of the recommendation stage are done based on this dataset and will be discussed in the experiments chapter, Sections 7.3.2 and 7.4.7.

Every time a recommendation is constructed we need to evaluate how good or bad it may be. How good a recommendation is not only depends on the accuracy with which we can predict each class, but also, on the gain or loss associated with choosing the predicted class. For example in the Performance problem, if we predict that a method is the optimal for solving problem A and this prediction is 90% accurate, it is also necessary to associate a penalty for the 10% chance that the prediction made was wrong; we will denote this as the penalty of missclassification. But it is important as well to associate a gain for the 90% possibility that the prediction is correct; this we will refer to as the gain of correct prediction.

We can base the calculation of the penalty of misprediction on the statistical errors described in Section 3.7. In our case, both problems of Reliability and Performance are associated with a performance measurement (time to solution) that determines if a method is the best solution to solve a problem. We can associate the classification errors with some loss of performance, for example, using a mispredicted method may result in waiting 10 hours to solve a problem, as opposed to waiting 1 hour if the optimal method had been chosen. In this sense, we can penalize the errors in terms of the additional waiting time (*performance loss*), and the more mistakes a classifier makes will result in a higher penalty.

On the other hand, we can also weight the ability of a classifier to predict the correct method. This is a *performance gain* that can be measured as the benefit of choosing an optimal method compared to a default (default methods are commonly used in numerical software). In our example, this means waiting less time.

These penalties are computed differently for the Reliability and Performance problems. However, they are determined based on the misclassification errors, and on the time loss. For the Reliability problem we use the number of false positive (FP) errors, and for the Performance problem we use the errors' data in the confusion matrix (described in Section 3.7.2).

Chapter 5

The Reliability Classification Problem

This chapter addresses the concepts and strategies concerned with the identification and recommendation of reliable solvers. This is probably the most critical part in the problem of choosing appropriate solvers, because we want to minimize as much as possible the risk of choosing an unreliable solver that will iterate forever. The first section contains specific definitions for this problem, and incorporates some of the statistical techniques from Chapter 3 to develop strategies for classification and recommendation. Section 5.2 describes in detail the algorithms that constitute the various classifying and recommending strategies for the problem of finding the best method in terms of *time to convergence*.

In Chapter 4 we stated the need to address two main problems – *reliability* and *performance*. This chapter concerns itself with the definition and description of the *reliability* problem.

In performance classification, there are three possible recommendation outputs: iterative method, transformation, a combination of these two. A recommendation is built using the classifier functions Υ. For the recommendation of iterative methods we need Υ_{ksp} (where $ksp \in \mathbb{K}$), and for preconditioners Υ_{pc} (where $pc \in \mathbb{D}$). For combination methods, we need to use both the classifiers for iterative methods and for preconditioners.

The algorithms that describe how set up and derive the classifiers are derived will be denoted as "Construction" algorithms. Those that describe the usage of the derived classifiers to form a recommendation will be known as "Recommendation" algorithms.

5.1 Concepts and Definitions

The general problem of *reliability classification* consists of determining which methods converge and which don't for any particular problem (observation) $F \in \mathbb{F}$. If we define the class space as $\mathbb{C} = \{converge, diverge\}$, then the classes represent the sets of those observations for which a method converged or diverged respectively. More specifically, these classes are defined for each method M as $converge_M$ and $diverge_M$, so when M is used to solve a problem A if M converged then F_i is assigned to $converge_M$, otherwise it is assigned to $diverge_M$.

Definition 18. *Let $M \in \mathbb{M}$. In the reliability problem, the class space for M is defined as*

$$\mathbb{C} = \{converge_M, diverge_M\}$$

where

$$converge_M = \{F|\, F = \Phi(A),\ A \in \mathring{A} \textit{ and } T(A, M) \neq \infty\}$$
$$diverge_M = \{F|\, F = \Phi(A),\ A \in \mathring{A} \textit{ and } T(A, M) = \infty\} \tag{5.1}$$

And the reliability classifier function for a particular method $M \in \mathbb{M}$ is:

$$\Upsilon_M : \mathbb{F} \mapsto \{convergence_M, divergence_M\} \tag{5.2}$$

such that for every $F \in \mathbb{F}$, $\Upsilon_M(F) = convergence_M$ when M applied to F converges and $divergence_M$ otherwise. And the corresponding accuracy evaluating function

$$\Omega_M(F) = \langle \alpha_{convergence_M}, \alpha_{divergence_M} \rangle. \tag{5.3}$$

5.1.1 Classification of Iterative Methods

To determine which iterative method is the most reliable or the fastest, we use as input observations the set of feature vectors resulting from preconditioned (and preprocessed) matrices. In section 4.2 we defined $\mathbb{M}$ as the set of methods that potentially solve a problem A. $\mathbb{M}$ includes all possible combinations of transformations, preprocessing steps and *ksp*s. In this particular problem we refer to the subset $\mathbb{K} \subset \mathbb{M}$ that includes *only* iterative methods. This implies that the observations considered for this experiment consist of the feature vectors extracted from preprocessed and preconditioned matrices. The class space is defined like in Definition 18 but using $\mathbb{K}$. It is also appropriate to consider the set of (possibly) preconditioned problems and its features, since the preconditioners may have an effect on the behavior of some ksps.

Let $ksp \in \mathbb{K}$ be an iterative method. In the reliability problem, the class space for ksp is

$$\mathbb{C} = \{converge_{ksp}, diverge_{ksp}\}$$

where

$$converge_{ksp} = \{F|\, F = \Phi(D(A)),\ A \in \mathcal{A} \text{ and } T(D(A), ksp) < \infty\ \}$$
$$diverge_{ksp} = \{F|\, F = \Phi(D(A)),\ A \in \mathcal{A} \text{ and } T(D(A), ksp) = \infty\}$$

and $D \in \mathbb{D}$ is a preconditioner. The classifier function for the reliability of ksp can be written as follows based on Equation 5.2:

$$\Upsilon_{ksp} : \mathbb{F} \mapsto \mathbb{C} \tag{5.4}$$

such that $\forall\, F \in \mathcal{F}$

$$\Upsilon_{ksp}(F) = \begin{cases} converge_{ksp} & \text{if } ksp \text{ converged for } F \\ diverge_{ksp} & \text{otherwise} \end{cases}$$

where $\mathcal{F} \subset \mathbb{F}$ is the set of preconditioned features $\mathcal{F} = \Phi(D(\mathcal{A}))$.

And the accuracy evaluation function for $ksp \in \mathbb{K}$ is defined as $\Omega_{ksp} : \mathbb{F} \mapsto \mathbb{Y}$ such that $\forall F \in \mathbb{F}$

$$\Omega_{ksp}(F) = \langle \alpha_{converge_{ksp}}, \alpha_{diverge_{ksp}} \rangle \tag{5.5}$$

The recommendation function, which determines the best choice of method for this case is:

Definition 19. *Let $\mathcal{K} \subset \mathbb{K}$ be a set of available iterative methods, $ksp \in \mathbb{K}$ and $F \in \mathcal{F}$. The method picking function is given by*

$$\Pi_{ksp}(F) = \mathcal{K} \equiv \forall ksp \in \mathcal{K} \ \Upsilon_{ksp}(F) = converge_{ksp}$$

where $\mathcal{F} \subset \mathbb{F}$ and $\mathcal{F} = \Phi(D(\mathcal{A}))$ where $D \in \mathbb{D}$

5.1.2 Classification of Transformations

This problem considers the case when we want to know if a transformation is reliable or not, regardless of the iterative method being used.

The criteria for convergence and divergence in the transformations case is different than the one used for the methods. For a problem $A \in \mathbb{A}$ and a transformation $D \in \mathbb{D}$:

D is said to converge $\iff \exists M \in \mathbb{M}$ that converges for $F = \Phi(D(A))$

D is said to diverge $\iff \forall M \in \mathbb{M}$ diverges for $F = \Phi(D(A))$

Using these criteria and following the same idea as we did for iterative methods, we can define the convergence and divergence classes.

Let $D \in \mathbb{D}$ be a transformation. The class space for the reliability problem for D is $\mathbb{C}_D = \{converge_D, diverge_D\}$ where

$$converge_D = \{F | F = \Phi(A) \ s.t. \exists \ ksp \in \mathbb{K} \ s.t. \ T(D(A), ksp) < \infty \text{ where } A \in \mathcal{A}\}$$

$$diverge_D = \{F | F = \Phi(A) \ s.t. \forall ksp \in \mathbb{K} \ s.t. T(D(A), ksp) = \infty \text{ where } A \in \mathcal{A}\}$$

and $F \in \mathcal{F}$ where $\mathcal{F} \subset \mathbb{F}$ is the set of original feature vectors (non-preconditioned) from $\mathcal{A}$.

However, we have found from numerous experiments that this definition cannot be used in all the cases. Unlike the classification of iterative methods which can be done independently from other transformations, the classification of certain transformations requires to evaluate the classification with respect to each particular method.

Theorem 2. *It is not possible to determine if a transformation $D \in \mathbb{D}$ converges or not for a problem $A \in \mathbb{A}$ using the class space $\mathbb{C}_D = \langle converge_D, diverge_D \rangle$ and non-transformed feature vectors.*

Proof. By counter example. Let $pc \in \mathbb{D}$ be a preconditioner transformation. Suppose for instance we apply different preconditioners to a problem A and we run two *ksp*s on the preconditioned systems. For a particular problem A_j preconditioned with pc_i, the iterative method ksp_1 may converge while ksp_2 does not. In terms of classification this creates a conflict, we have the same observation (corresponding to A_j) in both *convergence* and *divergence* classes for pc_i. This not only contradicts the *class space* Definition 12, but also makes impossible for classifiers to distinguish between the classes because observations could appear in both classes. □

This issue is also present in the other transformations, so we need to devise a different approach for these classification problems. We will refer to this approach as *conditional classification* and it is based on the fact that the behavior of many preconditioners is highly dependent on the iterative method used and vice versa.

The type of observations considered for this approach consist solely of *unpreconditioned* (non-transformed) feature vectors from the set of problems $\mathcal{A}$. The classes are formed by the set of observations for which a ksp converged (or not), given that A was first preconditioned using pc (i.e., $ksp(A')$ where $A' = pc(A)$).

Definition 20. *Let $pc \in \mathbb{D}$, $ksp \in \mathbb{K}$, and $pc, ksp \in \mathbb{M}$. The class space for the conditional classification of pc with ksp is given by the union of the following two classes:*

$$converge_{pc,ksp} = \{F | F = \Phi(A) \text{ such that } T(pc(A), ksp) < \infty \text{ where } A \in \mathbb{A}\}$$

$$diverge_{pc,ksp} = \{F | F = \Phi(A), \text{ such that } T(pc(A), ksp) = \infty \text{ where } A \in \mathbb{A}\}$$

and $F \in \mathbb{F}_{pc}$, where $\mathbb{F}_{pc}$ is the set of unpreconditioned feature vectors extracted from $\mathbb{A}$.

Definition 21. *Let $pc \in \mathbb{D}$, $ksp \in \mathbb{K}$ and $\mathbb{C} = \{converge_{pc,ksp}, diverge_{pc,ksp}\}$. The reliability classifier function for the transformation pc given the iterative method ksp is $\Upsilon_{pc,ksp} : \mathbb{F}_{pc} \mapsto \mathbb{C}$, such that $\forall\, F \in \mathbb{F}_{pc}$*

$$\Upsilon_{pc,ksp}(F) = \begin{cases} converge_{pc,ksp} & \text{if } ksp \text{ converges for } F = \Phi(pc(A)) \\ diverge_{pc,ksp} & \text{otherwise} \end{cases}$$

where $\mathbb{F}_{pc}$ is the set of unpreconditioned feature vectors.

To compute the classification for each pair of pc and ksp, we define the corresponding accuracy evaluating function:

Definition 22. *Let $pc \in \mathbb{D}$ and $ksp \in \mathbb{K}$. The accuracy evaluation function for the classification of the conditional combination pc, ksp is $\Omega_{pc,ksp} : \mathbb{F} \mapsto \mathbb{Y}$ such that $\forall F \in \mathbb{F}$*

$$\Omega_{pc,ksp}(F) = \langle \alpha_{converge_{pc,ksp}}, \alpha_{diverge_{pc,ksp}} \rangle$$

After obtaining the classification and accuracy evaluation for a particular pc with each available ksp (conditional classes), we "merge" the results to evaluate the pc's overall behavior. The main idea of this strategy is to pair a particular pc with every available ksp in $\mathbb{K}$ to form the conditional cases. The results of each conditional combination class where pc appears, are then merged to form a global result for that pc as we will explain in detail next.

We first designate a numeric value for the convergence or divergence of a pc, ksp class for $F \in \mathbb{F}$:

$$\beta_{\Upsilon_{pc,ksp}}(F) = \begin{cases} 1 & \text{if } \Upsilon_{pc,ksp}(F) = converge_{pc,ksp} \\ 0 & \text{if } \Upsilon_{pc,ksp}(F) = diverge_{pc,ksp} \end{cases} \tag{5.6}$$

where $F = \Phi(A)$ for $A \in \mathbb{A}$. We then create a value to measure the *convergence percentage* ($convp_{pc}$) of the pc:

$$convp_{pc}(F) = \frac{\sum_{\forall ksp \in \mathbb{K}} \beta_{\Upsilon_{pc,ksp}}(F)}{|\mathbb{K}|} \times 100 \tag{5.7}$$

Lastly, the classifier function for the preconditioner is constructed as follows:

$$\Upsilon_{pc}(F) = \begin{cases} converge_{pc} & \text{if } convp_{pc}(F) \geq threshold \\ diverge_{pc} & \text{if } convp_{pc}(F) < threshold \end{cases} \tag{5.8}$$

where $threshold$ is an experimentally determined heuristic to quantify the overall convergence of pc.

The accuracy evaluation for the convergence and divergence of a pc is processed in a similar manner. It is computed as the average of the accuracies of the individual conditional classes:

$$\begin{aligned} \alpha_{converge_{pc}} &= \frac{\sum_{ksp \in \mathbb{K}} \alpha_{converge_{pc,ksp}}}{|\mathbb{K}|} \\ \alpha_{diverge_{pc}} &= \frac{\sum_{ksp \in \mathbb{K}} \alpha_{diverge_{pc,ksp}}}{|\mathbb{K}|} \end{aligned} \tag{5.9}$$

For example, suppose that $pc = ilu$ and that $KSP = \{gmres, tfqmr\}$. The conditional classes would then be $ilu, gmres$ and $ilu, tfqmr$ with the corresponding accuracies $\alpha_{\langle ilu,gmres \rangle}$ and $\alpha_{ilu,tfqmr}$. The overall accuracy for ilu would be the average of these two amounts. The accuracy evaluation function for a precondtioner would be

$$\Omega_{pc}(F) = \langle \alpha_{converge_{pc,ksp}}, \alpha_{diverge_{pc,ksp}} \rangle \tag{5.10}$$

With the accuracy evaluation function we can obtain the accuracies for every possible combination of preconditioner and iterative method.

The method picking function, which determines the best choice of transformation in terms of reliability is:

$$\Pi_{pc}(F) = \mathcal{D} \equiv \forall pc \in \mathcal{D} \Upsilon_{pc}(F) = converge_{pc} \tag{5.11}$$

where $\mathcal{D} \subset \mathbb{D}$ is a set of preconditioners, $pc \in \mathbb{D}$, $\mathbb{F}_{pc}$ is the set of unpreconditioned feature vectors and $F \in \mathbb{F}_{pc}$.

5.2 Algorithms for Classification and Recommendation

This section we revisit various definitions from the previous section. These are incorporated into algorithms that overview the construction of classifier and recommendation functions for iterative methods and/or preconditioners.

We consider three types of recommendations: iterative method, preconditioner, or combination of both (e.g., composite methods). For the case of iterative methods and preconditioners, we will describe the algorithms to derive the classifiers and to construct the recommendation (using the derived classifiers). For combinations we only contemplate the recommendation function implementation. There is no need to construct an additional classifier for combinations since we can use the classifiers that have been derived for the iterative methods and preconditioners separately.

5.2.1 Algorithms for Iterative Methods

This considers the case where we only want to know which iterative methods are reliable for a problem, regardless of the preconditioner (or transformations) used. Note that for this particular

problem, it is possible that the input is a matrix of the type A, or a preconditioned matrix $A' = pc(A)$.

Construction of Classifiers for Iterative Methods

This algorithm describes the steps required to derive a classifier function for an iterative method ksp. The classifier can later be used to select converging methods, discard diverging ones, and form recommendations. Recalling from Section 4.1, a classifier is constructed based on decision rule. In the reliability problem, it is necessary to derive such a decision rule for each of the iterative methods. Thus, the following algorithm has to be applied for every $ksp \in \mathbb{K}$:

for each $ksp \in \mathbb{K}$

apply Algorithm 3 on $\mathbb{A}$ *using* ksp *as input*

Although the application of this algorithm is expensive, it is performed only once to train the learning system (or whenever new information needs to be added to a knowledge database).

Algorithm 3. *Derive Reliability Classifiers for Iterative Methods*

Input: *a set of problems* $\mathbb{A}$ *and an iterative method* $ksp \in \mathbb{K}$

Output: *a classifier function (discriminant)* Υ_{ksp}*, and a vector of accuracies* Y_{ksp} *in Equation 4.6*

1. *Obtain the feature space as* $\mathbb{F} = \Phi(\mathbb{A})$
2. *As an optional step, transform the feature space using PCA as described in Section 3.6:* $\mathbb{F}' = \Gamma(\mathbb{F})$ *and make* $\mathbb{F} = \mathbb{F}'$
3. *Randomly split the feature space into training* ($\mathbb{F}_{tr}$) *and test* ($\mathbb{F}_{te}$) *sets*
4. *Using* $\mathbb{F}_{tr}$*, derive a discriminant function* Υ_{ksp} *(a set of parameters that describe a probability density function) using a statistical learning methodology such as:*

   Kernel Mixture

   Decision Trees

   K-means Clustering

5. *Using* $\mathbb{F}_{te}$*, compute the accuracies for each of the two classes using* Ω_{ksp} *from Definition 15 to form* Y_{ksp}

Construction of Recommendations for Iterative Methods

This algorithm describes the way of using the classifiers Υ to form a set of reliable methods for a given problem. This implements the iterative method picking function (or recommendation function) $\Pi_{ksp}(F)$ (Equation 5.4). Observe that if the given problem A is not preconditioned, this algorithm is very expensive because it would be necessary to precondition A with every preconditioner to evaluate the overall performance of each ksp. However, the applicability of this particular algorithm is for problems A that are already preconditioned, and the main requisite is to find a set of reliable iterative methods.

Algorithm 4. *Use of Reliability Classifiers for Iterative Methods*

Input: *a problem $A \in \mathbb{A}$, a classifier function Υ_{ksp} and a vector of accuracies Y_{ksp} for each $ksp \in \mathbb{K}$*

Output: *a set of iterative methods $\mathcal{K} \subseteq \mathbb{K}$ that converge for A*

initialize $\mathcal{K} = \{\}$ and $Q = \{\}$

for each $ksp \in \mathbb{K}$

if A is not preconditioned then

for each $D \in \mathbb{D}$

extract the preconditioned features $F = \Phi(D(A))$

if $\Upsilon_{ksp}(F) = converge_{ksp}$ then

$\mathcal{K} = \mathcal{K} \cup \{ksp\}$

obtain $accuracy_{converge_{ksp}}$ from Y_{ksp} (Definition 4.6)

$Q = Q \cup \{accuracy_{ksp}\}$

else

extract the features $F = \Phi(A)$

if $\Upsilon_{ksp}(F) = converge_{ksp}$ then

$\mathcal{K} = \mathcal{K} \cup \{ksp\}$

obtain $accuracy_{converge_{ksp}}$ from Y_{ksp} (Definition 4.6)

$Q = Q \cup \{accuracy_{ksp}\}$

output $\Pi_{\mathbb{K}}(F) = \mathcal{K}$ and Q

5.2.2 Algorithms for Transformations

This is the case when we are interested only in the reliability of a transformation (preconditioner), independently of the iterative method being used. In order to simplify the description of the algorithms, we will limit the set of transformations to preconditioners as we have done in previous chapters. The problem A in this case is not preconditioned, so the features extracted are in their original form. Remember from Section 5.1.2 that in order to analyze a preconditioner, it is necessary to account for each of the individual combinations of the desired preconditioner with each iterative method and then merge these results.

Construction of Classifiers for Preconditioners

This algorithm describes the process to derive a classifier function for a preconditioner pc. In the reliability problem, it is necessary to construct a classifier for each preconditioner, like t was done for iterative methods. Thus it is necessary to apply this algorithm to every preconditioner to derive each of the corresponding Υ functions.

Algorithm 5. *Derive Reliability Classifiers for Preconditioners*

Input: *a set of problems $\mathbb{A}$ and preconditioner $pc \in \mathbb{D}$*

Output: *a classifier function (discriminant) Υ_{pc}, a vector of accuracies Y_{pc} as in Definition 4.6*

1. *Obtain the feature space as $\mathbb{F} = \Phi(\mathbb{A})$*
2. *As an optional step, transform the feature space using PCA as described in Section 3.6: $\mathbb{F}' = \Gamma(\mathbb{F})$, and make $\mathbb{F} = \mathbb{F}'$*
3. *Using $\mathbb{F}$, derive a discriminant function Υ_{pc} using the conditional combinations approach described in Definition 17*
4. *For each $ksp \in \mathbb{K}$*

    Create a conditional class for pc, ksp

    Derive a discriminant function $\Upsilon_{pc,ksp}$ (a set of parameters that describe a probability density function) using a statistical learning methodology such as:

        Kernel Mixture

        Decision Trees

        K-means Clustering

    Compute the accuracies for each of the two classes with $\Omega_{pc,ksp}$ from Definition 15 to form $Y_{pc,ksp}$.
5. *Merge the results for conditional $\Upsilon_{pc,ksp}$ using the steps described in Equations 5.6, 5.7, 5.8*
6. *Compute the accuracies for each of the two classes using Ω_{pc} (Definition 22).*

Construction of Recommendations for Preconditioners

This algorithm implements the preconditioner picking function $\Pi_{\mathbb{D}}(F)$ (Equation 5.8) to obtain the resulting set of reliable preconditioners. This is also the recommendation process.

Algorithm 6. *Use of Reliability Classifiers for Preconditioners*

Input: *a problem $A \in \mathbb{A}$, a classifier function Υ_{pc} and a vector of accuracies Y_{pc} for each $pc \in \mathbb{D}$*

Output: *a set of preconditioners $\mathcal{D} \subseteq \mathbb{D}$ for which some iterative method $ksp \in \mathbb{K}$ converges for F*

initialize $\mathcal{D} = \{\}$ and $Q = \{\}$

extract the features $F = \Phi(A)$

for each $pc \in \mathbb{D}$

    if $\Upsilon_{pc}(F) = converge_{pc}$ then
        $\mathcal{K} = \mathcal{D} \cup \{pc\}$
        obtain $accuracy_{converge_{pc}}$ from Y_{pc} (Definition 4.6)
        $Q = Q \cup \{accuracy_{pc}\}$

output $\Pi_{\mathbb{D}}(F) = \mathcal{D}$ and Q

5.3 Strategies for Combinations of Iterative Methods with Transformations

The algorithms for these approaches have the same input and output, but the construction of the function Π and the way of obtaining the accuracy differs for each of them. The main difference between these algorithms lies in the use of the classifier functions and the way the recommendation is built. We will describe the algorithm for each approach in the following sections.

5.3.1 Orthogonal Approach

To find the accuracy of classification α for the convergence and divergence classes we do:

$$\alpha_{converge_{(pc,ksp)}} = \alpha_{converge_{pc}} \times \alpha_{converge_{ksp}}$$

$$\alpha_{diverge_{(pc,ksp)}} = \alpha_{diverge_{pc}} \times \alpha_{diverge_{ksp}}$$

The method picking function is then constructed:

Definition 23. *Let $pc \in \mathbb{D}$, $ksp \in \mathbb{K}$, $\mathcal{D} \subseteq \mathbb{D}$, $\mathcal{K} \subseteq \mathbb{K}$, and let $F \in \mathbb{F}$. The function for choosing pc and ksp is*

$$\Pi^{orthogonal}(F) = \mathcal{D} \cup \mathcal{K} \equiv \forall pc \in \mathcal{D} \quad \Upsilon_{pc}(F) = converge_{pc} \\ \text{and } \forall ksp \in \mathcal{K} \quad \Upsilon_{ksp}(pc(F)) = converge_{ksp}$$

where $\mathbb{F} = \Phi(\mathbb{A})$.

Orthogonal Approach Algorithm

We assume that picking a pc is independent from picking a ksp (and that, numerically, one has no effect on the behavior of the other). It is called "orthogonal" because we classify and pick first the pc and then then ksp, separately. This algorithm implements the recommendation function $\Pi^{orthogonal}$ from Definition 37.

In this approach, it is important to differentiate between the *theoretical* version and the *practical* version. A *theoretical* version considers that in order to evaluate the overall behavior of a ksp it is required to analyze its behavior with all the available preconditioners. To do this, it is necessary to obtain the preconditioned features of the matrix A with each preconditioner, which is a very expensive process. For this reason, the *practical* versions contemplate the use of a reduced set of preconditioners.

Algorithm 7. *Use of Reliability Classifiers for Combination methods: Orthogonal*

> ***Input:*** *a problem $A \in \mathcal{A}$, a set of classifier functions Υ and a set of vectors of accuracies Y for each method and/or preconditioner*
>
> ***Output:*** *a set of converging combinations of iterative method and preconditioner subset of $\mathbb{D} \times \mathbb{K}$ expressed in the (pc, ksp) format, an accuracy of classification for each (pc, ksp)*
>
> *initialize $\mathcal{D} = \{\}$, $\mathcal{K} = \{\}$ and $Q = \{\}$*

extract the features $F = \Phi(A)$

for each $pc \in \mathbb{D}$

if $\Upsilon_{pc}(F) = converge_{pc}$ *(Definition 5.8) then*

$\mathcal{D} = \mathcal{D} \cup \{pc\}$

obtain $\alpha_{converge_{pc}}$ *from* Y_{pc} *(Definition 4.6)*

for each $D \in \mathcal{D}$ *(or* $D \in \mathbb{D}$ *for Theoretical algorithm)*

extract the preconditioned features $F = \Phi(D(A))$

for each $ksp \in \mathbb{K}$

if $\Upsilon_{ksp}(F) = converge_{ksp}$ *then*

$\mathcal{K} = \mathcal{K} \cup \{ksp\}$

obtain $\alpha_{converge_{ksp}}$ *from* Y_{ksp} *(Definition 4.6)*

$\alpha_{(pc,ksp)} = \alpha_{converge_{pc}} \times \alpha_{converge_{ksp}}$

$Q = Q \cup \{\alpha_{(pc,ksp)}\}$

output $\Pi^{orthogonal} = \mathcal{D} \times \mathcal{K}$ *and* Q

5.3.2 Sequential Approach

The accuracies for the resulting combination (pc, ksp) are then obtained as follows

$$\alpha_{converge_{(pc,ksp)}} = \alpha_{converge_{pc,ksp}} \times \alpha_{converge_{ksp}}$$

$$\alpha_{diverge_{(pc,ksp)}} = \alpha_{diverge_{pc,ksp}} \times \alpha_{diverge_{ksp}}$$

The method picking function is:

Definition 24. *Let* $pc \in \mathbb{D}$, $ksp \in \mathbb{K}$, $\mathcal{K} \in \mathbb{K}$. *Furthermore, let* $\mathcal{M} \subseteq \mathbb{M}$ *be a set of combined methods of the type* pc, ksp, *and* $F \in \mathbb{F}$. *The function for choosing* pc *and* ksp *is*

$$\Pi^{sequential}(F) = \mathcal{M} \equiv \forall pc, ksp \in \mathcal{M}\ \Upsilon_{pc,ksp}(F) = converge_{pc,ksp} \text{ and } \forall ksp \in \mathcal{K}\ \Upsilon_{ksp}(pc(F)) = converge_{ksp}$$

where $\mathbb{F} = \Phi\mathbb{A}$.

