Machine Learning Techniques to Improve Software Quality

by

Jaspar Cahill

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy (Information Technology)

Faculty of Science and Information Technology

The Queensland University of Technology

2010

Abstract

A significant proportion of the cost of software development is due to software testing and maintenance. This is in part the result of the inevitable imperfections due to human error, lack of quality during the design and coding of software, and the increasing need to reduce faults to improve customer satisfaction in a competitive marketplace. Given the cost and importance of removing errors improvements in fault detection and removal can be of significant benefit. The earlier in the development process faults can be found, the less it costs to correct them and the less likely other faults are to develop.

This research aims to make the testing process more efficient and effective by identifying those software modules most likely to contain faults, allowing testing efforts to be carefully targeted. This is done with the use of machine learning algorithms which use examples of *fault prone* and *not fault prone* modules to develop predictive models of quality. In order to learn the numerical mapping between module and classification, a module is represented in terms of software metrics. A difficulty in this sort of problem is sourcing software engineering data of adequate quality. In this work, data is obtained from two sources, the NASA Metrics Data Program, and the open source Eclipse project. Feature selection before learning is applied, and in this area a number of different feature selection methods are applied to find which work best. Two machine learning algorithms are applied to the data - Naive Bayes and the Support Vector Machine - and predictive results are compared to those of previous efforts and found to be superior on selected data sets and comparable on others.

In addition, a new classification method is proposed, Rank Sum, in which a ranking abstraction is laid over bin densities for each class, and a classification is determined based on the sum of ranks over features. A novel extension of this method is also described based on an observed polarising of points by class when rank sum is applied to training data to convert it into 2D rank sum space. SVM is applied to this transformed data to produce models the parameters of which can be set according to trade-off curves to obtain a particular performance trade-off.

Table of Contents

Statement of Original Authorship

The work contained in this thesis has not been previously submitted to meet requirements for an award at this or any other higher education institution. To the best of my knowledge and belief, the thesis contains no material previously published or written by another person except where due reference is made.

Signature: _____

Date: _____

Acknowledgements

Thanks are due to my supervisors, Jim Hogan and Richard Thomas, for their invaluable advice and support. Without it, it would not have been possible to complete the work, which has provided a great opportunity to make a research contribution and in doing so to develop valuable faculties in research.

SECTION 1 - Background

Chapter 1 - Introduction

Since the early days of software development, software has become larger and more complex, as demand for functionality has increased along with computer power. Early software programs might have been no longer than some pages, whereas today they can be millions of lines in length. While this has allowed software to perform tasks that weren't previously possible, it has brought upon software developers and the industry some significant problems. A large proportion of software projects run over time and over budget, and in many cases they fail outright. There is also a proportionally higher number of faults in software, leading to large costs in software testing.

An explanation for these problems in software is lack of quality. If software quality is to be improved a means to evaluate quality is required. Unfortunately however there is no direct way to measure this, and although currently available software tools attempt to assist in this regard, the assistance is rather limited. This has motivated efforts to develop predictive models of software quality. Techniques from the field of Machine Learning are applied to examples of good and bad quality software described in numerical terms by software measurements, to develop models that can discriminate between these quality classes on other software modules. The aim of this research is to use this approach to develop better performing predictive models. If this can be done successfully and the developed quality prediction models are accurate enough to be used in practice and incorporated into development tools, they would serve as a useful means with which to evaluate software quality to help avoid the aforementioned problems of software.

The rest of the chapter is structured as follows. The problems of software are discussed in more detail in section 1.1, along with how quality models can help with them. Next in section 1.2, the basis for the software quality modelling approach, software measurement, is described, as a means of providing a numerical description of software that can be used by machine learning methods. The current offering by metrics tools to assist in identifying software of poor quality is then briefly covered in section 1.3 with attention drawn to their limitations. This motivates the modelling effort using machine learning techniques described in section 1.4. And finally, there is a brief discussion in 1.5, and overview and roadmap in the subsequent sections leading into the other chapters.

1.1 The Problem of Software

The dramatic increase in size and complexity of software in recent decades lead to an increase in the rate at which projects ran over time and over budget, or failed completely. This phenomenon, first recognized in the late 1960s, lead to the coining of the term the "software crisis". Since then technologies have improved and efforts have been made within the field of software engineering to address these problems, and it has subsided somewhat. Project outcomes however are still far from ideal. This can be seen in the published results of project outcomes by the Standish Group (Standish Group International). This group first published its so called "chaos report" on project outcomes in 1994, so named because of the state of chaos thought to exist the software industry at the time. Since then, other issues of this report have been released, and the group has come to be regarded as the leading source of information on the state of the software industry. A plot of project outcomes as reported by the Standish Group from 2004 to 2009 is shown on the left in Figure 1. Outcomes are grouped into three categories: success, challenged (over time and budget), and failed (not deployed or abandoned). It will be noted that in 2009 only about one third of projects were successful, and one quarter ended in failure. Details on challenged projects are shown on the right in Figure 1. In 2004, of the half of all projects that were found to be challenged (53%), these on average ran both over time and over budget by 50%, and despite this only delivered 50% of required features. Clearly these figures do not suggest that the problems of software have been adequately dealt with, and that further improvement should be sought.

It should be mentioned, in the interests of fair analysis, that the negative picture portrayed in

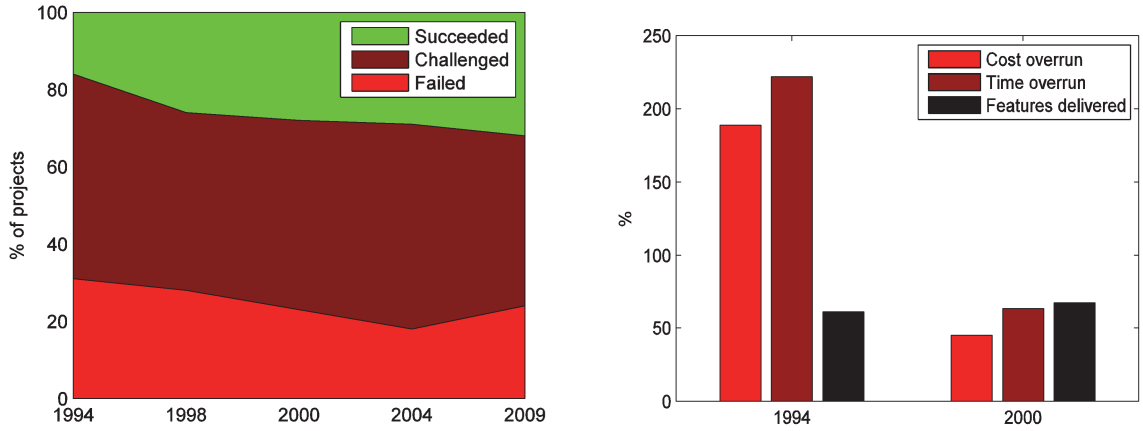


Figure 1 Standish Group report figures showing on the left the proportion of projects that failed, succeeded, or were challenged from 1994 to 2009, and on the right, time and budget overruns in the challenged projects.

the Chaos Reports has been questioned. A well respected author in the field of computing and software, Glass (Hartmann, 2009), points to the astoundingly successful software that exists today and contrasts this with the negative picture of a struggling software industry. More substantially however, he refers to three studies conducted by academic and industry researchers, which arrived at findings that were largely inconsistent with those of the Standish Group's. There is suggestion too that the latter's findings might be biased towards failure. Against such doubts, the Standish Group is unlikely to have remained as a market research leader, supported by paying industry customers, if its reports were in fact grossly inaccurate. The view one might adopt then, is that while there could be some negative bias in the Standish numbers, there is still a significant element of project failure and challenge which clearly indicates difficulty in achieving successful software project outcomes.

Achieving better project outcomes depends on achieving higher quality repeatably, both in terms of product and process, with the product, the software itself, the focus in this work. If the quality of software code can be maintained at a high level, then problems that might otherwise be experienced in software could be avoided, such as additional time spent on adding new features due to poorly written legacy software. The goal of this research, is to produce software quality models to identify modules lacking in quality, which could then be the focus of improvement, ideally earlier on in the development lifecycle. Models would also allow quality benchmarks to be produced against which later efforts could be compared, as part of an ongoing process of quality management.

The other problem of software, which relates to poor project outcomes, is the high rate at which faults occur in software, and the high cost involved in testing to identify and fix them. It has been estimated that software testing accounts for between 50% and 80% of software development costs (Lipton, 1991). In monetary terms the Standish Group has estimated the cost of faulty software to business to be 78 billion per year (Levinson, 2001). Any means to assist in testing efforts then, even marginally, can be of significant benefit.

Software quality models may assist by targeting software testing efforts, by identifying modules of lowest quality that are at highest risk of containing faults. Even if only the highest risk modules are identified, these typically contain most of the faults in the system, as faults tend to be distributed according to an approximate 80:20, in which 80% of the faults lie in 20% of software modules (Gondra, 2008). Using quality models to identify fault prone modules can lead to more effective testing, so that more faults can be found for a given testing budget. The effectiveness results from more efficient testing, in which less time is wasted on

inspecting modules that are unlikely to contain faults. The use of quality models may also make it more feasible to identify faulty components early in the lifecycle, and removal of faults then could avoid more costly removal later when the problem may be compounded and the source of the fault more difficult to find.

1.2 Software Measurement and Metrics

Measurement is an important part of the approach taken in developing quality models, in that it is through measurement that software modules are described. Measurement is the mapping of some relation in reality to a numerical measurement scale that maintains the relation e.g. the measurement of a person's height (Fenton & Pfleeger, 1997). A useful feature and requirement of measurements is that they are objective and repeatable, such that a measurement taken by another on the same object will produce exactly the same value. Also useful, particularly for modelling, is that measurements provide a mapping of an object into the mathematical domain, to which various mathematical devices and methods can be applied. In this work, the interest is in the measurement of software, which is the concern of the field of Software Metrics. This field continues to grow along with the efforts to further formalise and automate software development under the umbrella of software engineering. Software measurements are usually referred to as metrics.

There are various types of metrics, some distinctions between which are worth mentioning. Software metrics are often separated into two broad classes, process and product. The former are those concerned with the software development process, an example of which is development time. The latter are the type of metrics used in this research, which are measurements made of the software product. More precisely, these are measures of static source code, as distinct from dynamic measures made of aspects of software at runtime such as runtime couplings between components. These static measures are convenient as they can be easily obtained from the source code using a metrics collection tool. Perhaps one of the other worthwhile distinctions to be made is between that of internal and external metrics. As suggested by the names, internal metrics are those derived from inside the software system, and external from the outside of the system. Quality is an example of an external metric of software, as it often thought of in terms of the number of faults associated with it as encountered by end users. The task of modelling quality, is actually to estimate the external measure of quality from internal static code measures. A distinction is also often made between direct/primitive and indirect/derived/computed metrics. The former is a count

obtained directly from the source code, such as number of lines of code, while the latter is obtained by the combination of two or more direct metrics. The metrics in this research, and that are used typically, are of both types. It should be mentioned in relation to static and dynamic metrics that while dynamic metrics could potentially offer useful information for quality prediction, static metrics are believed to capture enough information that relates to quality to be useful in themselves for prediction. This will become evident in this and later sections of the chapter.

Measurement is an observation of some relation in reality. The relation sought is that of quality of a software module. Quality however is subjective – what is thought of the quality of an object depends on the observer's perspective. Hence models have been developed that reflect the various ways in which quality might be defined. One such model was proposed by McCall (McCall, Richards, & Walters, 1978), as shown in Figure 2. Quality is decomposed into factors such as reliability, then further still, until specific metrics are reached. The specific aspect of quality that is of interest in this research is fault proneness. That is, we seek metrics that are thought to have some association or correlation with fault proneness. An example of such a metrics is number of GOTO statements in a module. GOTO statements represent unstructured code, and it is thought that the more unstructured a module is the more likely it is to contain faults (Dijkstra, 1968). This kind of metric could be useful in developing predictive models of quality.

One of the earliest metrics is Lines of Code (LOC). Since then many more metrics have been defined (Sharma, 2004). In practice, typically only a small subset of these are used. Although only a smaller subset may be required, often metrics may not be used because there has been little association established between the metric and quality. This has lead more recently to efforts to find these associations in so called metric validation studies. These usually rely on empirical means rather than the theoretical, due to the domain of software quality being poorly understood.

In this research, the software metrics available for modelling are from two metrics suites, Halstead and McCabe. These suites are amongst the most commonly used static code metrics for quality modelling. The Halstead metrics are based on a theory of software called Halstead's Software Science developed in the 1970's. The metrics primarily measure program size and complexity, examples of which include Volume, Error Estimate and Program Level. The McCabe metrics measure structural complexity of a module from a representation of a module as a control flowgraph. The best known of this class of metrics is cyclomatic

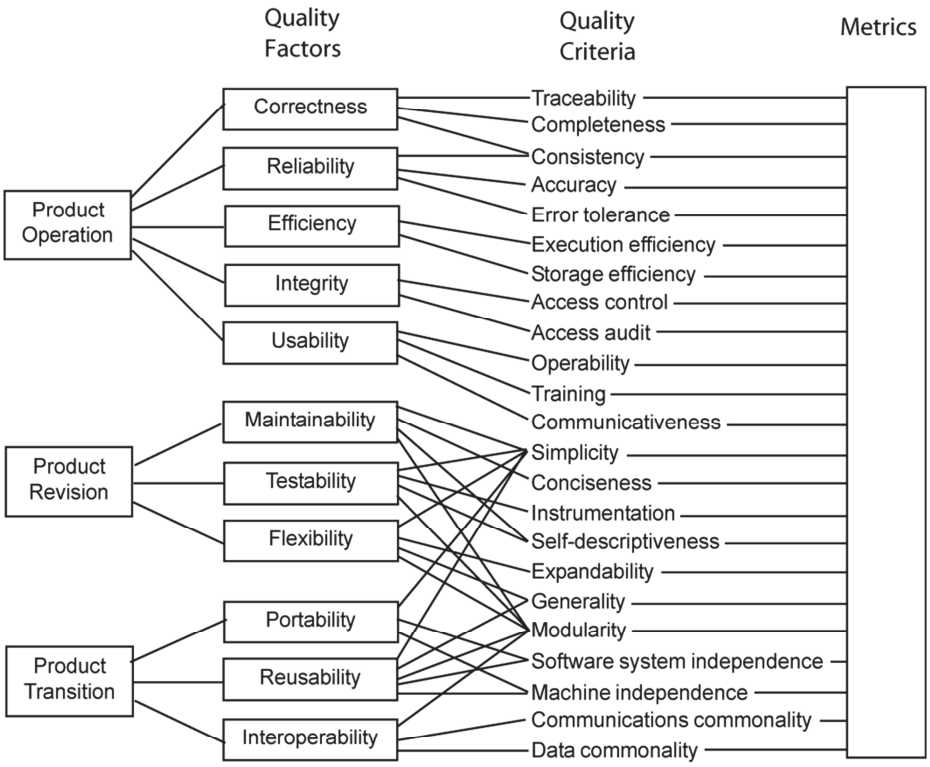


Figure 2 The McCall model of software quality illustrating various aspects by which the quality of software might be considered and evaluated.

complexity. A third metrics suite, that of Chidamber and Kemerer, provides class-level object-oriented metrics. Metrics in this suite are included in two of the available data sets, but were not used in the experiments in this research with the focus being on function-level procedural metrics, due to their broader applicability. Details on metrics in each of these suites are given in Appendix A for reference.

Given these metrics, it is of some interest whether any studies have empirically validated their relation to quality. It might be said at the outset that the use of thresholds on individual metrics as a basis for differentiating between high and low quality, is not soundly based, in that in the distribution of metrics values there is usually no threshold effect apparent that might be associated with quality. This approach is also rather simplistic. Thus finding any such thresholds in the literature, though some exist (e.g. McCabe proposed a threshold of 10 for cyclomatic complexity above which modules would be deemed fault prone), would likely be of little value in producing accurate quality models of broad usefulness.

Few studies appear to have been undertaken validating Halstead and McCabe metrics. The strongest report of association with fault count was found with McCabe's cyclomatic complexity (Shroeder, 1999). This is clearly shown in the graph from his study in Figure 3. The first two bars on the left of the graph indicate that models with low complexity made up a large percentage of code (red bar), and that these contained a small percentage of faults (yellow bar). As cyclomatic complexity increases, so does the percentage of faults found in these modules (despite comprising a decreasing percentage of the code). Less positive than this result was a review of other studies by Itzfeldt (1990) on both metrics suites, which, while finding an association with fault count, also found a strong relationship between these metrics and LOC. However it was suggested that there was still some residual association (beyond that with size), and the conclusion reached was that it was not clear which metrics were most useful, and that no single metric was sufficient to capture the broad quality aspect in this case of maintainability. A critique of these two metric suites that may be of interest is provided by Riguzzi (1996). A number of criticisms are made of both suites, but more so towards the Halstead metrics whose theoretical assumptions have been said to be flawed. The cyclomatic complexity metric is criticised in another study (Marco, 1997) for being too simple a measure of complexity, in that there can be differing levels of complexity in different parts of the control structure. In relation to the Chidamber and Kemerer metrics, more validation studies exist for these, e.g. (Basili, Briand, & Melo, 1996; Briand, Morasca, & Basili, 1999; Tang, Kao, & Chen, 1999; Cartwright & Shepperd, 2000; Emam, Benlarbi, Goel, & Rai, 2001; Emam, Melo, & Machado, 2001; Subramanyam & Krishnan, 2003) and in some, correlations

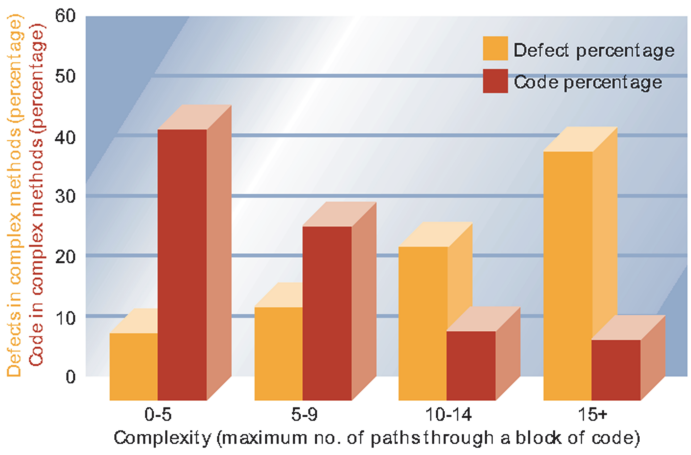


Figure 3 Plot of %code and %defects as cyclomatic complexity increases (Schroeder, 1999).

were found with fault count after size and other confounding variables were accounted for. However, due to some conflicting findings on validity and also the fact that these metrics have not been used in this work, they are not discussed any further.

The efforts in empirical validation as described above are evidently somewhat lacking. But there are criticisms levelled at static code metrics more generally in terms of their value in quality modelling. Two of these criticisms are described by Menzies (2007). It has been argued that metrics such as those of McCabe are nothing more than a proxy for LOC, due to high inter-correlation. Furthermore, it has been argued that it is possible to rewrite a function and obtain completely different metrics. Metrics thus may not be considered a complete characterisation of a module. Menzies responds to these criticisms, in the first case, in saying that predictors built from LOC alone perform sub-optimally compared to models built from multiple metrics, and in the second, that despite the incomplete characterisation and thus inability to provide certainty in quality prediction, they are useful in providing a probabilistic prediction. Perhaps the best argument in favour of the utility of static code metrics for quality prediction, is that quality models have been developed from them whose performance is good enough to be of practical use, providing an advantage used in conjunction with existing testing methods.

A potential problem with the use of metrics in quality modelling is that the same metric may be computed differently depending on the metrics collection tool used e.g. LOC. This may be due to the use of different definitions of the metric, or interpretations, especially as it may apply for different programming languages. Reported model accuracy may be overstated if metrics in the deployed environment are calculated using different methods from those used in the model development environment. There would seem to be little attention in the literature given to this issue. The main research issue is whether good or better model performance be obtained using some set of metric descriptions as used by NASA and Eclipse, rather than the implementation issue of how the models would actually be set up in practice. The assumption is that either tools could be found or modified to ensure consistency, or that any difference would be slight enough that its effect on performance will be limited.

1.3 Metrics Tools and their Limitations

Many tools have been developed to automate the collection of metrics from source code. The purpose of these tools is to gain insight into the software so that design improvements may be

made and to identify problematic modules. Excluded here are syntactic checkers that are mainly concerned with style conformance and identification of anomalies according to a defined rule base, and project management software that only makes use of a small subset of metrics such as those measuring size and change.

Metrics tools are typically used by specifying the location of source files which are then parsed by the tool to create some internal representation of the source code, such as an abstract syntax tree. This contains elements at different levels of scope, such as package and class. Metrics are computed over these elements and are then presented to the user in a table, with a row of metrics for each element. Sometimes this is accompanied by a graphical presentation of the data with the use of histograms and Kiviat diagrams (Kolence & Kiviat, 1973). For the purpose of identifying problem modules, some tools go further and flag values according to predefined thresholds for particular metrics. This is typically done by colour-coding cells in the metrics table. There may be multiple colours according to different estimated levels of risk. Some metrics collection tools are described in Appendix B.

The problem with these tools is that they are quite limited in terms of their usefulness for identifying poor quality or fault prone modules. The threshold based approach for indicating such modules, in which thresholds are determined arbitrarily rather than according to any solid theory, is simplistic and often inaccurate. The metrics collected are often not well understood by developers, either because the tools do not provide adequate documentation, or because there simply is not enough time to read through the documentation. Even if the metrics are understood, the developer still has the problem of how to interpret metric values as to what they might say about the quality of a software module. Clearly there is a need for a better solution than current software metrics tools provide.

1.4 The Quality Model Solution

The solution pursued in this research, is to development quality models, the input to which are the metrics collected for a given software module, and the output a quality rating, in this case fault prone, or not fault prone. It is assumed for this to be possible that the metrics have some relation to fault proneness. The model as such is basically a function, and the problem is how to obtain it, so that it can produce accurate predictions. One possibility is to explicitly define a formula that combines metrics in some way, but this is difficult to do as the nature of software and derived metrics as they relate to quality is not well understood. There is no theory that defines what quality is in terms of metrics.

So the method taken given the lack of domain information is to 'learn' the function from examples of good and bad quality software modules. Each example or instance is in the form $\mathbf{x} \rightarrow y$, where $\mathbf{x}$ is a description of the module in terms of a vector of metrics, and y is a quality label. The set of examples is referred to as the training set. The objective is to learn the function f from the training set, such that for any $\mathbf{x}$, $f(\mathbf{x}) \rightarrow y$ gives the correct quality label y. The field of Machine Learning provides many algorithms for learning functions in this way (Kotsiantis, Zaharakis, & Pintelas, 2006). This is the so called supervised learning problem, as each example has a class label/output value y associated with it. The learning problem can also be posed as an unsupervised one in which examples do not have labels, and the task is to simply to differentiate modules according to some emergent pattern. This approach however is less common than supervised learning for this problem, and is not used in this research. The supervised algorithms used are Naïve Bayes and the Support Vector Machine, the former being probabilistic and the latter based in statistical learning theory. These learning algorithms are described in detail in Chapter 3. The primary tool used in the research that contains implementations of these algorithms is Weka, the algorithms being accessible as a library or through a GUI environment (Witten & Frank, 2000). An important aspect of this machine learning based approach is determination of model performance, a common measure of which is accuracy. For this purpose, in order to obtain unbiased performance measures a separate set of data is usually held out in some way separate to the training set, called the test or validation set.

Two issues that arise with this approach are what metrics to use for $\mathbf{x}$, and what quality label to use for y. The former is usually selected by feature selection algorithms to avoid including redundant or irrelevant features that might interfere with the learning algorithm finding an optimally performing model. Feature selection is described in Chapter 4. For quality labels y, with quality viewed in terms of fault proneness, and modules labelled fault prone or not, labels are obtained based on fault count. It is assumed that fault count is available for each module, or can be obtained from fault logs. If fault count is low then the module is labelled *not fault prone*, otherwise *fault prone*.