Sequential Approach Algorithm

In this case we assume that picking a combined method pc, ksp is actually independent from picking *only* the ksp. In order to use pc, ksp we must first choose a ksp. We can pick a combination but we still need to consider the probability of actually picking a converging ksp. It is called simultaneous because we are picking pc, ksp at once (however, we then choose a ksp). This algorithm implements the recommendation function $\Pi^{simultaneous}$.

Algorithm 8. *Use of Reliability Classifiers for Combination methods: Sequential*

Input: *a problem $A \in \mathcal{A}$, a set of classifier functions Υ and a set of vectors of accuracies Y for each method and/or preconditioner*

Output: *a set of converging combinations of iterative method and preconditioner subset of $\mathbb{D} \times \mathbb{K}$ expressed in the (pc, ksp) format, an accuracy of classification for each (pc, ksp)*

initialize $\mathcal{M} = \{\}$ and $Q = \{\}$

extract the features $F = \Phi(A)$

for each $\langle pc, ksp \rangle \in \mathbb{D} \times \mathbb{K}$

if $\Upsilon_{\langle pc,ksp \rangle}(F) = converge_{\langle pc,ksp \rangle}$ (Definition 21) then
$\mathcal{M} = \mathcal{M} \cup \{\langle pc, ksp \rangle\}$
from $\langle pc, ksp \rangle$ take the iterative method's name ksp
use Algorithm 4 to obtain $\alpha_{converge_{ksp}}$ from Y_{ksp} (Definition 4.6)
$\alpha_{(pc,ksp)} = \alpha_{converge_{\langle pc,ksp \rangle}} \times \alpha_{converge_{ksp}}$
$Q = Q \cup \{\alpha_{(pc,ksp)}\}$

output $\Pi^{sequential} = \mathcal{M}$ and Q

5.3.3 Conditional Approach for Combination Methods

This approach is exactly the same as the *conditional approach* for transformations, except that we don't carry out the "merging" part to obtain the overall information for a pc. Instead, we focus on each of the combinations of the pc with the various ksps and treat these as if they were actually methods in $\mathbb{M}$ of the form pc, ksp. The accuracy evaluating function becomes simply

$$\Omega_{(pc,ksp)}(F) = \Omega_{pc,ksp}(F).$$

and the computation of the respective accuracies is

$$\alpha_{converge_{(pc,ksp)}} = \alpha_{converge_{pc,ksp}}$$

$$\alpha_{diverge_{(pc,ksp)}} = \alpha_{diverge_{pc,ksp}}$$

Experimental results show high accuracy of classification by using pc, ksp on its own. The method picking function Π for this approach is as follows:

Definition 25. *Let $\mathcal{M} \subset \mathbb{M}$ be a set of combined methods of the type pc, ksp, where $pc \in \mathbb{D}$ and $ksp \in \mathbb{K}$, and let $F \in \mathbb{F}_{pc}$*

$$\Pi^{conditional}(F) = \mathcal{M} \;\equiv\; \forall pc, ksp \in \mathcal{M} \;\; \Upsilon_{pc,ksp}(F) = converge_{pc,ksp}$$

where $\mathbb{F}_{pc}$ is the set of unpreconditioned feature vectors.

Conditional Approach

This approach assumes that the process of picking a pc is dependent on that of picking a ksp (and vice versa). This implies that the behavior of a pc is conditional based upon the effect of a ksp and conversely. The conditional approach is convenient, not only we don't need to compute preconditioned features for choosing ksps like we did in the sequential approach, but it also turned out to be very accurate on its own. This algorithm implements the recommendation function $\Pi^{conditional}$

Algorithm 9. *Use of Reliability Classifiers for Combination methods: Conditional*

> ***Input:*** *a problem $A \in \mathcal{A}$, a set of classifier functions Υ and a set of vectors of accuracies Y for each method and/or preconditioner*
>
> ***Output:*** *a set of converging combinations of iterative method and preconditioner subset of $\mathbb{D} \times \mathbb{K}$ expressed in the (pc, ksp) format, an accuracy of classification for each (pc, ksp)*
>
> *initialize $\mathcal{M} = \{\}$ and $Q = \{\}$*
>
> *extract the features $F = \Phi(A)$*
>
> *for each $\langle pc, ksp \rangle \in \mathbb{D} \times \mathbb{K}$*
>
> > *if $\Upsilon_{\langle pc,ksp \rangle}(F) = converge_{\langle pc,ksp \rangle}$ (Definition 21) then*
> > $\mathcal{M} = \mathcal{M} \cup \{\langle pc, ksp \rangle\}$
> > *obtain $\alpha_{converge_{\langle pc,ksp \rangle}}$ from $Y_{\langle pc,ksp \rangle}$ (Definition 4.6)*
> > $\alpha_{(pc,ksp)} = \alpha_{converge_{\langle pc,ksp \rangle}}$
> > $Q = Q \cup \{\alpha_{(pc,ksp)}\}$
>
> *output $\Pi^{conditional} = \mathcal{M}$ and Q*

This approach is convenient for the Reliability problem because it is not necessary to compute preconditioned features of the problems. The accuracy of classification is comparable to the sequential approach, which in turn requires the full analysis with preconditioned features making the process very expensive. The downside of this strategy is that sometimes there may be very little data in some of the classes. In our experiments, this has not posed a big issue, specially since the data in the convergence classes has been separated very well from the corresponding divergence classes. However, in the Performance problem (which we will cover in the next chapter) where the comparison is done between all the available methods, using conditional combinations makes the discrimination between classes very difficult.

5.4 Recommendation Evaluation

The recommendation for the Reliability problem consists of a set of methods that are likely to converge for solving problem A. As a part of the recommendation we need to evaluate the "risk" of choosing each method. In Section 3.7.1 we presented the types of statistical errors for a binary classification problem. We can use these concepts in the assessment of classification and recommendation results.

The penalty of misclassification is dependent on the action we take once the class membership has been determined for a matrix. The price of choosing a diverging method would be to iterate forever or to get a completely wrong solution, but discarding a converging method under the idea that it diverges means that we can choose another possibly slower method. Obviously the first mistake is more costly than the second one, so the associated penalty should be higher; this error is also a false positive.

For a method M we have defined our set of classes as $converge_M$ and $diverge_M$. The worst action to take is to recommend or use a method that diverges when the classifier has mistakenly said that it converges, so the errors with respect to class $convergence_M$ are:

FP: saying the method converges when in reality it diverges

FN: saying the method diverges when in reality it converges

Now we need to incorporate this error and its associated penalty into our decision making process.

The Bayes methodology provides a convenient approach to incorporate the notions of gain and penalty directly into the formulation of the classifiers and the decision rule. We can define two mutually exclusive actions to take for a method:

- a_1: run the method
- a_2: discard the method

Let us express the set of classes as ω, and define a penalty λ, which is associated with taking a particular action in a particular situation (e.g. running a method, given that it diverges): $\lambda(a_i|\omega_j)$. We can then express the risk R of taking the action a_i as follows:

$$R(a_i|F) = \sum_{c}^{j=1} \lambda(a_i|\omega_j) P(\omega_j|F)$$

where F is a given problem (vector of features in our case), and P is the computed posterior probability. To simplify the notation, we can express the penalties as λ_{ij}. Note that for a two-class problem like this one, the total penalty is given by $\lambda_{ij} + \lambda_{ii}$, but λ_{ii} is zero because we do not assign a penalty for making the right choice.

This is very convenient for those classifiers using Bayesian methodology, but since we also use other approaches like Decision Trees, we can use these penalty ideas to "weight" a recommendation. For example, the cases when the error is minimized we can have better recommendations from a classifier.

The challenge here becomes to choose an appropriate metric for the associated penalties in terms of the weight we want to give to each type of error. The proposed metric for this penalty is in terms of the maximum amount of time we are willing to wait to determine convergence as a *slowdown* or performance loss factor. In general, we define *slowdown factor* as follows:

$$slowdown = \frac{\text{time to solve } A \text{ using a slower method}}{\text{time to solve } A \text{ using a faster method}}. \tag{5.12}$$

In the Reliability problem, the time for the "slower" method is the maximum waiting time to convergence, and the time for the "faster" method could be that associated to the optimal method

or to any other converging method. Although the time-to-wait for divergence is determined by the particular stopping criteria used in the implementation of the iterative solvers, the slowdown should be a factor big enough to represent the maximum waiting time (in practice, this is perhaps comparable to the time for a very slow method to converge). This factor can be determined with the aid of preliminary experimental results; for example, in our experiments we have found that the highest slowdown factor for a method is 16 (a method is at most 16 times slower than any another one), therefore we assign 30 as the the penalty for an FP *.

The slowdown and number of FP errors can then be used to weight how good a recommendation is for a method M. We define a metric $\mathcal{Q}$ to measure how good is a recommendation using the accuracy of classification, the error and the slowdown:

$$\mathcal{Q}^{reliability} = \frac{\alpha_M}{slowdown \times FP_{ratio}}, \tag{5.13}$$

and the smaller $\mathcal{Q}^{reliability}$ is, the better the recommendation. In our experiments, the slowdown is constant for a test, so we can simplify Equation 5.13:

$$\mathcal{Q}^{reliability} = \frac{\alpha_M}{FP_{ratio}}. \tag{5.14}$$

The penalty for a FN error is given by the slowdown factor computed as

$$slowdown = \frac{\text{time to solve } A \text{ using the predicted converging method}}{\text{time to solve } A \text{ using the optimal method}}. \tag{5.15}$$

Note however, that when a Reliability recommendation is issued is not possible to determine whether a "converging" method is the optimal or not, and if not, how slow it is. The analysis of this type of error falls into the area of the Performance problem and it will be addressed in Section 6.3. The worse case is when the converging method is the slowest, and the best case is when the converging method is actually the optimal, in which case the slowdown factor is 1, in such case FN is not really an error.

*In our experiments we have determined these factors experimentally from the performance information from our available datasets, but the computation of this factor should be automated as a part of the adaptivity of the recommendation system over time

Chapter 6

The Performance Classification Problem

This chapter describes in detail the algorithms that constitute the various classifying and recommending strategies for the problem of finding the best method in terms of *time to convergence*. The first Section contains specific definitions for this problem, and incorporates some of the statistical techniques from Chapter 3 to develop strategies for classification and recommendation. Section 6.2 covers the description of the specific strategies and algorithms developed for this problem.

In Chapter 4 we stated the need to address two main problems: Reliability and Performance. This chapter concerns itself with the definition and description of the Performance problem. Like in the Reliability problem, there are three types of classification: iterative method, preconditioner, a combination of these two. A recommendation is built using the classifier functions Υ. For the recommendation of iterative methods we need Υ_{ksp} (where $ksp \in \mathbb{K}$), and for preconditioners Υ_{pc} (where $pc \in \mathbb{D}$). For combinations of these, we need to use both the classifiers for iterative methods and for preconditioners.

The algorithms that describe how set up and derive the classifiers will be denoted as classifier "Construction" algorithms. Those that describe the usage of the derived classifiers to form a recommendation will be known as "Recommendation" algorithms.

6.1 Concepts and Definitions

The performance problem consists of finding which method converges the fastest for a given problem (this implies that only cases where the methods converged are considered). There are always two or more classes, one for each method that is being compared. The set of methods taken into consideration is $\mathcal{M} \subseteq \mathbb{M}$ (Definition 5), for iterative methods and transformations in particular we use $\mathbb{K}$ and $\mathbb{D}$ respectively. A *performance class* $C_M \in \mathbb{C}$ is the set of problems (observations) for which the method M was the fastest . The cardinality of the performance class space $\mathbb{C}$ is the number of methods considered $|\mathcal{M}|$; i.e., there is one class per method. Similarly, given a transformation D the class C_D is defined as the set of problems transformed with D for which some method was the fastest.

In general, the classes and the class space for the performance classification problem is defined as follows:

Definition 26. *Let $A \in \mathbb{A}$, let I be the index set of methods and $i, j \in I$. For each method $M_i \in \mathbb{M}$ define:*

$$Class_{M_i} = \{F \,|F = \Phi(A),\, \forall j : T(A, M_i) < T(A, M_j)\}\,.$$

In other words, M is the fastest method for solving A. Note that for every solvable $A \in \mathbb{A}$ there exists some method M that is the fastest, thus $\mathbb{A} = \bigcup_{M} Class_M$, and for every observation in $\mathbb{A}$ there is only one method that is the fastest so the classes are disjoint.

Definition 27. *In the performance problem the class space is defined as*

$$\mathbb{C} = \{Class_M \mid M \, is \, a \, method \, in \, \mathbb{M}\} .$$

In particular, for performance classification we are interested in comparing several methods side by side, so the classifier is defined differently than for convergence:

Definition 28. *For a given problem $A \in \mathbb{A}$, let $\mathcal{M} \subseteq \mathbb{M}$ be the set of methods that solve A. The performance classifier for $\mathcal{M}$ is given by $\Upsilon_{\mathcal{M}} : \mathbb{F} \mapsto \mathbb{C}_{\mathcal{M}}$ where $\mathbb{C}_{\mathcal{M}}$ is a set of two or more performance classes of the type $Class_M$ (from Definition 26), such that for a given $F \in \mathbb{F}$ $\exists M \in \mathcal{M}$*

$$\Upsilon_{\mathcal{M}}(F) = Class_M$$

where $F = \Phi(A)$, and M is the method that solves A the fastest.

Remember that the accuracy for any type of class is computed as described in Definition 15. The accuracy evaluating function Ω for the performance problems has a general form. The only difference is the number of methods (or transformations) that are considered for each problem in particular.

Definition 29. *Let $n = |\mathcal{M}|$ where $\mathcal{M}$ represents the methods or transformations to classify. The accuracy evaluating function is given by:*

$$\Omega_{\mathcal{M}} = \langle \alpha_{M_1}, \alpha_{M_2}, \ldots, \alpha_{M_n} \rangle$$

As opposed to the reliability classification problem, by Definition 8, the method picking function Π for the performance problem outputs solely one method – the fastest.

Definition 30. *For a given problem $A \in \mathbb{A}$, let $F = \Phi(A)$ the function that picks the fastest method to solve A is constructed as:*

$$\Pi_{\mathbb{M}}(F) = M \;\equiv\; \exists M \in \mathbb{M} \;\; \Upsilon_{\mathbb{M}}(F) = Class_M$$

where $F \in \mathbb{F}$.

However, it is possible that certain methods are almost as fast as the chosen method. For such cases, we will consider the possibility of recommending more than one method, e.g., the first and second best methods. This will be described in the implementation of the recommendation algorithms in Section 6.2.

6.1.1 Classification of Iterative Methods

In this section we focus only on $\mathbb{K}$, the subset of $\mathbb{M}$ which includes only iterative methods (ksps), like we did in Section 5.1.1. The class space is constructed as in Definition 27, but based only on the set iterative methods considered $\mathcal{K} \subseteq \mathbb{K}$. A class in this space is defined:

Let $A \in \mathbb{A}$, and $\mathcal{K} \subseteq \mathbb{K}$ be the set of iterative methods available that solve A. Let I be the index set of methods and $i, j \in I$. For each method $ksp_i \in \mathbb{K}$ define:

$$Class_{ksp_i} = \{F \,|F = \Phi(D(A)),\, \forall j : T(A, ksp_i) < T(A, ksp_j)\} \tag{6.1}$$

where $D \in \mathcal{D}$.

Definition 31. *The class space for the performance problem, based on a set of iterative methods is constructed as:*

$$\mathbb{C}_{\mathcal{K}} = \{Class_{ksp} \,|\, ksp \in \mathcal{K}\}$$

Based on this, the classifier functions to find the best iterative method can be defined as follows:

Definition 32. *For a given problem $A \in \mathbb{A}$, let $\mathcal{K}$ be the set of methods that solve A. The performance classifier for $\mathcal{K}$ is given by $\Upsilon_{\mathcal{K}} : \mathbb{F} \mapsto \mathbb{C}_{\mathcal{K}}$ where $\mathbb{C}_{\mathcal{K}}$ is a set of two or more performance classes of the type $Class_{ksp}$ (Definition 6.1), such that for $F \in \mathbb{F}$ $\exists ksp \in \mathcal{K}$*

$$\Upsilon_{\mathcal{K}}(F) = Class_{ksp} \tag{6.2}$$

where $F = \Phi(D(A))$ for a transformation $D \in \mathbb{D}$, and ksp is the iterative method that solves A the fastest.

The trivial function for choosing the method is defined as follows:

Definition 33. *For a given problem $A \in \mathbb{A}$, let $F = \Phi(D(A))$ where $D \in \mathbb{D}$ and let $ksp \in \mathcal{K}$. The function that picks the fastest method to solve A is constructed as:*

$$\Pi_{\mathcal{K}}(F) = ksp \;\equiv\; \exists ksp \in \mathcal{K} \;\; \Upsilon_{\mathcal{K}}(F) = Class_{ksp}$$

where $F \in \mathbb{F}$.

6.1.2 Classification of Transformations

The problem of finding the "optimal" (fastest) transformation for a problem $A \in \mathcal{A}$, is that of finding which transformation makes an iterative method be the fastest. The comparison of transformations is performed only against other transformations of the same type. The different types of transformations were described in Section 4.2.3. Like in previous sections, the description and construction of the class space and other concepts for this problem will be based on the *preconditioner* transformation; so for clarity of the notation, we will assume that $\mathbb{D}$ contains only preconditioners (i.e., $\mathbb{D}_{pc} = \mathbb{D}$). The methods and definitions described here can be extended later to other types of transformations.

We denote the set of available preconditioners as $\mathcal{D}$, a subset of $\mathbb{D}$. A class can be defined as follows.

Let $A \in \mathbb{A}$, and $\mathcal{K} \subseteq \mathbb{K}$ be the set of iterative methods available that solve $pc(A)$, where $pc \in \mathcal{D}$. Let I be the index set of methods and $i, j \in I$. For each preconditioner $pc_i \in \mathcal{D}$ define:

$$Class_{pc_i} = \{F \,|F = \Phi(A),\, \forall j : T(pc_i(A), ksp) < T(pc_j(A), ksp)\} \tag{6.3}$$

where $ksp \in \mathcal{K}$.

We now define the class space for the transformation classification problem.

Definition 34. *Let $\mathcal{D} \subseteq \mathbb{D}$ be the set of available preconditioners. The class space for the performance problem is*

$$\mathbb{C} = \{Class_D | D \in \mathcal{D}\}$$

Next, we construct the corresponding classifier based on this class space:

Definition 35. *For a given problem $A \in \mathbb{A}$, let $\mathcal{D}$ be the set of preconditioners that can be applied to A. The performance classifier for $\mathcal{D}$ is given by $\Upsilon_{\mathcal{D}} : \mathbb{F} \mapsto \mathbb{C}_{\mathcal{D}}$ where $\mathbb{C}_{\mathcal{D}}$ is a set of two or more performance classes of the type $Class_{pc}$(Definition 6.3), such that for a given $F \in \mathbb{F}$ $\exists pc \in \mathcal{D}$*

$$\Upsilon_{\mathcal{D}}(F) = Class_{pc}$$

where $F = \Phi(A)$, and pc is the preconditioner for which a set of iterative methods $\mathcal{K}$ were the fastest for solving $pc(A)$.

The function for choosing the method is:

Definition 36. *For a given problem $A \in \mathbb{A}$, let $F = \Phi(A)$. The function that picks the preconditioner which yields to solve A the fastest with some iterative method $ksp \in \mathbb{K}$ is constructed as:*

$$\Pi_{\mathcal{D}}(F) = pc \equiv \exists pc \in \mathcal{D} \ \Upsilon_{\mathcal{D}}(F) = Class_{pc}$$

where $pc \in \mathcal{D}$ and $F \in \mathbb{F}$.

6.1.3 Strategy for Recommendation of Composite Methods

As opposed to the Reliability problem, in the Performance problem we have to compare side by side all available combinations, which results in a large class space. Suppose we have k available iterative methods and d available preconditioners; this results in $k \times d$ classes, which makes the classification problem very difficult. Since the class space is a partition of the dataset, the more classes there are, the fewer number of observations there is in each class (classes with very few observations can cause some classifiers to fail or result in overfitting and thus, poor generalization). It is also harder to distinguish between classes, especially among those that are very similar, because the intersection areas between the classes yield larger statistical errors relative to the size of the classes (experimental results about this are discussed in Section 7.4.6).

For this reason, we will focus on a strategy equivalent to the orthogonal approach from the Reliability problem (see Section 5.3.1). The density functions for the iterative method and the preconditioner are computed separately ($\Upsilon_{\mathcal{K}}$ and $\Upsilon_{\mathcal{D}}$), as well the accuracies of classification. Let $n = k \times d$ where k is the number of iterative methods available and d is the number of preconditioners available. The accuracy of classification for any combination (pc, ksp) is the product of the independent accuracies:

$$\alpha_{(pc,ksp)} = \alpha_{pc} \times \alpha_{ksp} \text{ where } pc \in \mathbb{D} \text{ and } ksp \in \mathbb{K}. \tag{6.4}$$

The method picking function is then constructed (based on the sets of available iterative methods and preconditioners):

Definition 37. *Let $pc \in \mathcal{D}$, $ksp \in \mathcal{K}$, $\mathcal{D} \subseteq \mathbb{D}$, $\mathcal{K} \subseteq \mathbb{K}$, and let $F \in \mathbb{F}$. The function for choosing the best composite method (pc, ksp) is*

$$\Pi^{orthogonal}(F) = \mathcal{D} \cup \mathcal{K} \equiv \exists pc \in \mathcal{D} \quad \Upsilon_{\mathcal{D}}(F) = Class_{pc} \\ \text{and} \quad \exists ksp \in \mathcal{K} \quad \Upsilon_{\mathcal{K}}(pc(F)) = Class_{ksp}$$

where $\mathbb{F} = \Phi(\mathbb{A})$.

6.1.4 Method Similarity and Superclasses

In classification problems, the more classes there are the more difficult it is to differentiate between each of them, in particular when we are dealing with high dimensional problems. For this reason, it is advisable to have only the basic number of classes that best represent the data. In our software implementation we have 8 iterative methods and 10 preconditioners. Several of these methods are potentially similar in performance and convergence behavior, whether it is because of their numerical nature or because of their implementation (same applies to preconditioners).

To compare and measure the *similarity* of the methods or transformations, we define the following measurements

Definition 38. *Let $x, y \in \mathbb{M}$. The independence of method x from method y is:*

$$In_y(x) = \frac{x_1 \wedge y_0}{x_1} \tag{6.5}$$

that is, the proportion of cases for which method x converged and method y diverged with respect to the total number of cases where method x converged. This quantity is always between 0 and 1, with being 1 complete independence.

Definition 39. *For any two different methods $x \in \mathbb{M}$ and $y \in \mathbb{M}$ we define covering ratio as:*

$$CR(x, y) = \frac{P(x_0 \wedge y_1) + P(x_1 \wedge y_0)}{P(x_1 \wedge y_1)} \tag{6.6}$$

where P represents a proportion. In other words, this ratio is the proportion of cases where either method converges with respect to the proportion of cases where both methods converge. This measurement can have values between 0 and ∞.

The method independence helps us identify those methods that cover other methods in the sense of reliability. It also help us evaluate how independent one method is with respect to another. Figures 6.1(a) and 6.1(b) explain these ideas in more detail. Also, note that method independence is not a symmetric relation, i.e. $I_{M_1}(M_2) \neq I_{M_2}(M_1)$.

The covering ratio can help us determine when a method covers another one and by how much, i.e. the size of intersection of cases that they solve. The closer its value is to 0, the more they intersect, and the more it grows to ∞ the less they intersect (refer to figures 6.1(a) and 6.1(b)). Essentially, a small intersection indicates that the types of cases that each of them cover is very different, and we can view a bigger intersection as the case when one can be used instead of the other to solve the problems (in terms of reliability).

The *similarity* of two methods can be measured in terms of these two quantities. Suppose for example we have methods M_1 and M_2, if $In_{M_1}(M_2) \approx 0$ and $In_{M_2}(M_1) \approx 0$ then both methods

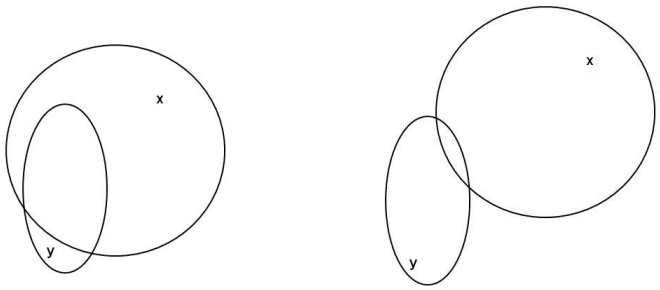


Figure 6.1: Example for method independence and coverage: (a) Method x converges for most of the cases where method y also converges; this figure also shows the case when x is very independent from y ($I_y(x) \approx 0.85$) but not otherwise ($I_x(y) \approx 0.25$). (b)Method x converges for most of the cases where method y diverges, x and y are both very independent from each other.

are highly dependent on each other. In addition, if $CR(M_1, M_2) > 1$ it means that these methods solve approximately the same number (and types) of cases. These amounts allow us to group methods (or transformations) into what we define as *superclasses*, which group the observations that are best solved by similar methods.

Experimental results have showed that independence values smaller than 0.4 already indicate a significant dependence of one method on the other. For covering ratio a value greater than 0.7 is an indication that the methods have a similar convergence and performance behavior if both independence measurements are below 0.4. Based on this we can now formalize the concept of similar iterative methods:

Definition 40. *Let I be the index set of iterative methods and $i, j \in I$. Let $ksp_i, ksp_j \in \mathbb{K}$. The methods ksp_i and ksp_j are similar if the following conditions hold for $i \neq j$:*

$$In_{ksp_i}(ksp_j) \leq 0.4 \text{ and } In_{ksp_j}(ksp_i) \leq 0.4 \text{ and } CR(ksp_i, ksp_j) \geq 0.7$$

A *superclass* can be defined as follows:

Definition 41. *Let I be the index set of iterative methods, and $i, j \in I$. Let $K \subset \mathbb{K}$ be a set of similar methods as defined above (Definition 40). A superclass, denoted as $CLASS_K$, is composed by the union of several classes like the $Class_{ksp}$ from Definition 26 where $ksp \in K$*

$$CLASS_K = \bigcup_{ksp \in K} Class_{ksp}$$

and none of these classes is contained in any other superclass.

Furthermore, $CLASS_K$ is a set of observations for which one of the similar methods is the fastest:

$$CLASS_K = \{A | \exists ksp_i \in K \text{ such that } \forall ksp_j \notin K : T(A, ksp_i) < T(A, ksp_j)\}$$

The following algorithm describes how to form the superclasses of iterative methods:

Algorithm 10. *Build Superclasses*

Input: *a set of methods* $\mathbb{K}$*, a set of classes* $\mathbb{C}_{\mathbb{K}} = \{Class_{ksp} | ksp \in \mathbb{K}\}$

Output: *a set of* N *superclasses* $\mathbb{C}_{\mathbb{K}_N}$*, a partition* $\mathbb{K}_N$ *of* $\mathbb{K}$ *(methods organized into superclasses)*

initialize $i = 1$, $CLASS = \{\}$, $\mathbb{K}' = \{\}$, $\mathbb{K}_N = \{\}$, $\mathbb{C}_{\mathbb{K}_N} = \{\}$, *and* $K = \mathbb{K}$

for each $ksp_1 \in K$

$CLASS_i = CLASS_i \cup \{Class_{ksp_1}\}$
$\mathbb{K}' = \mathbb{K}' \cup \{ksp_1\}$
for each $ksp_2 \in K$ *such that* $ksp_1 \neq ksp_2$
if ksp_1 *and* ksp_2 *are similar (definition 40)*
$CLASS = CLASS \cup \{Class_{ksp_2}\}$
$\mathbb{K}' = \mathbb{K}' \cup \{ksp_2\}$
$K = K - \{ksp_2\}$
$K = K - \{ksp_1\}$
make $CLASS_i = CLASS$ *and* $\mathbb{K}_i = \mathbb{K}'$
restore $CLASS = \{\}$ *and* $\mathbb{K}' = \{\}$
$i = i + 1$
$\mathbb{C}_{\mathbb{K}_N} = \mathbb{C}_{\mathbb{K}_N} \cup \{CLASS_i\}$
$\mathbb{K}_N = \mathbb{K}_N \cup \{\mathbb{K}_i\}$

output $\mathbb{C}_{\mathbb{K}_N}$ *and* $\mathbb{K}_N$

The class space for superclasses is defined as:

Definition 42. *Let* N *be the number of identified superclasses for iterative methods.* $Class\mathbb{K}_N$ *is the class space of superclasses and is a* partition *of the class space* $\mathbb{C}$ *from Definition 27 such that*

$$\mathbb{C}_{\mathbb{K}_N} = \{CLASS_K | CLASS_K \text{ is a superclass}\}$$

The classification is then made based on superclasses rather than the original classes. By grouping the observations in this way we may achieve better accuracy of classification for the classifiers. Once we have found the best superclass, we proceed to do classification using the subclasses.

The classification learning process using superclasses follows both the training and testing steps as described previously in Section 4.1. We can use the concept of superclasses to develop a *hierarchical classification* strategy, which carries out the classification in several levels of classes and subclasses. The algorithm for hierarchical classification using superclasses will be described in detail in Section 6.2.2.

6.2 Algorithms for Classification and Recommendation

The algorithms for the Performance problem are also divided in Classification (which concerns the learning process) and Recommendation (the actual method picking process). In here we also consider the three recommendation possibilities as discussed in previous chapters: iterative method, preconditioner or combination of both. There are classification algorithms for these three cases, but for combinations there are only recommendation algorithms since the classification is based on independent results of the first two cases.

6.2.1 Algorithms for Iterative Methods

The Performance problem is described as finding which iterative method is the fastest (in terms of time to convergence) for solving a particular linear system $A \in \mathbb{A}$, regardless of the preconditioner (or transformations) used. Note that for this particular problem, it is possible that the input is a matrix of the type A, or a preconditioned matrix $A' = pc(A)$.

Construction of Classifiers for Iterative Methods

This is the way to derive the classifier for a particular ksp in the performance problem. The classifier can later be used to form pick converging methods, discard diverging ones, and form recommendations.

Algorithm 11. *Derive Performance Classifier for Iterative Methods*

> ***Input:*** *a set of problems $\mathcal{A}$, a set of available iterative methods $\mathcal{K}$*
>
> ***Output:*** *a classifier function (discriminant) $\Upsilon_{\mathcal{K}}$, and a vector of accuracies $Y_{\mathcal{K}}$ as in Definition 4.6*

1. *Obtain the feature space as $\mathbb{F} = \Phi(\mathcal{A})$*
2. *As an optional step, transform the feature space using PCA as described in Section 3.6: $\mathbb{F}' = \Gamma(\mathbb{F})$ and make $\mathbb{F} = \mathbb{F}'$*
3. *Randomly split the feature space into training ($\mathbb{F}_{tr}$) and test ($\mathbb{F}_{te}$) sets*
4. *Using $\mathbb{F}_{tr}$, derive a discriminant function $\Upsilon_{\mathcal{K}}$ (a set of parameters that describe a probability density function) using one of the methodologies in Section 3.2, e.g.*

   Kernel Mixture

   Decision Trees
5. *Using $\mathbb{F}_{te}$, compute the accuracies for each of the $|\mathcal{K}|$ classes using $\Omega_{\mathcal{K}}$ as described in Definition 15 to form $Y_{\mathcal{K}}$*

Recommendation of Iterative Method

This algorithm describes the way of using the classifier $\Upsilon_{\mathcal{K}}$ to find the fastest iterative method to solve a given problem. It shows the implementation of the iterative method picking function $\Pi_{\mathcal{K}}(F)$ (Equation 6.2).

Algorithm 12. *Use of Performance Classifier for Iterative Methods*

Input: *a problem $A \in \mathcal{A}$, a classifier function $\Upsilon_{\mathcal{K}}$*

Output: *an iterative method $K \in \mathcal{K}$ which is the fastest for solving A*

initialize $K = \{\}$

if A is not preconditioned then

for each $D \in \mathbb{D}$

extract the preconditioned features $F = \Phi(D(A))$

if $\Upsilon_{\mathcal{K}}(F) = Class_{ksp}$ *then*

$K = \{ksp\}$

obtain α_{ksp} *from* $Y_{\mathcal{K}}$ *(Definition 4.6)*

else

extract the features $F = \Phi(A)$

if $\Upsilon_{\mathcal{K}}(F) = Class_{ksp}$ *then*

$K = \{ksp\}$

obtain α_{ksp} *from* $Y_{\mathcal{K}}$ *(Definition 4.6)*

output $\Pi_{\mathcal{K}}(F) = K$ *and the corresponding accuracy* $\alpha_K = \alpha_{ksp}$

6.2.2 Algorithms for Transformations

This is the case when we are interested only in finding the optimal transformation among the available ones, independently of the iterative method being used. In order to simplify the description of the algorithms, we will limit the set of transformations to preconditioners as we have done in previous chapters, and $\mathcal{D} \subseteq \mathbb{D}$ is the set of available preconditioners. The problem A in this case is not preconditioned, so the features extracted are in their original form. Unlike the Reliability problem described in Section 5.1.2, in the Performance problem it is not necessary to analyze individually each of the combinations like $\langle pc, ksp \rangle$. For the performance problem we want to know which is the *fastest* preconditioner, and since we can assume that the timings for each observation are different, then there is only one transformation that yields the fastest solution (i.e., there is only one "optimal" preconditioner no matter what iterative method is used, as opposed to the Reliability problem where a preconditioner can be reliable for more than one iterative method).

Construction of Classifiers for Preconditioners

Algorithm 13. *Derive Performance Classifier for Preconditioners*

Input: *a set of problems $\mathcal{A}$, a set of available preconditioners $\mathcal{D}$*

Output: *a classifier function (discriminant) $\Upsilon_{\mathcal{D}}$, a vector of accuracies $Y_{\mathcal{D}}$ as in Definition 4.6*

1. *Obtain the feature space as $\mathbb{F} = \Phi(\mathcal{A})$*
2. *As an optional step, transform the feature space using PCA as described in section 3.6: $\mathbb{F}' = \Gamma(\mathbb{F})$, and make $\mathbb{F} = \mathbb{F}'$*
3. *Randomly split the feature space into training ($\mathbb{F}_{tr}$) and test ($\mathbb{F}_{te}$) sets*
4. *For each $pc \in \mathcal{D}$*
   - *Using $\mathbb{F}_{tr}$, derive a discriminant function $\Upsilon_{\mathcal{D}}$ using the conditional combinations approach described in Definition 17, and one of the techniques described in Section 3.2 for example*

     Kernel Mixture

     Decision Trees
5. *Using $\mathbb{F}_{te}$, compute the accuracies for each of the $|\mathcal{D}|$ classes using $\Omega_{\mathcal{D}}$ as described in Definition 15 to form $Y_{\mathcal{K}}$*

Algorithm for Recommendation of Preconditioner

This algorithm details the construction of the preconditioner picking function $\Pi_{\mathcal{D}}(F)$ (Equation 35) to obtain the fastest preconditioner. This is the recommendation process for the performance of preconditioners.

Algorithm 14. ***Use of Performance Classifier for Preconditioners***

Input: *a problem $A \in \mathcal{A}$, a classifier function $\Upsilon_{\mathcal{D}}$*

Output: *a preconditioner $PC \in \mathcal{D}$ for which some iterative method $ksp \in \mathcal{K}$ is the fastest solving A*

initialize $PC = \{\}$

extract the features $F = \Phi(A)$

if $\Upsilon_{\mathcal{D}}(F) = Class_{pc}$ then

$PC = \{pc\}$

obtain α_{pc} from $Y_{\mathcal{D}}$ (Definition 4.6)

output $\Pi_{\mathcal{D}}(F) = PC$ and $\alpha_{PC} = \alpha_{pc}$

Hierarchical Classification for the Performance Problem

To find the fastest method, it is necessary to compare different methods at a time. The bigger the cardinality of the class space, the harder it is to distinguish between each of the classes. For this reason we need to develop a different approach for finding the fastest method for solving a problem A. *Hierarchical classification* consists of forming superclasses (Definition 41) that group similar methods (Definition 40) and perform the classification based on these classes. Then, once the superclass was picked, the classification proceeds to differentiate between the classes within that superclass.

The following algorithms show the strategy followed to carry out hierarchical classification for a set of $|\mathcal{K}|$ methods to find the fastest method $K \in \mathcal{K}$.

Methods are grouped based on *similarity* measures (In and CR) of convergence and performance, which were described in Section 6.1.4.

In hierarchical classification these two processes vary slightly so we will describe the additional processing. Using Algorithm 10 we form the super classes and create their class space $\mathbb{C}_{\mathbb{K}_N}$, this space also defines how the iterative methods are organized into each of the superclasses $\mathbb{K}_N$.

Algorithm 15. *Derive Performance Classifiers for Superclasses of Iterative Methods*

> ***Input:*** *a set of problems $\mathcal{A}$ organized as a set of superclasses $\mathbb{C}_{\mathbb{K}_N}$, a partition $\mathbb{K}_N$ of the iterative methods*
>
> ***Output:*** *a superclass classifier $\Upsilon_{\mathbb{K}_N}$, a vector of accuracies $Y_{\mathbb{K}_N}$ (Definition 4.6)*

1. *Obtain the feature space as $\mathbb{F} = \Phi(\mathcal{A})$*
2. *As an optional step, transform the feature space using PCA as described in Section 3.6: $\mathbb{F}' = \Gamma(\mathbb{F})$ and make $\mathbb{F} = \mathbb{F}'$*
3. *Randomly split the feature space into training ($\mathbb{F}_{tr}$) and test ($\mathbb{F}_{te}$) sets*
4. *Using $\mathbb{F}_{tr}$, derive a discriminant function $\Upsilon_{\mathbb{K}_N}$ (a set of parameters that describe a probability density function) using a statistical learning methodologies such as the ones described in Section 3.2, for example:*

   Kernel Mixture

   Decision Trees

5. *Using $\mathbb{F}_{te}$, compute the accuracies for each of the N superclasses using $\Omega_{\mathbb{K}_N}$ as described in Definition 15 to form $Y_{\mathbb{K}_N}$*

   for each $CLASS_i \in \mathbb{C}_{\mathbb{K}_N}$

   make $\mathcal{A} = CLASS_i$ and $\mathbb{K} = \mathbb{K}_i$

   apply Algorithm 11 to derive a discriminant function $\Upsilon_{\mathbb{K}_i}$ and the corresponding accuracies $Y_{\mathbb{K}_i}$ for the current superclass

   output $\Upsilon_{\mathbb{K}_N}$ and $Y_{\mathbb{K}_N}$

The following algorithm uses the derived classifiers of both superclasses and subclasses to build the recommendation.

Algorithm 16. *Use of Performance Classifiers for Superclasses of Iterative Methods*

Input: *a problem $A \in \mathcal{A}$, a classifier function for superclasses $\Upsilon_{\mathbb{K}_N}$, and a set of classifier functions $\Upsilon_{\mathbb{K}_i}$ (one for each ith superclass)*

Output: *an iterative method $K \in \mathbb{K}$ that is the fastest for solving A*

initialize $K = \{\}$

if A is not preconditioned then

for each $D \in \mathbb{D}$

extract the preconditioned features $F = \Phi(D(A))$

if $\Upsilon_{\mathbb{K}_N}(F) = CLASS_{\mathbb{K}_i}$ *then*

with the superclass $CLASS_{\mathbb{K}_i}$ *and the respective set of classifiers* Υ_{ksp} *where* $ksp \in \mathbb{K}_i$ *use Algorithm 12 to obtain* K *and* α_K

else

extract the features $F = \Phi(A)$

if $\Upsilon_{\mathbb{K}_N}(F) = Class_{\mathbb{K}_i}$ *then*

with the superclass $CLASS_{\mathbb{K}_i}$ *and the respective set of classifiers* Υ_{ksp} *where* $ksp \in \mathbb{K}_i$ *use Algorithm 12 to obtain* K *and* α_K

output $\Pi_{\mathbb{K}_N}(F) = K$ *and* $\alpha_{\mathbb{K}_N} = \alpha_K$

These algorithms are specific for iterative methods, however, they can be easily modified to work with preconditioners and other transformations by using $\mathbb{D}$ instead of $\mathbb{K}$ (and they can be extended to work with combinations of these by using $\mathbb{D} \times \mathbb{K}$). If we are dealing with transformations, the recommendation process in Algorithm 12 has to omit the preconditioning (or preprocessing) of features as follows:

Algorithm 17. *Use of Performance Classifiers for Superclasses of Preconditioners*

Input: *a problem $A \in \mathcal{A}$, a classifier function for superclasses $\Upsilon_{\mathbb{D}_N}$, and a set of classifier functions $\Upsilon_{\mathbb{D}_i}$ (one for each ith superclass)*

Output: *a preconditioner $D \in \mathbb{D}$ which makes some iterative method in $\mathbb{K}$ be the fastest solving A.*

initialize $PC = \{\}$

extract the features $F = \Phi(A)$

if $\Upsilon_{\mathbb{D}_N}(F) = CLASS_{\mathbb{D}_i}$ *then*

with the superclass $CLASS_{\mathbb{D}_i}$ *and the respective set of classifiers* Υ_{pc} *where* $pc \in \mathbb{D}_i$ *use Algorithm 14 to obtain* PC *and* α_{PC}

output $\Pi_{\mathbb{D}_N}(F) = PC$ *and the accuracy* $\alpha_{\mathbb{D}_N} = \alpha_{PC}$

6.2.3 Algorithms for Combinations of Iterative Methods with Transformations

As opposed to the Reliability problem, the treatment for combination methods for the Performance problem is different. Suppose that we create a class per combination (pc, ksp). If we have eight preconditioners and seven iterative methods available then we would have 56 classes to choose from. If we use this approach, the number of classes grows quadraticaly and is very hard for a classifier to distinguish among them. Hence, it is very difficult to follow the conditional approach that was so favorable for the Reliability problem.

In this case we need to take a simpler approach. Preliminary experiments indicated that the accuracy of different classifying strategies for combinations like (pc, ksp), used in the Reliability problem (Section 5.3), is minimal. Therefore, we can simply use the most straightforward approach, which is the *orthogonal* strategy. Naturally, a tradeoff between accuracy and simplicity may arise for certain methods or classifiers, but the use of certain recommendation measurements can help to improve predictions, these will be discussed in Section 6.3.

Algorithm 18. *Use of Performance Classifiers for Combination Methods: Orthogonal*

Input: *a problem $A \in \mathcal{A}$, functions $\Upsilon_{\mathcal{D}}$ and $\Upsilon_{\mathcal{K}}$*

Output: *a combination method expressed in the (pc, ksp) format which is the fastest combination for solving A.*

initialize $\mathcal{D} = \{\}$, $\mathcal{K} = \{\}$ and $Q = \{\}$

extract the features $F = \Phi(A)$

for each $pc \in \mathcal{D}$

  if $\Upsilon_{\mathcal{D}}(F) = Class_{pc}$ then

    obtain α_{pc} from $Y_{\mathcal{D}}$ (Definition 4.6)

for each $D \in \mathcal{D}$ (or $D \in \mathbb{D}$ for Theoretical algorithm)

  extract the preconditioned features $F = \Phi(D(A))$

  for each $ksp \in \mathcal{K}$

    if $\Upsilon_{\mathcal{K}}(F) = Class_{ksp}$ then

      obtain α_{ksp} from $Y_{\mathcal{K}}$ (Definition 4.6)

      $\alpha_{(pc,ksp)} = \alpha_{pc} \times \alpha_{ksp}$

output $\Pi^{orthogonal} = (pc, ksp)$ and $\alpha_{(pc,ksp)}$

This approach is also convenient for use with superclasses, e.g., instead of using the $\mathcal{K}$ and $\mathcal{D}$ sets we can use superclasses sets.

6.3 Recommendation Evaluation

The Performance problem is a multi-class classification problem. In a two-class problem, like in Reliability, the quality of the recommendation depends mainly on the statistical error. In a multi-class problem, both the quality of a recommendation and the penalties associated with error depend on how we measure the "badness" of a wrong recommendation considering all the classes involved. Like in the Reliability problem, we can measure how bad a predicted method is as the additional time t we have to wait for it to converge to the solution for a linear system A, compared to the time that the optimal method takes. This can be seen as a factor of time loss or speed loss for solving a problem; it was defined Equation 5.12 as the *slowdown* factor.

In the Performance problem, we can then express the quality of the recommendations as gain and loss of time (or any other performance metric). This problem is concerned with the prediction of an optimal method to solve a matrix, so the answers are not "binary" like in the Reliability problem (converges, diverges). For this reason, besides measuring "how bad" a method is, it is also important to measure "how good" it is. In other words, the recommendation goes beyond determining whether a method is the optimal, but should also considers these cases *:

- If the prediction is indeed the optimal method, how much faster it is compared to a default. This can be represented as the factor of speed gain

$$speedup = \frac{T(A, M_{default})}{T(A, M_{predicted})}$$

- If the prediction is wrong, how much slower the recommended method is compared to the actual best method. We use the following factor to evaluate this speed loss:

$$slowdown = \frac{T(A, M_{predicted})}{T(A, M_{optimal})}$$

The slowdown can be used as a factor to penalize mispredictions for each method together with the statistical error. The error of misprediction can be obtained from the confusion matrix(see Section 3.7.2) associated to the classification problem. We can compute the $slowdown$ factor for each method and then use it to weight the non-diagonal elements of the confusion matrix to obtain an *expected loss* associated with the method.

First we need to define intervals or "bins" of slowdown factor, e.g., "from 1 to 2", "from 2 to 20","more than 20". These bins will contain those errors whose associated slowdown factors fall in the range delimited by the bin; i.e., each bin is an error *counter* or accumulator. A *slowdown bin* is defined as:

$B_{M,l,c,u}$ where $M \in \mathcal{M}$, l is the lower limit of the bin u is the upper limit and c is the center.

The center of the bin can be for example the mean value between l and u or the median. It can be determined arbitrarily depending on the data, application, methods, etc. Then we compute the slowdown factor for every A in the unseen set that is misclassified and assign it to the corresponding

*Moreover, we have to ponder the possibility of recommending more than one method. For example, according to the classification recommend the "optimal" method and then from the remaining ones, select the second "optimal" choice.

bin (increase its count)

$$\text{if } l \leq \frac{T(A, M_{predicted})}{T(A, M_{optimal})} < u, \text{ increase} B_{M,l,c,u} \text{ by one.}$$

Observe that these errors originate from the non-diagonal elements of the confusion matrix associated with the classification, so the errors that account for a bin $B_{M,l,c,u}$ correspond to the column for method M. Based on this, we can compute a weighted average of errors and slowdown factors for a each method $M \in \mathcal{M}$; we call this *expected loss*.

Definition 43. *Let $S_1, \ldots, S_n$ be a series of speedup factors, where n is the number of bins in which we split the speedup range, i.e., $B_{M,l,c,u}$ is a bin, such that $1 \leq c < n$. Let $\mathcal{M}$ be the set of available methods. The expected loss for a method $M_i \in \mathcal{M}$ is given by*

$$L_{M_i} = \sum_{j=1}^{n} S_j \times B_{M_i,l,j,u}$$

This represents the expected loss of performance if we are to choose the recommended method M_i, i.e., if it happens that M_i is not the optimal method, how bad or slow it is expected to be with respect to the optimal method (whichever it was).

Similarly, we can use the $speedup$ factor to weight the diagonal elements of the confusion matrix and obtain an *expected gain* associated with the method. When we form the bins for each observation classified correctly, we are accounting only for these diagonal elements. The bins are created based on the speedup factor:

$$\text{if } l \leq \frac{T(A, M_{default})}{T(A, M_{predicted})} < u, \text{ increase} B_{M,l,c,u} \text{ by one.}$$

Definition 44. *The expected gain for method $M_i \in \mathcal{M}$ is*

$$G_{M_i} = \sum_{j=1}^{n} S_j \times B_{M_i,l,j,u}.$$

Note that when the recommended method is also the default this factor is 1.

Like in the Reliability problem, we can measure how good the recommendation is for some method $M \in \mathcal{M}$, by defining a metric $\mathcal{Q}$. In this case $\mathcal{Q}$ is defined in terms of L_M and G_M, which are derived from the confusion matrix and are intrinsically related to the accuracy of classification and associated statistical error:

$$\mathcal{Q}^{performance} = \frac{G_M}{L_M}, \tag{6.7}$$

and similar to the Reliability problem, the larger $\mathcal{Q}^{performance}$ is, the better the recommendation. This kind of measurement is useful when, for instance, we would like to provide more than one choice of method. For example, if internally, the recommendation system generates several recommendations and then compares between them, discarding those that are beyond certain value of $\mathcal{Q}$. Algorithm 19 shows how this process is carried out.

Algorithm 19. *Multiple Recommendations and use of $\mathcal{Q}$*

Input: *a problem $A \in \mathbb{A}$, a set of available methods $\mathcal{M}$, the number of performance recommendations requested n (where $n \leq |\mathcal{M}|$), and the associated performance gains and losses for each method.*

Output: *a method $M \in \mathcal{M}$*

for $i = 1, \ldots, n$

apply a recommendation algorithm such as Algorithm 12 or Algorithm 18 on problem A to find most suitable method M_i

obtain the $\mathcal{Q}_{M_i}$ for M_i with the associated expected performance gain G_{M_i} from Definition 44 and loss L_{M_i} from Definition 43 using Equation 6.7

add M_i to a temporary set of recommended methods $\mathcal{M}^{tmp}$

remove M_i from $\mathcal{M}$

end

choose M_i from $\mathcal{M}^{tmp}$ for which the associated $\mathcal{Q}_{M_i}$ is the largest

output M as M_i

To evaluate the quality of a recommendation for a method, based on reliability and performance we can use the tuple

$$\mathcal{Q}^{total} = (\mathcal{Q}^{reliability}, \mathcal{Q}^{performance}). \tag{6.8}$$

The recommendation system can output these values and the user can then decide which is more important for a particular problem – reliability or performance – and make a choice. This value is mainly based on the slowdown and speedup factors, which are determined upon the current experience from the available data. In other words this is dependent on the particular experiment.

Chapter 7

Experiments and Results

This chapter presents the results and discussion of various experiments. It also includes the discussion of a case study that exemplifies how the recommendation strategies work on new problems.

7.1 Introduction: Experimental Setup

The different experiments discussed in this chapter are based on the main steps of our methodology:

- Feature Analysis: feature evaluation using PCA.
- The Learning Process
  - The Reliability Problem: strategies for classification and recommendation.
  - The Performance Problem: strategies for classification and recommendation, including the hierarchical approach and a description of how the superclasses used in this strategy are formed.
- Building Recommendations: an example of how the reliability and performance strategies can be used to make predictions for new data.

Experimental Data We have used various datasets (described in Appendix A.2) in preliminary experiments to develop, test and tune the different strategies and concepts discussed in previous chapters. A description of these datasets can be found in Appendix A.2. The same set and type of experiments were performed individually on the different datasets. Results from these experiments have been qualitatively the same, except for those for the dataset JLAP40. JLAP40 is an "artificial" dataset that does not exhibit the feature variation that the other sets do; e.g., some of its features are constant for every observation in the set, making the feature characterization process fundamentally different.

The Matrix Market dataset is the most heterogeneous and provides a good and variated testbed for training and testing heuristics, and the variety of problems in this set is better for testing generalization. Considering that qualitative results across other datasets are the same, for the experiments presented here we focus on the analysis of the results from the Matrix Market dataset.

Like many other statistical and machine learning applications, the results of our methodology depend greatly on the type and amount of input data available for the learning process. Since we part

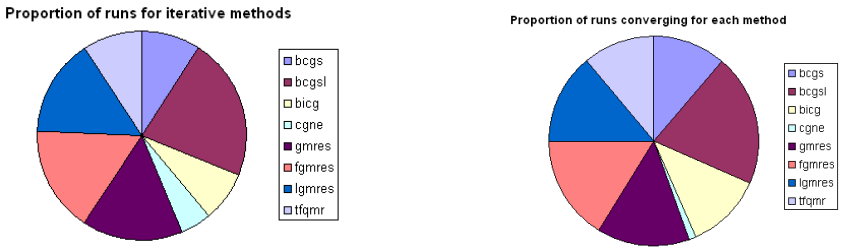


Figure 7.1: Proportion of examples for iterative methods in the database: (a) Proportion of examples for each method. (b) Proportion of runs for which each method converges.

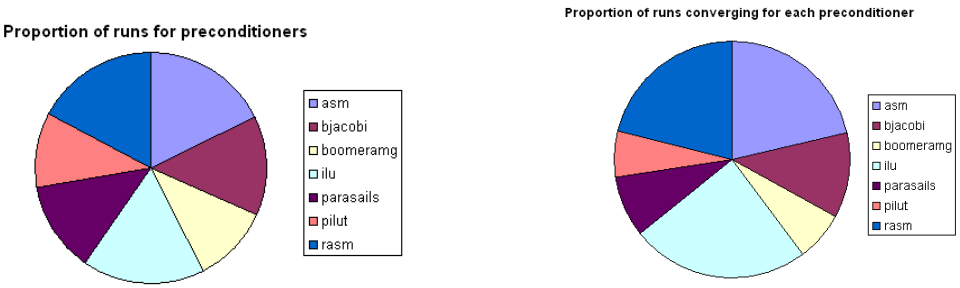


Figure 7.2: Proportion of examples for preconditioners: (a) Proportion of examples for each preconditioner. (b) Proportion of runs that converged when each preconditioner was used.

from the premise that the numerical properties affect the performance of solvers, we need to have a general perspective of the performance data in our database. Figures 7.1(a) and 7.2(a) present this information. For every available matrix we apply every possible combination of preconditioners and other transformations, then we try every possible solver and record the outcome. Each of these tests is a *run*. In particular, Figure 7.1(a) shows the proportion of *runs* available for each iterative method and Figure 7.1(b) shows the proportion of these cases for which a method converges. Figures 7.2(a) and 7.2(b) present the same type of information but for the available preconditioners*.

These types of measurements are very useful to tune classification strategies and interpret their results. For instance, $cgne$ in Figure 7.1(b) appears as a method that is not very reliable. In the Reliability problem, this reflects as a method for which the divergence class in the training process is very big, and there are almost no examples of cases where $cgne$ converges. This can cause *overfitting* [Winston, 1992], a common problem in supervised learning, which can make it nearly impossible for a classifier to *generalize* for problems where $cgne$ works. For the Performance problem, where we compare several methods at once, the $cgne$ class can also introduce noise, making the distinction between other methods more difficult.

*A "converging run" for a preconditioner pc is when some iterative method converged for a matrix preconditioned using pc

Learning Methodologies From the different types of classifiers described in Chapter 3, the kernel density estimators (using Bayes approach) and decision trees are the ones that experimentally were the most efficient. These methods are also the most suitable for the type of input problems that we here address, mainly because of their high dimensionality, feature interactions and large amount of data. It is useful to consider and test more than one classifier type in case one of them fails. The results presented in this chapter are from experiments using implementations of these two types of classifiers, however, decision trees was the one that yielded the best results in the majority of the tests. The accuracy of classification α presented in these results is an average accuracy taken over several trials on each experiment (the training and testing set are randomly picked from the original dataset each time, hence these sets are different each time). A confidence interval for α is also computed in most of the experiments; this indicates the interval in which the resulting accuracy for a random trial, will be away from the presented average α by $\pm z$ 95% [Douglas and Montgomery, 1999] of the time[†]. The interval can be used to determine how "stable" the resulting accuracy is for an experiment.