Important in this approach is having sufficient data, and sufficient quality data. The training sample has to be large enough that it is sufficiently representative of the input space $\mathbf{X}$ to be able to predict accurately over this space. The number of metric dimensions in the data ideally also would also include a wide variety of metrics from which to choose. In terms of data quality, mistakes in $\mathbf{x}$ are unlikely because metrics are collected by tools, but mistakes in

fault count and the derived class label may not be uncommon. This has the potential to disrupt learning and to impact adversely on model performance. Large data sets help in this regard, as if there are many other instances in the vicinity of the **x** whose class label is in error, the learner will likely adopt the majority label, which is the correct one. Some learning algorithms are also more tolerant of noise than others.

These issues in relation to data are addressed more extensively in Chapter 5. But suffice to say, that while there is a lack of software engineering data generally, particularly with reliable fault records, adequate data sets have been sourced online from NASA's Metrics Data Program, and also from a repository for the Eclipse IDE. These provide a number of data sets, some of which are very large, and include numerous metrics. Fault counts are also provided, and for the NASA sets in particular these would have to be regarded as fairly reliable.

The developed models may be useful in two ways. The first is to use the models directly. Performance obtained in development should be reasonably well reproduced in a commercial setting given that the NASA software projects from which the data sets were derived were outsourced to commercial developers. The projects would be somewhat representative of commercially developed software. However, reliability of an individual model in predicting on software from other software systems could not be guaranteed. The literature would suggest that 'localising' application to similar types of software is likely to give better results than less specific global application. For a given commercial project, models might be selected according to matching characteristics such as language. This would improve reliability. The use of multiple models with agreement on fault proneness by a larger proportion of models, thereby increasing confidence in classification, might also improve reliability. It might be said too, that the Eclipse models may be suited to other open source projects.

The second way developed models may be useful, is to apply the methods used to develop them to in-house data if it is available, so that a company may create models tailored to their own software. This is likely to give better predictive performance than by using the models developed here directly. Performance reported in this work would be indicative of what performance could be achieved in-house. The drawback with this of course, is that in order for a company to create its own models it must have fault data, for which software faults would need to have been tracked over some period of time for at least one project. Model performance would to some extent depend on rigour of the fault tracking effort and the

corresponding quality of fault data. Assuming these fault data were available, creating in-house models would clearly be preferable in terms of more reliable performance. However in its absence, direct use of the models would be preferable to the use of none at all, and their performance is at least indicative as to what performance may be achieved in the former scenario.

1.5 Discussion

There are a couple of ways of looking at the motivation for this work. One is a quality view in which the aim is to improve the quality of software, and the other is a fault detection view, in which the aim is to identify software modules that contain faults. Both motivations are well justified, but it is the latter that has come to the fore as the more immediate and practical motivation for the work, especially as methods and technologies in software engineering have raised overall standards in software projects with a trend of improving project outcomes.

The task that the research aims to assist with lies in the identification of software modules that are likely to contain faults, so that testing efforts, in particular code reviews, can focus on those modules, rather than waste time inspecting modules that are less likely to contain faults. A lot of money is spent on testing, and if this assistance can be provided even with just marginal success or improvement over existing methods, there is potential for considerable saving. The approach may also allow removal of more bugs and provide for more reliable software.

The basis for any approach to identify fault prone modules is seen to lie in software measurement. Metrics tools however, are considerably lacking in this regard. So the problem lies open as to how to make use of the static code metrics they collect to assist the developer to identify faulty modules.

The answer that has been found, with recent advances in the field of machine learning, is to apply these algorithms to examples of fault prone and not fault prone software modules, to learn from them a mapping between metrics and fault proneness label. The task is fundamentally as simple as that.

For the approach to work there is an important assumption made, which is that static code metrics capture information about a software module that is associated with its quality. Supporting this assumption, numerous studies provide empirical evidence that correlation exists between some software metrics and fault proneness (Gondra, 2008). Furthermore,

models developed from static code metrics have proven to perform well in fault proneness prediction, as will be seen in the next chapter. This is reflected in the fact that many researchers have used it to develop quality models.

Some of the efforts to develop quality models, some specifically on the NASA and Eclipse data, are described in the next chapter. Given these, the task then in this research, other than validating the work in other studies, is to improve upon the performance obtained. This is not an easy task, given that many other learning algorithms have been applied without a statistically significant improvement in performance, and the recent view that has come about that the information content of the static code metrics as it relates to quality has been fully exploited. The results of this study will serve to support or disprove that view.

1.6 Overview of Chapters

This chapter has described the motivation for the research and the quality model solution. The focus in the next chapter, Chapter 2, is to review previous attempts in quality modelling, and other relevant studies that set the context for this research. Of these the studies of Menzies are the most relevant, and are described in some detail, and importantly his baseline results are given with which the results of this work are compared.

Chapter 3 turns to machine learning, and algorithms for constructing predictive models from data. In the present work, we develop models of software quality from experience in the form of training examples belonging to the different classes of quality to be predicted. This chapter focuses mainly on the two key learning algorithms used in this thesis, Naïve Bayes (NB) and Support Vector Machine (SVM) classifiers.

The closely related issue of feature selection is examined in Chapter 4. Features (or attributes) of a module are the measurements on the module that are included in its description. It is often the case that using all features as input to a learning algorithm gives suboptimal performance. The selection of an optimal feature set for modelling is therefore critical, and this chapter provides detailed background for the data preparation of Section 2.

In Chapter 5, we move from the background chapters and literature review, and into the research proper. In this chapter, the two sources of data are introduced, the NASA and Eclipse metrics data sets. Some of the basic information for each data set is covered first, such as the number of modules and features in each, and the proportion of faulty modules. Exploratory analysis follows, covering various aspects of the data, such as typical class

distributions and correlation between features. Preceding this work, at the outset of the chapter, are sections that extend the previous review of the literature on topics that relate more closely to the identified sources of data.

In Chapter 6, the focus turns to preparation of the data for modelling. Significant elements of this work include labelling of instances according to a selected threshold on module error density, and the creation of two strands of data, untransformed and log transformed, the latter of which has been reported by Menzies to enhance performance (Menzies, Greenwald, & Frank, 2007). Most of the chapter however is devoted to the application of various configurations of feature selection algorithms and their resulting feature sets, the configurations being tailored separately for each data set group due to the Eclipse sets having a much larger number of features from which to choose.

In Chapter 7, the crux of the research is reached, with the application of the two learning algorithms, Naïve Bayes and Support Vector Machines, to all of the prepared data sets to develop predictive models. Various feature selections obtained in the previous chapter are utilised, and models are created separately on the two strands of data, untransformed and transformed. The performance of these models is evaluated and reported, and compared to the baseline results of the previous work of Menzies.

Chapter 7 concludes the primary investigation using standard modelling techniques and feature selection methods. The next chapter, Chapter 8, begins a new line of research, toward a new classification method which we call Rank Sum. The method developed from the idea that a class is more likely to be the correct prediction if for component features, values are closer to the peak of the distribution of that class. Proximity to the peak is measured by rank and these are summed across features, hence the name of the method. Much of this chapter focuses on methods for improving classification performance.

In Chapter 9, the work of the previous chapter is exploited to provide a novel two-dimensional feature set for standard classification methods, yielding strong classification performance and allowing a ready trade-off between precision and recall. It was noticed that rank sum values computed for each instance on the training data when plotted exhibited clear separation between the classes.

Finally, conclusions are given in Chapter 10. The contributions of the work are summarised, and the main findings of the chapters are discussed with a view to informing future research in this area.

1.7 Roadmap

This section describes more schematically the structure of the thesis and how readers with particular interests might make their way through the material. The structure of the thesis and pathways through it are shown in Figure 4.

The next chapter, Chapter 2, summarises and discusses relevant literature. This is not essential to understanding the rest of the research. It evidences other similar work in which the quality modelling approach has proven successful in support of its validity. It also provides some context in which to appreciate the current work, and raises some relevant issues that other researchers have encountered, notably in section 2.8 Menzies. It would be helpful to have covered this prior to reading the modelling chapter, Chapter 7.

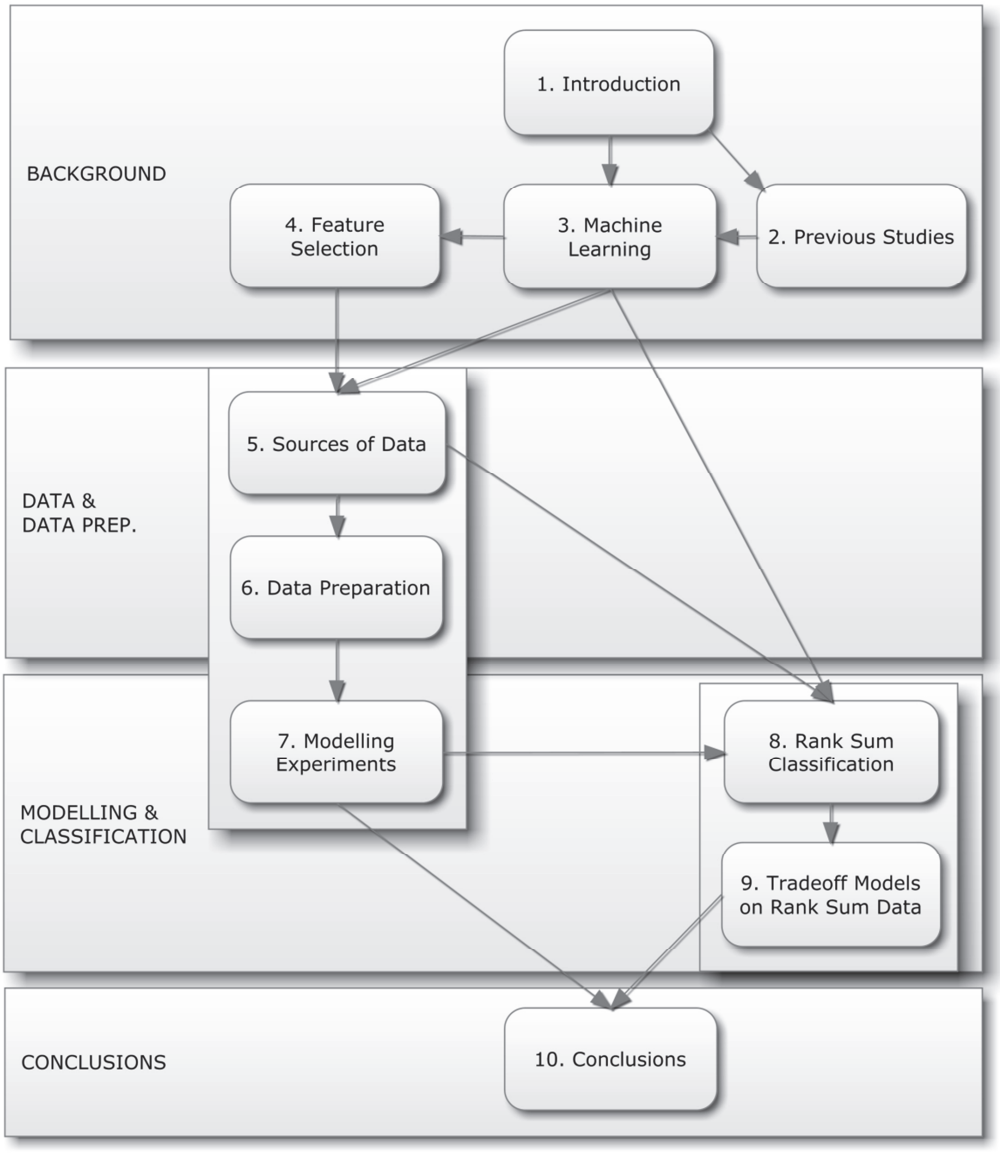


Figure 4 Thesis structure and pathways.

Chapter 3 introduces supervised learning which is the basis for the learning methods used in this research – both in standard modelling approaches in the earlier work, and the rank sum method in the latter. The two supervised methods described in detail in this chapter, Naïve Bayes and SVM, precede their application in Chapter 7.

Following Chapter 3, there are two lines of research. The main one is the use of standard modelling techniques to develop quality models from the sourced data sets. This is covered in chapters 5 to 7. The other line is in the development of the rank sum method and extension to trade-off models, covered in chapters 8 and 9. As written, these lines are covered in sequence, but they may be read independently and in any order. Optionally Chapter 5, Sources of Data, may be included prior to rank sum, in order for familiarity to be gained with the NASA data sets which are used in the rank sum chapters.

The final chapter to be addressed is Chapter 4, Feature Selection. There is a significant focus on feature selection and use of feature selection methods in the standard modelling techniques line of the research (Chapter 5 et seq.), and thus if covering this line of the work its inclusion is advised, particularly the opening sections, which present a more general overview of the subject.

Chapter 2 – Previous Studies

With the aim of the work having been set out in the previous chapter, we here consider some previous studies in software quality modelling. This provides some context for the work, informs it as to avenues that may be explored, and importantly provides performance benchmarks by which to compare results obtained. The application of machine learning algorithms to software engineering data, in particular to modelling fault proneness, is not an original idea. In developing quality models, the aspect of the effort that is usually of interest is the machine learning algorithm that was chosen. Given the data, this seems to be or is often thought to be the main factor that affects what sort of results are achievable. Accordingly, some of the various algorithms that have been used in studies are listed below. Those that used NASA data are marked with an asterisk. The purpose of this list is to provide an idea of the range of algorithms that have been used, and as a signpost to various directions in which the reader may wish to explore. Some of these studies are discussed later in this chapter. The list is as follows:

- Boolean Discriminant Functions (BDS) (Khoshgoftaar & Seliya, 2002),
- Random Forests ((Guo, Ma, & Harshinder, 2004)*, (Catal & Diri, 2008)),
- Binomial Regression (Ostrand, Weyuker, & Bell, 2004),
- SVM ((Xing, Guo, & Lyu, 2005), (Elish & Elish, 2008)*, (Gondra, 2008)*),
- Case Based Reasoning (Khoshgoftaar, Seliya, & Sundaresh, 2006),
- Spam Filtering (Mizuno, Ikami, Nakaichi, & Kikuno, 2007),
- Artificial Immune Recognition System ((Catal, Diri, & Ozumut, 2007)*, (Catal & Diri, 2008)),
- Fuzzy Neural Network (Yang, Yao, & Huang, 2007),
- Expected Maximisation (Seliya & Khoshgoftaar, 2007)*,
- Rough Hybrid approaches (Ramanna, Bhatt, & Biernot, 2007),
- Naive Bayes (Menzies, Greenwald, & Frank, 2007)*,
- Weighted Naive Bayes (Turhan & Bener, 2007),
- Radial Basis Function (Shin, Goel, Ratanothayanon, & Paul, 2007)*,
- Neural Network (Singh, Kaur, & Malhotra, 2008; Gupta, Brar, & Sandhu, 2008), and
- Fuzzy Integration (Pizzi, 2007).

References to older quality modelling studies prior to the year 2000 can be found in (Zhang & Tsai, 2003).

However, the long-held view that algorithm selection is the critical factor in performance has been subject to challenge. A telling paper published only recently, systematically applied an extensive 22 classifiers to 10 of the NASA data sets, and it was found that there was no significant difference in performance among the top 17 classifiers deployed (Lessmann, Baesens, Mues, & Pietsch, 2008). This paper further concludes that there is no convergence across studies about classifiers most suited to defect prediction. This would seem to lend weight to the view that choice of classifier may not be as important as dealing with issues in the data itself, those of noise, overlap and imbalance. Alternatively, this may support the view that focus should be to somehow improve the information content in the data (Menzies, Turhan, Bener, Gay, Cukic, & Jiang, 2008).

With this in mind, the chapter begins with an early exemplary study in software quality modelling in which remarkable results were obtained. Other studies are then summarised, with some brief discussion where appropriate, according to relevance to this research. We begin with those studies that focused on the NASA data, and then Eclipse. The issue of performance evaluation is then introduced, which raises a problem with fault data that makes evaluation less straightforward. Some studies on other data sets are mentioned thought still to be of some interest, one for example using the SVM algorithm with which to develop quality models. Further afield, studies that used different kinds of data to develop quality models with are described, to provide a broader perspective on data which is nevertheless a central element to the task of software quality modelling. Unsupervised modelling efforts are touched on. Finally, the work of one of the leading researchers in the field of software quality modelling, Menzies, is described in some detail. His work focuses on the NASA data, which makes it highly relevant to the current research.

2.1 Fault Proneness Exemplar Study

One of the most cited studies closely related to this research (Selby & Porter, 1988), and which appears to be among the earliest, identifies a problem with the use of machine learning methods as there being little systematic empirical evaluation, and corresponding empirical information concerning important system parameters with respect to an application area. It fills this void with respect to the decision tree algorithm as applied to software resource data for the purpose of prediction of fault proneness and effort. The data, sourced from NASA (not the MDP data sets), is notable for the fairly large number of attributes (74) available covering a multitude of aspects of the modules including design style, implementation style

and changes. It includes early availability metrics from the design phase, as well as code metrics. The top 25% of modules with faults were labelled positive (though class size is not mentioned). A number of factors affecting decision tree model performance are explored, but most are not of general interest. Standing out from this study is the mean predictive accuracy of 100%. This was significantly lower at 70% when only early availability metrics were used. This study is remarkable for the excellent result obtained. Though no comment was made about this result (only the average across predicted faults and effort was mentioned), it may be due to rich information in the data, exhaustive search of parameter combinations, and the small percentage of highly faulty modules targeted.

2.2 NASA Data Studies

Turning to studies on the NASA data sets, Elish & Elish (2008) investigate the capability of SVM for software quality prediction. SVM performance is compared with that of eight other classifiers. SVM is considered a good candidate for defect prediction because it is tolerant of high dimensional feature spaces, it finds a global optimum solution, it is robust to outliers by varying the cost parameter C, and it is able to model nonlinear functional relationships. A single automatic feature subset selection method is used, Hall's CFS (see section 3.7.2). Mentioned is the trade-off in performance between precision and recall, and suggested is the use of the F measure which combines them (the reader may refer to section 3.3 Performance Evaluation if unfamiliar with performance terms such as recall and false alarm rate). A limitation perhaps of this study, apart from the single feature subset selection, is that only the default parameters for each learning algorithm are used in Weka. Also, performance is only compared using an F value with an equal weighting on precision and recall, which when classes are imbalanced would favour classifiers that tend to produce very low FP rates. The overall finding was that SVM was at least as good as if not better than other classifiers. It was notable for its performance in particular in recall, which was higher than all other classifiers. However this came at the cost of precision in which it was often outperformed by other classifiers.

Vandecruys et al. (2008) applied a number of different machine learning algorithms to 3 of the NASA data sets, one of which, AntMiner+, was applied for the first-time to software mining and is a central element of the study. A benefit on this algorithm is that it produces comprehensible models as a set of rules. Rather than cross validation, each data set was simply split into a train and test set (using stratified sampling), in the ratio 70%/30%. The

justification for this presumably was that it was thought the data sets were large enough not to warrant a more rigorous validation procedure. Due to the well-recognised problem of class imbalance in the data which tends to adversely affect the performance of learning algorithms, positive instances in the training set were over-sampled by repeating each positive instance a number of times, which achieved a desired trade-off between sensitivity and specificity. One of the algorithms applied was the SVM, and its parameters were set using the grid search method. In the results, sensitivity and specificity varied between classification techniques, however the difference was not found to be statistically significant. Thus all were found to be evenly matched. The AntMiner+ algorithm was regarded as a good choice for modelling, as it performed as well as the other algorithms, with the advantage of offering comprehensible models. Its rules were also shorter than two other rule based classification techniques applied.

Investigating the merit of another learning algorithm, Guo et al. (2004) focused on the random forest, and compared its performance to that of 20 other classifiers (including SVM and Naïve Bayes) on 5 of the NASA data sets. In the first part of this study, results were obtained for the random forest algorithm alone. Accuracy and recall were chosen to report performance (rather than recall and false alarm rate as in Menzies' studies). The trade-off between these two measures was controlled by varying a "cut-off" parameter to the random forest algorithm, comprised of a value for each class. A range of cut-off values were tried. While no specific pair of best performance values was given, across the five data sets, accuracy ranged from 75% to 94%, and the best recall was at 87%. The random forests algorithm was also used to determine the 5 best features for each data set, however selecting these features rather than using all of them did not improve random forest performance. It was noted that the difference in the composition of the best set of features for each data set was a consequence of the different software environments and processes of each project, and suggests consequently that quality models should be tuned on the local project.

In the second part of the study, random forest performance was compared to that of the other classifiers. The main finding was that the random forest generally performed better in terms of accuracy and detection rate than all the other algorithms. It is more robust to noise and outliers than other methods, which gives it a greater advantage on larger data sets such as jm1, pc1, and kc1. One of the best performers on the NASA data sets was the Voting Feature Intervals algorithm (for all 5 of the data sets). Following this, was KStar and Naïve Bayes (for 3 of the data sets). SVM was not among the best for any of the data sets. It would not appear however that feature selection was applied in obtaining performance on the other

algorithms, which might unfairly bias the results against those more sensitive to feature selection. Also there was no mention of varying algorithm parameters, for SVM for example, and one assumes that only default ones were used which may for some algorithms undermine the validity of the results. Finally, it was mentioned that the occurrence of modules with faults in the NASA data sets agreed with the 80%/20% rule, except for two of the data sets, and in those cases the systems were mission-critical for which it would be expected there would be fewer faults.

Ensemble classification has also been applied to the NASA data sets although seemingly less commonly to the application of individual algorithms. Tosun et al. (2008) used a simple method in which three classifiers were combined, and a prediction made by majority vote. The three classifiers were Naïve Bayes (with log normalisation and information gain feature selection as used by Menzies described later in section 2.8 Menzies' Studies), Neural Network, and Voting Feature Intervals. The interest was to find out whether or not the ensemble improved over Naïve Bayes performance alone. On four of the NASA data sets there was no gain in performance with the ensemble method according to Menzies' 'balance' measure, however it increased the detection rate at the expense of a higher false alarm rate. Applying the ensemble models to 3 other non-NASA data sets, but also derived from embedded systems, the performance seemed quite good, with an average 76% detection rate and 22% false alarm rate. This compared to a slightly lesser NB performance of 70% and 24%. The cost benefit procedure proposed by Arisholm et al. (2007), described later in this section, was then applied. It stated that on average, the ensemble method detected 76% of defects and reduced verification effort by 56%, while Naïve Bayes detected 71% of defects and reduced verification effort by 55%. Thus, on the 3 non-NASA data sets, also in terms of cost benefit, the ensemble approached provided a small improvement over NB.

Data transformation as a pre-processing step before training has been suggested as a means to improve model performance on the NASA data (Menzies, Greenwald, & Frank, 2007). This avenue was explored in a study by Jiang et al. (2008) (co-authored by Menzies) in which four data transformations where applied: none, log normalisation, discretisation, and the latter two combined. Models were derived from the NASA MDP data sets using 10 different machine learning algorithms (with default parameters) in order to see what effect the transformations had on each learner. The learners included Naïve Bayes but not SVM. Model performance was evaluated using area under the ROC curve. For a given data transformation, performance for each learner was summarised across the data sets using a box plot, providing a highly

summarised view of the results. The most important finding of this study was that regardless of transformation, overall classification performance (as measured by AUC, across both data sets and learners) did not improve. In more detailed findings, the random forest learner was found reliably to be one of the best classification algorithms overall (except on discretised data, which removes noise that random forests handle well). Boosting (coupled with decision tree) was consistently one of the best performers (across transformations). The performance of Naïve Bayes was greatly improved by discretisation, and it was said to be desirable to do this prior to its application in developing software quality prediction models. And finally, log transformation, according to the first finding, rarely affected performance of predictive software quality models – if anything the increase in variance caused models to suffer. This last result is surprising given that log transformation provided a significant boost to performance with Naïve Bayes in earlier work of Menzies, discussed later in this chapter in section 2.8 Menzies' Studies. That this boost was not found in this study may be due to the different performance measure used, AUC. As well, no mention was made of feature selection in the experiments (although some relative improvement in performance with log normalisation might have been observed even with all features).