7.2 Feature Analysis

The feature analysis is one of the most important steps in the classification process. In this section we present the analysis of the features used in our experiments using PCA, and show the importance of PCA as an optional preprocessing step before classification.

"Bad" features commonly hinder the performance of different classifiers by introducing noise; it is necessary to do our best job to identify and filter out these features. These can be features that correspond to outliers, or features that remain constant across the different classes. On the other hand, "good" features help to achieve better classification for most kinds of classifiers; most importantly they provide valuable information to understand important characteristics of the classification problem being addressed.

Both problems of Performance and Reliability use the same set of numerical features. However, each problem partitions the dataset in a different way, so the same features may characterize the problems differently (e.g. feature i may affect the speed with which a method converges, but may have no effect on whether it is reliable or not). The categories in which we group these features were described in Section 4.1. Some features, particularly the ones in the *simple* and *structure* categories are not scale invariant, so it is necessary to scale them down or *normalize* them using other features such as number of rows or some of the norms, to keep these features independent from the size of the matrix, making the analysis scale-independent. A detailed description of these features and their normalizations can be found in Appendix A.2.

In the numerical context, all the features we have taken into account are intuitively important. In addition, the features characterized as relevant for both the Reliability and the Performance problems (Sections 7.2.3 and 7.2.4 respectively) make sense in the context of sparse matrices. Some of the most obvious ones are, based on expert intuition and heuristics [Langou, 2007]:

- $symmetry$: we cannot use a method for symmetric matrix on a nonsymmetric matrix. Additionally, the features $diag\_definite$ and $diag\_dominance$ together with $symmetry$ can determine when a mtrix is symmetric positive definite (SPD), which is extremely important

[†]We use z to delimit the confidence interval since we have used the *Z-test* [Douglas and Montgomery, 1999], commonly used in statistics

for the selection of methods. The feature ratio $symm_{anorm}/symm_{snorm}$ is redundant with $symmetry$.

- κ: the condition number is particularly important for SPD matrices, for convergence of methods like $CGNE$, and for some cases of nonsymetric matrices.
- $ellipse\_ax$ and $ellipse\_ay$: these features are related to κ and important for nonsymmetric matrices. These are quantities directly coming from the convergence theory of *GMRES*, and influence both robustness and performance.
- max_nnz_{row} and min_nnz_{row}: these quantities are useful mostly for unsymmetric matrices
- Features from the *Normal* category: these are important for convergence in nonsymmetric matrices. They are related to the cost of matrix-vector products. If the number of non-zeros per row is large then this means that matrix-vector products are very expensive compared to vector operations. As a consequence, Full GMRES may be used to minimize the number of iterations. If the number is small, then matrix-vector products are relatively cheap and a method like *BCGS* is a better choice.

Next, we walk through the process of feature characterization using as an example the data for the Performance problem. Then we present results for both Performance and Reliability.

7.2.1 Initial Elimination of Features

As described in Section 3.6.5, there are two types of features that are statistically problematic and need to be eliminated during the first stage of feature preprocessing. The first type are features that are constant across the different input problems and datasets. In our experiments we identified some of these features only in certain datasets (such as *FEMH* or *JLAP*). The second type of problematic features are those that introduce noise. Most of the features that had to be eliminated from our experiments correspond to this type. The following are some examples of these type of "noisy" features (see Section 3.6.5) that are eliminated in the Performance:

- $\lambda_{max,magnitude,\Im}/\lambda_{max,magnitude,\Re}$: $\mu = -6.39 \times 10^{-3}$, $stdv = 4.01 \times 10^{-1}$
- $\lambda_{max,magnitude,\Im}/diag\_avg$: $\mu = -2.44$, $stdv = 1.27 \times 10^{2}$
- $\lambda_{max,rp,\Im}/diag\_avg$: $\mu = -2.99 \times 10^{-2}$, $stdv = 1.12$

Observe how the order of magnitude of the standard deviation ($stdv$) is bigger than the mean's (μ), this implies that there is a lot of variation (possibly noise) on their values even after they are normalized to make them scale-invariant:

A difference in order of magnitude of 2 is already big enough to cause these features to become noisy, and makes it difficult to cluster them in any of the classes. Eliminating this type of features not only reduces the noise for the classification process, but is also a first step *dimensionality reduction*.

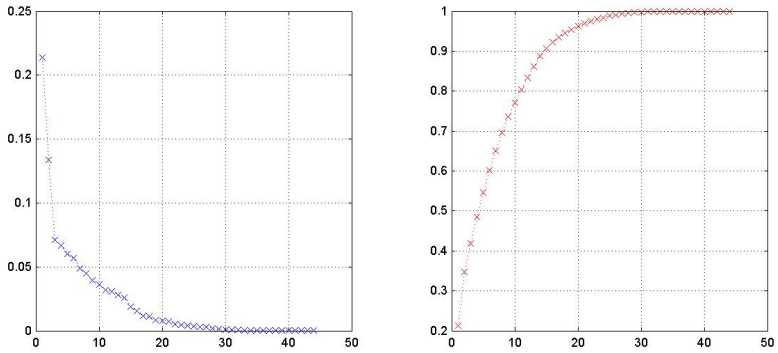


Figure 7.3: Scree plots: along the x-axis is the principal component index, and the y-axis shows proportion of the information captured by each principal component (cumulative on the right plot).

7.2.2 Characterization of Features Using PCA

The different products and stages of PCA have provided useful information for feature characterization (this was described in Section 3.6). We illustrate the feature analysis process by using an instance of features extracted for the Performance problem with three iterative methods. All the features extracted from input problems form the matrix X used for SVD (Section 3.6.1). After the initial elimination of statistically useless features, we are left with $n = 44$ features, supposing that the size of the dataset is m problems, then X is of size $m \times n$. There are as many principal components as the number of the original features, 44 in this example. By using only the first 30 principal components instead of the original 44, we can achieve some *dimensionality reduction*.

The *scree* plots help us decide how many principal components can be discarded; this is a type of dimensionality reduction. Figure 7.3(left) shows that the first two principal components capture more information than the others (approximately 0.21 and 0.13 respectively); in the same figure the cumulative plot (right) indicates that the first 30 principal components capture almost all the information from the original features, these can also be referred to as the "heaviest" principal components.

Next, the principal components are analyzed to evaluate the importance of each original feature in each as described in Section 3.6.3. Each loadings vector (LV) is a principal component, and each of them is analyzed using a bar plot (see Figures 7.4- 7.8). These plots have along the x-axis an index associated to each original feature, and the y axis shows how much each feature contributes to that particular vector (i.e. the higher the bar, the more meaningful the feature is). To simplify the notation, we will abbreviate the loadings vector charts as LVi, where i is the index of that vector. The analysis of these vectors consists of two parts: first, we explore the "heaviest" loadings vectors (the first 30 in this example) to identify significant and/or correlated features by looking at their magnitudes; and second, we look at the last 14 vectors to search for redundant and/or insignificant features. Next we will examine the type of information we can extract from these two parts.

Table 7.1: Feature correlations in the first Loadings Vectors. The columns *Positive* and *Negative* show those relevant features with positive and negative loading signs respectively.

LV	Positive Sign	Negative Sign
1	7, 8, 10, 11, 16, 17, 21, 25	2, 3, 5, 12, 14, 22
2	6, 10, 11, 17	19, 20, 26, 27
3	2, 3, 7, 9, 25	15, 22, 38, 40, 41, 42, 44
4	23, 41, 42	26, 27, 37, 38, 40
5	18, 28, 33, 36	7, 29, 38, 40

First Loadings Vectors

The heaviest loadings vectors are shown in Figures 7.4- 7.5. The features that are loaded the highest in $LV1$ can be "marked" as important because they contribute with the most information in the heaviest principal component. These features and their corresponding indexes are: 2 ($trace$), 3 ($trace_{abs}$), 5 ($Fnorm$), 7 ($symm_{anorm}/symm_{snorm}$), 8 ($symm_{fanorm}/symm_{fsnorm}$), 12 ($diagonal_{avg}$), 14 ($diagonal_{sign}$), 16 ($nnz$) and 22 ($diag\_definite$).

To find possible linear correlations, we compare the magnitudes of features in each PC. When two or more features are *positively correlated* their magnitudes on a particular principal component will be approximately the same, and with the same sign and they are *negatively correlated* when the signs are different For instance, features 2 and 3 ($trace$ and $trace_{abs}$) in the chart for $LV1$ in (Figure 7.4) are positively correlated, while features 2 and 16 are negatively correlated because when one grows, the other decreases. Note that features 2 and 3 also exhibit the same behavior in other important loadings vectors (such as $LV3, LV4, LV12$). Table 7.1 shows other important feature correlations in the first five LVs. For each LV, the features in the same column are positively correlated, and are negatively correlated with the ones in the other column. (See Appendix A.1 for the associated name, description and normalization type of each feature index).

Features like 26 and 27 are positively correlated in $LV4$, $LV6$, $LV18$, $LV20$, and they both score relatively high in these vectors. Furthermore, there is no principal component vector where their loadings have opposite signs, neither one scores high when the other one did not. We can conclude that these two features are positively correlated, and we can possibly eliminate one of them because they exhibit the same behavior. The identification of correlations is important, if several features are always correlated we can eliminate all but one of them, this is another way of reducing the number of dimensions in a problem.

The first loadings vectors can also be used to look for features that do not provide a significant amount of information. Figure 7.6 shows the magnitude of the loadings of each feature in the first five loadings vectors. Examples of features that are not relevant are 24 (n_ritz_vals), 30 (σ_{max}), 31 (σ_{min}), and 35 $\lambda_{min,magnitude,\Re}$ in Figure 7.6(d). These plots provide information of how significant a feature is in the most important pirncipal components.

Last Loadings Vectors

The loadings vectors corresponding to the last principal components also provide important information or redundant feature variables. The bar charts corresponding to these vectors appear in Figures 7.7, 7.8. From various experiments we have observed that the last loadings vectors are usually

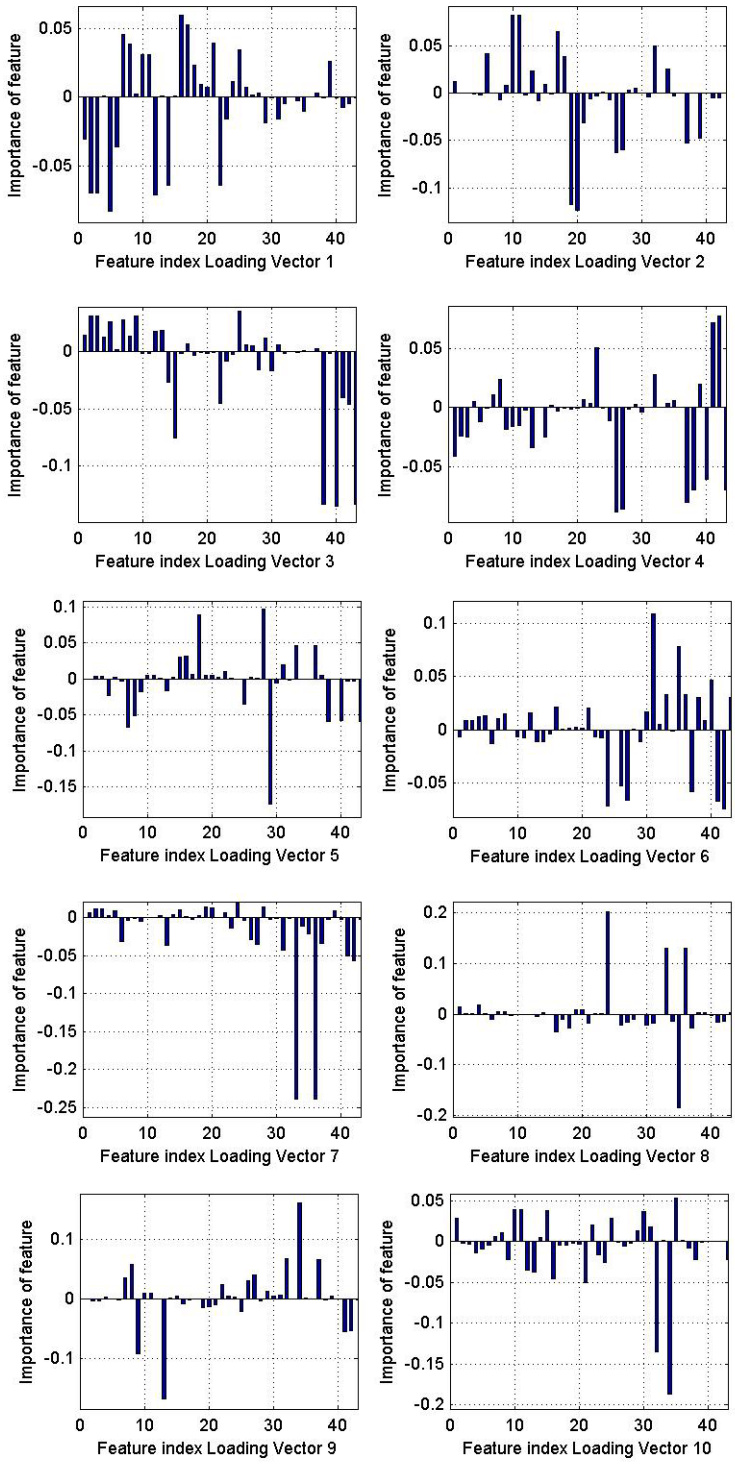


Figure 7.4: First 10 loadings vectors.

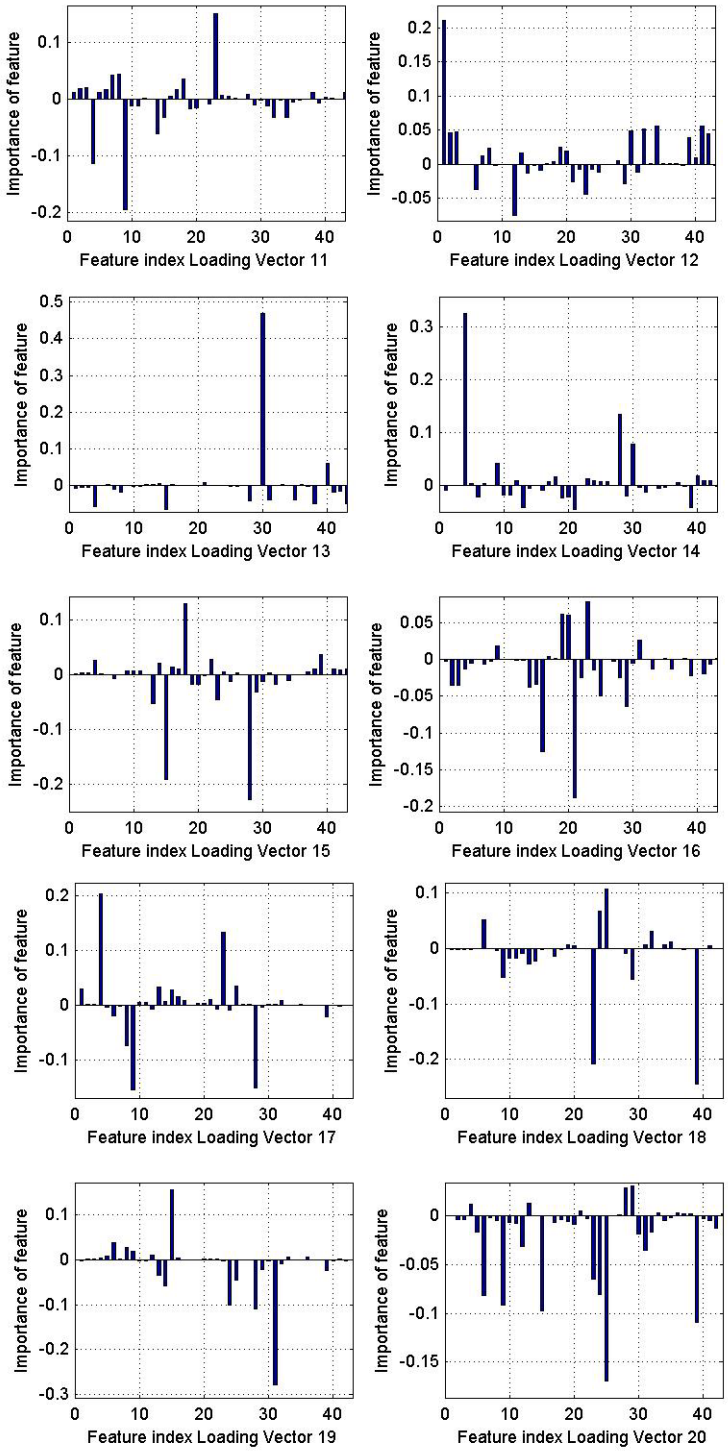
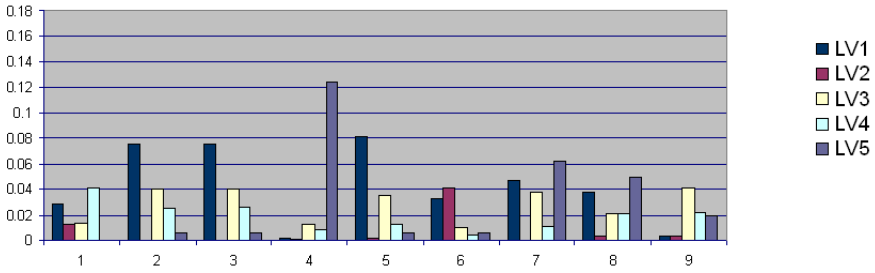
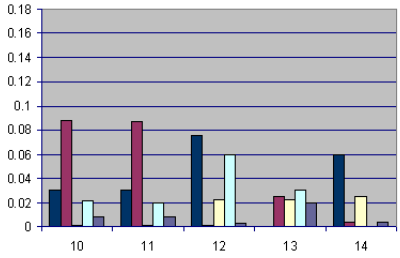


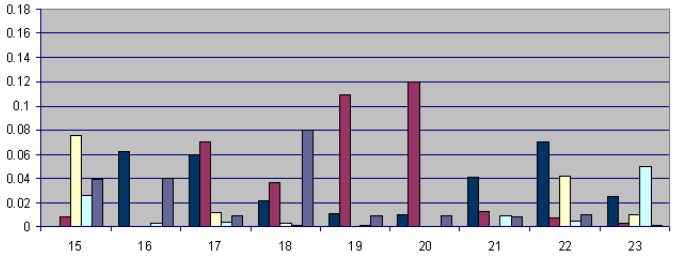
Figure 7.5: Loadings vectors 11-20.



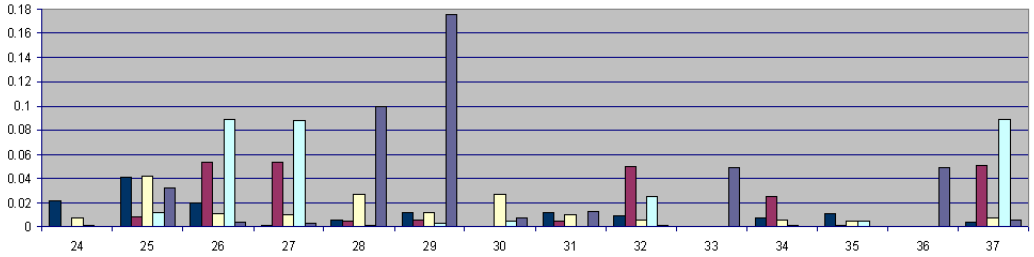
(a) Simple



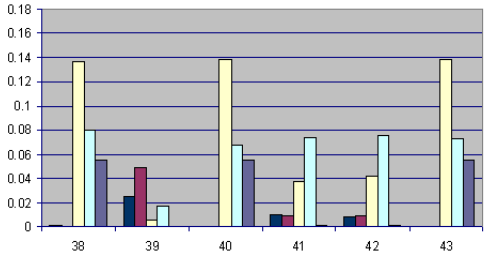
(b) Variance



(c) Structure



(d) Spectrum



(e) Normal and JPL

Figure 7.6: Loadings of each feature in the first five LVs for the Performance problem. Along the x-axis is the feature index; each bar in a feature index corresponds to a loadings vector. The y-axis shows the loadings magnitudes. See Appendix A.1 for the associated name, description and normalization type of each feature index.

not very populated with high-score features; instead, there are only a few that appear to be important along them (e.g. in $LV41$ through $LV44$). Since these vectors are the ones that capture the least information, the features that score high in these vectors are usually those that do not contribute with much information themselves. If such features do not score high in the first (and heaviest) principal components they can be considered as not representative of the data and can be excluded from the experiments. If a feature scores high in these vectors, it means that only a negligible number of observations have that feature as relevant property. Some of these observations, could very well correspond to outliers. One example is feature 13 ($diagonal_{var}/\infty norm$) in $LV29$, which does not appear in any of the first loadings vectors. A different case are those features that never stand out in any loadings vector, however, none of our features strongly displays such behavior.

Furthermore, remember that the information provided by these loadings vectors is better captured by at least one of the first vectors. This means that the behavior of features present along this vectors is better exposed by one or more features in the first vectors, which also represent the majority of the sample population. For example, in Figure 7.8, the plot for $LV40$ shows that features 10 and 11 ($row\_variability$ and $col\_variability$) are inversely correlated, but since this is one of the last loadings vectors it means that they exhibit this behavior *only* for a few observations in the whole sample. If we look instead at the plots for $LV1$ and $LV2$ in Figure 7.4 we can see that these two features are directly correlated. Similar examples are the cases of features 2 and 3 ($trace$ and $trace_{abs}$), 19 and 20 ($left\_bw$ and $right\_bw$), 26 and 27 ($ellipse\_ay$ and $ellipse\_ax$), 33 and 36 ($\lambda_{max,magnitude,\Re}$ and $\lambda_{max}\_rp\Re$), 40 and 41 ($ruhe75\_bound$ and $lee95\_bound$).

It is very difficult to set a threshold for deciding whether a feature is relevant along a loadings vector. This is generally determined depending on the application, preliminary experiments and desired outcome. In our case, the initial process of analyzing the loadings vectors was done manually. We inspected each LV to find what is the highest absolute score achieved by any feature in that particular LV; we then determined that we can mark as "important", those features whose score is at least within 70% of the highest score. Using this heuristic threshold yielded a reasonable outcome in our experiments as we will see in Section 7.4.7, subsequently, it can be used for automatizing the analyzis.

We can now summarize some of the feature-related findings resulting from the different stages of PCA.

7.2.3 Feature Characterization for the Performance problem

After the initial feature elimination described in Section 3.6.5 we are left with a set of features that behave "well" statistically. By looking at the first loadings vectors and the high-scoring features in them, we can identify several features that appear correlated to others. Features like 2 and 3 ($trace$ and $trace_{abs}$), are a good example of correlated features; not only they appear in $LV1$ with high magnitudes, but if we look at other loadings vectors like $LV3, LV6, LV7, LV11$ and $LV31$, we can see that even if they don't score high, they always appear together and their magnitudes are about the same. In our data, we have seen that such features occur in pairs and we can decide to eliminate either of them because the effect of one is mimicked by the other one. These features are:

- $trace$ and $trace_{abs}$
- $symm_{anorm}/symm_{snorm}$ and $symm_{fanorm}/symm_{fsnorm}$
- $row\_variability$ and $col\_variability$

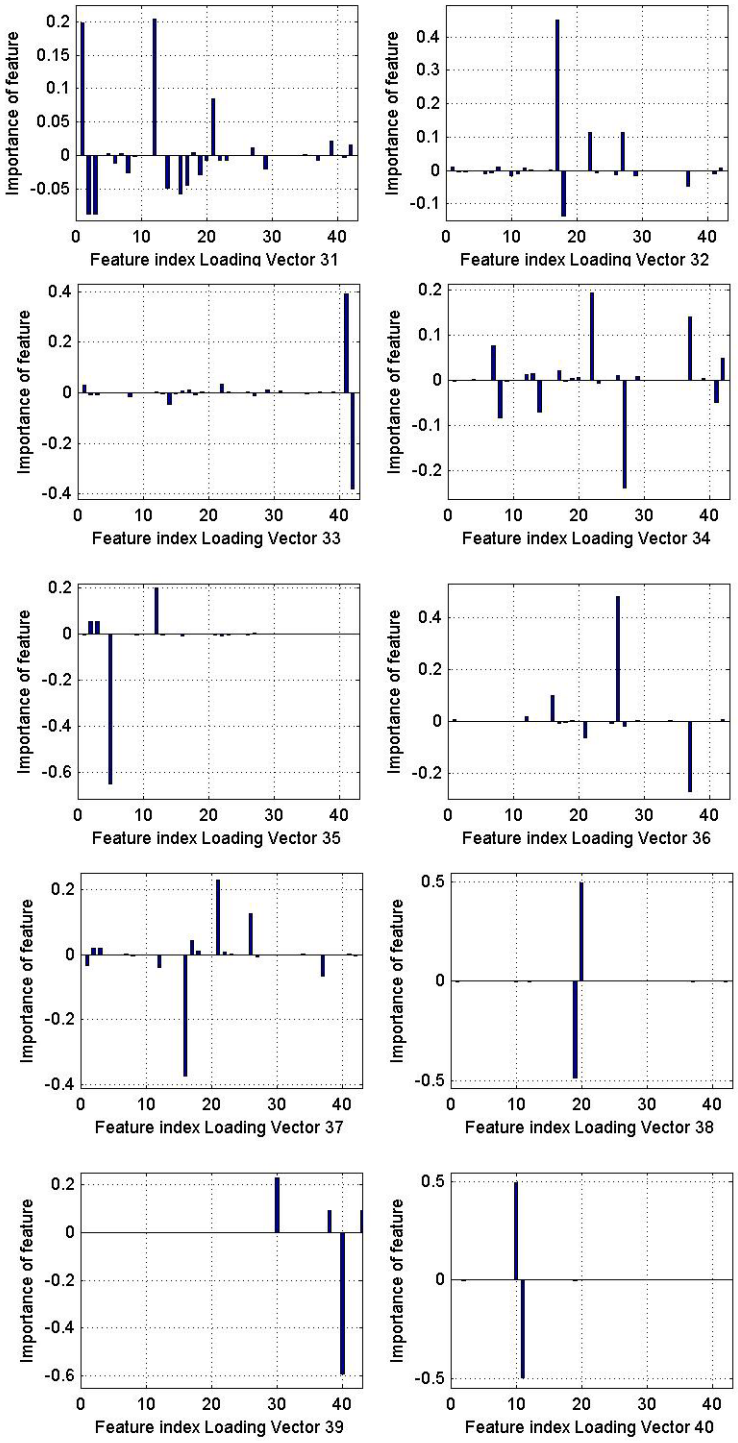


Figure 7.7: loadings vectors 31-40.

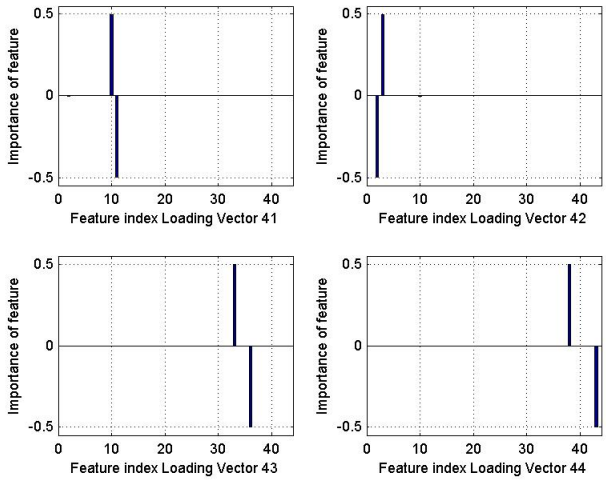


Figure 7.8: Last four loadings vectors (41 through 44).

- max_nnz_{row} and min_nnz_{row}
- $left\_bw$ and $right\_bw$
- $ellipse\_ay$ and $ellipse\_ax$ both normalized by $ellipse\_cx$
- $\lambda_{max,magnitude,\Re}$ and $\lambda_{max,rp,\Re}$ both normalized by $1norm$
- $lee95\_bound$ and $lee96\_lbound$
- $ruhe75\_bound$ and n_colors

Most of the correlations here found make sense from a linear-algebra point of view; the only unforeseen correlation is between $ruhe75\_bound$ and n_colors. This could be a behavior resulting from our dataset in particular, but it is an interesting finding that requires further research.

Also based on the analysis of loadings vectors we have identified several relevant features. These features have a significant contribution to the heaviest principal components. From Figure 7.3, we can see that these are the heaviest components (left) and account for almost 70% of all the information (right). The following is a list of relevant features (some in their normalized form) that load high in the first 7 principal components, we have discarded some of those that appear as correlated:

- $trace$
- $Fnorm/\infty norm$
- $symm_{anorm}/symm_{snorm}$
- $row\_variability$
- $diagonal_{avg}/\infty norm$

- $diagonal_{sign}$
- $symmetry$
- $nnz/nrows^2$
- $left\_bw/nnz^2$
- $diag\_definite$
- $ellipse\_ay/ellipse\_cx$
- κ
- $positive\_fraction$
- $\lambda_{max,magnitude,\Re}/1norm$
- $\lambda_{max,rp,\Im}/diag\_avg$
- $trace^2$
- $lee95\_bound$
- n_colors

In addition, Figure 7.3 shows that principal components 8 through 25 account for almost 30% of the remaining information. There are other features like $nrows$, n_ritz_values, $\sigma_{min}/1norm$, $\lambda_{max,magnitude,\Im}/diag\_avg$ and $ellipse\_ay/ellipse\_ax$, which may not load high in the first loadings vectors, however they are relevant in several of these "secondary" ones.

Although we can identify several relevant features, the behavior of iterative solvers does not depend individualy only on these, but rather on a composition of features that contribute in different amounts.

7.2.4 Feature Characterization for the Reliability Problem

In the Reliability problem, the observation space is split in two for every method available: "converge" and "diverge" classes. In this section we focus on the feature analysis for the reliability of iterative methods using as an example the results for $bcgs$, i.e. we want to determine the behavior of the features that can help us to determine whether $bcgs$ converges or not for a problem. The case of $bcgs$ is used as an example here, because for most of the other methods the feature arrangement along the loadings vectors are very similar.

First we detect and eliminate "noisy" features:

- $\lambda_{max,magnitude,\Im}/\lambda_{max,magnitude,\Re}$: $\mu = -2.02 \times 10^{-1}$, $stdv = 7.86 \times 10^{+1}$
- $\lambda_{max,magnitude,\Re}/\lambda_{max,magnitude,\Im}$: ∞
- $\lambda_{min,magnitude,\Im}/\lambda_{max,magnitude,\Im}$: NaN
- $\lambda_{max,rp,\Im}/diag\_avg$: $\mu = -2.32 \times 10^{4}$, $stdv = 4.58 \times 10^{6}$

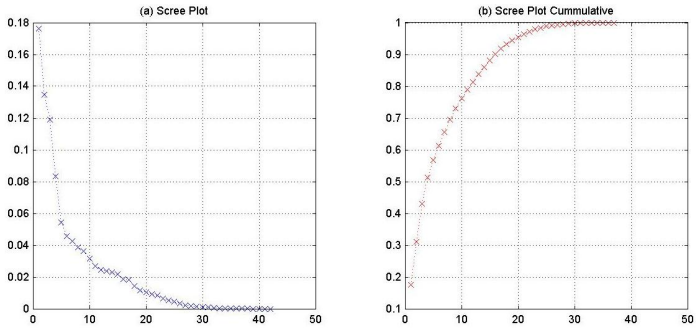


Figure 7.9: Scree plots for the Reliability problem.

Note that $\lambda_{max,magnitude,\Im}$ must be 0 in many observations in this class, which makes some of these normalized features impossible to use in global statistics. In such case we have to decide whether to eliminate the affected feature over all the set, or just eliminate those observations that exhibit this problem. For the learning and classification stage we chose the later option, but in the recommendation problem we have to eliminate the feature because if it is not available in the given problem we cannot use it as part of the decision process.

The rest of the features are analyzed with the same guidelines as we did for the Performance problem example. Figure 7.9 shows the scree plots, very similar to the ones for the Performance problem in Figure 7.3. These indicate that using approximately the first 25 principal components, is enough to capture almost the totality of the information. Also, we see that the first 7 principal components are the heaviest and account for more than 60% of the information.

The first loadings vectors appear in Figure 7.10. Important correlations found in these vectors are:

- $trace$ and $trace_{abs}$
- $symm_{anorm}/symm_{snorm}$ and $symm_{fanorm}/symm_{fsnorm}$
- $row\_variability$ and $col\_variability$
- $\sigma_{max}/1norm$ and $\sigma_{min}/1norm$
- $\lambda_{max,magnitude,\Re}/1norm$, $\lambda_{max,rp,\Re}/1norm$ and $ruhe75\_bound$
- $lee95\_bound$, $lee96\_lbound$ and $lee96\_ubound$

And the relevant features are:

- $trace$
- $Fnorm/\infty norm$
- $diag\_dominance/\infty norm$
- $symm_{anorm}/symm_{snorm}$

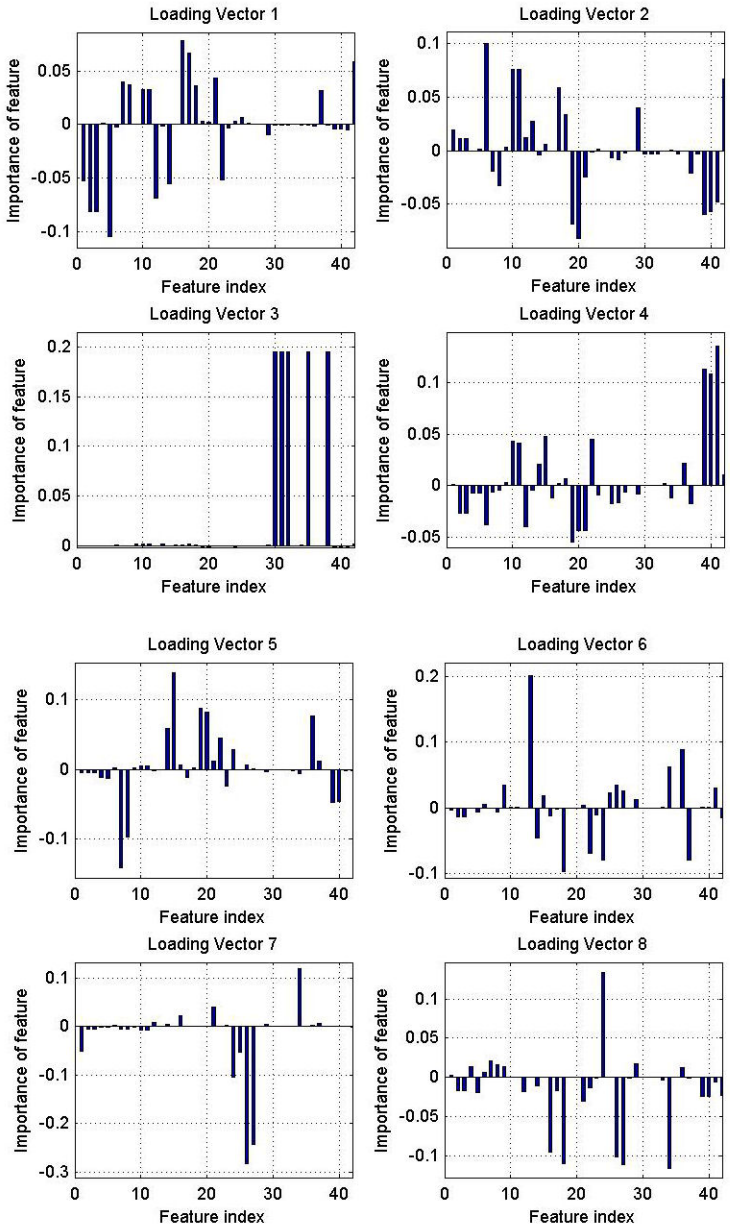


Figure 7.10: loadings vectors 1-8.

- $row\_variability$
- $diagonal_{avg}/\infty norm$
- $diagonal_{sign}$
- $symmetry$
- $nnz/nrows^2$
- $left\_bw/nnz^2$
- $diag\_definite$
- $ellipse\_ay/ellipse\_cx$
- $ellipse\_ax/ellipse\_cx$
- $\sigma_{max}/1norm$
- $\lambda_{max,magnitude,\Re}/1norm$
- $ruhe75\_bound$
- n_colors

The last four loadings vectors appear in Figure 7.11, and show some features and correlations that also appear in the first loadings vectors; however, this information most likely come from observations that are outliers.

It is important to note the differences and similarities between the lists of relevant features for the Performance problem (List 7.2.3) and for the Reliability problem (List 7.2.4). Many of these features seem to characterize both the Reliability and Performance behavior, although there are several differences, which show in the spectrum category of features. This can lead us to think that spectral features are indeed, very important for determining both the performance and the reliability of solvers, and unfortunately these features are also the most expensive to compute. Future research will focus on the study of correlations with other "cheaper" features, to determine whether it is possible, to some extent, to limit the use of spectral features as a basis for classification.

7.2.5 PCA and Classification Approaches

The information extracted from the Loadings Vectors is valuable and can be used to understand which features affect the most the behavior of certain iterative methods, and which ones do not have much influence. In Definition 10 the function Γ is presented as an optional function that can be used to preprocess the feature vectors extracted from input problems. PCA has proved a useful tool to evaluate the importance of different features, but it can also be used as a preprocessing step that can be advantageous for certain classification techniques. The transformation of the linear space shaped by the original features yields a new space with less dimensions, which allows us to visualize more easily how the different transformed features project into the new space.

Consider for example the Reliability problem for the iterative method $gmres$. Figure 7.12(a), which shows two plots of the observations along three randomly selected dimensions in the original feature space, does not exhibit any clear separation of the observations from the two different

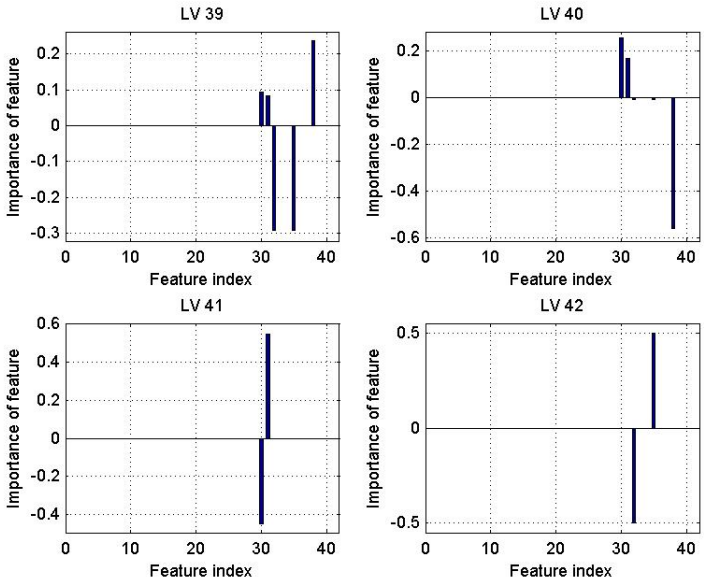


Figure 7.11: Last four loadings vectors (39 through 42).

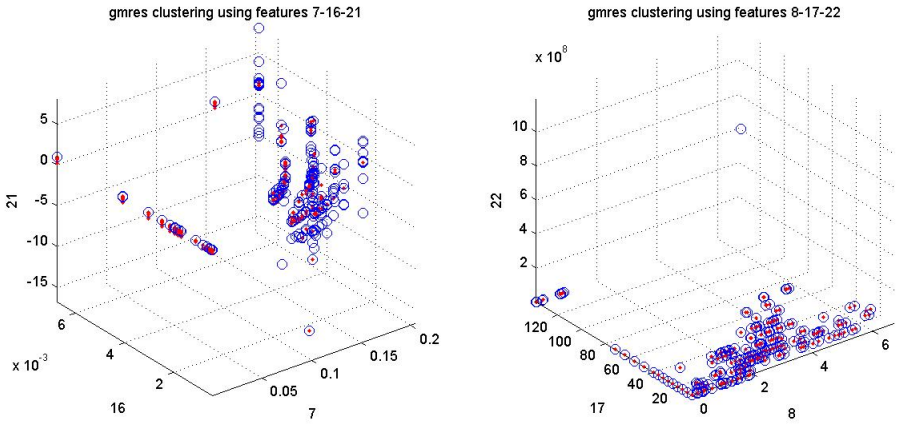
classes. Now, if we look at Figure 7.12(b), which plots the observations along three principal components in the transformed feature space. It shows a clearer separation of the observations of the two different classes. Just like it is easier for us to better distinguish between the different clusters in a transformed space it might be easier for certain classifiers to do the same, specially those classification methods based on spatial distances (e.g., simple Gaussian, K-means or SVM). Even some methods like the kernel density estimators can benefit from such a process. Furthermore, PCA constructs as a result from SVD an *orthogonal* new feature space, which is actually more applicable for multi-variate Bayesian approaches (see Equation 3.5).

7.3 The Reliability Problem

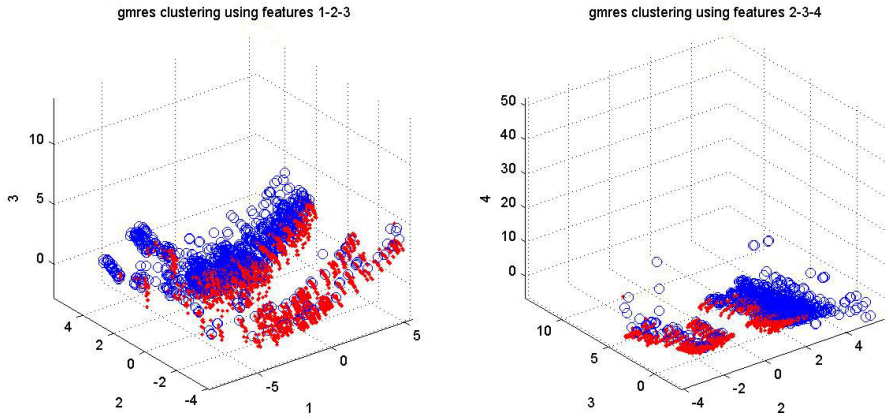
The Reliability problem is the first step for a recommendation. In some cases it may be more important to determine how safe a method is rather than its speed. In here we discuss the results obtained for the Reliability problem classification and recommendations for iterative methods, preconditioners and combinations of these. The experiments consisted of training classifiers to predict whether a method converges or not when trying to solve a linear system. Remember that in the Reliability problem, there are two classes per method: set of observations for which the method converges, and the set for which it diverges.

7.3.1 Reliability Classification

The second column of Table 7.2 shows the accuracies with which we can determine if an iterative method is reliable or not. The reliability of an iterative method in this case is evaluated from its overall performance with the different preconditioners, i.e. each preconditioned problem A is



(a) Distribution of observations in 3D in original feature space. (Left) The axes correspond to three randomly chosen original features. (Right) The axes correspond to each of the previous features incremented by one.



(b) Distribution of observations in 3D principal component space. Here the term "features" refers to the PCA-transformed new features, that is, the principal components. (Left) shows the plot using the first 3 principal components. (Right) shows the plot of the same data but using principal components 2,3,4.

Figure 7.12: 3D plot for the classes $gmres_{converge}$ and $gmres_{diverge}$: red markings correspond to the observations in the $converge$ class and blue markings to those in the $diverge$ class.

Table 7.2: Accuracy measurements for classification of *ksp*: the second column is the accuracy of prediction, third column is the false positive error and last column shows how good the recommendation is.

ksp	$\alpha_{converge} \pm z$	FP	$\mathcal{Q}^{reliability}$
bcgs	0.97±0.01	0.027	0.830
bicg	0.98±0.01	0.003	0.082
cgne	0.79±0.02	0.053	0.702
gmres	0.94±0.01	0.093	3.011
fgmres	0.92±0.01	0.073	2.150
lgmres	0.95±0.01	0.089	2.560
tfqmr	0.97±0.01	0.022	0.667

Table 7.3: Accuracy measurements for classification of preconditioner: the values in the second and third columns refer to accuracy of prediction for each class.

pc	Avg. $\alpha_{converge} \pm z$	Avg. $\alpha_{diverge} \pm z$
asm	0.84±0.02	0.81±0.04
bjacobi	0.89±0.01	0.84±0.01
boomeramg	0.80±0.02	0.80±0.03
ilu	0.86±0.01	0.93±0.02
parasails	0.81±0.02	0.80±0.02
pilut	0.81±0.02	0.80±0.02
rasm	0.82±0.02	0.85±0.02
silu	0.87±0.01	0.91±0.01

considered as a different observation, and is assigned to a class depending on the behavior of the iterative method. Observe that the accuracy of prediction for every method is very high, except for $cgne$; this is related to the fact that there is not enough experience stored in the database regarding cases where $cgne$ converged (see Figure 7.1(b)).

In Table 7.3 are the classification results for different preconditioners. The accuracies of classification of each preconditioner are the average taken over all the iterative methods as indicated in Equation 5.9. Although this gives a good approximation of how trustworthy a prediction may be, it disregards the fact that the reliability of a preconditioner is greatly related to the iterative method used. These results can be useful for preconditioner-only recommendations, where we are interested on evaluating the general reliability of a preconditioner.

The best type of reliability prediction we can obtain is for particular combinations of preconditioner and iterative method $\langle pc, ksp \rangle$. Determining whether such a combination converges or not for a problem A is more difficult because the training dataset contains only observations preconditioned with pc, which means fewer training data. This could possibly result in overfitting because there may be cases where there are too few observations in the corresponding convergence or divergence class. However, an important experimental result that yielded the conditional strategy for reliability classification, is that the behavior of classifiers is highly dependent on the iterative method used.

In Section 5.1.2 we discussed why it is not possible to make a prediction based solely on a preconditioner. Additional experiments showed that unless the iterative method is taken into account, is not even possible to correctly group observations into convergence and divergence classes. As an example, consider the 3D plot on PCA-transformed space (using the first three principal components) for the preconditioner $bjacobi$ in Figure 7.13 left, and the plot in the same space for the combination $\langle bjacobi, bcgs\rangle$ Figure 7.13 right. Observe the amount of overlap between the two classes on the left, as opposed to the plot on the right, which offers a "cleaner" distinction between the observations in each class.

Table 7.4 summarizes the accuracy of prediction (Reliability problem) for each combination of available preconditioners with available iterative methods using the *conditional approach* (Section 5.3.3). Each row corresponds to a preconditioner with each of the available iterative methods. Although this is a very economic and relatively accurate approach, it has the problem that it may result in overfitting because of the smaller size of sets. An example is the entry for $boomeramg$ and $cgne$, cases for which there are very few examples stored in the database. For such cases, the *orthogonal* or *sequential* approach would be more convenient. The selection of the approach depends on the data and also on the associated expected loss. For this particular combination $boomeramg, cgne$ the expected loss is maximum, because the false positive ratio is maximal (the accuracy for convergence is 0).

7.3.2 Reliability Recommendations

In the Reliability problem, a recommendation consists of a set of methods that are the most likely to converge to a solution when solving a problem A. Besides evaluating the accuracy of a classifier it is very important to measure the risk involved on taking a particular recommendation. In Section 3.7.1 we discussed the statistical error analysis for two-class problems and penalties.

For our type of problem, the "price" of making an error can be measured as the time we have to wait when picking the wrong solver. So the mistake of recommending a method that does not converge should be penalized more than the mistake of not recommending a method that in reality converges. The first error (and the most dangerous) is measured with the ratio of False Positive errors (FP for short): given problem A, for method M assigning $\Phi(A)$ to $M_{converge}$ when in reality belongs to $M_{diverge}$; in such case, we wait "infinite" time. The other possible error is to discard a converging method, in which case the worse situation is if that method had also turned to be the optimal solver, then we would have to wait the additional time that another method (or a default) takes to solve the problem A.

To assign the penalty consider the following iterative method classification example. Table 7.2 shows in the second column the False Positive error FP for each method, and the last column shows its effect on how good a recommendation is (see the definition for the reliability $\mathcal{Q}$ in Section 5.4). The classifiers for this examples are effective in general, even for the cases where the ratio of false positive errors are higher. However, suppose that for a particular problem A all these methods are recommended as "reliable"; if we want to go with safer choices we would rather not pick $gmres$, $fgmres$ or $lgmres$.

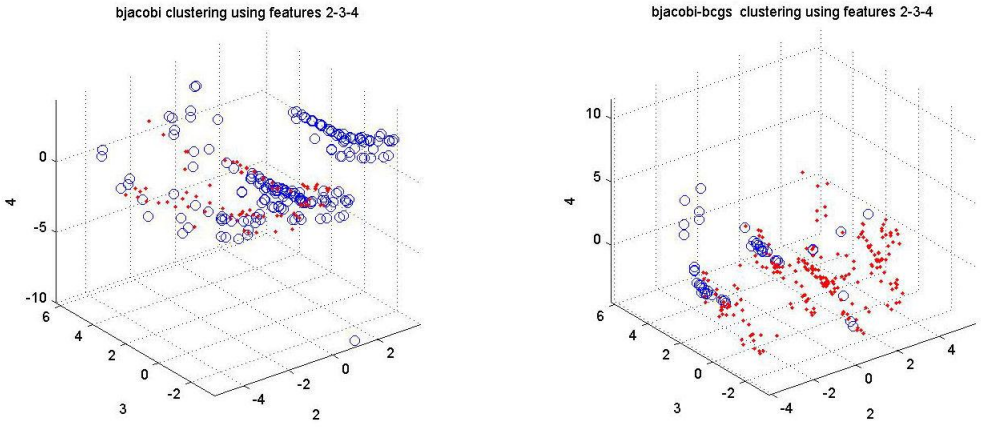


Figure 7.13: Distribution of observations in 3D principal component space for classes $bjacobi_{converge}$ and $bjacobi_{diverge}$: (left), and $\langle bjacobi, bcgs\rangle_{converge}$ and $\langle bjacobi, bcgs\rangle_{diverge}$ (right): red markings correspond to observations in the $converge$ class and blue markings to those in the $diverge$ class. The axes correspond to the first three principal components (or dimensions in the transformed space).

Table 7.4: Accuracy measurements α for classification of $\langle pc, ksp\rangle$.

	Reliability Class	$bcgs$	$cgne$	$fgmres$	$gmres$	$lgmres$	$tfqmr$
asm	converge	0.83	0.88	0.80	0.78	0.87	0.89
	diverge	0.72	0.80	0.82	0.83	0.78	0.95
bjacobi	converge	0.84	1	0.94	0.91	0.93	0.88
	diverge	0.89	0.82	0.86	0.85	0.77	0.89
boomeramg	converge	0.96	0	0.98	0.97	0.97	0.95
	diverge	0.98	0	0.93	1	0.92	1
ilu	converge	0.87	0.89	0.87	0.90	0.85	0.94
	diverge	0.97	1	0.96	0.96	0.89	0.99
parasails	converge	0.99	0	0.96	0.97	0.97	1
	diverge	0.98	0	0.92	1	0.98	0.95
pilut	converge	0.98	0	0.98	0.97	0.97	1
	diverge	0.98	0	0.91	1	0.98	0.95
rasm	converge	0.93	0.77	0.87	0.87	0.90	0.75
	diverge	0.81	0.82	0.98	0.92	0.80	0.96
silu	converge	0.98	0.88	0.88	0.93	0.86	0.92
	diverge	0.95	1	0.95	0.89	0.80	0.98

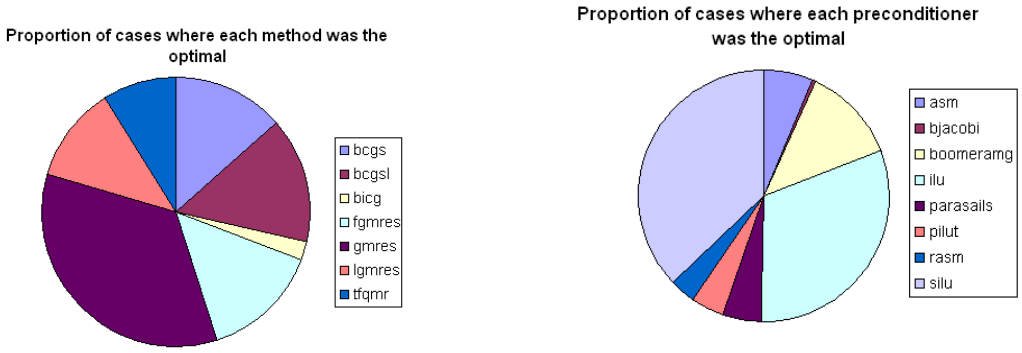


Figure 7.14: Proportion of cases where (a) each iterative method was the optimal and (b) each preconditioner was the optimal.

7.4 The Performance Problem

In order to understand and interpret the results from the Performance experiments it is important to acknowledge what the current "experience" is that is stored in the database, as far as methods performance is concerned. Figure 7.14(a) shows the proportion of the runs for which each method was the best, and Figure 7.14(b) those for which some method was the best using a particular preconditioner.

Figure 7.15 shows a performance comparison for different (reliable) solvers applied to problems from the Matrix Market dataset, similar to the one in Figure 1.1 of Chapter 1. The default method is also $gmres20$ and the slowdown and speedup ratios are also computed, for a problem A as:

$$\frac{T(A, slower\_method)}{T(A, faster\_method)}$$

In this case, again, the default is not much faster than other methods (Figure 7.15(a)), and other methods can be much faster than the default, in which case there is a lot of room for improvement by picking a different method.

Remember that for the Performance problem, the more methods or preconditioners we have available, the more classes there are and the harder it becomes to differenciate between these. The discrimination between all the classes becomes even more difficult for smaller datasets. The results in Table 7.5 show this problem. In Table 7.5(a) we have the classification accuracy for two classifiers (kernel density estimator and decision trees) working on seven methods. Table 7.5(b) shows the accuracies for these classifiers when trying to differenciate between eight preconditioners. In both cases, the accuracy of the classifiers is not very high, yet, it is still better than making a random choice that would yield an $\alpha = \frac{1}{number\ of\ available\ methods}$ (0.14 for this experiment).

These accuracies can be improved by using the hierarchical classification strategy (Section 6.1.4). In the following sections we will show how the use of this strategy and *superclasses* can help to obtain better predictions.

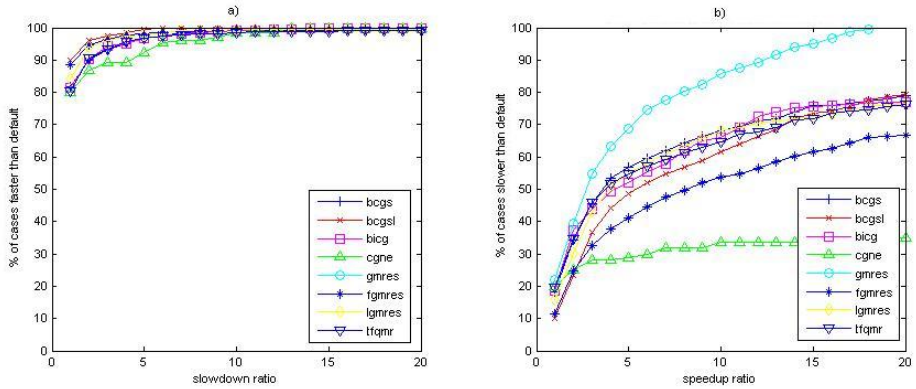


Figure 7.15: Performance profile for iterative methods with respect to default method $gmres20$: (a) Shows the cumulative proportion of cases for each ksp where the default method was faster for different slowdown ratios. (b) Shows the cumulative proportion of cases for each ksp where the default method was slower for different speedup ratios.