A study that should be mentioned focused on the size of modules in the NASA data sets, in terms of LOC (Koru & Liu, 2005). It was noted that there were two problems with module size. The first was that there are many modules in the data whose size was small, and consequently other metrics in these modules would have small values as well. This would provide difficulty for a learner in discriminating between modules of the two classes, of these modules of small size. The second was that large modules, which are associated with a higher probability of containing faults than smaller, are few in number. This would provide difficulty for a learner in that there may not be enough of them, in the presence of so many smaller modules, to properly recognize them. An experiment was conducted in which essentially sets of instances were created of different module size. A decision tree was developed from each set. It was found that the sets containing larger sized modules performed better. This is an interesting finding, but the intention is not to propose a better means of developing models on the NASA data. Performance appears to have been obtained over the training data, and thus the performance benefit in training on larger modules would likely be lost when the model was applied to smaller modules.

2.3 Eclipse Data Study

The other source of data used in this research is the publicly available Eclipse data sets, derived from the open source Eclipse project written in Java. A study by Zimmermann et al. (2007) introduces these data sets by essentially describing how they were obtained, their composition, and some efforts which demonstrate the use of the data sets in producing fault prediction models. The data sets were obtained from two sources, a bug database, Bugzilla, and a version data base, CVS. The former identifies failures in software, and the latter all changes made to the software source code. It is only in recent years that techniques have been developed to relate bug reports to fixes, and which were applied in this case to Eclipse. From these fixes, a defect count can be produced for each component. The released Eclipse data sets include defect counts both before and after release for each software module, as well as common complexity metrics and common syntactic elements obtained from abstract syntax trees. There is a data set for each granularity level, file and package, for three separate releases. It was observed that almost 50% of packages contained no defects and the remaining packages contained up to 103 defects. Based on this distribution two questions were raised that modelling may address: which modules contain defects, a classification task, and which contain the most defects, a ranking task. Both types of models were produced in this study.

In exploratory analysis of the data sets, a high correlation was found between pre-release and post-release defects, and correlations with metrics were positive, implying that more complex modules were likely to have more defects. In the classification experiments, logistic regression models were used with a threshold of 0.5 on the predicted likelihood to classify defective or not, with the use of post release defect counts for training. These were created for each release, both at the file level and package level. Testing was performed on each of the releases, to provide an indication of model robustness. Performance obtained would seem to be lacking, with recall ranging from only 19% to 38%. This was countered somewhat by a higher precision at around 60% although this still only means that of all predicted positives, just over half were actually positive. Recall improved somewhat at the package level. Performance held across versions. For the ranking experiments, linear regression models were used to predict the expected number of post release defects. Again, results were not impressive, with a Spearman correlation between predicted and actual post release faults of only 0.416. This does mean however that in the ranking, modules with a higher rank would be more likely to be faulty than those of a lower rank. It might be mentioned that in this case the

interest was in predicting post release faults, on the basis that it was more relevant to the successful release of a product, while in this research, the interest is in predicting faults prior to release. This is because pre-release fault counts affect more modules (relieving class imbalance), identifying faults early in the development process is desirable, and they may have a stronger relationship with static code metrics than post release defects.

2.4 Performance Evaluation Studies

An important issue in the development of predictive models is performance evaluation. Ma and Cukic (2007) provide an introduction to this issue in relation to software quality models. Due to imbalance between the classes, the unsuitability of accuracy as a performance measure is mentioned, due to its bias in reflecting accuracy more on the negative class than the positive, when the latter, in terms of detection rate, is of particular importance. A few other measures are suggested that can provide more insight on performance, namely, sensitivity, specificity and precision, although it is said these cannot be used to assess performance each alone. As this does not facilitate simple comparison or model selection, measures that provide a comprehensive summing up of performance are suggested. In this regard, the G-mean and F-measure are said to provide "more honest insights into model performance". The Receiver Operating Characteristic (ROC) curve is mentioned as providing a useful visual tool for examining the trade-off between the probability of detection (TPR) and the probability of false alarm (FPR) across all the possible experimental threshold settings. Area Under the Curve (AUC) derived from the ROC curve is commonly used to obtain a summary performance measure by which to compare classification methods on the same data set.

The study also mentions four characteristics of software metrics data, referring to the NASA MDP data sets in particular, which may make it more difficult to develop accurate predictive models. These problems arise if only a small proportion of software modules are defective, there is a high correlation between metrics; there are many small modules which provide little variance by which to discriminate; and that a significant proportion of close neighbours to defective instances are non-defective. With respect to the latter, percentages are provided on defective modules whose nearest neighbour is the majority class, and on defective modules whose 3 nearest neighbours comprise at least 2 which belong to the majority class, for a number of the NASA data sets. It is said this would pose a problem for all machine learning techniques, but especially instance-based learning methods.

Still on the subject of model evaluation, Arisholm et al. (2007) argue that standard performance measures based on the confusion matrix do not provide enough evidence in themselves that a model is cost effective. Larger modules typically contain most faults, because they are more complex. These modules are easy to target without a model. So the benefit of a model is in how well it is able to locate faults beyond the factor of size. It proposes that this cost effectiveness be measured by a ROC like graph, with percentage of code along the x-axis, and percentage of faults along the y-axis. Assuming faults are correlated with module size, then, if modules are ordered randomly, on average, as percentage of code increases, percentage of faults will increase correspondingly, to produce a diagonal line in the plot. This serves as a baseline, which assumes that faults are distributed purely according to module size, and therefore a model producing this baseline curve would offer little benefit. A cost effectiveness curve is produced for a given model, by ordering modules according to probability of being faulty (and then according to size). If the model is effective, then the initial modules will contain most of the faults, and the curve will rise abruptly above the diagonal reflecting the fact that a high percentage of faults had been uncovered in a relatively small proportion of the code. An optimal curve can be produced by ordering modules according to known faults rather than predicted. Cost effectiveness of a model is measured as the area between the model curve and the baseline. Typically however, budget may limit testing of code to only a percentage of the total, in which case, cost effectiveness of a model is measured as the area under the curve along the x-axis until the desired percentage is reached. What this amounts to is evaluating the effectiveness of a model in terms of uncovering as many faults as possible in the least amount of code. (Menzies (2004) proposed a similar measure to this, in terms of combining faults and the amount of source code to be inspected, in his 'effort' metric. This is the number of lines of source code in modules predicted faulty, divided by the total number of lines in the system.)

The above approach for gauging model effectiveness would appear useful. This may be more so the case in light of the results of the study, in which, in applying a number of different types of machine learning algorithm to the fault prediction task on a multiple release telecom Java system, the confusion matrix based measures did not produce much variation in performance, but the effectiveness measures did. Methods applied included SVM, NN, and boosting, and a couple of feature selection methods were used. Of further interest was the finding that, in considering the effectiveness measure, the algorithm which gave a consistently high result, regardless of the limit on percentage of code considered, was the decision tree,

C4.5. Some algorithms however performed better at different percentage levels, and optimal effectiveness could be achieved by choosing the algorithm accordingly. It is perhaps worthy of mention, that the precision values obtained were very low regardless of modelling technique used, usually below 5%, but this was complemented with a high recall of up to 80%. Low precision was attributed to class imbalance, but one would nevertheless hope for higher precision levels than those reported. Because of the problem of imbalance, training sets (for all modelling techniques applied) were balanced by oversampling the positives to be of equal size to the negatives.

2.5 Other Data Set Studies

A couple of studies used particular learning algorithms to create fault prediction models on non-NASA data sets with success. Xing et al. (2005) applied SVM to a small data set of 203 modules, derived from a medical imaging system. Fault counts were estimated based on change reports (only those changes that affected the executable code) – atypically classes were roughly balanced in this data set. Models achieved "relatively good performance" (e.g., 5% FNR and 12% FPR). SVM was recommended for software quality modelling because it is known to generalise well even in high dimensional spaces and small training set conditions, and it is adaptive to model non-linear functional relationships that are difficult to model with other techniques. A variant of SVM was proposed in which rather than there being a single error penalty on both types of error, there is an error penalty for each class. This allows for some control over the trade-off between error rates, in view of the FNR (type 2 error rate) being more important to lower than the FPR (type 1 error rate) in order to maximise detection of faulty modules, usually at the expense of a higher FPR.

Denaro et al. (2002) developed fault prediction models using logistic regression on a fairly small data set (139 modules) derived from an antenna configuration system. Results were excellent, with a TP rate of around 90%, and precision above 85%. It was emphasised that attempts to produce general fault proneness models have been unsuccessful and that they must be tailored to specific application domains. Instead of feature selection, all possible combinations of size from 1 to 6 features were generated from the 33 available software metrics, which made for a large number of models. This is effectively feature selection using a wrapper with exhaustive search, but only of subsets up to the specified size. The metrics found to relate to fault proneness, based on the features of the four best models, were those related to size, comments, interface complexity, and internal complexity. Comments were

thought to be related to fault proneness, in that the lack of them can denote high-pressure on programmers which increases the probability of introducing error. A benefit of logistic regression is that it produces a continuous fault proneness indicator that allows models to be ordered according to their fault proneness, and this can be used to define effective testing strategies, in which the models with a higher probability of faults may be targeted first.

2.6 Other Kinds of Data Studies

A few other fault prediction studies are worth mentioning for the use of data beyond, or instead of, complexity metrics measured on source code. Shroter et al. (2006) developed models using only the classes and packages imported into a component, typically available at design time. This information was mined from CVS and Bugzilla repositories for plugins from the Eclipse project. Analysis initially focused on each import individually, for which a probability was calculated of its inclusion in a component being associated with the component being faulty. The use of an internal class was found to increase the likelihood of a post release failure. But the more interesting analysis was on whether combinations of import usages make good fault predictors. To do this, prediction models were developed, based on training data that contained a feature for each possible import, of binary type, indicating inclusion or not. Prediction was made on files and also packages, with the input vectors representing import inclusions of either classes or packages. Predictions were made within the same version on a holdout test set, as well as a subsequent version to evaluate model robustness. The SVM was one of four learners used. Both ranking and classification models were generated. Classification was based on a labelling with a decision boundary of zero, fault free components with -1, and faulty components with the number of faults. The best result was achieved by only considering the top 5% of packages predicted as fault prone, in which 90% were correctly classified. The SVM classifier was found to have the best predictive power, and predictive performance held across versions. The results show a correlation between imports and fault proneness that can be useful for prediction, and that a particular set of imports can be more fault prone than another and this relates to the problem domain of the component (e.g., UI, or compilation).

A study by Nagappan and Ball (2005) looked at whether defects reported by static analysis tools - used in the development environment at Microsoft - could be useful in predicting the number of pre-release faults found by other testing methods such as manual code inspection. This was performed on a large amount of code (22M LOC) from Windows Server 2003.

Static analysis defect density was found to correlate with pre-release defects, and prediction models were developed from static analysis defect density that were able to predict pre-release defect density at statistically significant levels.

Bernstein et al. (2007) used historical data mined from bug tracking and version control systems for a number of plug-ins for the Eclipse project. Data over most of the period of time for which data was collected, was used to predict both fault proneness and the number of faults in each file, for the last month in the period only. An important element of the study, or the data in particular, was the inclusion in collected metrics, of temporal/historical information. For example, the number of revisions, or defects, over a varied window of time. These, and other non-temporal metrics, such as LOC, age, and number of lines added to fix a bug, for each file, were selected carefully, partly based on a review of measures mentioned in the literature, to make up a small set of 22 metrics that comprised the data. The decision tree learner C4.5 was used to build the fault proneness models, and a regression decision tree M5, the fault count models. Remarkable in this study was the predictive performance achieved, which the authors would seem justified in describing as excellent. The best fault prone model had a TPR of 84% with 0% FPR. That is, high detection with virtually no false alarms. The corresponding F value was 88%. The results of the fault count prediction models had the predicted fault count for each Eclipse plug-in within 80% to 90% accuracy of the actual fault count. This performance for both types of prediction was attributed largely to the inclusion of temporal features. In the fault proneness models for example, models trained without temporal features performed consistently worse than those with. Another important factor in performance, in relation to the fault count models, was the use of a non-linear regression model, which it was concluded was necessary to take advantage of the non-linear relationships between temporal features and the number of bugs. Confirmed in this study, and supported in other studies, is the view that change history is a better predictor than traditionally used code metrics. The drawback with the approach used in this study, is that in order to use the models, it is necessary to have a repository of historical information from which to compile the needed metrics as the models input. This would still have application in many contexts, however it does not have the ease of use of models developed on static source code metrics alone, as with the approach taken in this research and which has been the focus of many other studies.

2.7 Unsupervised Studies

Finally, all the modelling efforts described above have relied on supervised learning methods. The use of unsupervised learning methods is much less common. A couple of studies which have explored its use however are briefly mentioned. Dick et al. (2004) investigated the unsupervised learning method of fuzzy cluster analysis. It provides an obvious advantage in that it does not require failures to have been observed in order to assign class labels to training instances. Each formed cluster was treated as a class. It was claimed that fuzzy modelling of classes provides a better representation of the true state of nature in software failure analysis. Criticism was made of one study on the subjective categorisation of modules into classes based on fault count. It was argued that there was no analysis performed to support the view that a module with 9 faults was substantially different to one with 10, where a module with 10 or more faults was considered to be at high risk. For the neural network classifier at least, which seeks to find a smooth mapping from the 'actual mapping' of inputs to outputs, the subjective judgment on class categories was thought to destroy the actual mapping. This study also makes two observations about software modules. The first was that most modules have a low degree of complexity. This means that most will be simple modules, with low metric values, and a low occurrence of failures. This would concur with the often quoted rule of thumb that 80% of the systems bugs are found in just 20% of its modules. It was suggested this can be an especially serious problem for machine learning approaches in that in optimising a global measure of predictive accuracy they tend to guess the majority class. A solution to this problem is to differ misclassification penalties. The second observation made was that modules with high metric values seemed to have a larger variation in their change (fault) distribution (in the Medical Imaging System data set studied). This could pose a problem for machine learning algorithms, in particular function approximation ones, making it more difficult to recognise the high faults classes.

Perhaps the best known study of the application of unsupervised learning methods to software fault prediction is that by Zhong et al. (2004). Unsupervised methods are seen to offer a solution to the problem of lack of fault measurements. The study proposes a semi- supervised clustering and expert based method, in which instances are clustered first, and then a label is assigned to each cluster based on the judgment of an expert on a representative cluster instance. The assumption is that based on measures for each instance, faulty instances will cluster together, as will non-faulty instances. The results of this approach suggested it had promise. It was hypothesised that minority instances of a particular class by the clustering

method are likely to be noise. This was confirmed using another method for noise detection referred to as an ensemble noise filter approach, the noisy instances identified by which largely matched the minority instances in clustering. While these authors have a focus on removal of noise from the NASA data sets, it would not seem to be standard practice to do so perhaps because some classification algorithms are more tolerant of noise. Prior to analysis as a preliminary noise filtering step, inconsistent instances were removed from the data sets, as those which had similar feature values but different class labels. It is also noted in this study that the jm1 dataset is a difficult one to classify, even for many state-of-the-art classifiers.

2.8 Menzies' Studies

The previous sections have covered studies by various authors. Here, the focus is on the work specifically of Menzies, one of the leading researchers in the development of software quality models from fault data. As such his work is highly relevant. His studies are summarized in the following paragraphs chronologically. Perhaps the most important element to come from them in the context of this work is the benchmark results obtained on the NASA data sets using the Naïve Bayes classifier. It is against these that the results of this research will be compared.

In one of Menzies' earlier papers (Menzies, Ammar, Nikora, & Di Stefano, 2003) the main focus was on the use of feature subset selection methods as an alternative to previously used principle component analysis (PCA) for feature selection. In PCA, the original set of metrics is mapped to an effectively smaller set that captures most of the observed variation and in which inter-correlation is removed. Feature subset selection was found to give better model performance, and this was achieved with a significant reduction in features, down to between 1 and 6 from upwards of 20.

In another paper in the same year (Menzies, et al., 2003), alternatives to the performance measure of accuracy were defined, as more suitable measures given the imbalance between the classes where modules without faults greatly outnumber modules with faults. These measures and others are discussed in Chapter 3. In all of Menzies' studies, and for that matter the studies of most others as well, examples are labelled fault prone if the fault count is 1 or greater, otherwise not fault prone.

Subsequently he focused not on how many features could be removed, but on how many training instances could be discarded, without loss of performance (Menzies, Raffo,

Setamanit, DiStefano, & Chapman, 2004). Surprisingly, only a relatively small number of instances of between 200 and 300 were sufficient to give the same performance as on the whole data set. Also noted in this study was a "stratification effect" in which sub-sub-sub-system level data sets performed better than higher level ones and required fewer instances as shown in Figure 5. Seen too was the well-recognized trade-off phenomenon between the performance rates "true positive rate" (TPR) and "false positive rate" (FPR), also referred to as detection rate and false alarm rate. The former is the proportion of positives (faulty instances) correctly predicted positive, and the latter, the proportion of negatives incorrectly predicted positive. A high detection rate and low false alarm rate is desirable. However, as noted in the study, any increase in the detection rate is had at the cost of an increase in the false alarm rate, resulting in the trade-off dilemma. This is evident in a plot of performance of 30 models shown in Figure 6. Models are sorted according to an effort metric, which is inspection effort measured as the proportion of lines of code in detected modules over the total number of lines of code in the system.

Another point of interest was that defect model performance compared favourably to manual code inspection, and it is faster and cheaper. Probability of detection for manual inspection effort from the literature ranges from 13% to 30% and for more elaborate inspections from 35% to 65%, which compared with the detection rate obtained for the NASA kc1 data set of 55% (which rose further in a later study).

The next study (Menzies T. , 2006) sought to compare the performance of a few learning

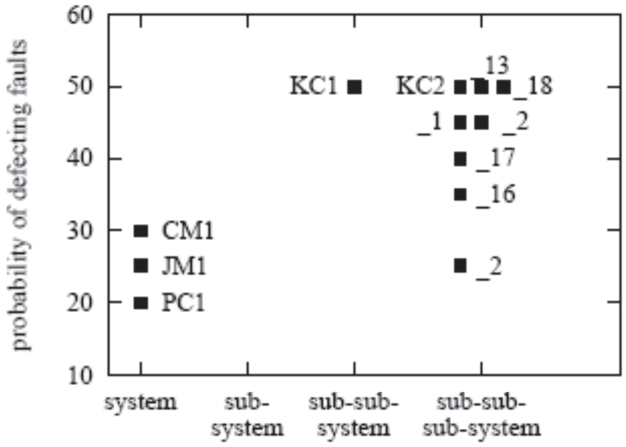


Figure 5 Menzies' detection rates for the NASA data sets ordered horizontally by sub system level revealing the better performance of more specialised sub-systems.

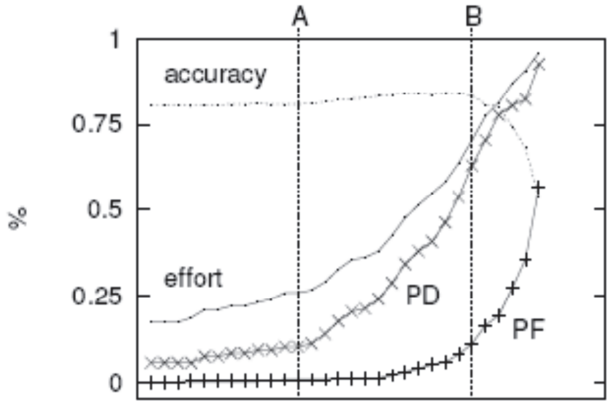


Figure 6 The performance of a range Naive Bayes models generated by Menzies, sorted by 'effort' showing a trade-off between the detection rate (PD) and false alarm rate (PF), and along with the increase in both (predicted positives) an increase in inspection effort as well.

algorithms, and determine whether more sophisticated methods offered any advantage over simpler ones (among a number of other hypotheses). The simplest method, OneR, referred to a "straw man" approach, is essentially based on a threshold on a single metric. The other two algorithms compared were the decision tree J48, and Naïve Bayes. Unusually, metric selections were chosen manually, rather than with the use of automatic feature selection algorithms, to test specific groups of metrics. The idea of log normalising metrics was also explored. It was found that more sophisticated methods gave better results, Naïve Bayes outperformed the other classifiers, and that log normalisation of the data significantly boosted performance. Log normalisation gives feature distributions more gradient assuming values are plotted in ascending order, rather than most values being close to zero.

Up to this point in the discussion, the work of Menzies has been more exploratory in nature. Menzies' subsequent work turned to the task of using the best methods found from the exploratory work to produce baseline performance measures on the NASA data sets (Menzies, Greenwald, & Frank, 2007). Multiple learning algorithms were used again but with automatic feature selection. Feature selection algorithms applied were the ranking methods of Information Gain and Relief, and subset methods Correlation Based Feature Selection and Consistency Based Selection[1]. Both normal and log normalised data were used. Naïve Bayes again performed better than the other learners, with log normalisation. Information Gain was

[1] Ranking methods evaluate features individually as to their predictive value according to which a ranking of features is produced, while subset methods evaluate groups of metrics to find the one with highest merit. More detail on these two approaches and each of the feature selection algorithms mentioned are given in Chapter 3.

found to perform was well as other feature selection methods. Just the top 3 ranked features were used for most data sets (except for pc1, for which the top 7 from the information gain (IG) ranking were reduced to 3 by running selection exhaustively on all subsets of size 3). Interesting insight was gained into selection in that there was little repetition of features between selections across data sets, and this was found to be due to many features at the top of the information gain ranking having similar weighting as shown in Figure 7. The conclusion from this was that identification of most relevant or informative features as had been the subject of earlier studies was somewhat futile. The different orderings were also seen as a reason for Naïve Bayes performing better than other learners, as its method of polling of multiple Gaussians across features was seen to be robust to the brittleness of the data in terms of variation in features selected depending on training sample. Baseline results obtained are shown in Table 1. The mean performance on sub-sub system level data sets was a detection rate of 71% and false alarm rate of 25%. This is similar to the performance obtained in other research in binary classification on standard data sets from the University of California Irvine repository. This is a good result, and is seen as evidence that static code metrics are useful in capturing information about software quality. Methods suggested for improving performance include developing new metrics, particular inter-module ones rather than intra-module, using other learning algorithms and others ways of pre-processing the data apart from simple log filtering.

In Menzies' most recent work on the subject (Menzies, Turhan, Bener, Gay, Cukic, & Jiang, 2008), any optimism for improving performance further was replaced with a view that a "performance ceiling" had been reached. This was based on his own efforts in trying many

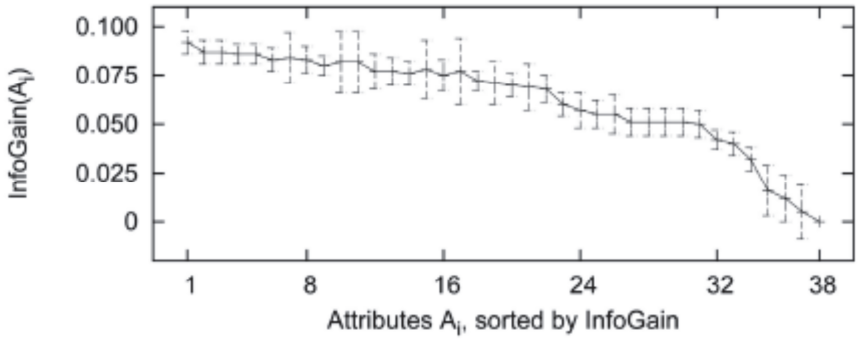


Figure 7 Information gain of features in a NASA data set indicating that high feature relevance with respect to class is shared among many features. This suggests it may be mistaken to be looking for a small set of features that best characterise quality, and instead that any subset could be taken from those plateauing at the high end of

	TPR	FPR	F1	F1.5	F2	Menzies Attribs.	Selection method
cm1	68	28	32	40	47	lC, lnOpdU, lnOptU	Iterative subsetting
kc3	68	28	31	39	46	lE, hN, hT	Iterative subsetting
mw1	55	14	34	40	44	hB, gN, lnOpdU	Iterative subsetting
pc1	51	18	26	32	37	cCP, lnOpdU, lN	Exhaustive subsetting
pc2	78	14	4	7	10	lC, lCp	Iterative subsetting
pc3	80	35	33	42	51	lB, hI, lN	Iterative subsetting
pc4	98	29	48	60	69	lB, lCC, lCp	Iterative subsetting
jm1	25	18	25	25	25		
kc1	50	15	43	46	47		
Avg:	64	22	31	37	42		

Table 1 Baseline results of Menzies for the NASA data sets (Menzies, Greenwald, & Frank, 2007).

different learning algorithms, including random forests and bagging (which yielded slight but negligible improvement), and on those of another study which at the time was unpublished in which two dozen learning algorithms yielded no significant performance improvement. Naïve Bayes tied first place with 14 other methods. The view concluded was that static code metrics contain limited information content (that relates to defects), and it can be, and has been, quickly and completely discovered by even simple learners. The way forward is seen not to lie in better algorithms, but in improving the information content of the training data. This is thought best done through a Case Based Reasoning approach in which there is a human in the loop, which makes use of any information available, and in which business knowledge can be contributed. Also it was found that by sampling specifically to balance the classes, as few as 50 instances were required to build a model.