Table 7.5: Accuracy of classification and statistical error in the Performance problem using one class per available method (left) or preconditioner (right). The second column in each table is the accuracy of prediction using Kernel Density Estimators classifier and the third column is using Decision Trees.

(a) Iterative Methods

ksp	$\alpha^{KDE} \pm z$	$\alpha^{DT} \pm z$
bcgsl	0.29±0.06	0.59±0.02
bcgs	0.19±0.15	0.71±0.03
bicg	0.28±0.03	0.68±0.06
fgmres	0.26±0.03	0.80±0.02
gmres	0.19±0.03	0.59±0.04
lgmres	0.19±0.07	0.81±0.03
tfqmr	0.21±0.05	0.61±0.05

(b) Preconditioners

pc	$\alpha^{KDE} \pm z$	$\alpha^{DT} \pm z$
asm	0.25±0.04	0.72±0.05
bjacobi	0.52±0.22	0.11±0.11
boomeramg	0.11±0.06	0.71±0.06
ilu	0.18±0.06	0.66±0.02
parasails	0.35±0.16	0.46±0.12
pilut	0.17±0.11	0.80±0.06
rasm	0.14±0.06	0.70±0.04
silu	0.17±0.18	0.83±0.02

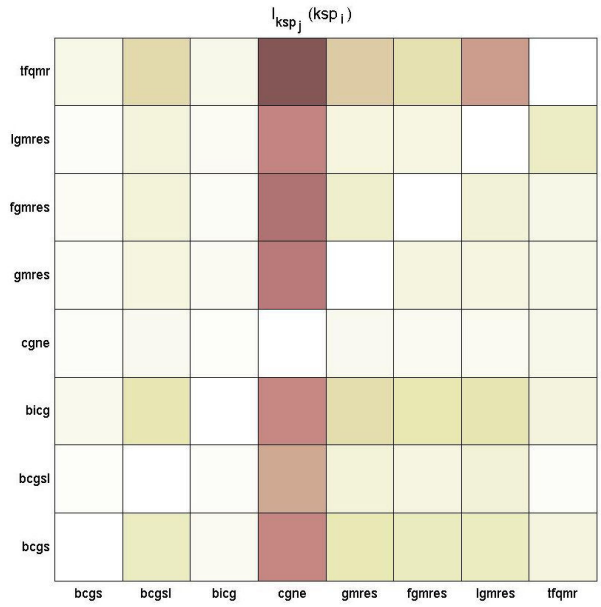


Figure 7.16: Independence ratios for iterative methods: (a) the shade of color indicates the ratio between two ksps. The darker the color the more independent two methods are; "i" represents the rows and "j" the columns.

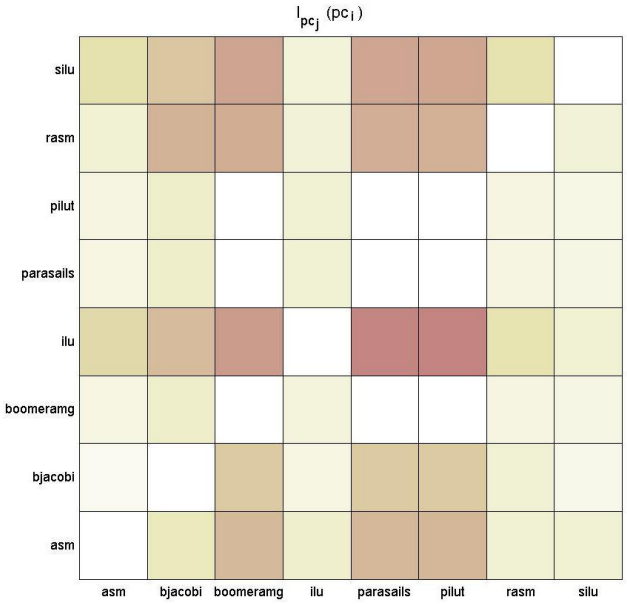


Figure 7.17: Independence ratios for preconditioners: (a) the shade of color indicates the ratio between two pcs. The darker the color the more independent two methods are; "i" represents the rows and "j" the columns.

7.4.1 Building Superclasses for Hierarchical Classification

Hierarchical classification is based on creating superclasses, which are formed by grouping methods or preconditioners based on their similarity. Similarity can be measured using the independence ratio In, covering ratio CR and visual comparisons of the $speedup$ factor trends (these were discussed in Section 6.1.4).

An interesting matter is to see whether there are some methods that "cover" others, i.e. almost all the problems that a method can solve are also solved by another method. On the other hand, there is also the question of whether there are methods that work *only* for a subset of problems for which other methods break down. We use this information to create superclasses (Definition 41), which can improve the accuracy of prediction in the Performance problem as we will see in Section 7.4.4. In this section we show how we determine if two methods are similar or not; similar methods can then be grouped into superclasses.

A preliminary analysis of results from exhaustive comparisons among all methods (using different datasets) motivated the development of the concepts of method independence In and covering ratio CR (Definitions 38 and 39 respectively), and the determination of the corresponding experimental thresholds discussed in Section 6.1.4. Figure 7.16 illustrates the In measurements for different pairs of iterative methods. In this figure, we can see that other methods and $cgne$ are very independent (look at the column $I_{cgne}(M)$), numerically this also makes sense because $cgne$ is an iterative method very different to the others. On the other hand, the row $I_M(cgne)$ has lighter shade indicating that $cgne$ is less independent from other methods, the reason is mainly because there are not enough examples of converging runs with $cgne$ in our database. The darker regions for $gmres$, $fgmres$ and $lgmres$ indicate these methods are more independent from $tfqmr$, $bcgs$, $bcgsl$ and $bicg$ than they are among themselves. These results are consistent with the fact that the numerical implementation of the "$gmres$" methods is different than those for the "$bcgs$" and $tfqmr$ methods. Additionally "$bcgs$" and $tfqmr$ are numerically similar techniques, and this figure shows more dependence between them.

Similarly, Figure 7.17 shows the independence ratios for every pair of preconditioners. In this figure we can obtain important regarding different preconditioners. For instance, $parasails$ and $pilut$ are very independent from any other preconditioner excluding $boomeramg$, suggesting that we can group these into a superclass. However, ilu and $pilut$ seem to be very independent, which is an interesting result because $pilut$ is a parallel implementation of ilu and they would be expected to behave similarly. However, all of our experiments were done in single-processor, which suggests that in such case ilu and $pilut$ behave very differently. This poses an interesting research question, but it is beyond the scope of this dissertation. Overall, this figure shows that there are very few preconditioners that are very independent from others. This affects the classification process because it is more difficult to distinguish between preconditioners that are more dependent. It is also harder to group in superclasses.

To determine similarity, it is also important to study those cases where one method is faster than the other and by how much. Similar behavior of the speedup factor of two methods is another indication that the methods are similar; remember that the *speedup* factor expresses how much faster one method is with respect to the other for a problem A

$$speedup = \frac{T(A, M_2)}{T(A, M_1)}.$$

Next, we present examples of how we use these ideas to find which methods and preconditioners are similar and which are not to form superclasses. Once we have formed superclasses for iterative methods and preconditioners, we can test the classification process; these results will be presented in Sections 7.4.4, 7.4.6 and 7.4.6. Results presented for performance recommendation are also from experiments where superclasses were used.

7.4.2 Superclasses for Iterative Methods

First, consider methods $bcgs$ and $bicg$. Figure 7.18(a) shows that the proportion of cases for which either of them converged is very small compared to that where both of them converged and to that for which neither of them did. This suggests that these methods may not be independent. Additionally, the associated CR is 0.152355, which indicates that they both can solve almost the same problems. Based on this, these two methods are catalogued as similar and put into the same superclass. Furthermore, Figure 7.18(b)(c) shows that the speedup trends are very close; for instance, note that one method is faster than the other by a factor of 1 for almost all the cases.

Furthermore, another way to determine method similarity is to compare methods which we suspect to be similar with other methods, side by side. For example, if $bcgs$ and $bicg$ are potentially similar we can compare $bcgs$ vs. $gmres$ and $bicg$ vs. $gmres$. From Figures 7.19 and 7.20 we can see that both methods behave comparably with respect to $gmres$, which in turn is not similar to either of them, observe for example the big difference in the speedup trends.

Now consider the example of the methods $tfqmr$ and $cgne$. If we look at Figure 7.21(a), we can see that there are very few methods for which both work as opposed to the number of methods for which only $tfqmr$ works; the corresponding $CR = 5.584337$ shows that the intersection of problems that they both solve is very small. These methods are marked as different. In this example, there is no other method similar to either of them, so they consitute their own superclasses.

By comparing side by side all the available methods in this manner, we can form a set of superclasses for the hierarchical classification of iterative methods:

- **B**=$\{bcgs, bcgsl, bicg\}$: from Figure 7.16 we can see that the independence In for these three methods is nearly 0, which indicates that they are very dependent on each other; also the associated CRs values are less than 0.2, which suggests that one method covers the other. This is also depicted in Figure 7.18(a), which shows how the number of cases for which both converge and diverge is much bigger than the number of cases when either of them did.

- **G**=$\{gmres, fgmres, lgmres\}$: in a similar manner, we found experimentally that these methods behave in a similar way and they are highly dependent from each other (see the corresponding entries in Figure 7.16). CR values for each pair of these methods is below 0.4.

- **T**=$\{tfqmr\}$: numerically, this method is a modification of $bcgs$, however, there is not enough experimental evidence to group them together in the superclass **B**, so we put it in its own class [‡].

- **C**=$\{cgne\}$: this method is quite different from any other presented here, both in terms of coverage and independence. Numerically, we know that the nature of this approach is different than other methods. Unfortunately, there is very little experience in our database regarding

[‡]By definition, $tfqmr$ is a modification of $bcgs$ [Freund and Nachtigal, 1991, Freund, 1993] as mentioned in Section 2.1.1.

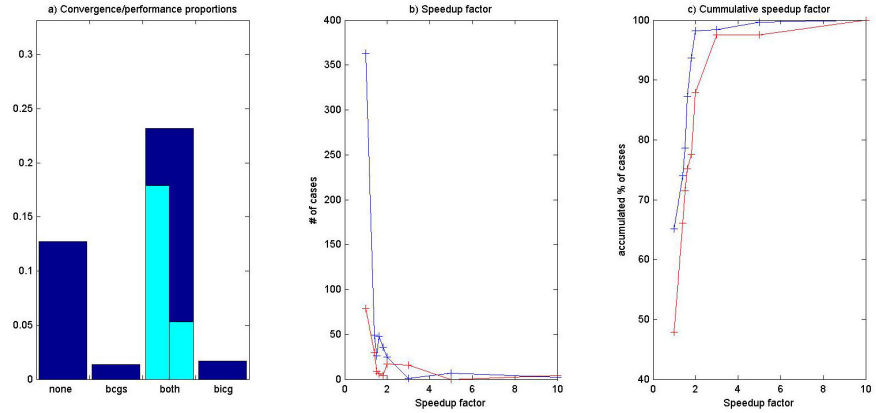


Figure 7.18: (a) Proportion of problems for which none of these methods converged, only of them converged, and both of them converged. For the last case, the bin labeled as "both" shows the proportion when each was faster than the other method. (b)(c) Distribution of number cases by speedup factor: the red line corresponds to the case when $bcgs$ was faster than $bicg$ and blue line when it was slower. Along the x-axis is the factor by which either method is faster than the other one. Along the y-axis are (b) the number of cases per each factor, and (c) the accumulated percent of cases at each factor.

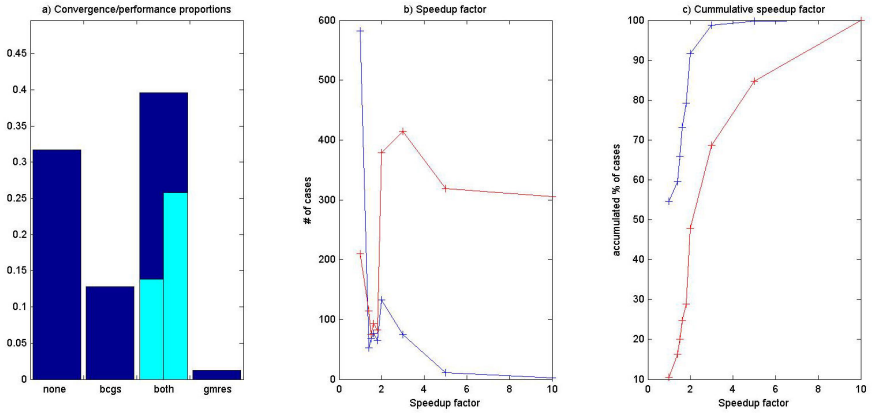


Figure 7.19: (a) Proportion of problems for which none of these methods converged, only of them converged, and both of them converged. For the last case, the bin labeled as "both" shows the proportion when each was faster than the other method. (b)(c) Distribution of number cases by speedup factor: the red line corresponds to the case when $bcgs$ was faster than $gmres$ and blue line when it was slower. Along the x-axis is the factor by which either method is faster than the other one. Along the y-axis are (b) the number of cases per each factor, and (c) the accumulated percent of cases at each factor.

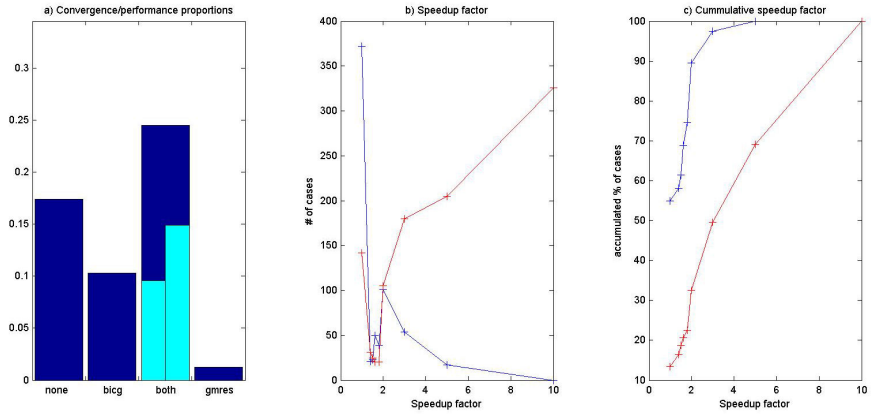


Figure 7.20: (a) Proportion of problems for which none of these methods converged, only of them converged, and both of them converged. For the last case, the bin labeled as "both" shows the proportion when each was faster than the other method. (b)(c) Distribution of number cases by speedup factor: the red line corresponds to the case when $bicg$ was faster than $gmres$ and blue line when it was slower. Along the x-axis is the factor by which either method is faster than the other one. Along the y-axis are (b) the number of cases per each factor, and (c) the accumulated percent of cases at each factor.

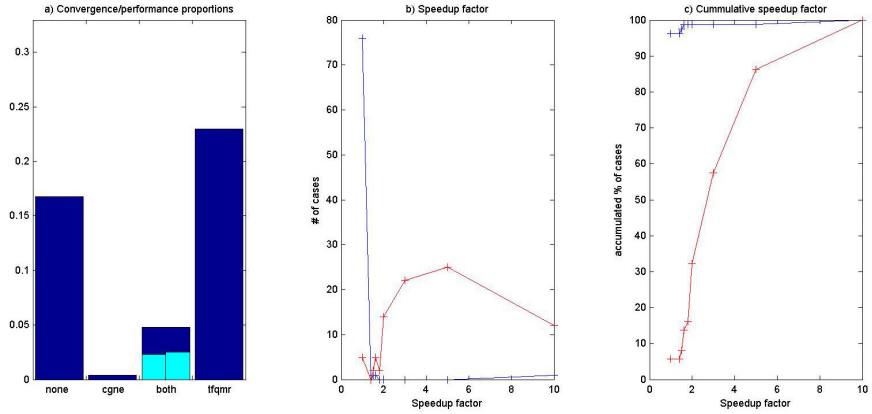


Figure 7.21: (a) Proportion of problems for which none of these methods converged, only of them converged, and both of them converged. For the last case, the bin labeled as "both" shows the proportion when each was faster than the other method. (b)(c) Distribution of number cases by speedup factor: the red line corresponds to the case when $cgne$ was faster than $tfqmr$ and blue line when it was slower. Along the x-axis is the factor by which either method is faster than the other one. Along the y-axis are (b) the number of cases per each factor, and (c) the accumulated percent of cases at each factor.

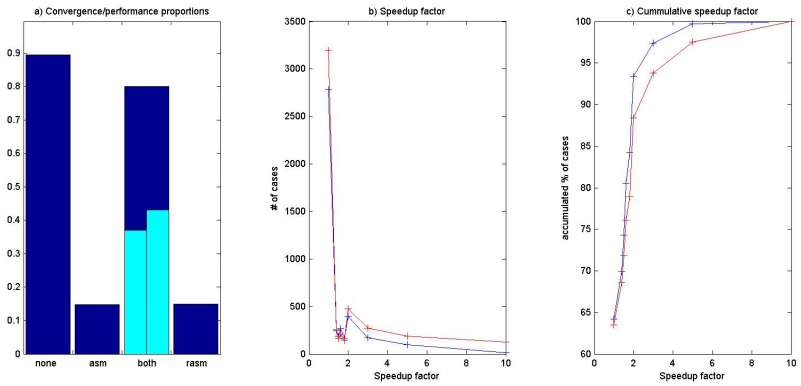


Figure 7.22: (a) Proportion of problems for which none of these methods converged, only of them converged, and both of them converged. For the last case, the bin labeled as "both" shows the proportion when each was faster than the other method. (b)(c) Distribution of number cases by speedup factor: the red line corresponds to the case when asm was faster than $rasm$ and blue line when it was slower. Along the x-axis is the factor by which either method is faster than the other one. Along the y-axis are (b) the number of cases per each factor, and (c) the accumulated percent of cases at each factor.

this method, so for statistical purposes we have not included this experiment in several experiments because it may introduce noise.

7.4.3 Superclasses for Preconditioners

We can do the same type of comparisons for preconditioners. Take for instance Figure 7.22, which shows preconditioners asm and $rasm$ that are similar ($rasm$ is a modification of asm). Figure 7.17 (both $In_{asm}(rasm)$ and $In_{rasm}(asm)$) indicates that they are more dependent than independent, and the corresponding CR for these preconditioners is 0.366132. On the other hand, Figure 7.23 shows an example of two preconditioners, $boomeramg$ and ilu, that behave differently; Figure 7.17 shows that while ilu is dependent from $boomeramg$, $boomeramg$ is very independent from ilu, in such case the CR measurement becomes useful and in this case it is 1.310578

Following the same guidelines as with the iterative methods, for preconditioners we have formed the following superclasses:

- **A** = $\{asm, rasm, bjacobi\}$
- **BP** = $\{boomeramg, parasails, pilut\}$
- **I** = $\{ilu, silu\}$

7.4.4 Classification of Iterative Methods

Table 7.6 shows the accuracy outcome using the hierarchical classification approach with two classification methods $-$ kernel density estimation and decision trees. Compare the values in the columns

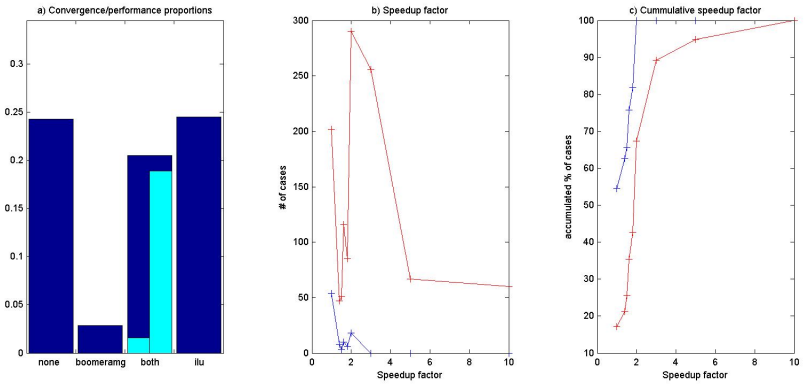


Figure 7.23: (a) Proportion of problems for which none of these methods converged, only of them converged, and both of them converged. For the last case, the bin labeled as "both" shows the proportion when each was faster than the other method. (b)(c) Distribution of number cases by speedup factor: the red line corresponds to the case when $boomeramg$ was faster than ilu and blue line when it was slower. Along the x-axis is the factor by which either method is faster than the other one. Along the y-axis are (b) the number of cases per each factor, and (c) the accumulated percent of cases at each factor.

Table 7.6: Hierarchical classification for iterative methods

Superclass	Class	α^{KDE}	KDE Totals	α^{DT}	DT Totals
B		0.73		0.95	
	bcgs	0.59	0.43	0.93	0.87
	bcgsl	0.69	0.51	0.92	0.87
	bicg	0.15	0.11	0.89	0.84
G		0.87		0.98	
	fgmres	0.88	0.77	0.96	0.94
	gmres	0.53	0.46	0.91	0.89
	lgmres	0.75	0.66	0.94	0.93
T		0.79		0.91	
	tfqmr	–	0.79	–	0.91

Table 7.7: Hierarchical classification for preconditioners

Superclass	Class	KD Accuracy	KD Total	DT Accuracy	DT Total
A		0.97		0.95	
	asm	–	–	0.98	0.93
	bjacobi	–	–	0.67	0.64
	rasm	–	–	0.82	0.78
BP		0.93		0.99	
	boomeramg	0.65	0.61	0.80	0.80
	parasails	0.75	0.70	0.78	0.77
	pilut	1.00	0.93	0.97	0.96
I		0.92		0.94	
	ilu	0.82	0.75	0.82	0.75
	silu	0.97	0.90	0.97	0.91

labeled as "Totals" for the two classifiers with the values in Table 7.5(a). Observe the improvement for both classifiers using superclasses and the hierarchical approach.

One of the main benefits besides improving the accuracy of classification is that it can better help in handling overfitting. Remember that a reason for overfitting is having very small training sets, and the more classes we have, the smaller number of observations per class. The use of superclasses can help to overcome this issue; although the classification inside a superclass may suffer from overfitting it does not affect other superclasses, and the methods in a superclass are similar so mistakes within this are less dangerous. The effect of hierarchical classification in recommendations is also important, and will be discussed in Section 7.4.7.

7.4.5 Classification of Preconditioners

Similarly, Table 7.7 shows the results for accuracy using the hierarchical approach on preconditioners. Note that in this table there are some subclasses for which the result is marked as '$-$'; this is a problem that occurs in some cases with kernel density estimators. The reason is that this particular technique makes use of a covariance matrix for each class which becomes indefinite in cases where one or more features are constant within the class. A good way to overcome this problem is either filtering these features previously or to use a different classifier (like decision trees) if such a problem is encountered.

7.4.6 Classification of Combinations of Preconditioners and Iterative Methods

Classifying combinations of preconditioners with iterative methods in the Performance problem is more difficult than in the Reliability problem because we have to distinguish among several methods at a time. If we consider each composite method as a class we would have $|\mathcal{D}| \times |\mathcal{K}|$ classes, in our case it would be around 56 classes. For this reason the conditional approach, that was so effective for the Reliability problem, is not convenient in the Performance problem. The sequential approach does not perform much better since it follows some of the principles of the conditional

approach. The orthogonal approach, which ignores the interaction between the preconditioners and the iterative methods, yields better results in this case.

To show the effectiveness of these strategies in the problem of classifying the best composite method $\langle pc, ksp \rangle$, we use the confusion matrix concept from Section 3.7.2. The more diagonal a confusion matrix is, the better the classification process is. We generate a confusion matrix for each approach, and use a color scale to represent the number of occurrences in each entry of the matrix. Higher values in entries of the confusion matrix correspond to darker colors, conversely smaller numbers are lighter, where white indicates an entry with value 0. For this example we use a reduced number of iterative methods (four) and preconditioners (three), those for which we have the most number of observations.

Figure 7.24 shows the confusion matrix for the conditional approach. In this figure we can appreciate the interaction between iterative methods and preconditioners and how it affects the classification process. This matrix has "blocks" along the diagonal, and asi turns out each of these blocks correspond to a preconditioner. Take for instance the first four rows and columns corresponding to various iterative methods with the preconditioner asm; there we can find a group of errors. This means that a classifier can accurately distinguish between preconditioners, but once the preconditioner is selected it is difficult to differentiate between the iterative methods. In other words, it is easy to predict an optimal preconditioner regardless of the iterative method, and in general a preconditioner will be the optimal choice for a problem no matter the method. In addition, this result is very important for Performance classification, because it makes it easier to exploit the hierarchical classification (described in Section 6.1.4) together with the orthogonal classification strategy(see Section 6.2.3), where we pick the iterative method and the preconditioner separately. Figure 7.25 shows the confusion matrix for the sequential approach, although there is a reduction of errors off the diagonal, its pattern is very similar to that of the conditional approach so there is not much gain in the accuracy of classification. Finally, Figure 7.26 shows the confusion matrix obtained using the orthogonal approach; this matrix has a nice diagonal pattern meaning that this is the best approach in the Performance problem.

7.4.7 Performance Recommendation

The recommendation process for the Performance problem involves more factors than the Reliability problem. The different classes are based on the performance of each method, so there is a performance loss and gain associated with making wrong and correct predictions. In this section we present and discuss these factors and how they are affected by other factors such as hierarchical classification, PCA preprocessing and using different sets of fetures. Remember that recommendation tests are made on "unseen" data, and they also make use of the accuracy α information resulting from the testing process.

First, let us see how the L and G (Definitions 43 and 44 respectively) are obtained. In the case of performance loss, a slowdown factor greater than 20 would include those cases where the recommended method did not converge; however, in the performance recommendation process the cases where a method diverged are not considered[§]. In the case of performance gain, the factor $20+$ includes those cases for which the default method did not converge so the recommended method is not only more efficient but also more reliable. The performance loss is, in general, lower than the ones for performance gain, and the particular advantage lies in the $20+$ factor, meaning that the

[§]In our experiments, 20 was the largest slowdown factor encountered between two methods.

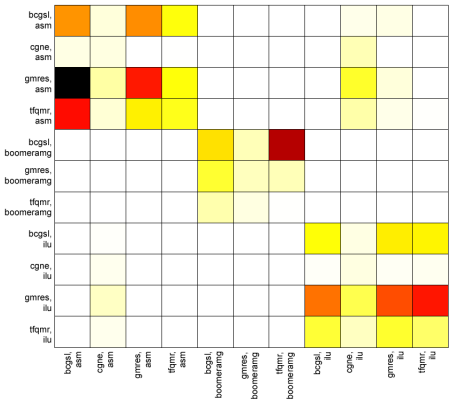


Figure 7.24: Confusion matrix for conditional approach for classifying (pc, ksp).

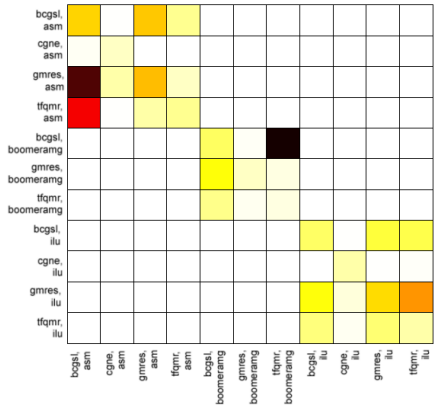


Figure 7.25: Confusion matrix for sequential approach for classifying (pc, ksp).

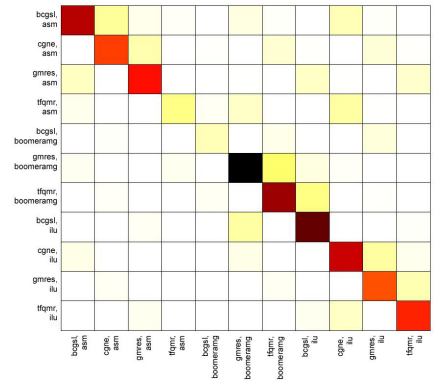


Figure 7.26: Confusion matrix for orthogonal approach for classifying (pc, ksp).

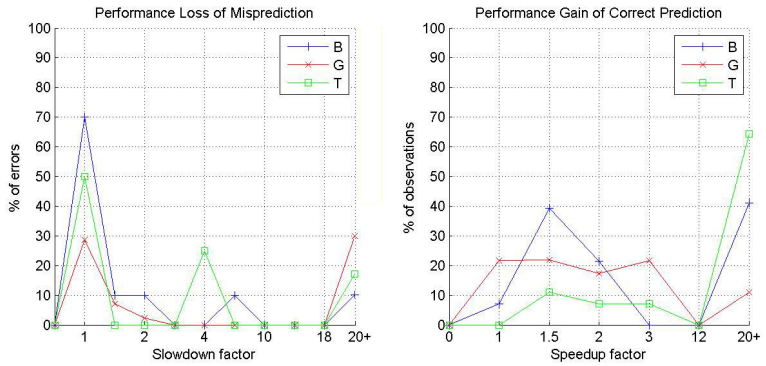


Figure 7.27: Performance Loss and Gain factors for unseen problems: this chart shows results for *superclasses* only. Along the x is the slowdown factor (left) and the speedup factor (right). The y is the proportion of misclassified cases that fall within each slowdown factor.

Table 7.8: Accuracy of classification α and expected performance loss and gain for the iterative methods in superclass **G** using decision trees.

	As Individual Classes			As part of Superclass **G**		
ksp	$\alpha_{ksp} \pm z$	L	G	$\alpha_{ksp} \pm z$	L	G
fgmres	0.80±0.02	8.178	7.73	0.94±0.02	6.62	8.32
gmres	0.59±0.04	13.167	3.40	0.88±0.03	10.61	6.01
lgmres	0.81±0.05	11.222	28.27	0.92±0.02	3.67	28.65

gain of using the recommendations goes beyond the possible loss and results also in more reliable choices even compared to a default choice (see the example in Figure 7.27). Even if the gain and loss factor values are similar, the probability of recommending a "bad" method is very low (0.01), which scales down the loss factors.

The recommendation process can also benefit from using hierarchical classification. For example, on the left columns Table 7.8 we can see the accuracies for *fgmres*, *gmres* and *lgmres* as individual classes (as taken from Table 7.5(a)) and their corresponding expected loss L and gain G values; on the right columns it shows the same measurements but for these three methods as part of the superclass G. As we can see, the expected loss inside the superclass is less, and the expected gain is greater.

Effect of the Feature Set and PCA Preprocessing

In Section 7.2.5 we discussed how the use of PCA can affect the classification process in the Reliability problem. The effect of choosing the correct set of features for classification has a bigger impact in the Performance problem because we have to differentiate between more classes. In here we illustrate how the selection of features can affect the performance of a classifier. We use superclasses since these provide the larger number of observations per class so we can have a better general view of the effect. We also compare the effect of using PCA with all the transformed features and using a reduced set (using the most important principal components).

Table 7.9: Accuracy, Expected Performance Gain and Loss for classification of iterative methods using Kernel Density Estimation.

	Using *all* features			Using *good* features			Using $(all - good)$ features		
ksp	$\alpha_{ksp} \pm z$	L	G	$\alpha_{ksp} \pm z$	L	G	$\alpha_{ksp} \pm z$	L	G
B	0.80±0.03	4.77	11.73	0.79±0.03	3.91	11.70	0.57±0.07	3.99	11.22
G	0.67±0.02	3.74	9.02	0.53±0.03	3.85	8.97	0.51±0.06	3.30	8.17
T	0.71±0.05	3.76	19.50	0.89±0.03	3.68	19.80	0.84±0.02	3.85	18.30

Table 7.10: Accuracy, Expected Performance Loss (L) and Gain (G) for classification of iterative methods using Kernel Density Estimation and PCA-transformed feature space. The first 25 principal components account for 90% of the information.

	Using *all* PCs			Using *first* PCs		
ksp	$\alpha_{ksp} \pm z$	L	G	$\alpha_{ksp} \pm z$	L	G
B	0.93±0.01	2.83	11.121	0.85±0.03	4.487	11.125
G	0.94±0.01	2.07	8.258	0.87±0.03	3.385	9.018
T	0.90±0.01	2.01	21.750	0.88±0.04	3.792	20.367

Table 7.9 shows the results for kernel density classifier without using PCA, and Table 7.10 using PCA. And Table 7.9 and 7.10 show the corresponding results for decision trees. Using these tables we can evaluate how these two classifiers are affected by the feature selection as well as by the use of PCA. The accuracy of classification is the measurement that is the most affected and in less degree the expected gain and loss for predictions.

In these tables, the accuracy for each superclass corresponds to the average of the accuracies of its subclasses, similarly the expected loss L for a superclass is as well the average of the expected losses of the individual classes when measured as part of a superclass. For example the accuracy α for class G is the average of the values in the fourth column of Table 7.8, and the L is the average of the values in the second to last column in that same table.

First we look at the kernel density estimator classifiers, in Table 7.9 the *good* features correspond to those we characterized in Section 7.2.3 as relevant; observe how the use of this set do not affect much the overall performance of the classifier for each class, and it shows particular benefit for class T, where it seems that the use of all features introduces some noise that affects the ability of the classifier to distinguish this class. The use of the remaining features (denoted as $(all - good)$), greatly affects the accuracy α_{ksp}.

Table 7.10 shows what happens when we use a transformed set of features using PCA, first using all the principal components, and second using only the most relevant. The use of transformed features is, in general, better for this type of classifier: the accuracy α_{ksp} increases and the expected loss decreases. The overall performance of the accuracy is improved even with dimensionality reduction using only the principal components that account for a 90% of the information.

Now we take a look to these results but using decision trees classifier instead. Table 7.11 shows how the use of *all* features is very convenient in decision trees. Remember that this type of classifier can take advantage of the information originating from all the features. The use of the reduced set of *good* features slightly affects the L and the G, but also slightly decreases the accuracy, and not

Table 7.11: Accuracy, Expected Performance Gain and Loss for classification of iterative methods using Decision Trees.

	Using *all* features			Using *good* features			Using $(all - good)$ features		
ksp	$\alpha_{ksp} \pm z$	L	G	α_{ksp}	L	G	$\alpha_{ksp} \pm z$	L	G
B	0.86±0.03	10.69	16.07	0.83±0.02	9.77	10.58	0.66±0.04	6.28	10.87
G	0.91±0.02	7.00	14.32	0.89±0.03	6.27	13.36	0.79±0.03	8.73	10.42
T	0.89±0.04	4.75	20.88	0.743±0.04	15.11	22.22	0.257±0.05	3.11	24.22

using these features dramatically affects it. Table 7.12 shows the results using PCA, as opposed to the kernel density estimator classifier, there is some decrease of the overall effectiveness of the decision tree classifier. Several reasons, discussed in Section 7.2.5 could account to this effect.

Table 7.13 shows the effect of using different sets of features on the classification results. Using only one category (except the case of Spectrum), leads into less accurate classifications, in particular for the superclasses B and T. Observe that the classification excluding the Normal category is good, which may indicate that the features in this category introduce noise into the classification process. The Spectrum category seems to be important in this process, not only it yields good accuracy when used alone but also, the classification is not as successful when this category is excluded.

7.5 Case Study: "Blind" Test

In this section we discuss an example of how the classification and recommendation are carried out on eight completely new matrices, we will call this small set $\mathcal{G}$ (the description of this set can be found in Appendix A.2.4). Four of these matrices are symmetric, and the other four are not. The user that provided these matrices has specified the methods and preconditioners available for this particular application in a single-processor machine. We first obtain a recommendation for each problem, then we run every available method and preconditioner on these matrices and we compare the recommendations' results with the results from the actual runs.

The user also defines which is the set of available solvers:

$$\mathcal{M} = \{bcgs, bicg, cgne, gmres, tfqmr\}$$

The first stage is the learning process 4.1, covers the training of the classifiers:

1. Train a Reliability classifier. This classifier will determine whether a method converges for the input matrix. It is trained using all available methods on the data in the database. The resulting set of decision rules for each method is a discriminant function $\Upsilon_M^{reliability}$ (Definition 5.2), where $M \in \mathcal{M}$.

2. Train a Performance classifier. The performance classifier predits which of the available methods will achieve the best performance on the particular input. It is trained by using all available methods on the data in the database. The resulting set of decision rules is the discriminant function $\Upsilon^{performance}$ (Definition 28), which compares between the methods in $\mathcal{M}$.

Table 7.12: Accuracy, Expected Performance Loss (L) and Gain (G) for classification of iterative methods using Decision Trees and a PCA-transformed feature space. The first 25 principal components account for 90% of the information.

	Using all PCs			Using $first$ PCs		
ksp	$\alpha_{ksp} \pm z$	L	G	$\alpha_{ksp} \pm z$	L	G
B	0.84±0.04	14.58	11.67	0.83±0.03	10.98	12.02
G	0.93±0.03	6.93	8.34	0.92±0.03	5.70	8.32
T	0.66±0.03	3.66	20.87	0.74±0.03	4.00	21.92

Table 7.13: The first column indicates which set of features was used (based on the categories in Appendix A.1). These results were obtained using decision trees.

Feature Set	Method Superclass	α	L	G
Simple	B	0.67	7.40	10.64
	G	0.80	10.59	8.37
	T	0.23	2.91	26.75
Variance	B	0.56	6.72	10.67
	G	0.82	10.20	8.50
	T	0.23	3.11	26.75
Structure	B	0.71	5.40	10.36
	G	0.80	12.06	8.83
	T	0.40	3.67	22.57
Spectrum	B	0.79	13.61	12.34
	G	0.91	4.60	8.26
	T	0.77	2.00	21.22
Normal	B	0.78	10.02	10.82
	G	0.83	8.77	8.64
	T	0.62	8.86	20.45
All except Spectrum	B	0.76	7.26	10.42
	G	0.88	10.33	8.63
	T	0.54	3.33	23.05
All except Normal	B	0.93	15.61	11.07
	G	0.92	7.62	8.41
	T	0.69	4.00	20.12
All except Spectrum and Normal	B	0.66	6.98	10.87
	G	0.79	9.73	8.42
	T	0.26	3.11	24.22

Table 7.14: Output from reliability classifiers.

matrix $G \in \mathcal{G}$	Recommended reliable	Actual reliable
sym_pdl	$bicg, gmres, tfqmr$	$bcgs, bicg, cgne, gmres, tfqmr$
sym_pdl-20	$bcgs, bicg, gmres, tfqmr$	$bcgs, bicg, cgne, gmres, tfqmr$
sym_pd2	$bicg, gmres, tfqmr$	$bcgs, bicg, cgne, gmres, tfqmr$
sym_pd2-20	$bcgs, bicg, gmres, tfqmr$	$bcgs, bicg, cgne, gmres, tfqmr$
nsym_pdl	$bcgs, gmres, tfqmr$	$bcgs, bicg, gmres, tfqmr$
nsym_pdl-1.6e4	$bcgs, bicg, gmres, tfqmr$	$bcgs, bicg, gmres, tfqmr$
nsym_pd2	$bicg, gmres, tfqmr$	$bcgs, bicg, gmres, tfqmr$
nsym_pd2-20	$bcgs, bicg, gmres, tfqmr$	$bcgs, bicg, gmres, tfqmr$

Training the classifiers is a process that usually takes minutes (perhaps even hours depending on the amount of data in the database), it can be done offline and the decision rules saved for later use (e.g., store the decision tree structure in a file).

Using the trained classifiers we can proceed to find the set of reliable methods:

for each matrix $G \in \mathcal{G}$

  for each $M \in \mathcal{M}$

    if $\Upsilon_M^{reliability} = converge_M$ *then*

      add M *to the set of reliable methods for solving* G

    end

  end

end

Table 7.14 shows the following is the output of this process, and the results from the actual tests. Observe that sometimes the classifiers do not identify as converging some methods that actually are, however, methods that diverge are not recommended – that would be a dangerous error.

We also have to find which preconditioners are also reliable with each of these "reliable" methods. The set of preconditioners marked available for this experiment is:

$$\mathcal{D} = \{asm, ilu, bjacobi, boomeramg, parasails, pilut\}.$$

Using the conditional approach (Section 5.3.3), we try to identify the set of reliable combinations. Take for instance the set of reliable methods $bcgs$, $bicg$, $gmres$, $tfqmr$ for the matrix nsym_pd2-20, the result for the recommendation of reliable combinations (pc, ksp) is in Table 7.15.

The best recommendations are those that have a minimal false positive ratio and higher accuracy, which results also in a minimal $\mathcal{Q}^{reliability}$ (see Equation 5.13). Except for $(bcgs, asm)$ and $(gmres, jacobi)$, all the other recommendations are good, specially in terms of the penalty they incur in, these bad penalties happen to coincide with the worse accuracies too.

Now we try to find what is the optimal method to solve each of these matrices. For each matrix, two choices of optimal method are recommended. The first is the one that according to the classifier is the optimal among $\mathcal{M}$, this method is then removed from $\mathcal{M}$ and we let the classifier choose

Table 7.15: Recommended combinations (pc, ksp) for the matrix nsym_pd2-20.

(pc, ksp)	$\alpha_{(pc,ksp)}$	FP_{ratio}	$\mathcal{Q}^{reliability}$
$(bcgs, asm)$	0.83	0.270	0.087
$(bcgs, boomeramg)$	0.96	0.011	2.976
$(bcgs, ilu)$	0.84	0.023	1.191
$(bcgs, parasails)$	0.99	0.012	2.724
$(bcgs, pilut)$	0.98	0.012	2.695
$(bicg, asm)$	0.99	0.009	3.676
$(bicg, ilu)$	0.99	0.006	5.524
$(gmres, bjacobi)$	0.91	0.244	0.109
$(gmres, boomeramg)$	0.98	0.001	33.33
$(gmres, ilu)$	0.90	0.073	0.420
$(gmres, parasails)$	0.97	0.001	32.25
$(gmres, pilut)$	0.99	0.001	33.33
$(tfqmr, asm)$	0.90	0.047	0.666
$(tfqmr, boomeramg)$	0.95	0.001	33.33
$(tfqmr, ilu)$	0.94	0.012	2.695

another method, which is the second choice. On Table 7.16 we present the output of the recommendation process together with the real optimal method (obtained later by doing exhaustive tests on each matrix). We also present the expected gain and loss values for each recommendation as well as the $\mathcal{Q}^{performance}$, which the smaller it gets the better a recommendation is (see Equation 6.7).

These measurements can help us to decide if the optimal method recommended is indeed a good choice, or if taking the second option may be a better decision. Observe first the recommendations for the symmetric problems (first four). The classifier made a mistake predicting the optimal methods for these and in two of these cases the real optimal was $cgne$[¶]; observe from Figure 7.1(a) that there are almost no examples of cases using $cgne$ in our experience database, so even if that is indeed the optimal method for these particular matrices, it is very difficult to generalize from a few examples, so the classifier recommends instead something else.

On the other hand, the last three matrices from the blind test (non-symmetric) had better recommendations. The classifier was able to predict which was the actual optimal method, even when it was a second choice. The difference with these matrices is that we have recorded plenty of experience and examples in the database where these methods were successful for matrices sharing similar numerical properties, which results in a better generalization for new problems.

Let us analyze individually a couple of these examples:

- sym_pd1: the first recommendation is $gmres$ with $\mathcal{Q} = 1.76$ and an accuracy (α) of 0.89, while the second recommendation is $bcgs$ with $\mathcal{Q} = 0.38$ and accuracy of 0.87. In this case, selecting the first recommendation is a better choice. After checking the result from the real test, we find that this recommendation is slower than the real optimal by a factor of 1.25.

[¶]A common rule of thumb for solving positive-definite symmetric matrices is to use the conjugate gradient method cg, however, at the time we trained the classifiers we did not have this method available so there are is no experience recorded in the database about its behavior.

Table 7.16: Output from performance classifiers.

matrix $G \in \mathcal{G}$	Recommended optimal (1st and 2nd)	Actual optimal	$\alpha_{\mathcal{M}}$	L	G	$\mathcal{Q}^{performance}$
sym_pdl	*gmres*	*cgne*	0.89	10.61	6.01	0.568
	bcgs		0.87	4.16	10.95	2.632
sym_pdl-20	*gmres*	*tfqmr*	0.89	10.61	6.01	0.568
	tfqmr		0.89	4.75	20.88	4.405
sym_pd2	*tfqmr*	*cgne*	0.89	4.75	20.88	4.405
	gmres		0.89	10.61	6.01	0.568
sym_pd2-20	*tfqmr*	*bicg*	0.89	4.75	20.88	4.405
	bcgs		0.87	4.16	10.95	2.632
nsym_pdl	*tfqmr*	*bicg*	0.89	4.75	20.88	4.405
	bicg		0.84	3.333	30.00	9.09
nsym_pdl-1.6e4	*tfqmr*	*tfqmr*	0.89	4.75	20.88	4.405
	gmres		0.89	10.61	6.01	0.568
nsym_pd2	*tfqmr*	*tfqmr*	0.89	4.75	20.88	4.405
	bcgs		0.87	4.16	10.95	2.632
nsym_pd2-20	*tfqmr*	*tfqmr*	0.89	4.75	20.88	4.405
	gmres		0.89	10.61	6.01	0.568

- sym_pd1-20: the second recommendation with $\mathcal{Q} = 0.227$ and same accuracy as the first one is a better choice, and after checking the real result from tests we find that $tfqmr$ is indeed the optimal method.

- nsym_pd1: the first choice of recommendation is not the actual optimal method, however the second recommendation is. The accuracy of the recommendation for both cases is very close, but the $\mathcal{Q}$ for $bicg$ is better. Based on the order of the recommendation and on the accuracy of the prediction (α), choosing the first method is a good option (although it would not be the optimal in reality); however, the $\mathcal{Q}$, L and G tell us that with $bicg$ there is not much loss expected compared to the expected gain so $bicg$ becomes a better option. When we check the actual results from tests, we find that for this problem $tfqmr$ was slower than $bicg$ by a factor of only 1.02. So picking either of the methods recommended is a good choice.

- nsym_pd2: the first recommended method happens to be the optimal method too. This is an example of a perfect case, where the accuracy of the first recommendation is higher and its $\mathcal{Q}$ is smaller.

In general, we have found that for this test the methods recommended for the Performance problem using our strategies are good choices, even for the cases when these were mispredictions. To achieve better recommendations it is necessary to incorporate into our database more examples to homogenize the sample of data for different methods.

To exemplify how we can use the results from both the reliability and performance recommendations, lets use again the matrix nsym_pd2-20. From Table 7.14 we see that $gmres$ and $tfqmr$ are reliable, and in Table 7.15 we have a list of the recommended most reliable combinations of

preconditioner and iterative methods for this matrix. From Table 7.16 we have discussed that the best choice, in terms of performance, is $tfqmr$. However, from Table 7.15 we can see that $gmres$ converges for more choices of preconditioners than $tfqmr$.

The importance that each type of recommendation gets, reliability or performance, greatly depends on the application for which it is required and the requirements of the final user. In this particular example, although $tfqmr$ is a better choice for fastest method, $gmres$ is more promising in terms of reliability and more likely to work with more preconditioners.

Chapter 8

Conclusions and Future Work

In this section we describe the current achievements of this research as well as the possibilities for future work and research.

8.1 Conclusions

We have developed and tested various strategies that can be used for algorithm selection in numerical problems. These strategies have been based on the study of the numerical properties of sparse matrices paired with the behavior analysis of different solvers. A priori, the behavior of the classifiers is completely determined by features, and we have shown that sufficient features can be computed. The properties of these types of problems have a strong effect on the effectiveness of the algorithms, whether this is in terms of reliability or performance.

We have identified, via PCA, several correlations between features. We have also pointed out a group of relevant features for the Reliability problem and for the Performance problem. These features seem to influence the outcome of the method selection process, which implies that they affect as well the behavior of the algorithms. Central to the feature analysis problem was the determination of the normalizing factors for each feature; this was essential to make the problems scale-invariant. This makes our analysis independent (to some extent) from some architecture-related parameters and timing measurements that are dependent on the size of the problem.

Overall, PCA is a powerful tool for the study and comprehension of numerical problem features. It may also be used to improve the classification accuracy of some classifiers. It has been proved very useful in particular with reliability classification problems (both iterative methods and/or preconditioners). However, this approach should be used with caution since it could be useless (maybe even harmful) for other classifiers like decision trees.

Over the course of the research we carried out experiments with different datasets. Some of these guided us in the development and tuning of the different strategies presented here, by exposing characteristics of the features.

We have found that the interaction between preconditioners and iterative methods is very significant not only in the numerical sense. This reflects directly on the behavior and effectiveness of statistical learning techniques, particularly in the Reliability problem, where determining the convergence of a method is critical. Among the different strategies for recommendation we have tested, the conditional approach was the most effective and the one that best captured this interdependence. Experimentation with other approaches led us into the development of the conditional approach,

with which we were able to better represent the dependence of iterative methods on preconditioners, and use this to achieve better classification results and thus, more reliable recommendations.

The problem of finding an optimal method, i.e. the Performance problem, depends greatly on the way that methods behave compared to each other. We developed certain concepts, independence and covering ratio, to measure and determine when two methods are similar not only in terms of reliability but also in terms of performance. By determining which methods are *similar* we were able to group them in superclasses to improve the effectiveness of performance classifiers by using the Hierarchical classification strategy. We have found that it is easier to discriminate between superclasses, and then by restricting the classification to the classes within a superclass we can achieve better accuracy of classification, as well as better ratios of expected performance gain and loss for recommendations made using this approach. The determination of method similarity is a heuristic strategy, but it is a reasonable approach based on experimental results. Although the similarity of methods and preconditioners is consistent across different datasets, we have to keep in mind that for other datasets outside the current scope, the similarity measurements may vary; however, the superclasses may still be determined using the proposed methodology.

8.2 Future Work

A fact that we have not explicitly considered in our current experiments is that many iterative solvers, such as $gmres$ and $bcgs$, have parameters that can make a method perform better or make it more reliable. In order to incorporate these parameters into our analysis we could consider each parametrized solver as a different solver, e.g., $gmres5$, $gmres20$, etc. However, this creates many more classes which are more difficult to separate and could result in overfitting of some classifiers. A possible alternative could be to do Hierarchical classification (Section 6.1.4) in more steps, for instance: superclass G has classes $gmres$, $fgmres$ and $lgmres$, and in the class $gmres$ we could have subclasses $gmres5$, $gmres20$, $gmres50$. The main issue with either of these approaches is that many of these solver parameters are not discrete; they extend over a continuous range that is difficult to split in intervals. A different kind of modeling is required for this problem, such as regression or different machine learning techniques such as genetic algorithms or regression decision trees. Furthermore, parameters are not only particular of solvers, but they also appear in preconditioners and other preprocessing steps.

We have mentioned that there are other types of transformations that can be applied to a problem A prior to the application of a solver. Here we have focused on the use of preconditioners because of the important effect they have on iterative solvers. Other important transformations such as *scaling* also have parameters that must be taken into account. For this, it is very important to gather more data and examples from different datasets.

Another important issue for future investigation is the incorporation of parallel implementations of solvers and experimentation with the different preconditioners that are known to be affected by parallelism. We have characterized several features that are relevant for the decision making process in single processor experiments; some of these are intuitively important no matter the implementation. However, the amount of information that they may capture or the type of correlations they may exhibit could be very different in a parallel environment.

Further analysis of the adaptivity of the system over large periods of time is also important. This concerns mainly the constant evaluation of the recommendation system with the incorporation of

new “experience”, and how the particular strategies can be tuned improved when more data (and perhaps different) data is incorporated.

With the use of the techniques strategies here proposed, we have been able to address a certain spectrum of numerical problems. It is interesting to extend these techniques to other types of problems, not only those originating from the solution of linear systems and iterative solvers, but also those arising from different application areas such as the eigenvalue problem. A preliminary look at experiments with the “blind” test dataset described in Appendix A.2.4 (originally used for eigenvalue problem), has indicated a promising possibility for the use of our proposed methodology of feature characterization, given that many of the features we have analyzed are intuitively important in numerical research. We have stressed the importance of taking into consideration different problem features. Further analysis and experimentation with those features identified as relevant can allow numerical researchers to focus on particular areas of the problems affected by these properties.

Bibliography

Bibliography

[Aizerman and Braverman, 1964] Aizerman, M. and Braverman, E. (1964). and Rozonoer, L.(1964). Theoretical foundations of the potential function method in pattern recognition learning. *Automation and Remote Control*, 25:821–837.

[Alpaydin, 2004] Alpaydin, E. (2004). *Introduction to Machine Learning*. MIT Press.

[Angerson et al., 1990] Angerson, E., Bai, Z., Dongarra, J., Greenbaum, A., McKenney, A., Du Croz, J., Hammarling, S., Demmel, J., Bischof, C., and Sorensen, D. (1990). LAPACK: A portable linear algebra library for high-performance computers. *Supercomputing'90. Proceedings of*, pages 2–11.

[Apostol and Holbrow, 1963] Apostol, T. and Holbrow, C. (1963). *Calculus, Volume II*, volume 31. AAPT.

[Axelsson, 1987] Axelsson, O. (1987). A survey of preconditioned iterative methods for linear systems of equations. *BIT*.

[Balay et al., 2004] Balay, S., Buschelman, K., Gropp, W., Kaushik, D., Knepley, M., McInnes, L., Smith, B. F., and Zhang, H. (2004). PETSc users manual. Technical Report ANL-95/11 - Revision 2.2.1, Argonne National Laboratory. http://www.mcs.anl.gov/petsc.

[Balay et al., 2007] Balay, S., Buschelman, K., Gropp, W., Kaushik, D., McInnes, L., and Smith, B. (2007). PETSc home page. http://acts.nersc.gov/petsc/.

[Barrett et al., 1994] Barrett, R., Berry, M., Chan, T. F., Demmel, J., Donato, J., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C., and der Vorst, H. V. (1994). *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, Philadelphia, PA.

[Benzi, 2002] Benzi, M. (2002). Preconditioning Techniques for Large Linear Systems: A Survey. *Journal of Computational Physics*, 182(2):418–477.

[Bhowmick et al., 2007] Bhowmick, S., Eijkhout, V., Freund, Y., Fuentes, E., and Keyes, D. (2007). Application of Machine Learning in Selecting Sparse Linear Solvers. *Accepted for publication on the International Journal of High Performance Computing Applications*.

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

[Bhowmick et al., 2005] Bhowmick, S., Raghavan, P., and Teranishi, K. (2005). A Combinatorial Scheme for Developing Efficient Composite Solvers. *Lecture Notes in Computer Science, Eds. PMA Sloot, CJK Tan, JJ Dongarra, AG Hoekstra*, 2330:325–334.

[Bosq, 1998] Bosq, D. (1998). *Nonparametric Statistics for Stochastic Processes: Estimation and Prediction*. Springer.

[Breiman et al., 1983] Breiman, L., Friedman, J., Olshen, R., and Stone, C. (1983). CART: Classification and Regression Trees. *Wadsworth: Belmont, CA*.

[Brewer, 1995] Brewer, E. (1995). High-level optimization via automated statistical modeling. *Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 80–91.

[C. Carstensen and Tomov, 2005] C. Carstensen, R. L. and Tomov, S. (2005). Explicit and averaging a posteriori error estimates for adaptive finite volume methods. *SIAM J. Numer. Anal.*, vol. 42/6:2496–2521.

[Cai and Sarkis, 1999] Cai, X. and Sarkis, M. (1999). *Restricted additive Schwarz preconditioner for general sparse linear systems*, volume 21.

[Choi et al., 1992] Choi, J., Dongarra, J., Pozo, R., and Walker, D. (1992). ScaLAPACK: a scalable linear algebra library for distributed memoryconcurrent computers. *Frontiers of Massively Parallel Computation, 1992., Fourth Symposium on the*, pages 120–127.