2.9 Discussion

At the outset of this research it is essential to consider whether the approach has merit; the studies just described suggest it has. A number of the studies mentioned obtained convincing predictive performance e.g. TPR of 90% and precision of 85% (Denaro, Morasca, & Pezzè, 2002), albeit not on the NASA data sets. This is without the benefit that change history seems to provide, using just static code metrics.

The studies on the NASA data sets, other than Menzies', did not reveal much systematic analysis of performance levels. In one study for example (Vandecruys, Martens, Baesens, Mues, De Backe, & Haesen, 2008), the focus was on a new algorithm AntMiner+ and how it compared with a number of other algorithms on just three of the NASA data sets. Simpler

validation in this case, not cross fold but holdout, might undermine the reliability of the results. The only result that can be reported here (other than Menzies') as an indication of performance that might be achieved on the NASA data, is the one given in the random forest study by Guo & Singh (2004). High recall was achieved, at 87%, and accuracy ranged from 75% to 95%. The use of accuracy though, is not particularly helpful. The 95% result, with 5% of classifications in error, could be the result of misclassification on a large proportion of the positives. If this was the case, the model would be rather useless. The best indication of performance on the NASA data is given by Menzies, the work of which will be discussed shortly.

What can be gleaned from the NASA data studies largely relates to choice of algorithm. In the recent study by Elish and Elish (2008), a number of desirable characteristics of the SVM algorithm were mentioned in support of its use for quality modelling. Comparing SVM against the performance of eight other algorithms, SVM was found to perform at least as well if not better than them. This would support the use of the SVM algorithm in this research. SVM was also recommended in the study by Xing et al. on non-NASA data (Xing, Guo, & Lyu, 2005). Customisation of the SVM algorithm in this study to allow penalty on error, usually through the parameter C, on each class individually, to control the trade-off in performance between precision and recall, could be a useful one. Apportioning higher penalty on misclassification of positives would provide for better recognition of positives, which might otherwise be under recognized due to class imbalance.

AntMiner+ might be of interest in quality modelling, given it performed as well as other algorithms, if it were desirable to be able to comprehend the models, but that is not the case in this research with performance being the prime consideration. The random forest algorithm was found by Guo & Singh (Guo, Ma, & Harshinder, 2004) to generally perform better on the NASA data than the 20 other classifiers studied (bearing in mind the shortcoming of accuracy used in this study as an evaluation measure when classes are imbalanced). An advantage of this algorithm, relevant to the NASA data, is robustness to noise. The random forest algorithm was also found to be one of the best algorithms overall in the data transformation study by Jiang et al. (Jiang, Cukic, & Menzies, 2008) of 10 classifiers also applied to the NASA data (and boosting was another). The only other algorithm consistently among the best performing in the random forest study was Voting Feature Intervals (VFI). Merit for this algorithm might also been seen in its selection by Tosun et al. (Tosun, Turhan, & Bener, 2008) as part of an ensemble of classifiers trained on the NASA data. It was selected because in

other research it was found to be more robust and accurate than Naïve Bayes on real world data sets. Although there is merit for this algorithm in these two studies, it has not been used in this research. The next best algorithms in the random forest study, after VFI, were KStar and Naïve Bayes. Curiously, SVM was not among the best performing on any of the data sets, perhaps due to difficulty in recognizing the minority class. A shortcoming with this study, and perhaps other studies in which a large number of learning algorithms are employed, is the lack of different parameter values tried. Only a single SVM kernel may have been used for example, and only a single feature selection.

On a side note, the authors of the random forest study commented that different optimal feature sets selected for each data set, as determined by the random forest algorithm (as a possibly novel means of selection), were probably due to the different environments and processes of each project, suggesting that quality models be tuned on local projects. However, as mentioned later in this discussion, Menzies found that many features share the same level of predictive relevance at the higher end of the relevance range. There may be many subsets drawn from these features that perform optimally. Variation in subset between data sets may therefore not necessarily be due to the different projects from which they were derived.

Three other issues were touched on in the NASA studies: ensemble classification, data transformation and module size. One study focused on ensemble classification in an attempt to improve on Naïve Bayes performance alone (Tosun, Turhan, & Bener, 2008). Two other classifiers were combined using majority vote. But this attempt at improving performance was not successful, at least on the NASA data. This might be of some discouragement as to what benefit ensemble approaches could provide (on the NASA data). It might agree with Menzies' view of exhaustion of information content in the data. But in the data transformation study of Jiang et al. (Jiang, Cukic, & Menzies, 2008), also on the NASA data, boosting was consistently found to be one of the best performers. If ensemble classification were to be explored therefore, although they have not been in this research, boosting might have more potential than simple voting schemes. It may not be fair though to favour the former on the basis of a single study.

The transformation study by Jiang et al. explored whether transformations on the NASA data could be of any benefit to predictive performance. This was undertaken based on the finding of Menzies' earlier work, mentioned in the later discussion of Menzies, that log transformation provided a significant boost to Naïve Bayes performance. This study was very broad in scope, looking at the overall effects of transformation on performance across

data sets. Based on Menzies' finding, this work has included log transformation in experiments. However the finding of the transformation study was that no transformation improved overall performance, and indeed, log transformation might even diminish performance. This gives pause for its use. It would appear to support the findings of this work, in which log transformation did not consistently produce better results with Naïve Bayes, and in fact only worsened performance with the new classification method, rank sum. That contradictory findings were obtained, could possibly be due to the performance measure used, AUC. Another potentially useful finding of this study was that Naïve Bayes performance, while not improved by log transformation, was greatly improved by discretization. Transformation was recommended prior to any application of Naïve Bayes for building software quality models.

The final NASA data specific study discussed is one that investigated the effect of module size on performance. This study by Koru and Lie (2005) made two observations about the data that could perhaps be of use:

- Most modules were small in size and therefore metric values were small, if not zero. This could provide difficulty for learners to discriminate by.
- Large modules, with the benefit of larger metric values, were unfortunately relatively few in number. Similarly this could also be of difficulty to learners.

Modelling on only larger modules improved performance. This prompted some investigation as described in 5.9.2 Module Size.

Few studies appear to have been made on the Eclipse data sets, these only having recently become available. The study described in the chapter was one to accompany their release, to introduce them and provide results on some very preliminary modelling efforts. A point that might be made is that while open source projects and corresponding online development databases have been available for some time, fault data sets have not been generated due to the difficulty in associating fixes to actual source modules. This link is not made formally in the development records maintained. The achievement made then, and from which it became possible to produce the Eclipse data sets, was the finding of a technique to process pieces of information in the CVS and Bugzilla databases, that allowed the link mentioned to be inferred, if sometimes without certainty of accuracy. This may see new fault data sets becoming available in the future on other open source projects. That there may be some degree error in the inference made in relating bug reports to fixes, could introduce noise into fault counts. This issue is discussed further in 5.8 Eclipse Data Sets.

The study suggests based on the distribution of modules by fault count, which resembles an exponential one, that two modelling situations might be addressed: classification in terms of which modules contain faults, and ranking in terms of which modules contain the most faults to which testing efforts might first be directed. In this research the focus is only on classification, recognizing faulty modules with high fault density. Performance obtained with a logistic regression model with classification threshold of 0.5 on predicted likelihood, was far from impressive.

The topic of model performance evaluation was also given attention in the literature. The main issue here is that accuracy is a poor measure of performance for fault data where there is a much stronger presence in the data of negatives than positives. Accuracy then tends to reflect only on the majority class. Other measures may be suggested instead, such as precision, and single measures of performance such as the F measure, which provides support for its use in this research. Performance measures are discussed in more detail in 3.3 Performance Evaluation. This study mentions four characteristics of the NASA data that makes modelling more difficult:

- Small proportion of defective modules.
- High correlation between metrics.
- Many small modules which provide little variation with which to discriminate.
- Significant proportion of close neighbors to defective modules are non-defective.

The latter point is said to be particularly an issue for instance based learning methods.

Exploration of the literature revealed another means of model evaluation in the fairly recent study by Arisholm et al. (Arisholm, Briand, & Fuglerud, 2007). Performance of a model was not seen as sufficient in itself to prove cost effectiveness of a model. That is, that a model provides practical benefit beyond simply targeting modules by size – which tends to be somewhat effective as larger modules are more likely to contain faults. A technique for measuring cost effectiveness was described. Though not as yet widely adopted, and not used in this research, it was used in the ensemble study mentioned previously by Tosun et al. (Tosun, Turhan, & Bener, 2008). It is interesting to note that while little variation in performance might be detectable using standard confusion matrix based measures, there can be more differentiation between models in terms of cost effectiveness. As well, on a number of different machine learning algorithms employed in this study with which models were developed (on a Java telecom system), which included SVM and NN, the algorithm which

gave consistently high results according to the cost effectiveness measure, was the decision tree C4.5.

In relation to other types of data for modelling, module change information over multiple releases of a system, while providing unquestionably for more accurate models, requires in order to make classification on a module, record of the change history of the module. This may not be available. This is why there is an effort in this research and other studies to develop models from static source code metrics alone – for convenience, and necessity in the case where change information is not available. But two of the studies mentioned used data other than static source code metrics that were easily obtained and had predictive value. This is import statements to a module, and reported faults from a static analysis tool. These could potentially be useful in quality modelling efforts, perhaps in combination with static source code metrics.

Unsupervised studies were mentioned more for scope than being of practical value in this research. The fundamental assumption of these methods is that features capture information that relates to quality, and by grouping modules according to similarity by those features, models will be grouped according to quality. Given the nature of the NASA data however, where PCA plots of the data show what appears to be virtually completely random interspersion of points of both classes, the potential for accurate grouping by quality from clustering would not seem to be very great. Some exploratory work with clustering seemed to confirm this. However, Zhong et al. (Zhong, Khoshgoftaar, & Seliya, 2004) found this approach to have promise as a means to label data in the absence of class labels, with the assistance of an expert to effectively assign a quality category to formed clusters. With fault information in the NASA and Eclipse data sets with which to label instances, this technique then would not be of interest.

Perhaps a more interesting aspect of this work was in using clustering to identify noisy instances. Minority instances by class in each cluster may be considered noise. While most studies would seem not to take explicit action to remove noise from the data, Khoshgoftaar (as a co-author of the aforementioned study) seems to believe its removal from the data could be beneficial, in particular with the NASA data, and a number of his studies have focused on this issue. In this work, only the most extreme outliers are removed, as potentially distracting to feature selection and learning algorithms, according to the Mahalanobis distance (see 6.1 Initial Filtering).

The other unsupervised learning study described by Dick et al. (Dick, Meeks, Last, Bunke, & Kandel, 2004) in which fuzzy clustering was used, is mentioned here only for the opinion made about class labeling in supervised studies. It was said that subjective categorization of modules into classes by fault count, by selecting an arbitrary threshold, was an approach that did not have support based on analysis of the data. In doing so the actual mapping in the data is replaced by an artificial one. The neural network was singled out as a classifier that might suffer from this manipulation. For this reason fuzzy clustering was used in their study, in the belief that it provides a better representation of the true state of nature in software failure analysis. This position would perhaps be critical of the class labeling approach taken in this research.

Finally, to the work of Menzies. His studies have guided the modelling efforts in this research more than any others. The main findings of his studies from 2003 onwards are listed below:

- Feature subset selection can dramatically reduce the number of features without loss of performance.
- Accuracy as a performance measure was found to be lacking, and he proposed instead that two measures be used, the probability of detection (PD, or detection rate), and the probability of false alarm (PF, or false alarm rate). Mentioned was a trade-off between these measures, although this is a common phenomenon, not specific to the NASA data.
- As few as 200 or 300 training instances could be used without loss of performance. Later with "micro" sampling from each class, this dropped to only 50 training instances.
- A "stratification" effect was observed in that sub-sub-sub system data sets performed better than higher-level ones.
- Detection rates from the NASA data sets were found to compare with those reported in the literature for manual code inspections. Later, on sub-system datasets, performance was found to match that of standard data sets from University of California Irivine repository – roughly 80% detection rate, and 20% false alarm rate.
- Naïve Bayes was considered a better classifier overall to the decision tree J48 and the simple classifier OneR.
- Log normalization was found to boost Naïve Bayes performance significantly.

- "Iterative subsetting" was used to select features based on IG ranking of features, and Naïve Bayes performance on top down subsets from the ranking. These selections were found to perform as well as two subset selection methods CFS (correlation based) and CBS (consistency based).
- Many features have high relevance according to IG, so there is little point in trying to identify a small set of features that could be seen to have special significance in prediction.
- In Menzies' own work and other studies the application of many machine learning algorithms to the NASA data has not resulted in any significant gain in performance, and it is believed this is due to the information content in the static code metric data that relates to fault proneness having already been exploited – even by simple algorithms. It is believed that a "performance ceiling" has been reached.

These studies as discussed provide some context for this work. The task in practical terms then, is essentially to apply two learning algorithms Naïve Bayes and the Support Vector Machine, to get the best predictive performance possible from the data, and compare obtained performance with that reported in the literature, specifically that of Menzies. Some of the issues raised in this discussion of the literature are explored along the way.

The latter part of the research in development of the rank sum classification method in being novel does not relate to existing literature as far as it is known. But the results of this are also compared to the baseline.

Chapter 3 - Machine Learning

Machine learning is a field related to artificial intelligence (AI) that came about in large part to address the problem with expert systems of knowledge acquisition, often referred to as the 'knowledge acquisition bottleneck' (Buchanan & Wilkins, 1993). Programming knowledge into these systems is time consuming and error-prone. A means to automatically acquire knowledge from experience or data was needed. Acquiring knowledge from experience is learning, which is usually done with the goal of improving at some task. Machine learning came about to provide computers with this learning ability.

In practice, machine learning is concerned with algorithms to extract useful information or recognize patterns in data. While algorithms have been classified according to the primary type of inference used, inductive or deductive for example, or whether they use background knowledge or rely only on data, machine algorithms are usually classified according to input. The most common approach involves learning from input data whose instances contain two components, which are the input and the output of the function to be learnt. This approach is referred to as supervised learning, and is discussed first. The two algorithms of focus in this study, Naïve Bayes and the Support Vector Machine, take this approach, and are described in detail. The other less common approach, involves learning from input data whose instances contain only a single component – they do not contain a function output component. This approach is referred to as unsupervised learning, and is discussed briefly but has not been applied in this research.

3.1 Supervised Learning

Most of the research in Machine Learning has been devoted to supervised learning. The objective of supervised learning is to learn a mapping between an input space and an output space to approximate a target function $X \rightarrow Y$ from labelled observations e.g. from disease symptoms to diagnosis, or as is of interest in this study, software measures to a rating of software quality. In learning that mapping, a model or function is created that captures the input-output relationship. To learn the mapping the learner is supplied with examples, as a collection of input-output pairs, by a teacher with knowledge of the environment and the mapping of interest. From these examples, the learner performs an inductive search through hypothesis space to find an hypothesis that provides the best generalization of the mapping, that is, which is most consistent with the training data, and which generalizes well to unseen

instances, the latter of which is assumed based on the form and selection criteria. Common mapping representations include polynomial functions, and neural networks. The basic idea of supervised learning is illustrated in Figure 8.

More formally, if X is the input space, and Y the output space, then the training data D = { d_1, d_2,…d_n } where d_i = <$\mathbf{x}_i$, y_i>, $\mathbf{x}_i$ being the input vector and y_i the target output. The objective of supervised learning is to learn from the training data the function f:X→Y such that y_i = f($\mathbf{x}_i$) for all i=1..n.

The search through hypothesis space to find the hypothesis that best fits the training data is typically implemented in an iterative fashion over the training examples such that for each example <$\mathbf{x}_i$,yi>, the hypothesis is adjusted in order to minimize the error between the actual output of the current hypothesis V($\mathbf{x}_i$), and the target output y_i. This approach is illustrated in Figure 9.

Supervised learning is used in two types of problems: regression in which the output is continuous (e.g. mapping from system specification to project cost), and classification where the output is discrete (e.g. mapping from source measures to software quality rating). This study is concerned with classification problems.

Many supervised learning algorithms exist, but the two that will be employed in this research are Naïve Bayes and Support Vector Machines. These will be described in detail in the sections which follow.

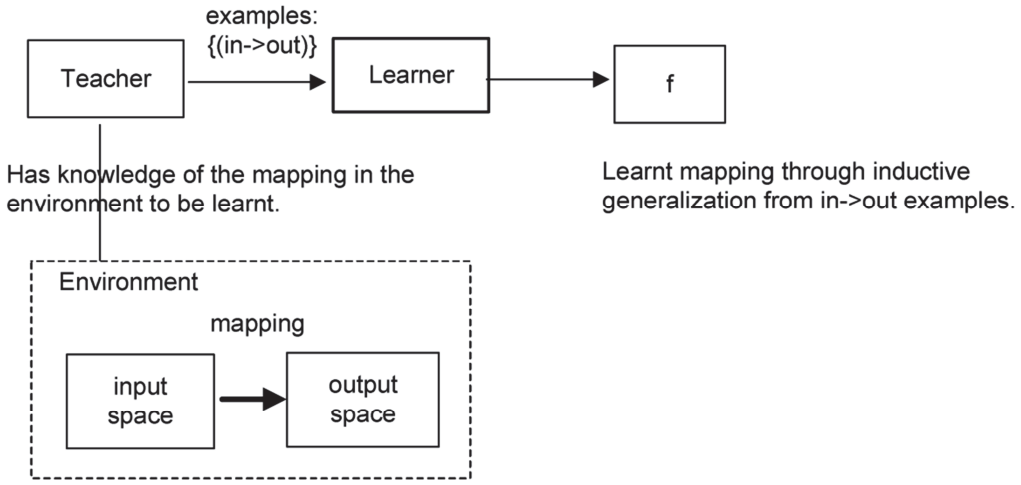


Figure 8 Supervised learning in which a function from the environment is described in terms of examples, and the learner aims to approximate it.

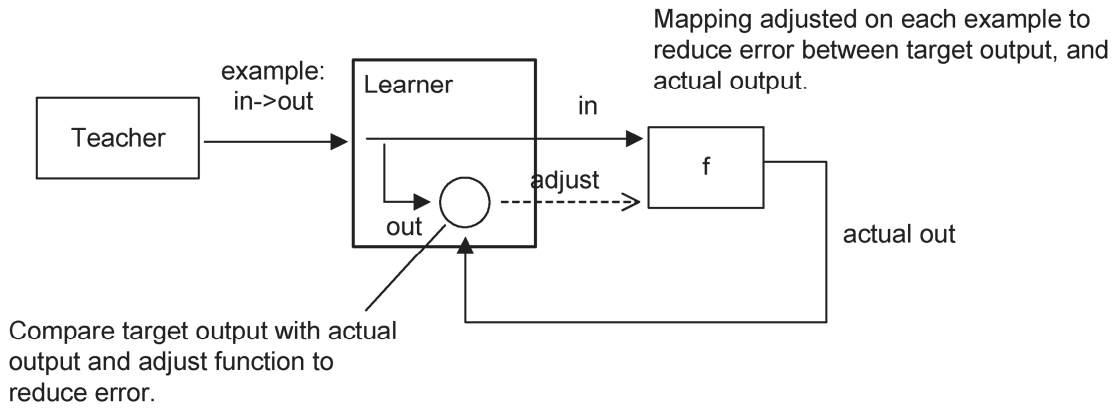


Figure 9 Typical implementation of supervised learning in which the learnt function is adjusted by comparing predicted output with actual for each example.

3.1.1 Naive Bayes

The Naïve Bayes (NB) classifier is based on direct application of Bayes Theorem. It is a relatively simple method, is easily implemented and efficient to run, yet achieves impressive results. It has been shown many times to outperform more sophisticated classifiers on many data sets (Witten & Frank, 2000). For these reasons it has become a widely used method, especially in the areas of text classification, information retrieval, and medical diagnosis (Yang, Xia, Chi, & Muntz, 2003).

As a Bayesian learning method, the approach taken in reasoning is a probabilistic one which may reflect uncertainty in the data and the relation. Predictions or classifications consequently have probabilities associated with them e.g. a software module has a 90% chance of being faulty. They are obtained by applying various probability theorems, primarily Bayes Theorem, to probability distributions over quantities of interest that are derived from the training data, the sample of instances which represent a joint probability distribution of vector attributes.

There are a number of advantages to the Bayesian approach to learning apart from probabilities being associated with predictions. It enables the incorporation of prior knowledge into learning, in a move from purely empirical learning, from data alone, as in decision trees, towards constructive induction. The approach also allows for less sensitivity to noise as hypotheses upon first inconsistency have their probability reduced rather than

eliminated. Due to the latter it is also possible to combine the probabilities of all hypotheses to make optimal predictions.

3.1.1.1 Bayes Theorem

Bayes theorem is used to calculate the revised probability, also known as posterior probability, of an event given that another event has occurred. In machine learning, the revised probability of interest is that of a hypothesis h, given that data D has been observed and is written in conditional probability notation as P(h|D).

The posterior probability that Bayes calculates is illustrated in Figure 10. Prior probabilities for h and D are shown as boxes which represent areas or events in sample space. The posterior P(h|D) is the probability of an event occurring in h, given that it is known the event belongs to D, and this probability is equivalent to the intersection of h and D, as indicated by the shaded region. The probability is divided by D for normalisation so that the sum of conditionals adds to 1. In Naïve Bayes classification, P(h|D) is computed for a number of hypotheses to find out which one has the greatest probability given the evidence or data. An example is shown in Figure 11 where there are two hypotheses h_1 and h_2, h_2 having the greater posterior probability. The posterior is expressed algebraically using the conditional probability rule as:

$$P(h|D) = \frac{P(h,D)}{P(D)}$$

Bayes theorem is obtained by expressing the joint probability P(h,D) in this equation in terms of an inverse conditional probability, P(D|h), using the product rule:

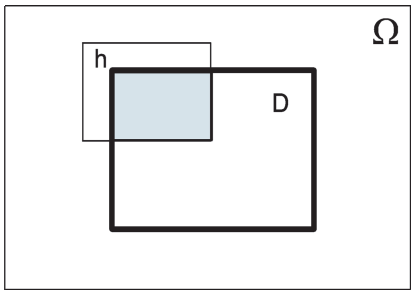


Figure 10 Bayes theorem calculates the revised probability of h (the shaded area, or the shaded area as a proportion of D) given that D is observed.

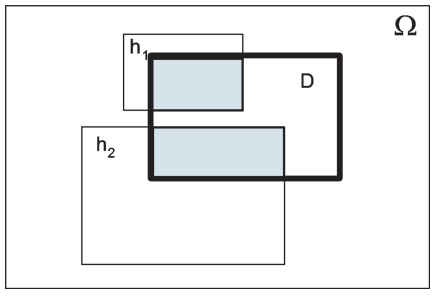


Figure 11 Bayes theorem in calculating the posterior probability of multiple hypotheses, in this case with h_2 having the greater posterior probability.

$$P(h, D) = P(h) \times P(D|h)$$

It can be seen in Figure 12 that the two sides of this equation are equivalent. The expansion is just a different way of expressing the same probability. Substituting this back into the first equation gives Bayes Theorem:

$$P(h|D) = \frac{P(D|h)P(h)}{P(D)}$$

The theoretical derivation of Bayes theorem as described is straightforward. The key point is that the posterior P(h|D) is calculated from a multiplication of the inverse P(D|h), the probability that the data is observed given h is true, or likelihood as it is called, and the prior P(h). The theorem is useful because often P(D|h) is known or is easier to obtain than the posterior, and from which the posterior P(h|D) can be computed. The theorem is sometimes written informally as:

$$posterior = \frac{likelihood \times prior}{evidence}$$

The posterior increases with an increase in either the likelihood or the prior.