[Demmel et al., 2005] Demmel, J., Dongarra, J., Eijkhout, V., Fuentes, E., Petitet, A., Vuduc, R., Whaley, R., and Yelick, K. (2005). Self-Adapting Linear Algebra Algorithms and Software. *IEEE Proceedings*, 93(2):293.

[Douglas and Montgomery, 1999] Douglas, C. and Montgomery, G. (1999). *Applied Statistics and Probability for Engineers*.

[Duda et al., 2000] Duda, R., Hart, P., and Stork, D. (2000). *Pattern Classification*. Wiley-Interscience.

[Duff et al., 1989] Duff, I., Erisman, A., and Reid, J. (1989). *Direct Methods for Sparse Matrices*. Oxford University Press.

[Dunham, 2002] Dunham, M. (2002). *Data Mining: Introductory and Advanced Topics*. Prentice Hall PTR Upper Saddle River, NJ, USA.

[Eijkhout, 2005] Eijkhout, V. (2005). Salsa test problems. About built-in test problems of the SALSA code.

[Eijkhout, 2007] Eijkhout, V. (2007). Anamod online documentation. http://www.tacc.utexas.edu/ eijkhout/doc/anamod/html/.

[Elomaa, 1999] Elomaa, T. (1999). The Biases of Decision Tree Pruning Strategies. *Advances in Intelligent Data Analysis: Proc. 3rd Intl. Symp*, pages 63–74.

[Falgout and Yang, 2002] Falgout, R. and Yang, U. (2002). hypre: a Library of High Performance Preconditioners. *Lecture Notes in Computer Science*, 2331:632–641.

[Faloutsos et al., 1997] Faloutsos, C., Korn, F., Labrinidis, A., Kotidis, Y., Kaplunovich, A., and Perkovic, D. (1997). Quantifiable Data Mining Using Principal Component Analysis.

[Fletcher, 1975] Fletcher, R. (1975). Conjugate gradient methods for indefinite systems. *Numer. Anal., Proc. Dundee Conf.*

[Freund, 1993] Freund, R. (1993). A transpose-free quasi-minimal residual algorithm for non-Hermitian linear systems. *SIAM J. Sci. Comput*, 14(2):470–482.

[Freund and Nachtigal, 1991] Freund, R. and Nachtigal, N. (1991). QMR: a quasi-minimal residual method for non-Hermitian linear systems. *Numerische Mathematik*, 60(1):315–339.

[Freund and Schapire, 1999] Freund, Y. and Schapire, R. (1999). A short introduction to boosting. *Journal of Japanese Society for Artificial Intelligence*, 14(5):771–780.

[Fukui and Hasegawa, 2005] Fukui, Y. and Hasegawa, H. (2005). Test of Iterative Solvers on ITBL. *High-Performance Computing in Asia-Pacific Region, 2005. Proceedings. Eighth International Conference on*, pages 422–425.

[George and Sarin, 2007] George, T. and Sarin, V. (2007). An approach recommender for preconditioned iterative solvers.

[Häfner et al., 1998] Häfner, H., Schönauer, W., and Weiss, R. (1998). The parallel and portable linear solver package LINSOL. *Proceedings of the Fourth European SGI/Cray MPP Workshop, IPP R/46, Max–Planck–Institut für Plasmaphysik, Garching bei München*.

[Hastie et al., 2001] Hastie, T., Tibshirani, R., Friedman, J., et al. (2001). *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer.

[Henson and Yang, 2002] Henson, V. and Yang, U. (2002). BoomerAMG: A parallel algebraic multigrid solver and preconditioner. *Applied Numerical Mathematics*, 41(1):155–177.

[Houstis et al., 1998] Houstis, E., Rice, J., Weerawarana, S., Catlin, A., Papachiou, P., Wang, K., and Gaitatzes, M. (1998). PELLPACK: a problem-solving environment for PDE-based applications on multicomputer platforms. *ACM Transactions on Mathematical Software*, 24(1):30–73.

[Houstis et al., 1995] Houstis, E., Weerawarana, S., Joshi, A., and Rice, J. (1995). The PYTHIA project. *Neural, Parallel, and Scientific Computations*, pages 215–218.

[Houstis et al., 2000] Houstis, E. N., Catlin, A. C., Rice, J. R., Verykios, V. S., Ramakrishnan, N., and Houstis, C. E. (2000). PYTHIA-II: a knowledge/database system for managing performance data and recommending scientific software. *TOMS*, 26(2):227–253.

[Jackson, 2003] Jackson, J. (2003). *A User's Guide to Principal Components*. Wiley-IEEE.

[Jones and Plassmann, 1993a] Jones, M. and Plassmann, P. (1993a). A parallel graph coloring heuristic. 14.

[Jones and Plassmann, 1993b] Jones, M. and Plassmann, P. (1993b). Parallel solution of unstructed, sparse systems of linear equations. pages 471–475.

[Katagiri et al., 2004] Katagiri, T., Kise, K., Honda, H., and Yuba, T. (2004). Effect of auto-tuning with user's knowledge for numerical software. *Proceedings of the first conference on computing frontiers on Computing frontiers*, pages 12–25.

[Klösgen and Zytkow, 2002] Klösgen, W. and Zytkow, J. (2002). *Handbook of data mining and knowledge discovery*. Oxford University Press, Inc. New York, NY, USA.

[Kunkle, 2005] Kunkle, D. (2005). Automatic algorithm selection applied to high performance solving of linear systems of equations.

[Kuroda et al., 1999] Kuroda, H., Katagiri, T., and Kanada, Y. (1999). Performance of Automatically Tuned Parallel GMRES (m) Method on Distributed Memory Machines. *proceedings of Hakken Kagaku Team, University of Tokyo, Japan*, pages 11–19.

[Kuroda et al., 2002] Kuroda, H., Katagiri, T., and Kanada, Y. (2002). Knowledge Discovery in Auto-tuning Parallel Numerical Library. *Progress in Discovery Science, Final Report of the Japanese Discovery Science Project, Lecture Notes in Computer Science*, 2281:628–639.

[Lagoudakis and Littman, 2000] Lagoudakis, M. and Littman, M. (2000). Algorithm selection using reinforcement learning. *Proceedings of the Seventeenth International Conference on Machine Learning*, pages 511–518.

[Lanczos, 1952] Lanczos, C. (1952). Solution of systems of linear equations by minimized iterations. *J. Res. Nat. Bur. Standards*, 49(1):33–53.

[Langou, 2007] Langou, J. (2007). Personal communication and consultation. On Iterative Methods and Sparse Linear Systems.

[Lee, 1995] Lee, S. L. (1995). A practical upper bound for departure from normality. 16:462–468.

[Lee, 1996] Lee, S. L. (1996). Best available bounds for departure from normality. 17:984–991.

[M3D-Home, 2007] M3D-Home (2007). http://w3.pppl.gov/∼jchen/index.html.

[Market, 2007] Market, M. (2007). Available on WWW at URL http://math. nist. gov. *MatrixMarket*.

[Martínez and Martinez, 2002] Martínez, A. and Martinez, W. (2002). *Computational Statistics Handbook with MATLAB*. Chapman & Hall/CRC.

[McInnes et al., 2003] McInnes, L., Norris, B., Bhowmick, S., and Raghavan, P. (2003). Adaptive sparse linear solvers for implicit CFD using Newton-Krylov algorithms. *Proceedings of the Second MIT Conference on Computational Fluid and Solid Mechanics*, 2:1024–1028.

[Oreizy et al., 1999] Oreizy, P., Gorlick, M., Taylor, R., Heimbigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D., and Wolf, A. (1999). An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3):54–62.

[Parsons, 2007] Parsons, M. J. (2007). Active vibration control using optimized piezoelectric topologies. Master's thesis, University of Tennessee.

[Pješivac-Grbović et al., 2007a] Pješivac-Grbović, J., Angskun, T., Bosilca, G., Fagg, G. E., Gabriel, E., and Dongarra, J. J. (2007a). Performance analysis of MPI collective operations. *Cluster Computing*, 10(2):127–143.

[Pješivac-Grbović et al., 2007b] Pješivac-Grbović, J., Bosilca, G., Fagg, G. E., Angskun, T., and Dongarra, J. J. (2007b). Decision trees and MPI collective algorithm selection problem. (4641):105–115.

[Provost et al., 1998] Provost, F., Fawcett, T., and Kohavi, R. (1998). The case against accuracy estimation for comparing induction algorithms. *Proceedings of the Fifteenth International Conference on Machine Learning*, pages 445–453.

[Quinlan, 1986] Quinlan, J. (1986). Induction of decision trees. *Machine Learning*, 1(1):81–106.

[Quinlan, 1990] Quinlan, J. (1990). Decision trees and decision-making. *Systems, Man and Cybernetics, IEEE Transactions on*, 20(2):339–346.

[Quinlan, 1993] Quinlan, J. (1993). *C4. 5: programs for machine learning*. Morgan Kaufmann Publishers Inc. San Francisco, CA, USA.

[Ramakrishnan and Ribbens, 2000] Ramakrishnan, N. and Ribbens, C. (2000). Mining and visualizing recommendation spaces for elliptic PDEs with continuous attributes. *ACM Transactions on Mathematical Software (TOMS)*, 26(2):254–273.

[Rice and Rosen, 1966] Rice, J. and Rosen, S. (1966). NAPSSa numerical analysis problem solving system. *Proceedings of the 1966 21st national conference*, pages 51–56.

[Russell and Norvig, 1995] Russell, S. and Norvig, P. (1995). *Artificial intelligence: a modern approach*. Prentice-Hall, Inc. Upper Saddle River, NJ, USA.

[Saad, 1993] Saad, Y. (1993). A flexible inner-outer preconditioned GMRES algorithm. *SIAM J. Sci. Comput*, 14(2):461–469.

[Saad, 1995] Saad, Y. (1995). *Iterative Methods for Sparse Linear Systems*. PWS Publishing Company.

[Saad and Schultz, 1986] Saad, Y. and Schultz, M. (1986). *GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems.*, volume 7.

[SALSA, 2007] SALSA (2007). Self-adapting large-scale solver architecture website. http://icl.cs.utk.edu/salsa/.

[Scott, 1992] Scott, D. (1992). *Multivariate Density Estimation: Theory, Practice, and Visualization*. New York: John Wiley & Sons.

[Smith et al., 2004] Smith, B., Bjorstad, P., and Gropp, W. (2004). *Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations*. Cambridge University Press.

[Sonneveld, 1989] Sonneveld, P. (1989). CGS, a fast Lanczos-type solver for nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 10(1):36–52.

[Sussman, 1992] Sussman, A. (1992). Model-driven mapping onto distributed memory parallel computers. *Proceedings of the 1992 ACM/IEEE conference on Supercomputing*, pages 818–829.

[Titterington et al., 1985] Titterington, D., Smith, A., Makov, U., et al. (1985). *Statistical analysis of finite mixture distributions*. Wiley New York.

[Trefethen and Bau, 1997] Trefethen, L. and Bau, D. (1997). *Numerical linear algebra*. SIAM.

[Troyanskaya et al., 2003] Troyanskaya, O., Dolinski, K., Owen, A., Altman, R., and Botstein, D. (2003). A Bayesian framework for combining heterogeneous data sources for gene function prediction (in Saccharomyces cerevisiae). *Proceedings of the National Academy of Sciences*, 100(14):8348–8353.

[van der Vorst, 1992] van der Vorst, H. (1992). BI-CGSTAB: a fast and smoothly converging variant of BI-CG for the solution of nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 13(2):631–644.

[Vapnik, 1998] Vapnik, V. N. (1998). *Statistical Learning Theory*. Wiley.

[Wall et al., 2001] Wall, M., Dyck, P., and Brettin, T. (2001). SVDMANsingular value decomposition analysis of microarray data. *Bioinformatics*, 17(6):566–568.

[Winston, 1992] Winston, P. (1992). *Artificial intelligence*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA.

[Wolfram, 2007] Wolfram, M. (2007). Conjugate gradient method on the normal equations. http://mathworld.wolfram.com/ConjugateGradientMethodontheNormalEquations.html.

[Xu and Zhang, 2004] Xu, S. and Zhang, J. (2004). A data mining approach to matrix preconditioning problem. *Proceedings of the Eighth Workshop on Mining Scientific and Engineering Datasets (MSD05)*.

[Ye, 2003] Ye, N. (2003). *The Handbook of Data Mining*. Lawrence Erlbaum Associates.

Appendix

Appendix A

Experimental Feature Set, Datasets and Implementations

A.1 The Feature Set

This section contains definitions and descriptions of the features considered in our research and experiments. These features are available as extracted using the AnaMod [Eijkhout, 2007] software library.

Tables A.1, A.2, A.3, A.4, A.5 and A.6 contain, organized by category, the specific features that are considered in our experiments. Each row of these tables includes the feature name, a brief description, and the feature used to normalize it (if necessary). The first column of these tables contains an index that indicates the order the Loadings Vectors used in Section 7.2.2. If the index is − it means the feature was not used in the analysis.

A.2 Experimental Datasets

In this part of the Appendix we describe in detail the datasets we have used as a source for our experiments.

A.2.1 The Matrix Market Dataset

The matrix collections [Market, 2007] in this dataset are both "wide" and "deep". The matrices from this dataset come from a wide range of applications, originating from PDE problems. From each application there is a group of similar matrices, and matrices from different datasets are dissimilar from the ones in other groups. Some of examples of the applications from which these matrices are collected from are: structural engineering, finite element structures problems in aircraft design, fluid dynamics, plasma physics. This set provides a good and variated testbed for training and testing heuristics.

A.2.2 The Finite Element Modeling Dataset FEMH

This is a set of matrices resulting from a finite element modeling for a homogenization of materials [Parsons, 2007]. The homogenized material introduces a material density dependancy. This density

Table A.1: Simple Category

Index	Feature Name	Description	Normalize With
1	$nrows$	number of rows	–
2	$trace$	Trace, sum of diagonal elements	$\infty norm$
3	$trace_{abs}$	Trace Absolute, sum of absolute values of diagonal elements	$\infty norm$
4	$1norm$	1-Norm, maximum column sum of absolute element sizes	$\infty norm$
5	$Fnorm$	Frobenius-norm, square root of the sum of elements squared	$\infty norm$
6	$diag\_dominance$	least positive or most negative value of diagonal element minus sum of absolute off-diagonal elements	$\infty norm$
7	$symm_{anorm}$	infinity norm of anti-symmetric part	$symm_{snorm}$
8	$symm_{fanorm}$	Frobenius norm of anti-symmetric part	$symm_{fsnorm}$
9	n_struct_unsymm	number of structurally unsymmetric elements	$\infty norm$

Table A.2: Variance Category

Index	Feature Name	Description	Normalize With
10	$row\_variability$	$max_i log_{10} \frac{max_j \lvert a_{ij} \rvert}{min_j \lvert a_{ij} \rvert}$	–
11	$col\_variability$	$max_j log_{10} \frac{max_j \lvert a_{ij} \rvert}{min_j \lvert a_{ij} \rvert}$	–
12	$diagonal_{avg}$	average value of absolute diagonal elements	$\infty norm$
13	$diagonal_{var}$	standard deviation of diagonal average	$\infty norm$
14	$diagonal_{sign}$	indicator of diagonal sign pattern	–

Table A.3: Structure Category

Index	Feature Name	Description	Normalize With
15	$symmetry$	1 for symmetric matrix, 0 otherwise	
16	nnz	number of nonzero elements in the matrix	$nrows^2$
17	max_nnz_{row}	maximum number of nonzeros per row	$\infty norm$
18	min_nnz_{row}	minimum number of nonzeros per row	$\infty norm$
19	$left\_bw$	$max_i\{i - j : a_{ij} \neq 0\}$	nnz^2
20	$right\_bw$	$max_i\{j - i : a_{ij} \neq 0\}$	nnz^2
21	$diag\_zerostart$	$min\{i : \forall_{j>i} a_{jj} = 0\}$	$nrows^2$
22	$diag\_definite$	1 if diagonal positive, 0 otherwise	–
23	$blocksize$	integer size of blocks that comprise matrix block structure, 1 in the general case	–

Table A.4: Spectrum Category

Index	Feature Name	Description	Normalize With
24	n_ritz_vals	number of stored Ritz values	–
25	$ellipse\_ax$	size of x-axis of the enclosing ellipse	$ellipse\_cx$
26	$ellipse\_ay$	size of y-axis of the enclosing ellipse	$ellipse\_ax$ and $ellipse\_cx$
27	$ellipse\_cx$	x-coordinate of the center of the enclosing ellipse	–
–	$ellipse\_cy$	y-coordinate of the center of the enclosing ellipse	–
28	κ	estimated condition number	–
29	$positive\_fraction$	fraction of computed eigenvalues that has positive real part	–
30	σ_{max}	maximum singular value	$1norm$
31	σ_{min}	minimum singular value	$1norm$
33	$\lambda_{max,magnitude,\Re}$	real part of maximum eigenvalue by magnitude	$1norm$
32,34	$\lambda_{max,magnitude,\Im}$	imaginary part of maximum eigenvalue by magnitude	$\lambda_{max,magnitude,\Re}$ (index 32) and $diag\_avg$ (index 34)
35	$\lambda_{min,magnitude,\Re}$	real part of minimum eigenvalue by magnitude	$\lambda_{max,magnitude,\Re}$
–	$\lambda_{min,magnitude,\Im}$	imaginary part of minimum eigenvalue by magnitude	$\lambda_{max,magnitude,\Im}$
36	$\lambda_{max,rp,\Re}$	real part of minimum lambda by eigenvalue	$1norm$
37	$\lambda_{max,rp,\Im}$	imaginary part of minimum lambda by eigenvalue	dia_avg

Table A.5: Normal Category

Index	Feature Name	Description	Normalize With
38	$trace^2$	an auxiliary quantity	–
39	$commutator\_normF$	the Frobenius norm of the commutator $AA^t - A^tA$	$Fnorm^2$
40	$ruhe75\_lbound$	the bound from [**?**]	–
41	$lee95\_bound$	the bound from [Lee, 1995]	–
42	$lee96\_lbound$	the lower bound from [Lee, 1996]	–

Table A.6: : Functions for computing a multicolour structure of a matrix [Jones and Plassmann, 1993a, Jones and Plassmann, 1993b].

Index	Feature Name	Description	Normalize With
43	n_colors	the number of colours computed	–
–	$color\_set\_sizes$	an array containing in location i the number of points of color i	–
–	$colors$	points sorted first by color, then increasing index	–

dependency is composed of an infinite number of small holes which are periodically distributed in the material structure. This can be thought of as regularly distributed porus material. The unit cell, defined as a single cell with a void, is the microscopic hole that is modelled using the finite element technique. Using the finite element discretization on a 2D domain, the unit cell model can be written as:

$$[K] \cdot [\hat{\chi}] = [f] \tag{A.1}$$

where $[K]$ is classically refered to as the stiffness matrix. This is the term of interest in this research. The stiffness matrix is varied by changing the number and location of the descretization points and material properties.

A.2.3 The "Artificial" Dataset JLAP40

These matrices can be generated from the SALSA [SALSA, 2007] software implementation; they constitute a built-in test set for this code. They are based on a general form of a Laplace problem. The following description is from [Eijkhout, 2005].

The general form of a Laplace problem is

$$-\nabla \cdot (a(\bar{x})\nabla u(\bar{x})) + \beta v(\bar{x}) \cdot \nabla u(\bar{x}) + \gamma u(\bar{x}) = f$$

In our test code, $\bar{x}$ is a two-dimensional vector, and

- the diffusion coefficient $a(\bar{x})$ is piecewise constant, with a step at $x = 0.5$ and $y = 0.5$;
- the convection vector $v(\bar{x}) = (sin(4p(x + y)), cos(4p(x + y)))^t$, and
- γ is negative.

The matrix is formed by discretizing this with a five-point stencil (which has 2nd order accuracy) on a square 2D domain.

The relation between differential equation coefficients and program parameters is as follows:

- `domain_size` determines the mesh parameters: $1/h = domainsize + 1$.
- the diffusion function is scaled by h^2:

- the vector $v(\bar{x})$ has unit length, so the amount of diffusion is determined by the parameter β:

$$\beta = \texttt{skew}h$$

- finally, $\gamma = -\texttt{shift}$.

The scalings by powers of h imply that even modest numbers for the program parameters may imply rather badly conditioned differential equations, if h is small enough.

In this dataset, the features in the categories $structure$ and $simple$ do not change in any of these samples. The two parameters shift and skew affect the spectral (and other) properties of the matrices when a transformation is applied, so the analysis will concentrate in the behavior and study of these properties.

A.2.4 Other Datasets

The M3D Dataset

This dataset includes problems from plasma physics projects (from Princeton, University of Wisconsin, and University of New Hampshire) and one geophysics project (at Columbia). This dataset contains very few matrices, a full description of the application that originates these matrices can be found in [M3D-Home, 2007]. The matrices are of three general types based on how diagonally dominant they are: weak, medium and strong.

The "Blind" Test Dataset

The sparse matrices for this case study are obtained using finite element method (FEM) discretization of convection-diffusion-reaction equations. In particular, the problem is in 3D, the FEM uses piecewise linear continuous finite elements over tetrahedral mesh. The PDEs describe a real 3-D application in fluid flow and transport in porous media. The symmetric matrices used are for problems without convection (e.g., solving for the pressure), and the non-symmetric are for problems with convection, discretized using up-winding. In both cases the problems have singularities due to boundary data, inhomogeneity, and localized source terms. The computational mesh is obtained with automatic mesh refinement around the singularities (just mentioned) using a posteriori error analysis. For more information see [C. Carstensen and Tomov, 2005].

There are several matrices based on the discretization just mentioned. These matrices are also used to obtain a few others by subtracting from them αI where α is constant chosen in the following way. The original matrices are positive definite and estimates about their smallest eigenvalues λ_{min} are obtained using SALSA. Several choices for $alpha$ are considered:

1. $alpha$ close to λ_{min} and $alpha < \lambda_{min}$,
2. $alpha$ close to λ_{min} and $alpha > \lambda_{min}$, and finally
3. $alpha$ further away from λ_{min} towards the middle of the spectrum.

Note that these choices correspondingly give us matrices with much larger condition number (than the original) but still positive definite, matrices with much larger condition number and indefinite (with possibly just a few eigenstates with negative eigenvalues), and highly indefinite matrices (many eigenstates with negative eigenvalues).

A.3 Code Implementations

Here we briefly describe the implementation of various methodologies used in this research and their corresponding software modules.

One of the main goals of developing a smart agent component for the *SALSA* project is to implement an automated decision making process that will eventually help to find the most "suitable" method. As mentioned in previous sections we study the performance using statistical machine learning methods to develop and test various types of classification approaches.

For this study, we need to consider two important aspects: first, we need to research the behavior of different classification techniques that can help the system to understand and eventually recommend a "good" method for solving a specific problem, and second we need a way to store the information that is inherently required (and generated) by these techniques. Since most of these techniques are statistical methods-based, we need to collect as much data as possible, and with a wide range of problem variety.

To address the first point we have investigated the behavior of different types of classification methods; the various implementations of these methods we will refer to as classifiers. These classifiers are implemented using Matlab scripts and toolboxes. To address the second point, we have created a database using MySQL, which stores both properties of the matrices (and several transformations) available so far, and the performance of different iterative methods applied to solve these matrices.

Another important aspect mentioned before is to identify and characterize the features or properties of the numerical problems that make them be solvable or more efficiently solved by particular methods.

A.3.1 Classification Modules Using Matlab

All the classifiers that have been studied so far have been implemented using Matlab and its Statistical Toolbox. Matlab provides a convenient `C` API, so it will be possible to integrate and run the classifier modules with the rest of the *SALSA* environment. We concentrate on statistical methods, and so far, the following types of classifiers have been implemented and tested with the data currently available:

- Bayesian with Gaussian assumption (parametric approach)
- Bayesian using Kernel Mixture (non-parametric approach)
- K nearest-neighbors (KNN) which is a clustering technique
- Principal Component Analysis (PCA), applied to both parametric and non-parametric Bayesian approaches
- Decision Trees
- Support Vector Machines (SVM): currently the implementation only supports 2-dimensional data, but will be extended for $multi$-dimensional analysis

Each classifier is implemented as a separate module in Matlab script with a standard API.

A.3.2 Accuracy Evaluation Modules using Matlab

Here we define some classification terms we will use. The following definitions help with the generalization and design of the classifier interface.

Let $\mathbb{X}$ be the set of numerical problem (in this particular case, the set of linear systems with sparse coefficient). $\mathbb{X}$ is composed of a finite number of observations, we define an observation $x_i \in \mathbb{X}$ as a vector of numerical properties (features) that describe a single numerical problem, where $i = 1, 2, ..., n$ and n is the number of elements in $\mathbb{X}$.

Let $\mathbb{M}$ be the set of numerical methods that can potentially solve a problem, or that can solve it in the most efficient way according to a designed performance measure (e.g. smallest time to convergence, least number of iterations, best accuracy).

Let $\mathbb{C}$ be a finite set of classes or groups defined within the context of a particular classification problem. In our case, a class $c_i \in \mathbb{C}$ can be defined in two different ways depending on the problem we want to address in terms of recommendation system (as described in Section 4.4):

- *Best performance recommendation*: a class is defined as a set problems subset of $\mathbb{X}$ with certain features such that all the members of the class are best solved by a particular method $m \in \mathbf{M}$ with respect to a particular performance measurement.
- *Reliability recommendation*: a class is defined as a set of problems that have certain features, such that a particular method $m \in \mathbb{M}$ converges (or not) regardless of its speed.

In general, each class in $\mathbb{C}$ is a subset of $\mathbb{X}$ $c_i \subseteq \mathbb{X}$, and the union of the elements of these classes compose the totality of $\mathbb{X}$: $c_1 \cup c_2 \cup, ..., \cup c_n = \mathbb{X}$.

It is important to note that the classes in in either case, are mutually exclusive, once a problem is assigned as a member of certain c_i it cannot belong to any other.

Let $\mathbb{P}$ be the set of input parameters that a statistical-based classifier can take. Depending on the type of classifier (parametric, non parametric, supervised, etc.) a subset p of $\mathbb{P}$ is a particular set that is adequate for a classifier. For example, consider a simple Gaussian based classifier. Here p would be composed of new observation, mean, standard deviation, and accuracy threshold.

A.3.3 Database Interface

The second part of our problem involves the manipulation of the database based in MySQL. MySQL also provides a C API which allows the integration of SQL queries embedded in C programs. This will allow the main C program to work as a link between the database information and how it can be processed using the Matlab scripts for the classifiers. Presently, we have several MySQL scripts that extract information from the database in different ways depending on the format and input required by the classifiers, and on the different types of statistics that we are interested in exploring. In general, we can group the types of scripts in the following categories:

- Counting and comparing statistics: with this module it is possible to count and compare the behaviour of different methods or transformations, such as convergence and speed rates of one method compared to other(s).
- Data grouping by best performance-measure: these scripts are used for dumping data grouped by method; that is, all the data dumped into one file corresponds to those matrices that are

best solved by a specific method. The data can be grouped in different ways depending on the way the data is to be analyzed; e.g., data groups could correspond to four different types of iterative methods.

- Data grouping by transformations applied to a matrix: these scripts are similar to the previous category, but in these cases we might be interested in the performance of different preprocessing steps applied to a matrix and the ultimate impact on the performance of the iterative method.

- Data display for comparing performance: these scripts are mainly used for PCA. The data is not grouped in any way, but it rather consists of a block of features followed by a block of performance measures (of the same type) corresponding to the different methods or groups of interest, to compare side by side the performance of the methods when applied to the same matrix but with different transformations, as well as investigating how the performance is affected by the difference in the properties or features of each specific matrix.

Vita

Erika Fuentes obtained her PhD degree at the University of Tennessee, Knoxville in 2007. 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 with specialization in Networking. During her PhD studies she worked as a graduate research assistant for the Innovative Computing Laboratory in the Self-Adaptive Large-scale Software Architecture project, with main research area in Self-Adaptive Numerical Software and applied statistical methods and Machine Learning in numerical problems.