3.1.1.2 Bayes Theorem and Machine Learning

Learning involves finding the hypothesis that best fits the training data. In Bayesian learning, which takes a probabilistic point of view, each hypothesis is seen as having an associated probability. The hypothesis that best fits the training data is the one that is most probable given the data. The probability of a hypothesis is calculated based on its prior probability, which is revised in light of observation to yield. As such, the probability is called the

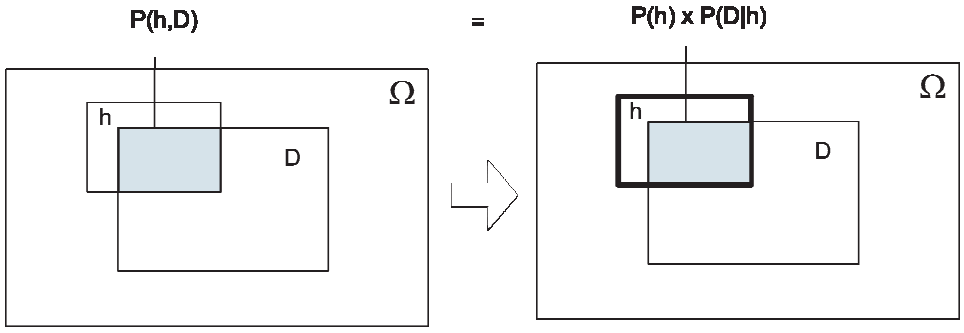


Figure 12 Numerator of Bayes theorem in which the probability that both h and D occur is described in terms of the prior probability of h multiplied by the likelihood that D will be observed if h is true (by the product rule).

posterior probability, and is calculated using Bayes Theorem. The maximum a posterior hypothesis sought, referred to as the MAP hypothesis, is defined as:

$$h_{MAP} = \underset{h \in H}{\operatorname{argmax}} P(h|D)$$

$$= \underset{h \in H}{\operatorname{argmax}} \frac{P(D|h)P(h)}{P(D)}$$

$$= \underset{h \in H}{\operatorname{argmax}} P(D|h)P(h)$$

The denominator P(D) is omitted as it is a normalizing factor which in terms of the ordering of posteriors relative to each other, has no bearing. A special case exists when the priors are uniform, in which the prior P(h) can be omitted from the formula for h_{MAP}. This effectively removes the contribution of background knowledge to give what is referred to as the maximum likelihood hypothesis, denoted by h_{ML}. Both h_{MAP} and h_{ML} are central to Bayesian learning and Bayesian learning algorithms.

A practical application of this Bayes-based MAP hypothesis to a machine learning problem, is the learning of a concept, such as 'good quality software'. The hypothesis that best fits the data is the one that best represents the concept and is the MAP hypothesis. The concept may be described in the data by mapping outputs to 0 or 1, to represent concept inclusion or exclusion. A way to find this hypothesis is by brute force: find the posterior probability for every possible hypothesis in the hypothesis space. In practical learning problems however, this may be infeasible due to the size of the hypothesis space, which motivates use of the Naïve Bayes method described shortly. In the next section one other theretical aspect will be touched on first, which is best possible classification, as that produced by the Bayes Optimal Classifier.

3.1.1.3 Bayes Optimal Classifier

In the concept learning scenario mentioned in the previous section, the interest was in finding the hypothesis that best fitted the data, the MAP hypothesis. However, in the supervised learning problem, with categorical outputs, the objective is not to find so much the best fitting hypothesis, but the best classification. While the MAP hypothesis would produce overall the most accurate predictions of all hypotheses individually, it is possible to obtain better accuracy by combing the predictions of all hypotheses. This is the basis for the Bayes Optimal Classifier, which finds the best possible classification of any method for a given hypothesis space and prior knowledge.

The method finds the maximally probable *classification* based on the posterior P(v|D) where v is a category label, rather than maximally probable hypothesis. This posterior probability for a given v is found by multiplying the degree to which a hypothesis fits the data, P(h|D), by the probability it gives that v is the correct classification, P(v|h), and summing this across all hypotheses. Effectively this represents summing the probabilities given for a prediction value across hypotheses, each weighted according to hypothesis fit. This may be defined formally as:

$$P(v_j|D) = \sum_{h_i \in H} P(v_j|h_i)P(h_i|D)$$

and the optimal classification is given by:

$$\underset{v_j \in V}{\text{argmax}} = \sum_{h_i \in H} P(v_j|h_i)P(h_i|D)$$

The classification given by this method may be different to that given by the MAP hypothesis alone. As with the concept learning example, here again, in practice it may not be feasible to compute when hypothesis space is large.

3.1.1.4 Naive Bayes Classifier

The Naive Bayes Classifier is a highly practical Bayesian learning method and one which will be employed regularly in the present study. In the best case it offers optimal classification, and it is very efficient, requiring at most linear time on the size of the input. It is a widely used classification method, especially in the areas of diagnosis and text classification.

As a supervised learning method for classification, it is trained from a set of examples in the form { (**x**,v_j) } where **x** is a description of an instance as a vector of attributes (x_1..x_d) and v is the category associated with it which belongs to a set of categories {v_1..v_c}. Given a new instance **x**, the task of the classifier is to produce the most probable classification. This is distinct from finding the MAP hypothesis, and as such the most probable classification is denoted by v_{MAP}. This is the *classification* (value v) with maximum a posterior probability. The posterior probability is calculated using Bayes Theorem (as for the MAP hyphothesis). The posterior probability for value v, given by P(v_j | **x**), is the revised probability of the category v_j given that the instance **x** has been observed. According to the theorem, the posterior is calculated from the prior P(v_j) and likelihood P(**x**|v_j). The first of these components is easy to calculate. It is simply the proportion of instances with category label v_j. The likelihood component however, is based on a joint probability distribution over the

instance space. It requires that the probability is known for the conjunction of any vector of values that **x** may assume, i.e. $p(x_1..x_n)$ given a category v_j. This can amount to a very large space of probabilities. In practice, training data does not usually contain instances that cover the whole range of possible conjunction of feature values by which to compute these probabilities.

So in order to estimate the likelihood probabilities on realistic training data whose instances cover only some of the input space, a simplifying assumption is made, by which the method gets its name. It is assumed that attributes $x_1..x_d$ are conditionally independent given a category. This allows the joint probability $p(x_1..x_d|v_j)$ to be calculated as $p(x_1|v_j)\times p(x_2|v_j)\times..p(x_d|v_j)$ according to the Chain Rule. The total number of probabilities required then is far fewer. This is only the number of attributes multiplied by the number of categories, which is quite feasible to calculate from the training data.

With this simplifying assumption, the maximum a posterior classification, $v_{MAP,}$ for Naive Bayes is defined as:

$$v_{map} = \underset{v_j \in V}{\operatorname{argmax}} P(x_1, x_2, .. x_n | v_j) P(v_j)$$

$$v_{NB} = \underset{v_j \in V}{\operatorname{argmax}} \prod_i P(x_i | v_j) P(v_j)$$

There is no loss in predictive accuracy by computing the joint probability in this simpler way, as long as features meet the assumption of conditional independence. While there is sometimes violation of this assumption, seldom does it occur to the extent that classification is poor. The main reason for this is that while this may affect the posterior probabilities arrived at for each category, it often is not enough to cause a change in the predicted, most probable, category.

In the learning step of Naive Bayes, frequencies are obtained from the training data from which the component probabilities for v_{NB} can be estimated. This amounts to a learnt hypothesis. For calculating the likelihood in particular, a frequency distribution is obtained from the data for each feature for each class. These provide the likelihood component probabilities $P(x_i|v_j)$. It is perhaps of interest to note that assuming the data satisfies the assumption of conditional independence, the hypothesis effectively learnt is the MAP hypothesis. The method is summarized in Figure 13.

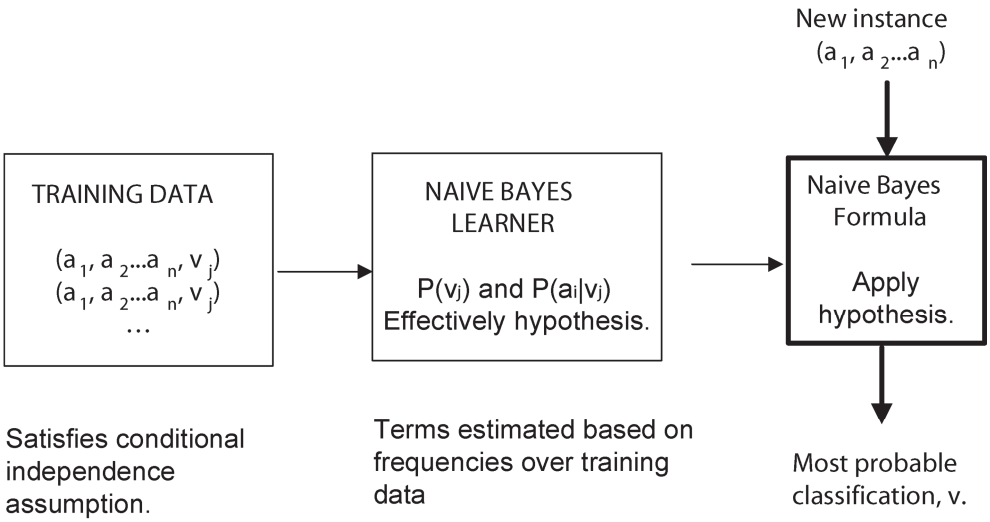


Figure 13 The Naive Bayes classifier: priors $p(v_j)$ and likelihoods $p(a_i|v_j)$ are easily calculated from the training data, and these are combined using Bayes rule to obtain a posterior probability, upon which classification is based.

3.1.2 Support Vector Machines

Of the many machine learning algorithms that have been developed, one which is currently considered to be amongst the best performing is the Support Vector Machine. It has been used for many applications, including handwritten digit recognition, text categorisation, and speaker identification (Burges, 1998), and for a wide variety of problems it has been found to perform as well or better than earlier learning algorithms. Crucial in classifier development is their performance on unseen data, their ability to generalize and avoid overfitting. Support vector machines tend to perform well at this task, as the statistical theory on which they are based is geared towards it. Apart from this core feature of the algorithm, it has other qualities which add to its appeal, such as it being based on simple linear classification, but with the use of kernels that efficiently map data between spaces to find non-linear classification boundaries. The geometry on which it is based is straightforward, as is the means by which optimum generalization performance is achieved in maximizing the margin between classes of points on either side of the linear decision boundary. The approach is solidly grounded in statistical learning theory (Vapnik, 1995). The essence of this theory is described first in section 3.1.2.1 Structural Risk Minimisation, leading into the geometric formulation of the SVM and its solution as a convex quadratic programming problem in section 3.1.2.2 The Basic Problem. Having covered the method in this basic form the issues of how non

separable data and kernel mapping are dealt with are addressed in sections 3.1.2.3 Slack Variables for the Non Separable Case and 3.8.2.4 Kernel Mapping for Non Linear Decision Boundaries.

3.1.2.1 Structural Risk Minimisation

The goal in developing any classifier is to minimize error, also referred to as risk. In the case of the support vector machine, the classifier is binary and of the form:

$$f(\boldsymbol{x}, \alpha) = 1 \; or \; -1 \tag{1}$$

where α is a particular set of algorithm parameters. The empirical risk – the error rate over the training data - is easily obtained for the usual zero-one loss, as:

$$R_{emp}(\alpha) = \frac{1}{2l}\sum_{i=1}^{l}|y_i - f(\boldsymbol{x}_i, \alpha)| \tag{2}$$

This is essentially the sum of errors, the deviation of function output from actual class for each instance. While this risk is of interest, in application of the classifier what is most relevant is the actual or expected risk. Theoretically this has been formulated as follows:

$$R(\alpha) = \int \frac{1}{2}|y - f(\boldsymbol{x}, \alpha)|dP(x, y) \tag{3}$$

To compute this however, the joint probability distribution for the input space is required and this is rarely available and is therefore not of practical use. It is possible though to calculate an upper bound on this risk which is independent of P(**x**,y). Vapnik formulated this as follows, a bound central to the underlying theory of SVMs:

$$R(\alpha) \leq R_{emp}(\alpha) + \sqrt{\left(\frac{h\left(\log\left(\frac{2l}{h}\right) + 1\right) - \log(\frac{\eta}{4})}{l}\right)} \tag{4}$$

The rightmost term is referred to as the VC confidence, part of which is the parameter η, the complement of which is the probability with which the bounds holds. This term is largely dependent on the variable h, the VC dimension of the classifier for which risk is being calculated. The VC dimension is a measure which gives some concreteness to the notion of classifier capacity. Before going further with this formula and expected risk, the VC dimension is very briefly described.

The notion of capacity is captured in terms of the maximum number of points which a classifier is capable of 'shattering'. A set of points can be shattered if the classifier, f, is able

to correctly classify each point, for every possible combination of class labels. For a binary classifier, if there are n points, there are 2^n combinations of labels to be predicted correctly if a classifier is to be said to be able to shatter them and to have VC dimension n. Clearly, if a classifier is able to shatter a larger number of points, it is of higher capacity.

Equation 4 bounds actual risk by the sum of the empirical risk and a monotonically increasing function of the capacity of the classifier. It may be thought that in developing a classifier, low error on the test set would likely work in favour of better generalization performance, but this may not be so if the test performance is achieved due to a classifier with excessive capacity. A higher capacity classifier will increase the upper bound on actual risk, allowing for the possibility of overfitting, which would result in a higher error rate on unseen data. Conversely, choosing a lower capacity classifier will not necessarily bring about better generalization performance, as it may result in higher empirical error. Thus, there is a trade-off between empirical risk and capacity to be found in order to obtain the minimum bound for actual risk. This is illustrated in Figure 14. It can be seen that as capacity/VC confidence increases, there is less error on the test set, or lower empirical risk. But around the middle point, the point of optimum trade-off where expected risk is lowest, while empirical risk continues to decrease with increasing capacity, overfitting starts to occur. This results in a turnaround in the expected risk curve which starts to increase.

The inclusion of the size of the training set, *l* (from equation 4), in VC dimension decreases the contribution of capacity to a higher bound on expected risk as training set size increases.

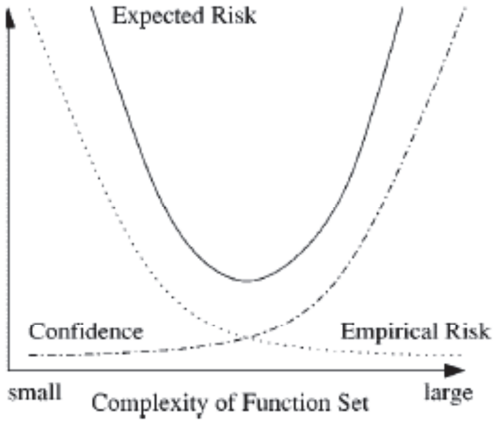


Figure 14 The trade-off between empirical risk and capacity/confidence, with the optimum trade-off at the minimum of the consequent expected risk curve, an optimum which evidence suggests SVM approximates.

This reflects a lower risk of excess capacity resulting in overfitting as more training data becomes available. The SVM can perform well on small training sets, despite being able to find quite complex decision boundaries and correspondingly being able to exhibit high capacity. This is due to it implementing a form of capacity control as part of the parameter optimization process.

Given this trade-off for the bounds on actual risk, a learning algorithm can use this to find a set of classifier functions with minimum bound. In theory, this is done by dividing the entire class of functions into nested sets each having different capacity h. This is the 'structure' referred to in the term structural risk minimization. The risk bound is then calculated for each set, and the one with the lowest bound is chosen. In practice, SVM achieves this by finding the separating hyperplane with maximum margin. While it has not been shown that this guarantees a function with minimum bound on actual risk, there is strong evidence to suggest that it provides a good approximation to it. A good example of this is shown in Figure 15. Although a polynomial kernel is used which is capable of defining a non-linear boundary, in the linearly separable case, the capacity is constrained so that the boundary is close to linear.

3.1.2.2 The Basic Problem

The basic problem to be solved on SVM is based is straightforward. SVM is a binary linear classifier and its task is to find a plane which separates the two classes of points in the training set such that the distance between the nearest points on either side of the hyperplane is maximal. It is assumed that the points are separable at this stage of the discussion. The data, in typical supervised form, is composed of points $\{\mathbf{x}_i, y_i\}$, where $y \in \{-1, 1\}$. The hyperplane is described by:

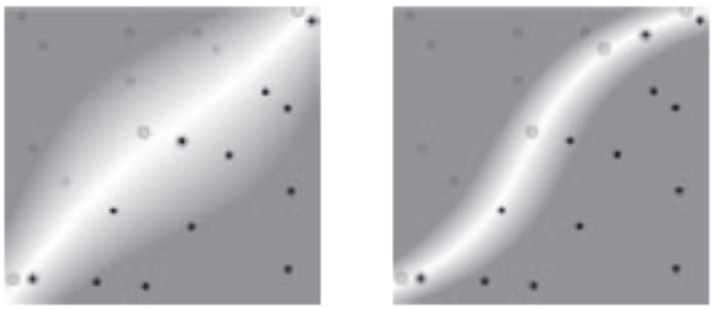


Figure 15 Limiting capacity of 3rd degree polynomial to near linear decision boundary in case where classes are linearly separable. (Burges, 1998)

$$\mathbf{w}\cdot\mathbf{x}+b = 0 \tag{5}$$

which is a vector equation of a line, where **w** is the normal to the hyperplane, and |b| / ||**w**|| is the perpendicular distance from the hyperplane to the origin. In this case with a functional output of 0, the points $\mathbf{x}_i$ which satisfy the question would lie on the hyperplane. If the perpendicular distance to the nearest positive is d_+, and negative d_-, then the margin that the hyperplane creates between the two classes, is $d_+ + d_-$. The objective of the SVM algorithm is to find the separating hyperplane with maximal margin. To find this, constraints are first placed on (5) such that its solution in terms of **w** and b defines any separating hyperplane. Separation is effected by specifying that the functional output of all positive points is positive, being 1 or greater, and negative points, -1 or less. This is equivalent to ensuring that positives lie on or beyond a hyperplane parallel with the central separating hyperplane whose functional output is 1, and similarly for negatives but on the opposite side. The nearest points $\mathbf{x}_i$ to the central hyperplane will thus lie on the hyperplanes $\mathbf{w}\cdot\mathbf{x}+b = +-1$. These constraints are defined as follows:

$$\begin{aligned} \boldsymbol{x}_i \cdot \boldsymbol{w} + b \geq +1 \quad \text{for } y_i = +1 \\ \boldsymbol{x}_i \cdot \boldsymbol{w} + b \leq -1 \quad \text{for } y_i = -1 \end{aligned} \tag{6}$$

These can be expressed in one set of inequalities as:

$$y_i(\boldsymbol{x}_i \cdot \boldsymbol{w} + b) - 1 \geq 0 \;\; \forall i \tag{7}$$

These ensure that the hyperplane will separate the two classes of points with the function output as desired. Required further however is the component which ensures that the hyperplane separates maximally. The distance specifically to be maximized, the margin, is the sum of the distance on each side of the hyperplane between it and the nearest point of each respective class. It can be shown through a simple geometric argument that this is equal to $2/||\mathbf{w}||$, and maximizing this is equivalent to minimizing $||\mathbf{w}||^2$. The constraints are incorporated through a formularisation of the problem described next.

A solution for **w** and b is to be found, and this is done using a Lagrangian formulation of the problem, or more particularly the dual formulation of it. A Lagrangian formulation is used because the constraints are replaced by constraints on Lagrange multipliers which are easier to handle, and it allows training data to appear only as dot products between vectors, which as discussed later, provides for non-linear decision boundaries. Positive Lagrange multipliers are introduced for each inequality constraint and these are subtracted from the objective function to be minimized. The primal Lagrangian formulation is thus given by:

$$L_P \equiv \frac{1}{2}\|\boldsymbol{w}\|^2 - \sum_{i=1}^{l} \alpha_i y_i (\boldsymbol{x}_i \cdot \boldsymbol{w} + b) + \sum_{i=1}^{l} \alpha_i \tag{8}$$

Rather than solve by minimizing the above objective function with respect to **w** and b, the problem is recast into a more convenient 'dual' form in which the objective function is to be maximized. This is possible because of the convex nature of the problem. The values of **w** and b for which the primal is minimal are exactly the same as those for which the dual is maximal. The problem then is to maximize L_P, subject to the constraints that gradient of L_P with respect to **w** and b vanish. The gradient gives the following conditions:

$$\frac{\partial \varphi}{\partial b} = 0 \quad \Rightarrow \quad \sum_{i=1}^{l} \alpha_i y_i = 0 \tag{9}$$

$$\frac{\partial \varphi}{\partial \boldsymbol{w}} = 0 \quad \Rightarrow \quad \boldsymbol{w} = \sum_{i=1}^{l} \alpha_i y_i \boldsymbol{x}_i \tag{10}$$

As (8) is an equality constraint it can be substituted into the primal function to give the equivalent in dual form:

$$L_D = \sum_i \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j \boldsymbol{x}_i \cdot \boldsymbol{x}_j \tag{11}$$

Support vector training therefore involves maximizing L_D subject to (9) and α_i being >=0. The solution for **w** is given by (10). The value for b can be found from the Karush-Kuhn-Tucker (KKT) conditions which are satisfied at the solution of any constrained optimization problem. Only shown here is the "complementary condition" for the primal problem from which b can be solved:

$$\alpha_i (y_i (\boldsymbol{w} \cdot \boldsymbol{x}_i + b) - 1) = 0 \quad \forall i \tag{12}$$

Any i for the set of instances in the training set can be chosen to substitute values into the other variables in the equation whose multiplier α is > 0.

The values for which $\alpha > 0$ are the "support vectors", after which the method is named and from which **w** in (10) is calculated. These are the points which lie along the boundaries of the margin on each side of the separating hyperplane, and are the operative points in defining the hyperplane. The same plane would be found if the training set contained only the support vectors, or if any of the other points were rearranged as long as they didn't cross into or beyond the margin tube.

The above describes the training procedure in determining **w** and b to find the separating hyperplane. Classification of a new instance **x** requires substitution into the equation of the hyperplane **wx** + b, the class being determined by the sign of the result, indicating on which side of the hyperplane the point lies.

3.1.2.3 Slack Variables for the Non Separable Case

In most classification problems, the two classes of points are interspersed and there is no hyperplane which can be found which separates them. However, the method described above can be augmented so that this situation can be handled. This is done with the introduction of slack variables e_i for each training instance. Strictly, a point is supposed to lie on the outside of one of the planes which border the margin 'corridor', depending on its class. A slack variable added to the original inequality constraints allows for a point to have some slack with respect to the strict requirement in that it must lie inside the boundary defined by **wx**+b = 1, say. The value of this slack variable is the amount by which it lies inside this region. If the error exceeds 1, then the point lies on the other side of the decision boundary, and the point would be incorrectly classified. The revised constraints are:

$$\boldsymbol{x}_i \cdot \boldsymbol{w} + b \geq +1 - \xi_i \quad \text{for } y_i = +1$$
$$\boldsymbol{x}_i \cdot \boldsymbol{w} + b \leq -1 + \xi_i \quad \text{for } y_i = -1$$
$$\xi_i \geq 0 \ \ \forall i$$

The slack variables are incorporated into the objective function to be minimized by simply adding to it the total error multiplied by an error penalty C chosen by the user, given by

$$C\left(\sum_i \xi_i\right)^k$$

When k = 1 none of the slack variables are introduced into the objective function and the dual formulation remains the same. The only difference is the addition of the constraint that α_i be less than or equal to C. The primal Lagrangian needs to be revised however in order to obtain revised KKT conditions which include the slack variables:

$$L_P \equiv \frac{1}{2}\|\boldsymbol{w}\|^2 + C\sum_i \xi_i - \sum_i \alpha_i\{y_i(\boldsymbol{x}_i \cdot \boldsymbol{w} + b) - 1 + \xi_i\} - \sum_i \mu_i \xi_i$$

From this the KKT conditions for calculating b are derived:

$$\alpha_i\{y_i(\boldsymbol{x}_i \cdot \boldsymbol{w} + b) - 1 + \xi_i\} = 0$$
$$\mu_i \xi_i = 0$$

3.1.2.4 Kernel Mapping for Non Linear Decision Boundaries

SVM is a linear classifier, but it is able to form non-linear boundaries by mapping the input data in Euclidean space into a feature space H using a mapping for each instance vector, $\varphi(\mathbf{x})$: $X^n \rightarrow H$. The separating hyperplane is found in feature space, defined by $\mathbf{w}$, the normal to the hyperplane within feature space, which when mapped back to input space, forms a non-linear boundary. As instances appear only as dot products of the form $\mathbf{x}_i.\mathbf{x}_j$ - or $\varphi(x_i).\varphi(x_j)$ if the mapping is applied - for certain classes of functions known as Mercer Kernels it is possible to replace the dot product with a kernel function $k(\mathbf{x}_i,\mathbf{x}_j)$ everywhere in the training algorithm. In the case of the decision function $f(\mathbf{x})$, in which the dot product is made between each support vector and the new instance, this is replaced by $k(\mathbf{s}_i, \mathbf{x})$, so here too, the instance vector is mapped to feature space before being tested as to which side of the hyperplane it lies. In the case of Mercer Kernels, an efficiency is achieved in using a kernel because a simpler calculation can be used to compute the result of the dot product and mapping of the vectors. The mapping function φ is thus implicit. In defining a kernel it is important therefore to ensure that for a given kernel formula to compute the dot product result, that it is a dot product in some space. Whether or not a kernel is allowable in this sense is determined by Mercer's condition. This states that there is a mapping φ for a defined function K if and only if for any $g(\mathbf{x})$ such that:

$$\int g(\boldsymbol{x})^2 d\boldsymbol{x} \text{ is finite}$$

then

$$\int K(\boldsymbol{x}, y) g(\boldsymbol{x}) g(y) d\boldsymbol{x} dy \geq 0$$

In some cases it may not be easy to check whether this condition holds.

This assumes that the kernel function is defined without explicit regard for the mapping φ. It is possible however to start with φ, and from this derive a formula for K, although this approach is unusual.

Many kernels have been defined, some for specific applications, and many in the more general class of kernel methods, which SVMs can take advantage of. There are a few which are however more commonly used with SVM which are listed beneath:

- Polynomial: $K(\boldsymbol{x}, \boldsymbol{y}) = (\boldsymbol{x} \cdot \boldsymbol{y} + 1)^p$
- Gaussian Radial Basis Function (RBF): $K(\boldsymbol{x}, \boldsymbol{y}) = e^{-\|\boldsymbol{x}-\boldsymbol{y}\|^2/2\sigma^2}$

- Hyperbolic tangent: $K(\boldsymbol{x}, \boldsymbol{y}) = \tanh(\kappa \boldsymbol{x} \cdot \boldsymbol{y} = \delta)$

3.2 Unsupervised Learning

Unsupervised learning is a more general learning task than supervised learning. In supervised learning the objective is essentially to learn to imitate a mapping in the environment, based on input-output examples of the mapping provided by a teacher. In unsupervised learning, the problem is more open. Given only inputs or instances from the environment, without any predetermined outputs, the objective of the learner is, in general terms, to learn something interesting about them - typically some grouping or association between them. Without the need to supply the learner with predetermined outputs there is no need for a teacher.

More specifically, the objective of the learner is to model relationships between inputs, which often involves partitioning examples into subsets based on similarity, or to model the relationships between components of inputs, such as identifying correlations among variables. To identify these relationships induction is used to generalize from the input facts. The general idea of unsupervised learning as described is illustrated in Figure 16.

As the learner does not have a teacher, in order to learn something interesting about the inputs, learning must be guided through some mechanism. This takes the form of goals and cost functions that are built into the learner. Learning an hypothesis is typically implemented iteratively: producing an output on the current hypothesis, evaluating this in the light of defined goals and cost functions, and adjusting the hypothesis to better satisfy those goals.

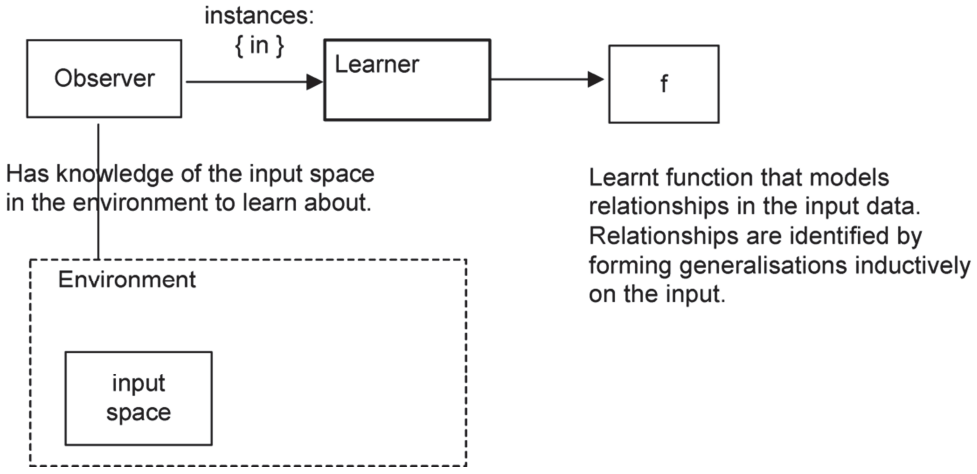


Figure 16 Unsupervised learning in which instances over the input space from the environment are examined by the learner for patterns or relationships that might be useful.

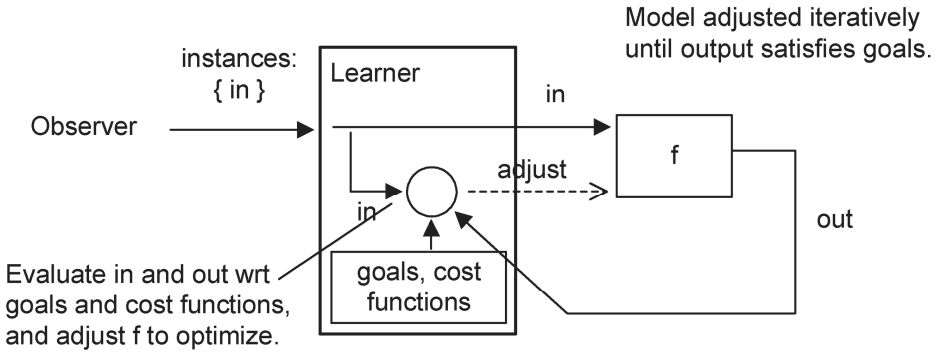


Figure 17 Typical implementation of unsupervised learning.

This typical implementation is illustrated in Figure 17. It will be noticed that there is some similarity between this and Figure 9 for supervised learning, with the difference being that instead of hypothesis change being guided by teacher supplied output, it is guided by the input goals and cost functions. The latter effectively takes the place of the teacher and the class label. Unsupervised learning can therefore be thought of as involving a teacher that is tightly coupled with the learner, rather than one that is separate and environment specific as in supervised learning. Because of this view that there is an implicit teacher involved, an unsupervised learning system is sometimes instead referred to as a self-organizing system.

There are a variety of tasks to which unsupervised learning can be applied including finding clusters in data, reducing data dimensionality, and modelling the density of the data. The most common application lies in clustering, in which input instances are grouped into categories based on similarity. For the purpose of software quality estimation, recent studies have demonstrated that unsupervised learning can be usefully applied and may be particularly useful in that it does not require instances with quality labels.

A commonly used unsupervised learning algorithm is k-means. While this method was used in some early investigations in this research, it did not prove to be useful, and so is not described here.

3.3 Performance Evaluation

An important element of machine learning in the context of this research is evaluation of model performance. The reason for this is that the models are to be applied to unseen data, so generalisation performance is critical rather than empirical. More to the point, that classes are

imbalanced means that the standard measures of performance which assume balance are of limited use.

Evaluation involves testing the learnt model on instances with known labels by which the predictions can be compared and evaluation made. The simplest approach in which this may be done is to use instances from the training set, in which case typically the whole data set would be used for training. This approach is sometimes referred to as training set evaluation. The problem with it is that performance over the training set is usually an overly optimistic gauge of how the model will perform on instances it was not trained on. This is particularly so the less well the training set sample is representative of the population from which the sample was drawn, and the more inclined the learning algorithm is to overfitting the data.

An improvement on this method of evaluation is to partition the data into separate train and test sets. The test set is usually specified as a proportion of the whole, e.g. 10%. This gives a better estimate of performance on new instances, since the test set provides an effective trial run on them. However, model performance can vary considerably depending on the train and test set samples, and hence does not provide a reliable means of determining and reporting performance.

Because of these shortcomings, the method often used to evaluate performance is N-way cross-fold evaluation. This is simply a sampling method which involves taking N different train and test sets, training and testing on each, and taking the average of performance across test sets. It is important though that the samples are taken such that train and test sets do not overlap, and test sets include eventually after multiple samples, all of the data set. The specific method used is to partition the data into N subsets, and from them form N different train and test set pairs, the training set being composed of N-1 buckets and the test set the remaining bucket. For better results this may be repeated M times, which is referred to as M×N cross validation. This overcomes any biases due to the partitions formed in a single N-way partitioning.

The above refers to the sampling method used for evaluation and the method of aggregation of performances, which is by taking their mean. Another aspect of it is the specific measures taken on the predictions of each test set. These are essentially derived by comparing test instance predictions with their actual labels. The most basic measures so derived are collectively referred to as the confusion matrix, which counts True Positives (predicted positive and actual positive), True Negatives (predicted negative and actual negative), False Positives (predictive positive and actual negative), and False Negatives (predicted negative

and actual positive). These numbers vary according to the size of the data set and thus may not provide a useful measure for comparison of model performance which is a frequent requirement. Other measures are therefore typically used instead which combine them in some way.

Often used instead are normalised "rate" versions which provide proportions of interest based on the counts: accuracy, error rate, TP rate, TN rate, FP rate and FN rate. Accuracy is the proportion of correctly classified instances of all instances:

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

This can be a useful overall measure in cases where classes are balanced. But otherwise, when there is imbalance (as with software fault data in which faults are the exception), it can be fairly useless. If the proportion of positives is small, a high accuracy can be achieved simply by classifying all instances negative. Given that the class of interest for which accurate predictions are desirable is the positive one, it can be seen why this measure is unsatisfactory. It effectively provides a measure of performance on the negatives to the extent that negatives predominate. For this reason accuracy is not used in this research. Error rate is the 1-accuracy, or the proportion of instances misclassified.

Of the TP, TN, FP and FN rate measures, two of these used in this research are the TP and FP rate. The TP rate, also called "recall" in the information retrieval domain, is a measure of predicted positives relative to actual positives. It is the proportion of correctly predicted positives to all actual positives:

$$TP\ Rate = \frac{TP}{TP + FN}$$

Because of this it is has also been referred to as the detection rate, as the rate at which actual positives are correctly predicted. The FP rate is similar in that it considers predicted positives, but it is in relation to actual negatives. It is the proportion of incorrectly predicted positives (i.e. negatives predicted as positive) to all actual negatives:

$$FP\ Rate = \frac{FP}{FP + TN}$$

This has been referred to as the false alarm rate, as the rate of actual negatives incorrectly classified positive. The two rates are illustrated in Figure 18. Together they provide a useful means of evaluating performance, particularly in respect to quality models, where these rates are of particular interest. Desired is a high detection rate to identify most modules which are

fault prone, and a low false alarm rate to avoid unnecessary checking of code that contrary to the prediction is not fault prone. However, it has been pointed out that they may not be adequate when classes are imbalanced (Zhang & Zhang, 2007). The reason for this is to do with the balance of classes among the predicted positives. The TP rate and FP rate may be at desirable levels (the TP rate high, and FP rate relatively low), but because of the substantially larger number of negatives, the FP rate even though low, could result in many more false positives than true positives. In other words the rates do not give an accurate indication of the number of false positives compared to true positives. The adequacy of these rates could thus be questionable particularly if it is considered desirable for a model whose detection number is not far outnumbered by false alarms. The measure it is suggested be used to deal with this shortcoming is precision, which captures the relationship between classes within the set of predicted positives. This is shown in the bottom diagram in Figure 18. Precision is defined as:

$$precision = \frac{TP}{TP + FP}$$

Juggling three measures to evaluate and compare performances however is not ideal. This is especially so when it comes to automatically identifying best performing models which requires some formalised method for their combination to arrive at a single evaluation for each model which can be difficult to determine. Other evaluation methods that combine measures to provide a single performance measure have thus been proposed. The one chosen for this research is the F measure (van Rijsbergen, 1979) which is defined by combining

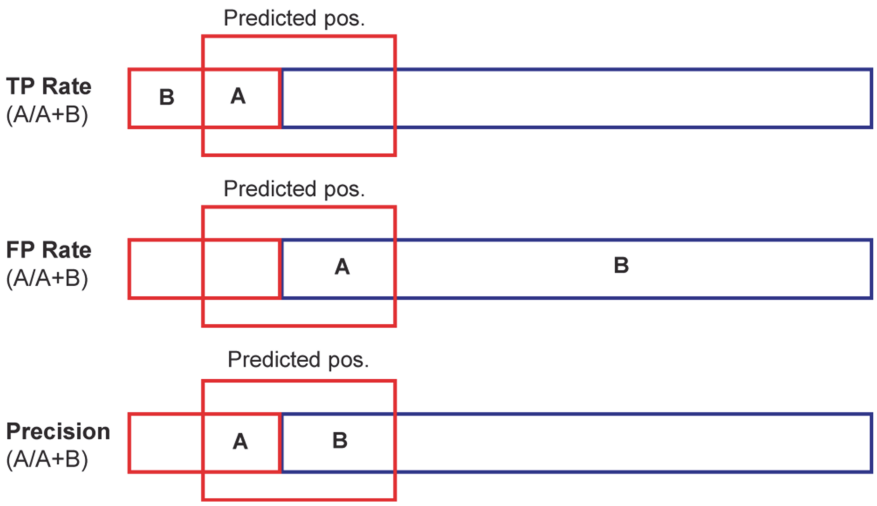


Figure 18 Performance evaluation measures: TP Rate, FP Rate and Precision, which in each case is calculated as A / (A+B).

precision and recall:

$$F\ measure = \frac{(1+\beta^2)recall \times precision}{\beta^2 recall + precision}$$

where β is a parameter which determines the weighting of recall relative to precision. The two are of equal importance when β is 1. Recall is twice as important as precision if β =2. The default is for both to have equal importance in which case the formula simplifies to:

$$F1\ score = \frac{2 \times recall \times precision}{recall + precision}$$

This may be interpreted as the weighted average of recall and precision. By convention the F measure is named with the β value as a suffix. However as the balanced version is most often used, the suffix is usually omitted, and is only included for other values of β. The range of the F measure is 0 to 1, with 1 representing maximal accuracy. It will be noted that by including recall and precision, both of which are measures with respect to the positive class, the F measure is sensitive to the positive class and thus class imbalance. This is another benefit the measure provides apart providing a single summary measure of performance.

Another summary measure not used in this research and apparently not as widely adopted as the F-measure, is the Expected Cost of Misclassification (ECM). It accounts for both the prior probabilities of classes and the costs of misclassification (Khoshgoftaar, Zhong, & Joshi, 2005). It is defined as:

$$ECM = C_I Pr(fp|nfp)\pi_{nfp} + C_{II}\Pr(nfp|fp)\pi_{fp}$$

where C_I and C_{II} are the costs for each type of error, $Pr(fp|nfp)$ is the probability that a module is fault prone given that it has been classified not fault prone and similarly for $\Pr(nfp|fp)$, and π_{nfp} and π_{fp} are the class priors.

For evaluating a model in terms of the trade-off between FP rate and TP rate with different model settings, a ROC curve is often used which plots these two rates along the X and Y axes respectively. The ideal performance is where FP rate is 0 and TP rate is 1, which is top left corner of the plot.

3.4 Discussion

The early part of the chapter focused on some fundamentals of machine learning, which resulted from an effort to find out how machine learning algorithms used related to learning

and more generally intelligence, and the relationship between machine learning and artificial intelligence, as this was not evident from the texts referred to. This lead to the Michalski's model for learning, involving inference, memory and background knowledge, and due to Dietterich, that machine learning algorithms are effectively a cached result of first principles inference which provided the missing connection between learning inference and many of the machine learning algorithms. There were also important elements of learning algorithms mentioned such as representation that describes a hypothesis space over which learning involves a search to find the hypothesis that best describes the data, and various biases which assist in making an inductive leap to arrive at a particular hypothesis. The taxonomy of methods for learning was of interest to put those methods commonly used, supervised and unsupervised learning in context of various approaches.

With this background, the two supervised learning methods focused on in this chapter were described in some detail. Naive Bayes as mentioned has been found to perform well in many applications, often performing better than more sophisticated algorithms. On the NASA data Menzies has reported that Naive Bayes performed better than other algorithms he tried including the decision tree on average, and also that in another study in which nearly two dozen different algorithms were tried, Naive Bayes tied first place along with 14 other methods. He was of the view that there is "brittleness" in the data, in that slight changes in the data make different attributes appear more useful, and that the polling method of multiple Gaussians across attributes as occurs in Naive Bayes helps smooth over the brittleness problem.

The support vector machine algorithm seems to be regarded as one of the better learning algorithms, it seemingly being popularly adopted in recent years for many applications. Perhaps the most significant reason for this is the statistical theory which underlies it, that seeks to find an optimal balance between model complexity and empirical risk which enables it to generalise well to new data which is clearly important when it comes to practical application of developed models. Another related quality is its robustness to noise, which it has been reported exists in the NASA data at considerable levels. It is for these reasons that this algorithm, as well as Naive Bayes, has been employed to develop quality models.

Chapter 4 - Feature Selection

Common to most machine learning algorithms both supervised and unsupervised is a data set comprised of instances, **x,** each of which is composed of a set of feature values $\{x_1, x_2, x_3,..\}$, on which the learning algorithm is trained. Typically these values measure different aspects of some entity, in the present research software modules, and are thought to potentially be related to the target concept, thus providing information relevant to the function to be learnt. Prior to training, it is usual practice to select a reduced set of features from the original set. The task of which features to include is called feature selection. The aim of feature selection is to select features such that they are optimal with regard to some evaluation criterion, commonly predictive accuracy or minimal use of input features. This chapter begins by discussing the reasons why feature selection is necessary and given that it is, what exactly is meant by an optimal subset, and by which there is some sensible rationale for choosing features. Two types of algorithms that are commonly used to select features are then discussed, filter and wrapper. A number of algorithms themselves are then described in some detail to provide an understanding of them as a basis for their use.

4.1 Motivation

From a purely theoretical standpoint, feature selection is not of much interest (Kohavi & John, 1997). Assuming the full distribution is known, Bayes rule gives the highest possible accuracy. However in practice, the underlying distribution is not known, and the algorithmic computation involved in finding the optimal target hypothesis is usually intractable necessitating reliance on approximation. Thus in practice, an optimal feature set must be considered with respect to a particular induction algorithm and its heuristics and biases.

The more practical motivation for feature selection derives from the effects of the so called curse of dimensionality (Cunningham, 2007). Larger numbers of features tends to lead to degradation in performance. One explanation for this is that the more features that are used to describe instances, the more similar they look on average. More features also tends to cause an algorithm to overfit the data – the essential structure of the data is missed for the excess of information included in additional features. By reducing the dimensionality of the data performance accuracy can be improved. There are other benefits as well such lower computation costs and more interpretable results. Sometimes the purpose in selection is more

reduction in dimensionality without loss of accuracy for the sake of algorithm efficiency rather than performance improvement.

4.2 Optimality

In order to find an optimal feature set, what is optimal must first be defined. Conceptually, features can be described as relevant or irrelevant (Kohavi & John, 1997). A relevant feature is one that affects the underlying structure of the data in some way. Not all relevant features however are necessarily optimal. A feature may relate to the underlying structure, but the contribution it makes may be not different to that of another feature, in which case the feature is said to be redundant. Relevance has also been described in terms of the Bayes classifier, the optimal classifier for a given problem (Kohavi & John, 1997). Actually a number of definitions have been proposed of which this is just one. Relevance is split into two levels, strong and weak. A strongly relevant feature is one whose removal from the data would result in performance deterioration of the Optimal Bayes classifier (see section

3.1.1.3 Bayes Optimal Classifier). This implies that the feature is indispensable in the sense that it cannot be removed without loss of prediction accuracy. A weakly relevant feature is one that is not strong, but whose inclusion in some subset of features provides for better performance than without its inclusion. Any other feature, that is not strongly or weakly relevant, is irrelevant.

Having covered the different types of features in relation to optimality, an optimal set of features can be defined in general as one comprised mainly of strongly relevant features, and possibly some weakly relevant features, but without irrelevant features. Not all strongly relevant features should necessarily be included as some may be redundant. In the presence of noise for example, the inclusion of irrelevant features may improve class separation.

In terms of supervised learning, the problem of finding the optimal feature set is rather well posed (Cunningham, 2007). The objective is typically to find features that are correlated with or predictive of the class label. More comprehensively, it is to select features that will construct the most accurate classifier.

4.3 Filter and Wrapper Approaches

Moving on to algorithms for finding optimal feature subsets, there are two types, filter and wrapper (Blum & Langley, 1997; Guyon & Elisseeff, 2003), differentiated according to

whether they use an inductive algorithm (there are also embedded methods which are embedding in the learning algorithm itself, but these seem less common).

In the filter approach, features are selected prior to induction. It is a filter in the sense that undesirable features are filtered out before learning begins. The filter method can be described as involving a search through feature subset space, where for each subset arrived at, a heuristic function is used to evaluate its merit based on general characteristics of the data. The simplest filtering scheme is ranking in which each feature is evaluated individually based on its correlation with the target function, using a measure such as information gain. A subset is then selected by taking some number of features from the top of the ranking.

In the wrapper approach, a similar search is conducted, with the difference being that the evaluation function evaluates a subset using an induction algorithm, usually the same algorithm as the one used to develop the predictive model after feature selection. In order for the evaluation made on the subset using the induction algorithm to be sound, taking into account performance on unseen data, cross validation is applied, in which the average performance is taken on folds of the data of the given selection. The goals of these approaches are different. In the filter case, the goal is often to identify relevant attributes, while for the wrapper approach, it is to maximise classification accuracy on unseen data. Different categories of evaluation functions for both approaches have been identified as being based on distance, information or uncertainty, dependence, consistency and classifier error rate (as used by wrapper methods) (Dash, Liu, & Motoda, 2000).

Whether to use filter or wrapper selection methods depends on the modelling problem. The filter approach has the advantage of speed, which may be important if the number of the features in the data set is large. The features are also generic, not aligned to any particular algorithm bias, and can thus be used with any learning algorithm, with the expectation being that since the structure of the data has been captured, performance should be satisfactory regardless of learning algorithm. It may be to that generic selection will produce simpler models. However if the objective is for optimal predictive performance then the wrapper approach is recommended (John, Kohavi, & Pfleger, 1994; Foreman, Guyon, & Elisseeff, 2003), depending on the number of features in the data as this approach is more computationally expensive involving the application of an inductive algorithm for every feature subset evaluated. Mitigating the computation problem to some extent are efficient search strategies that can be used such as greedy search. These can also help to avoid overfitting and provide better performance.

4.4 Feature Ranking

Feature ranking methods as a special case of filter selection seem recommended only as an initial step for removing irrelevant attributes, after which filter or wrapper subset methods can be used to eliminate redundancy to some extent (Guyon, 2008). It has been mentioned that a potential shortcoming with these methods when correlation is used as a basis for ranking, is that only linear dependence is captured. A feature with a correlation of zero may therefore still be relevant if there is some non-linear dependence. Taking the log or square root of the data can reduce the impact of any nonlinearities. Other ranking measure such as signal to noise ratio (described later in section 4.6.6 Signal-to-Noise) already do this.

While ranking methods are useful for removing irrelevant features, there have been criticisms made of them. The main shortcoming is that features are only considered individually, which can lead to the exclusion of features which are relevant in complement with other features and thus potentially beneficial. Only if the features are truly independent can the ranking method be reliable in arriving at an optimal feature set, an assumption which in practice for many data sets is often violated to some extent. Other arguments for the need to consider variable complimentarity through the use of subset rather than ranking methods are that adding redundant features can improve performance, high correlation and by which presumably a feature would be considered redundant does not mean the absence of complimentarity, a feature that is useless by itself can provide a significant performance improvement when taken with others, and two features that are useless by themselves can be useful together (Guyon & Elisseeff, 2003). Some ranking criteria however can take into account context using the notion of conditional dependency (Guyon, 2008). The Relief ranking algorithm in sampling nearest neighbours can also handle dependency and has been shown to be useful in practice (described later in section 4.6.4 Relief).

4.5 Search

An aspect of feature selection in subset methods is the type of search used. The search is through the space of possible feature subsets or states, each one being evaluated to find the one with highest merit. Different search methods cover the space in different ways as it is impractical usually to look at all possible subsets. These search methods are common to both filter and wrapper selection approaches. A search method has a starting point, which is either an empty set of features, or the full set, the former case being called forward selection, and the latter, backward elimination. The starting point affects the direction of search and the

operators used to generate successor states. Forward selection adds features that are relevant given previously selected variables, while backward selection removes features which are irrelevant given the remaining selected features. Organisation of the search is usually not exhaustive, with other methods such as greedy search being used instead. In greedy search, local changes are considered at each point in the search. An example is stepwise selection or elimination in which features are added or removed at each point, making it possible to retract earlier decisions but without keeping track of an explicit search path. Greedy schemes choose the next state by either selecting the best from all next states generated by the operators, or simply the first state that improves accuracy over the current set. Greedy search or closely related methods are recommended for the wrapper approach (unless a lot of data is available) as they tend to avoid overfitting, but for the filter approach it is suboptimal as once a feature has been added or removed its inclusion in the selected feature set is not questioned again (Guyon, 2008). There are other schemes which are more expensive computationally such as best first search, but which are still often tractable. Other elements of the search method are evaluation method of feature subsets, already discussed (usually ability to discriminate among classes or learning algorithm performance), and the criterion for termination.

One of the choices to be made in relation to search method is whether to use forward or backward selection. Backward selection is the more computationally intensive method and which has the potential to find better feature subsets as it uses all of the features not eliminated so far, making it more capable of finding complimentarity between features. However, it suffers from catastrophic performance degradation of subsets when they become small (Guyon, 2008). Thus it is not the most suitable for finding small subsets that are well performing. On the other hand, forward selection is faster and scales well to larger data sets, and even the smallest subsets are usually predictive.

4.6 Filter Ranking Methods

A few of the more common filter ranking methods are described below in detail.

4.6.1 Information Gain

Information gain (IG) is a measure of the amount of information a feature brings to a training set. Amount of information is measured in terms of the reduction of entropy or uncertainty involved in determining the class of an instance, from that if only the class priors are known, given the entropy of the class over the partitioned values of the feature. It has become one of

the most popular measures for feature selection in recent years (Guyon & Elisseeff, 2003). As for most heuristic methods for assessing feature quality, attribute independence is assumed, and to the extent this is violated, performance of selections derived using this method may be lacking. A more detailed discussion of the measure follows, with an initial focus on entropy and how it is used to evaluate the worth of an attribute in the context of class prediction.

Entropy is the minimum number of bits required to encode a stream of values, given the prior probabilities of each value, or knowing that they occur according to some probability distribution. In relation to feature evaluation, entropy may be considered in the context of a predictive model where the output of the model is a stream. Given only the class variable Y and its priors with which to make predictions, it is not possible to accurately predict the output. To correct mistakes in the stream so that it contains only correct predictions, entropy gives how many bits are required to encode predictions based on the priors. If a feature X is known, it may provide additional information that can help in predicting class on the stream. It will then be possible to encode the predictions accurately with less bits. In other words, the entropy of Y given X is less than the entropy of Y. This difference is what information gain measures. It measures how much information has been gained about Y, from variable X, in terms of the number of fewer bits required to correctly encode model output.

Entropy, which measures minimum bits required to correctly encode the stream, is defined by:

$$h = \sum p \log p$$

where p is the prior probability of each output or class on the stream. For the binary case this is ($p_N \log p_N$) + ($p_P \log p_P$), where p_N and p_P are the probabilities for each class, and the range is 0..1. The maximum of 1 occurs when priors are both 0.5.

Information gain is the reduction in entropy or bits required to encode the prediction accurately given that X is known:

$$IG = H(Y) - H(Y|X)$$

The latter term is calculated as the expected entropy for Y over each of the values v of X:

$$H(Y|X) = \sum p(v)H(Y|X = v)$$

This method for calculating information gain is illustrated in Figure 19 for the binary case. In order to compute conditional entropy over values, the data must be discrete, and if continuous, first discretised. A problem with information gain is that it tends to be larger for features with

more values, although this may be less of a problem if the data is discretised depending on the partitioning bias of the discretisation method. One solution to this is the symmetrical uncertainty coefficient, which compensates for that bias, and as well normalises its values into the range [0, 1]:

$$symmetrical\ uncertainty\ coefficient = 2 \times \left(\frac{IG}{H(Y) + H(X)}\right)$$

Another solution is the Gain Ratio measure described in the next section.

4.6.2 Gain Ratio

This measure, which is similar to information gain as the value of an attribute with respect to the class, was proposed by Quinlan (1986). Information gain was being used to determine which of the remaining features would serve to best partition the data in the next decision node of a decision tree. However, information gain was not producing trees with optimal structures due to its bias towards features with larger numbers of values. The reason for the bias can be seen by considering the case where every value in a feature is unique. This results

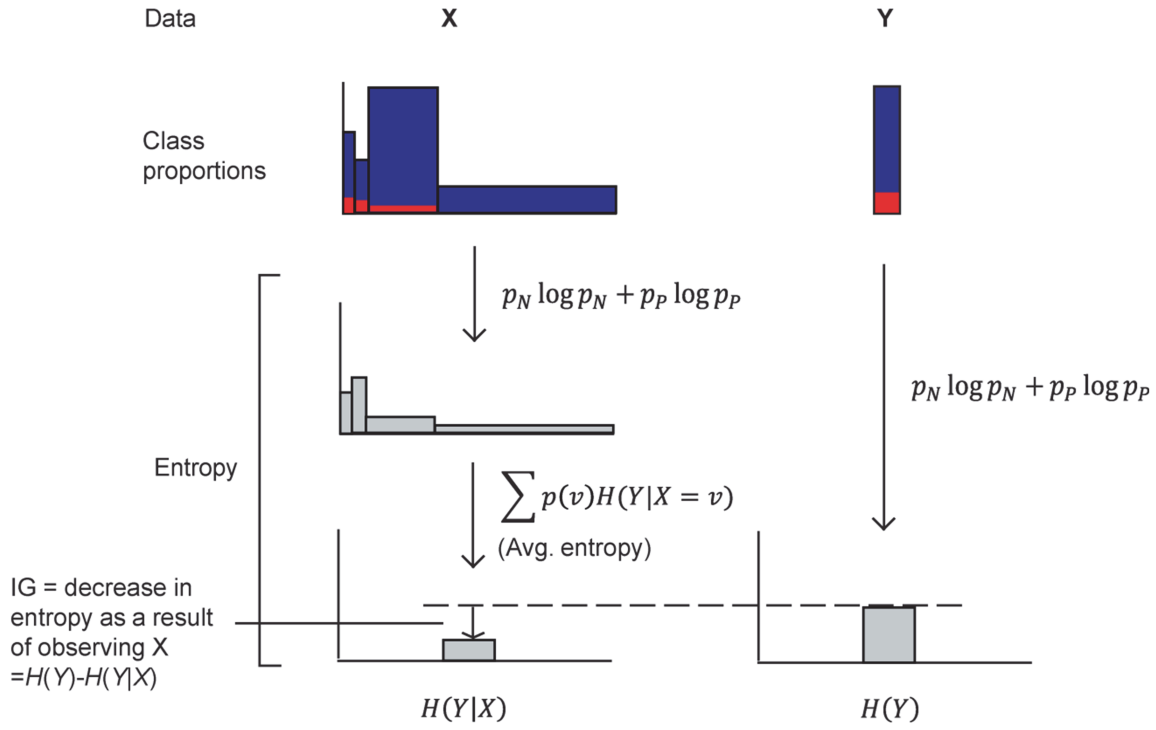


Figure 19 Information gain is the improvement (decrease) in entropy on the class that an attribute X provides (averaging over the entropy for each value of the attribute) over the entropy on the class labels Y.

in pure partitions, containing only one value of a single class, giving the highest possible entropy of 1 for each value. The problem is also evident in a less extreme case of a single value being split into two values to create a new feature, with instances retaining their class labels. The new feature would have higher information gain than the original feature (or equal gain if the proportions of the classes were the same for both). A solution was proposed in which IG was normalised to account for partitioning of the data across feature values – for example to penalize gain if there were a large number of small partitions as in the first example. A measure of information value was defined (also called intrinsic information), computed as the entropy of the splits across values of the feature:

$$info\ value = \sum H\left(\frac{\#v}{\#instances}\right)$$

This is may be interpreted as the information needed to tell which branch an instance belongs to. The measure of gain ratio is obtained by dividing information gain by information value: IG / IV. While this is effective in reducing the bias of information gain, sometimes the information value can overcompensate if it is very low.

4.6.3 Chi Squared

The chi squared measure, as for information gain and gain ratio, evaluates the worth of a feature with respect to the class, but does so using the chi squared statistic. As for the other measures, each value of the feature is considered, but computation over the values differs. A feature is considered to have no value with respect to the class if for every value of the feature, the class distribution (that is the count of instances for each class) matches the class priors. This can be stated more precisely by defining count terms. For a given value *i* of the values of the feature, counts for the positive class can be defined as $\underline{ni}+$ for the number of positives and μi+ for the expected number of positives which is *prior*+ × *ni* where *ni* is the number of instances with value *i*, and similarly for the negative class. A feature is considered to have no information value if for each of its values *i*, *ni*+ = *μi*+ and *ni*- = μi-.

To the extent that there is difference between these counts, between actual and expected distributions, which is *ni*+ - *μi*+ for the positive class, and *ni*- - *μi*- for the negative, the feature is considered to have information value. Chi squared sums the sum of these differences (for each class) across feature values i, to obtain a total:

$$\chi^2(D,c,f) = \sum_{i=1}^{r}\left(\frac{(n_{i+}-\mu_{i+})^2}{\mu_{i+}} + \frac{(n_{i-}-\mu_{i-})^2}{\mu_{i-}}\right)$$

where D is the data, c is the class and f is the feature to be evaluated. The squared difference is taken to produce a score that follows the X^2 distribution. In comparison with information gain, while information gain is higher the further the class splits are from 50/50 which gives maximum entropy and lowest gain, chi squared is higher the further the actual value-based class distributions are from the expected distribution.

4.6.4 Relief

One of the shortcomings of many ranking methods is that they consider each feature individually in terms of its relation to class. They do not take into account context or the relationship the feature has with other features. This can be important as a feature that is irrelevant by itself may be relevant in the presence of other features (Guyon, 2008). That is, a feature X may be independent of Y, but conditionally dependent given another feature X1. However, taking into account context information can become computationally impractical as the number of other features considered increases. One method which does manages to do this efficiently and which has been shown to work well in practice for classification problems is Relief (or ReliefF, where the letter F refers to a revision of the original method). The method ranks features using a heuristic based on nearest neighbours. A score is computed for a feature by iterating through each instance, and for each finding the K nearest neighbours of the same and different classes, using all features. The nearest neighbours of the same class, called hits, may be denoted by $\{\boldsymbol{x}_{Hk(i)}\}$ for k=1..K and where i is the current feature index, and similarly for the miss set of points of a different class $\{\boldsymbol{x}_{Mk(i)}\}$. The distance from the current instance $\boldsymbol{x}_i$ to each miss point in dimension j, $|x_{i,j} - x_{Mk(i),j}|$ for each k, is added to the score, and to each hit point, $|x_{i,j} - x_{Hk(i),j}|$ for each k, subtracted from it. This computation for a feature j, and given M instances, may be expressed as (based on (Guyon, 2008) but with subtraction of $M(j)$ and $H(j)$ as in the original version of Relief rather than taking the ratio):

$$Relief(j) = M(j) - H(j) = \sum_{i=1}^{M}\sum_{k=1}^{K}|x_{i,j} - x_{Mk(i),j}| - \sum_{i=1}^{M}\sum_{k=1}^{K}|x_{i,j} - x_{Hk(i),j}|$$

If a feature discriminates well between the classes, then the distances to the nearest instances of a different class will be larger than distances to those of the same class, resulting in a

higher relief score. If a feature is useful in predicting class, then instances should generally be amongst instances of the same class and further from those of different classes. In probabilistic terms, Relief estimates the following probability for feature j (based on (Robnik-Sikonja & Kononenko, 1997)):

$$weight(j) = P\left(x_{i,j} \neq x_{q,j} \middle| x_q \in M\right) - P\left(x_{i,j} \neq x_{q,j} \middle| x_q \in H\right)$$

It is desirable that the probability of the first term be high, and the second, low.

Apart from the Relief method taking into account dependencies between features (and not assuming their independence like many other ranking methods), it is tolerant of noise because it considers K nearest neighbours. Using the nearest neighbour approach also makes it well suited to nearest neighbour based classification algorithms. A criticism that can be made of it is that it selects all relevant features (Kohavi & John, 1997). It does not select a useful subset of weakly relevant features. As such it does not remove redundant features, which can be detrimental to the performance of some algorithms such as Naive Bayes, and may result in larger feature selections.

4.6.5 One R

The One R feature selection method is one of the most simple, the One R standing for One Rule. For a given feature, a classification rule is created for each value of the feature. The rule for each value is to predict the majority class of training instances with that value. The set of rules thus created is used to classify all instances from which an error rate is obtained, which is the sum of misclassifications for each value. The error rate is the score assigned to the feature and by which features are ranked. An assumption made is that a single feature is sufficient for classification. While this assumption often would not hold, this method in a limited way is able to evaluate the usefulness of a feature in being predictive of class, it can be used to produce benchmark performances by which to compare performances of other learning algorithms, and it has been found to perform well with the decision tree algorithm C4.5 compared with other classification methods (Witten & Frank, 2000).

4.6.6 Signal-to-Noise

Another simple method for ranking features is the signal-to-noise ratio, proposed for feature selection in gene data in which features are typically in large number (Golub, et al., 1999; Liu

& Wong, 2003). The ratio considers the mean of each class with respect to their standard deviations. The ratio is:

$$w_i = \frac{\mu_1 - \mu_2}{\sigma_1 + \sigma_2}$$

where μ_1 and μ_2 are the means of the classes, and σ_1 and σ_2 the standard deviations, from the log of the data. A feature is considered to have greater signal to noise if the means of each class are far apart and the standard deviations are smaller. As the means become closer or the standard deviations spread, the signal reduces. For the purpose of ranking, the absolute value of the result is taken. Otherwise the sign of the result indicates to which class the feature is more strongly correlated. This may be used to select features with correlations to both classes.

There are other measures similar to the signal to noise measure. Fisher's ratio is similar except that the difference of the means is squared and variance is used instead of standard deviation. Another is the "independent significance measure" (Weiss & Indurkhya, 1988), which differs in that the variances in the denominator are each divided by the respective number of instances in each class, and the square root is taken of their sum. It is recommended that only features with values above 2 be kept, a threshold with which the risk of removing useful features is said to be low.

4.7 Filter Subset Methods

4.7.1 Consistency

The consistency feature selection method evaluates subsets according to a measure of their 'consistency'. Importantly, it considers subset size, with strong preference for smaller subsets. The central idea of this method is that while often the criterion for selection is maximisation of class separability, in this method it is instead to retain the discriminating power of the data defined by the original features, or in other words, to find the smallest set of features than can distinguish classes as if with the full set (Dash, Liu, & Motoda, 2000). This is achieved by reducing feature set size while maintaining consistency. Unlike most of the ranking methods, the consistency subset method helps in removing redundant features (as well as irrelevant).

The term consistency of a subset is basically formalised as the degree to which no two examples with the same values have different class labels. It is assumed that the data is

discrete in order that this evaluation can be made. More specifically, consistency of a subset is measured by an inconsistency rate. This rate would not seem dissimilar to the error rate computed in the One R ranking method, except that in this case it is computed over a subset rather than a single feature. The rate is calculated by first identifying groups of examples with matching values, without regard to class label. For each group an inconsistency count is calculated as the number of examples in the group minus the number of examples of the majority class within the group. An inconsistency rate is obtained by summing the inconsistency counts across groups and dividing by the total number of examples in the data set. The consistency algorithm maintains a best set through the subset search space, initialized to the full set of attributes. If any subset visited is larger in size than the best set it is not considered any further – hence the bias to finding smaller subsets. Otherwise, the inconsistency rate is calculated on the subset, and if it is found to be smaller than that of the current best subset, the best subset is replaced. A variation of this is to define a preset maximum tolerance for the inconsistency rate, allowing a subset to be replaced as the best one so far if its inconsistency rate is less than this tolerance, rather than having to be less than the rate of the existing best set. This would mean giving less importance to minimising inconsistency and more to smaller subset size, given the inconsistency rate is tolerable.

4.7.2 Correlation

The Correlation-based Feature Selection method (CFS) is based on the view that a useful feature is one whose values vary systematically with class membership. This relationship between feature and class can be measured through correlation. In addition, a feature is useful only if it is not redundant (generally the case), which is to say not correlated with another feature. This ensures that each selected feature is predictive of the class in areas of the instance space not already predicted by other features. Satisfying these two requirements, a subset is evaluated using the following formula:

$$G_s = \frac{k\overline{r_{cl}}}{\sqrt{k + k(k-1)\overline{r_{ll'}}}}$$

where G_s is the heuristic merit of the subset S with k features, $\overline{r_{cl}}$ is the average correlation between each feature and the class, $\overline{r_{ll'}}$ is the average correlation between features. The formula is based on Pearson's correlation coefficient, but with standardized variables. Merit is higher the greater the feature-class correlation (relevance) in the numerator, and the lower the feature-feature correlation (redundancy) in the denominator. Higher k increases the merit,

but usually when this is the case there will be redundant features for which a higher value of $\overline{r_{ii'}}$ will compensate. Different measures can be used to measure correlation between features and feature and class. If two features are continuous, Pearson's correlation coefficient can be used. In this case it is assumed that features are normally distributed and independent of each other. In fact, regardless of correlation method used, it is assumed that features are independent given the class, and while this can be moderately violated, if there are strong dependencies relevant features may be omitted. Other correlation measures that could be used include symmetrical uncertainty (a normalised version of information gain), and minimum description length. Usually the data is discretised first to have a common basis for computing correlations.

4.7.3 Fast Correlation

The Fast Correlation Based Feature selection method (FCBF) was developed as a more efficient means of correlation based selection for high dimensional data (Yu & Liu, 2003; Koller & Sahami, 1996; Yu & Liu, 2004). Like CFS, a useful feature is considered to be one that is relevant to the class concept and is not redundant to any other features. It also uses correlation as a goodness measure, but selection is made based on the concept of Markov blankets. For measuring correlation, an entropy based measure, symmetrical uncertainty, is used rather than Pearson's correlation because entropy does not assume correlation is linear. Symmetrical uncertainty is essentially information gain, and this can be used as a correlation measure as for a feature X to provide information gain about another feature Y there must be some correlation between the two features. Entropy does however assume that features are either nominal or discrete. Two types of correlation are defined: C-correlation which is the correlation between feature and class, denoted by $SU_{i,c}$, and F-correlation, the correlation between a pair of features F_i and F_j, denoted by $SU_{i,j}$. The selection algorithm begins by first removing irrelevant attributes, which is done by filtering out features whose C-correlation is less than some threshold. The threshold can be adapted to get the expected number of features. The second step selects "predominant" features by which redundant features are removed. Features F are ranked by C-correlation. For each feature F_j starting from the top of the ranked list, all features below it, F_i, for which F_j forms a Markov blanket are removed. Formally a Markov blanket M exists when $P(F\text{-}M\text{-}F_i,C|F_i,M) = P(F\text{-}M\text{-}F_i,C|M)$. F_i is omitted in the conditional of the second term, meaning that the feature M (which is F_j) subsumes the information F_i has about about C and the other features, $F\text{-}M\text{-}F_i$. In other words, if a Markov

blanket exists for F_i it is redundant. To determine whether F_j forms a Markov blanket for F_i, an approximation is made. F_j forms an approximate Markov blanket iff it is a better predictor of the class Y than F_i ($SU_{jc} >= SU_{ic}$) , and if F_i is more similar to F_j than to Y ($SU_{ij} >= SU_{ic}$). Any feature F_i that does not have an approximate Markov blanket is called "predominant", and what remains of this procedure is a set of predominant features, which approximates an optimal subset.

4.8 Discussion

The purpose of this investigation on feature selection has been to acquire better understanding of it so that algorithms are not simply applied blindly and informed choices may be made in finding optimal feature selections. Feature selection is an important part of the process of model development, and if it is omitted or mishandled, model performance may suffer.

The task of feature selection has been formalised as that of finding an optimal feature set and this tends to be thought of in terms of the subset that will give best classification performance. While this is a common goal particularly when wrapper methods are used where subset evaluation is based on classifier performance, it would seem helpful to consider the problem of finding an optimal selection more generally in terms of excluding irrelevant variables and including some subset of features that are strongly and weakly relevant in terms of defining the underlying structure of the data while avoiding redundancy. The automatic selection methods described are capable of finding better selections than by non-automatic means and they are usually faster. However, except for the wrapper method, it is apparent that finding optimal subsets heavily relies on heuristics and approximation, and to the extent that this is so, selections may be somewhat less than optimal, particularly in respect to a given classifier.

Of the two recognized approaches to feature selection, filter and wrapper, the former is more efficient, but the latter provides potentially better predictive performance. Because of the better performance, the wrapper approach has been recommended over filter, in cases where this approach is feasible depending on the size of the data and speed of learning method considering that a model has to be created for evaluation of each subset. Where possible therefore, in this research, especially with the objective being to maximise performance, the wrapper approach should be preferred. This would probably be feasible for many of the data sets using Naive Bayes, for which models can be built quickly, but perhaps less so with SVM and for any data sets of high dimensionality or a large number of instances for which computational expense may be prohibitive. Different search methods can be used to reduce

wrapper-based processing time, and as mentioned those which explore only a subset of the search space may give better performance as they are less likely to overfit the data.

A number of feature ranking methods were described. They have been used in many studies, with Information Gain being one of the more popular. They also provide some insight into the problem of selecting useful features. One of their formalised uses in order to find a subset of useful features once the features have been ranked, is to obtain the performance on top-down subsets of increasing size from the rank list using some classification algorithm and then select the top n features according to which performed best. However, there is a significant shortcoming with this approach which is that ranking methods rank only by relevancy and do not take into account redundancy. Any subset of most relevant variables from the top of the ranking list is therefore likely to include some degree of redundancy between features. While algorithms such as SVM may be tolerant of this, others would be less so, such as Naive Bayes due to its assumption of feature independence. Another argument against the ranking approach is that most of the ranking methods, in considering each feature individually with respect to the class, do not take into account complimentarity between features. An exception to this is Relief, and for this reason is perhaps one of the better ranking methods. Perhaps for these reasons it was noted that in a recent study focusing on feature selection for fault detection modelling (Rodriguez, Ruiz, Cuadrado-Gallego, & Aguilar-Ruiz, 2007), that only subset selection methods were used. This may be reflective of a recent shift towards methods that consider subsets of features together facilitated by increasing computer processing power. While this trend may be so for data sets of manageable size, for larger sets of high dimensionality, ranking methods still have utility. It has been suggested for example that they may be used as an efficient means of removing irrelevant features as those at the bottom of the ranking, and possibly as a pre-processing step before applying one of the subset filter methods or the wrapper method. In summary, the overall impression is that ranking methods should only be used if other methods are not computationally feasible, and possibly as a pre-processing step to a subset selection method. However, it could be of interest to use ranking methods, as they are quickly applied, to obtain performance benchmarks by which to compare other non-ranking methods.

In relation to search, for filter methods the more computationally expensive would probably give better performance, including best first and possibly exhaustive, rather than greedy. Backward search would be preferable to forward although more time consuming as it has the potential to find better subsets even though it may result in larger feature sets. For wrapper

methods, as mentioned, non-exhaustive search methods are faster and are less likely to overfit.

Of the ranking methods described, the signal to noise is one of the simpler ones. It was originally proposed to handle the high dimensionality of gene data. That it is a very simple measure is not to say it is without merit. However, it is possible that more sophisticated ranking methods such as Relief and Information Gain would provide better selections. The interesting feature of this particular measure is its sign in being indicative of in which class the feature is more highly expressed, which might be useful in selecting subsets that are not unduly biased towards the majority class.

The most appealing subset filter method described is Hall's CFS because it captures the essence of what is entailed in an optimal feature set which is to include relevant features and exclude irrelevant and relevant redundant ones. One shortcoming that may be identified with this method is the use of Pearson's correlation when features are continuous which only measures linear dependence. The consistency method could easily be included in experiments for comparison, bearing in mind its strong bias towards smaller subsets.

One of the characteristics of the data used in this research is class imbalance, where the number of fault prone modules is much less than the number of modules that are not fault prone. Imbalance has been recognized as a significant issue that can affect the performance of learning algorithms. Similarly it is believed that it can also affect feature selection algorithms, resulting in feature selections that may not be optimal. While this problem is recognized, it has received little research attention, and that has been mostly in the area of text classification or Web categorisation (Guo, Yin, Dong, Yang, & Zhou, 2008).

One novel approach in this area described in (Zheng, Wu, & Srihari, 2004) is to select features for each class separately and then optimally combine them. An argument was made for the inclusion of negative features in order to confidently exclude non relevant classes. In the same study it was found that the rank scores of methods such as IG and Chi Squared for positive features were much higher than those for negative, and thus for the purpose of selection, the scores between features of different classes were thought not to be comparable. This relates to the view expressed that the reason why feature selection is not optimal for imbalanced data is that in treating positive and negative features equally, methods effectively optimise accuracy (as the number of correctly classified positives and negatives as a proportion of the total number of instances) which clearly is not optimal for imbalanced data sets (as high accuracy is achieved by predicting only the majority class), when they should

instead seek to optimize measures which account for the imbalance such as the F measure. Another approach to deal with imbalance at the feature selection level is to resample the data prior to selection (Guo, Yin, Dong, Yang, & Zhou, 2008), such as oversampling the positives, although typically studies only use sampling in the learning phrase rather than feature selection phase.

Against the need to account for imbalanced data is evidence that in some cases learning algorithms have performed well despite imbalance, which would imply that feature selection also was not adversely affected, with any performance issues being attributed to complimentary factors such as class overlap. Also, if wrapper methods are used, and the search is reasonably thorough, the imbalance problem might be dealt with in so far as the features selected provide best discrimination between the classes. It might also be asked if imbalance is such a problem in feature selection, why it has received such little attention. The problem with exploring too much in this area is that there are few existing methods, let alone proven methods, and the benefit performance-wise may be minimal when there could be more significant problems such as high level of redundancy between features or an intrinsic lack of discriminative information in the data. Only a brief foray in this area, if any, may be justified.

SECTION 2 - Data & Data Preparation

Chapter 5 - Sources of Data

This chapter introduces and explores the two sources of data for this research, NASA and Eclipse. The data serves as a source of information with which to train classifiers to discriminate between modules as fault prone and not fault prone. The data is comprised largely of measurements on the source code which are easily collected by metrics collection tools. The other part that is required is information by which labels can be assigned to modules so the data can be used in the training of a supervised learning model. In this research the aspect of quality of interest is faults, so the information must give some indication of this for each module. This sort of information is difficult to come by, as it is valuable for modelling purposes and it is typically proprietary in nature.

The chapter begins covering some aspects of data that have arisen in the literature that are seen as relevant to the current study. Following this, the data sets in each repository, NASA and Eclipse, are introduced. The number of modules, number of features and percentage of faulty modules are significant aspects of each data set that are mentioned. The data sets are then analyzed in various ways to gain a better understanding of them, and to serve as a basis for model development which begins with data preparation in the next chapter.

5.1 Availability

A significant issue in developing quality models is sourcing data from which to train them. More specifically, there is a lack of software engineering data available which contains fault information from which class labels can be derived (Seliya & Khoshgoftaar, 2007). This has hampered the development and deployment of software quality models despite the many machine learning techniques that have come about in recent years. The reason for the lack of data is that software projects are usually proprietary, and even if there were not so much concern about what project information might reveal about standards of software development in a company, the information may be of value for their own modelling purposes. It has been possible however in some studies under confidentiality agreements to have access to large propriety projects, but this uncommon. Among the studies that have had access to such projects, they are few in number and are often reused. More recently publicly available repositories of software engineering data have been established such as the NASA MDP data sets (Metrics Data Program), and the PROMISE repository (Promise Data Repository, 2009). There are also now open source projects such as Eclipse (Eclipse Bug Data) whose code and

bug tracking databases are freely available and which have been mined for information that may be useful. The work in this research has been on the NASA and Eclipse data which will be introduced shortly.

5.2 Class labels

Assuming one has data, an issue which arises in the supervised learning case, is assigning quality labels to instances. Typically fault count is used for this purpose, assigning labels of either fault prone or not fault prone. A threshold on the count is used to choose between the two labels. This threshold varies between studies - some use a threshold of ≥1 and others above that such as 2. It is perhaps surprising that there were no studies found in which fault density was used instead of fault count. The former could be seen to provide more resolution on error and to remove from the error measurement the effect of module size. Also used to determine fault proneness labels, has been change history (Kastro & Bener, 2008; Zimmerman, Nagappan, & Zeller, 2007). A module has been assigned faulty if it had more than 3 lines of code added to it over some period of time or since the last release, a change referred to as debug code churn (Zimmermann, Premraj, & Zeller, 2007). It has been reported that in general, such process measures based on change are better predictors of fault proneness than standard product metrics such as LOC (Graves, Karr, Marron, & Siy, 2000). Interestingly, it was also found that modules more than a year old were found to contain roughly 1/3 fewer faults, so time information if available could be a factor to consider as well in assigning labels.

5.3 Metrics

Training data is comprised of a number of features which are usually measurements on the source code. The number of actual features used varies dramatically, from as few as 2 as in (Shin, Goel, Ratanothayanon, & Paul, 2007) in which only LOC and cyclomatic complexity were used, to hundreds (in the Eclipse data). Apart from variation in number, they may also vary by type. Though usually static code metrics, data may include other types of metrics as well, such as change metrics. In one study the source of data was the OpenBSD project (Li, Herbsleb, & Shaw, 2005) mined from which was not only the code repository but also the request tracking system and mailing list archives. Various metrics types were extracted including development, deployment and usage metrics. The metric found to be the best predictor of fault proneness was one of the more novel ones and perhaps thought less likely to

be the most predictive, the number of messages to the technical discussion mail list. Also used may be object oriented metrics in addition to procedural if the source code is object oriented, which may assist in better characterising modules and in quality prediction. Metrics have also been extracted for the purpose of quality modelling from early lifecycle data such as requirements specifications (Jiang, Cukic, & Menzies, 2007) and UML specifications (Wagner & Jurjens, 2005) sometimes used together with code derived metrics.

5.4 Noise

Also a significant problem apart from paucity of data is quality, the lack of which largely manifests in the form of noise. Noise can make it difficult for a learning algorithm to detect the underlying characteristics of the data, and this can reduce performance and robustness of prediction models. Noise is often thought of as belonging to one of two categories, class noise and attribute noise (Tang & Khoshgoftaar, 2004). In the former case, noise may occur if for example faults have not been discovered (or discovered but not fixed), and as a result of which a module receives a different class label. An example of attribute noise is redundancy, often occurring in software engineering data as many metrics can be easily collected but many of which add little more information than other attributes. Attributes whose values are all close to zero which provide little variation by which to discriminate between the classes may also be considered as a form of attribute noise. This necessitates the need for feature selection discussed earlier and applied in the next chapter. The noise issue has been related to outlier detection, an outlier being an instance whose values differ greatly from those of most of the other instances. However not all outliers are necessarily noisy in that they may be simply be exceptions or non-trivial instances that are properly part of the population from which the data was drawn. Thus noise detection may be considered a subset of outlier detection. An example cited of the existence of noise in data used is this research, NASA data sets jm1 and kc2, is given in (Zhong, Khoshgoftaar, & Seliya, 2004). These two projects are from the same organisation, have the same software metrics collected and used, yet there was a dramatic difference in their misclassification rates. As well it was found in jm1 that roughly 2000 of the 10,000 instances had the same attribute values but different class labels, which is a significant amount of class noise. While the importance of assuring quality in data has been advocated (Khoshgoftaar & Seliya, 2004) there would not appear to have been much research done in this area. One approach to detection of class noise involves the use of k-means clustering (Tang & Khoshgoftaar, 2004). This is applied to instances and those with labels of

the minority class in each cluster are considered to be noisy. These can then be removed from the data set or the labels changed to that of the majority class. The problem with this approach and noise handling in general would seem to be distinguishing between legitimate outliers and noise.

5.5 Imbalance

Apart from lack of data, and noise afflicting the data that is available, perhaps one of the most significant problems in developing fault proneness models is class imbalance. This occurs when one class is significantly more represented in the data than the other e.g., when the class ratio is 100 to 1. In the case of fault proneness modelling, fault prone instances occur much less frequently than those without faults. For example in the NASA data sets, fault prone modules comprise usually less than 15% of the instances. This is in accordance with the software engineering rule of thumb that 80% of faults lie in only 20% of the modules. Class imbalance is found not only in fault proneness modelling, but in other areas as well such as text classification, fraud/intrusion detection, and medical diagnosis/monitoring to name a few (Chawla, Japkowicz, & Kolcz, 2004). The study of imbalance is a relatively new one in machine learning (Chawla, Japkowicz, & Kolcz, 2004). The problem with class imbalance as often reported in the literature is that it tends to degrade model performance. Experimentally, this can easily be found by taking samples of equal size of each class and creating a model from them using a standard learning algorithm, and then gradually increasing the size of one of the samples. Eventually performance will drop significantly. Another way in which the problem can be seen is in considering how a nearest neighbour learner would behave in this situation. As the majority class predominates and any positives become crowded with negatives, the nearest neighbours of a test instance are more likely to be of the majority class, resulting eventually in only majority class prediction. This leads to the perspective of accuracy and the objective function of the learner. The objective function typically aims to maximize accuracy, and this can be achieved by simply predicting all instances as being of the majority class. Exacerbating the problem of class imbalance is noise, which can make it even more difficult to accurately recognize the minority class (Weiss G. M., 2004). A related issue is overlap of classes (Prati, Batista, & Monard, 2004; Batista, Prati, & Monard, 2004). In some imbalanced domains it has been reported that standard learning algorithms were able to produce accurate models despite the imbalance. The loss in predictive accuracy in other domains or data sets with imbalance was seen then not necessarily to be due to the learning

algorithm mishandling imbalance but other factors such as class overlap and within-class imbalance (Japkowicz, 2001). Thus lack of performance attributed to imbalance may have other causes.

Methods have been proposed to deal with class imbalance, although it must be said that there has not been much systematic evaluation of them, and where studies have been done conclusions are not always consistent (Prati, Batista, & Monard, 2004). There have been positive results reported however. Methods for dealing with imbalance are either at the data level or algorithmic. At the data level, solutions include many different forms of re-sampling, such as oversampling the minority class by repeating points or generation new points synthetically as with SMOTE (Chawla, Bowyer, & Kegelmeyer, 2002), and undersampling the majority class. The latter has been reported to be more effective in general, although both types have been combined and results this can also be very effective. SMOTE was applied to the NASA data sets in (Pelayo & Dick, 2007) and this brought about up to a 23% improvement in geometric mean classification accuracy. The drawback with undersampling is that valuable instances may be removed, and with oversampling there is the risk of overfitting. Random sampling of either type has proven to be effective compared with more sophisticated techniques. At the algorithmic level, solutions include cost sensitive learning where the objective function of the algorithm is to minimise cost rather than accuracy and where higher costs are associated with false negatives rather than false positives, resulting in improved performance with respect to the positives. Another solution is the use of recognition-based algorithms, such as one class SVM, rather than discrimination based. The former may still use data from all of the classes. Other solutions include:

- choosing algorithms with a more appropriate inductive bias (such as instance-based learning),
- partitioning the data so that in the partition that contains most of the positive instances their representation becomes greater,
- use of non-greedy search techniques such as genetic algorithms (Drown, Khoshgoftaar, & Narayanan, 2007),
- combining predictions of multiple models obtained using different under or oversampling rates,
- selecting features that lead to higher separability between the classes (e.g. in the information retrieval domain, odds ratio was found to be better than information gain

as a ranking measure because the former ranks documents according to their relevance to the positive class (Chawla, Bowyer, & Kegelmeyer, 2002)), and

- boosting, a well-known algorithm of which is AdaBoost.

In the last case, boosting is seen to have potential to deal with class imbalance because the problem with imbalance is misclassification of positives, and boosting gives more weight to instances that are misclassified. In the case of the SVM classifier, two studies concentrated specifically on its application to imbalanced data (Akbani, Kwek, & Japkowicz, 2004; Raskutti & Kowalczyk, 2004).

When working with imbalanced data, an important issue that is often raised is evaluation of model performance. Typically used performance measures of accuracy and error rate are not suitable because they attach equal weight to errors of both classes, and this leads to a bias in evaluation towards the majority class. Other ways of measuring performance of models learnt from imbalanced data were discussed in section 3.3 Performance Evaluation.

5.6 Strategies in the Absence of Labelled Data

Means of dealing with the data paucity problem include use of semi-supervised and unsupervised learning methods, which have been advocated in the absence of fault proneness labels in (Zhong, Khoshgoftaar, & Seliya, 2004; Zhong, Khoshgoftaar, & Seliya, 2004). One semi-supervised approach that has been used is to make use of an expert software quality assessor in conjunction with clustering (Seliya & Khoshgoftaar, 2007; Zhong, Khoshgoftaar, & Seliya, 2004). The underlying assumption is that instances of similar quality will have similar software measurements and will belong to the same cluster. The instances are clustered first, following which an instance is taken from each cluster, which is assigned a quality label by the expert, which all other instances in the cluster are then assigned. This has been shown to work quite well, particularly if the software modules are of similar type, or the metrics used are able to characterise quality and enable clustering to partition instances into different quality level groups. Another semi-supervised approach is the Expected Maximisation algorithm (EM), in which only some of the training instances are labelled to begin with. In iterative fashion the remaining unlabelled instances are evaluated and assigned a label according to which class each is determined most likely to belong to. This has been applied to quality modelling in (Seliya & Khoshgoftaar, 2007; Gau & Lyu, 2000; Seliya & Khoshgoftaar, 2007). While these approaches can improve on the performance achieved on

the small number of labelled examples alone, better performing models can be obtained if more labelled examples are available and supervised learning methods can be used instead.

An unsupervised learning study that deserves mention is one in which all software modules were Java software agents, and were described without labels using metrics selected to capture information about the modules thought to relate to quality (Benedicenti, Wei Wang, Lee, & Paranjape, 2001). A simple learning approach was used in which k-means algorithm was used to cluster the instances. The results were remarkable in that the clusters formed matched the quality levels assigned by an expert assessor for 98% of the modules. These results may not ordinarily be expected however for the reason that the software modules in this case were all of the same type, and similar results would not appear to have been obtained in other studies in applying the same approach to other datasets.

5.7 NASA MDP Data Sets

The data with which this research is mainly concerned was sourced from the NASA Metrics Data Program website (Metrics Data Program). There were 9 data sets obtained from this site which are listed in Table 2. As can be seen the data sets are derived from various NASA projects, such as ground and satellite control systems. Most of the projects are implemented in C, except for kc1 and kc3 which are implemented in C++ and Java respectively. Language would have some bearing on metric values as many are language dependent. However in terms of the set of attributes available, all are procedural static code metrics, with the exception of kc1, which also has a separate set of object-oriented metrics, those of Chidamber and Kemerer. The level of granularity on which metrics are computed is the function, to which the term module refers. The size of the data sets varies in terms of number of modules varies considerably, from 403 to 10878. The total number of lines of code for each data set is not shown as the number of modules provides a more useful indication of size. Most of the data sets have 43 attributes, almost all of which are static measures derived from the source code. This includes error count and error density, from which class labels can be derived - an error being recorded when a module is changed due to an error report at any point over the lifetime of the project. The only exceptions are jm1 and kc1, which have about half of the standard set. Attribute metrics are listed in Appendix A. They are largely comprised of the LOC metrics and those of Halstead and McCabe. In terms of modules which contain faults, the proportion is 20% or less for all data sets and the average is 10%, which is quite low, all data sets suffering significant class imbalance. Of all these sets, only jm1, cm1 and pc1 are at

Name	Description	Lang.	#Mod.	#Attrib.	Faulty %
JM1	Real-time predictive ground system Uses simulations.	C	10878	24	19.0
CM1	Spacecraft instrument.	C	505	43	9.5
KC1	Group of software components within a large ground system.	C++	2107	27	15.4
KC3	Collection, processing and delivery of satellite metadata.	Java	458	43	9.4
MW1	Zero gravity experiment related to combustion.	C	403	43	7.7
PC1	Flight software from satellite.	C	1107	43	6.9
PC2	Dynamic simulator for altitude control systems.	C	5589	43	0.4
PC3	Flight software from satellite.	C	1563	43	10.2
PC4	Flight software from satellite.	C	1458	43	12.2

Table 2 The NASA MDP data sets.

the system level, the uppermost level within the subsystem hierarchy. The remaining sets are at subsystem levels, and because of which may be considered to be more specialized, or functionally cohesive. Menzies found classifier performance to be better from these data sets.

5.8 Eclipse Data Sets

The Eclipse data sets, derived from the open source Eclipse project, are somewhat different to the NASA ones. They are however both comprised of static code metrics describing modules, and module fault counts, which makes them potentially useful for quality modelling purposes. The NASA fault counts were obtained from formal error records, and as such the number of faults attached to each module could be considered fairly reliable to the extent that all errors were found and recorded. The fault counts for the Eclipse data were obtained differently, as the Eclipse development system does not provide a formal means of associating bugs with the particular modules that were changed in order to correct them. To obtain counts, the authors of the Eclipse data sets searched the CVS repository for bug identifiers which are noted by convention, and strings such as "fault" and "fixed". If these items were found and it was determined there was enough evidence to believe a module was involved in a bug fix, its error count was incremented.

This method for counting faults could obviously be prone to error, more so than through the use of a formal error tracking system as with the NASA data. Countering this is a potentially rigorous testing effort arising from the open source approach leading to the discovery of more

faults and correspondingly more accurate fault counts. More importantly, the quality models to be developed do not rely on fault counts directly, but simply on whether they belong to one class or another as binary predictors. For most instances fault count is likely to accurately indicate class membership whether or not fault count is mildly in error, with possible exception for modules whose count or density lies near the threshold on which class membership is determined. Of course, an assumption is made that the method used to obtain fault counts from the development databases is nevertheless reasonably accurate. Comment on accuracy is not made by the authors of the data however, as there is no means by which to measure it, as there is no bug database rigorously maintained in parallel, from which accurate fault counts could be obtained for comparison. However, that the data sets have been included in the PROMISE repository (Promise Data Repository, 2009), and a paper about their release has been published (Zimmermann, Premraj, & Zeller, 2007), would support the assumption of adequate accuracy. If further support were sought, it may be among the references in the aforementioned Zimmermann study on the techniques that exist for obtaining fault counts from online repositories in relating bug reports to fixes. The Eclipse sets are used here with the above justification, and on the basis that there are few alternatives in terms of publicly available software fault data sets. As data sets derived from open source repositories they also provide for comparison with those derived from the 'in-house' development databases of NASA.

There are other differences apart from the method by which fault counts were obtained. Faults counts are provided separately for pre-release and post-release, unlike fault count in the NASA data sets which is measured across both these phases. The Eclipse module is at the file level rather than function. As far as modelling is concerned, this probably would not have too much bearing, except that it affects metrics used and it would have some effect on the target function to be learnt in mapping from file instead of function. The Eclipse data sets contain different metrics, and many more metrics. They contain only a small set of specially selected metrics - which includes McCabe's cyclomatic complexity but not Halstead metrics unlike the NASA sets - and a large number of metrics which are direct counts of elements in the Abstract Syntax Tree (AST) parsed from the source code. The Eclipse metrics are listed along with the NASA ones in Appendix A. In number they total 200, which is far greater than the 43 metrics in most of the NASA sets, mainly attributable to the AST metrics. While the AST metrics may be found to be of predictive value themselves, their inclusion is also to provide raw information about the source from which to derive other metrics, that may have

Name	Release	Lang.	#Modules	#Attribs.	% Faulty Pre release	% Faulty Post release
E1	2	Java	6729	200	39	14
E2	2.1	Java	7888	200	27	11
E3	3	Java	10593	200	27	15

Table 3 The Eclipse data sets.

more predictive value. The Eclipse sets also include module file names, which allows reference to the source code from which additional metrics might also be computed.

In total there are 6 Eclipse data sets, with 3 at the file level and 3 at the package level. However only the file level ones are of interest in this research, with packages being at too course a level to be considered useful for fault prediction. The 3 file level data sets, listed in Table 3, correspond to different releases of the Eclipse project. As can be seen, Eclipse is implemented in Java, and all metrics are Java derived. The total number of modules in each data set is relatively large compared to most of the NASA sets, all having at least 6000 instances. The number of metrics collected for each version is the same. It will be noticed in the table that percentage of faulty modules is provided for both faults found prior to release, and after. As would be expected the proportions for the latter are less with many faults already having been fixed prior to release. Of interest in this research are just faults prior to release, as the objective is to develop models for use during software development to identify faulty modules as early on in the process as possible, avoiding more costly later correction. As well, there may be a stronger relationship between static code metrics and faults prior to release, than with faults found following deployment which may be more runtime related. The proportion of modules with faults in the Eclipse sets is significantly larger than for the NASA sets. The average proportion for the NASA sets it was mentioned is 10%, which compares with around 30% for Eclipse - three times the size. This may be the result of poorer quality code to begin with, rather than testing being more thorough. This higher fault rate may reflect on the Eclipse system having fundamentally different characteristics being an open source project to the NASA projects which were developed by NASA or its contractors.

5.9 Exploratory Data Analysis

5.9.1 Feature Distributions

Perhaps the broadest view of the data is in the distributions of its features. These show where most of the values lie, and how they are spread out across the range of each feature. A

frequency histogram would be one way to show these distributions, however, as there are many features and a number of data sets, a box and whisker plot is used instead for each data set. In a boxplot, each box indicates spread of the data across values at the centre of the distribution, specifically, from the value at the 25th percentile to the 75th percentile, the nth percentile being the value below which n% of the observations fall. A line across the box at some distance along its length indicates the centre of the distribution - the median, or 50% percentile. This point being away from the centre of the box indicates skewness in the distribution. Lines extend from each end of the box, the so called whiskers, to indicate the range of values on either side of the box, +-1.5 times the value range of the box (the value range is referred to as the interquartile range). Beyond the whiskers, values are considered to be outliers and are drawn as points. Some jitter is added to the outliers to have a better sense of their density, in the case where many values are the same or similar. Because the ranges of features are very different from one feature to another, which would make the boxes of features with small values hardly visible next to features with much greater values, all features are normalised first into the range 0..1 before plotting. This still allows the distribution of values to be accurately represented, and for comparison to be made between features.

Box plots for the NASA data sets are shown in Figure 20. Only those for 7 of the sets are shown as they all contain the same features. Omitted are jm1 and kc1 as they have a different number of features. Those shown though however give a sufficient idea of how the distributions behave. Notable in these plots is that most features are positively skewed, with most values lying closer to the 0 end of the range than 1. These are either exponential distributions, or log normal. There are also appear to be quite a number of outliers present in many of the features. These are points which lie away from the box which contains 50% of the values, by a distance at least 1.5 times the range of those values. In many cases this probably reflects the skewed distributions having long tails. This behavior applies also to error count and density, in which most of the values are small, but which also have quite a number of outliers that are well obove these. A few features are less skewed such as percent comments (lCp) in the Lines category, and McCabe's maintenance severity (mMS) which is slightly skewed in the opposite direction. The plots in this figure give an overall impression of how the data is spread out across features, for each data set. That there is significant skew may have some bearing on algorithms that assume normal distributions. It may also suggest the use of transformations to better shape the data, such as taking the log of values, as Menzies has had some success with in his modelling with the NASA data. It should also be

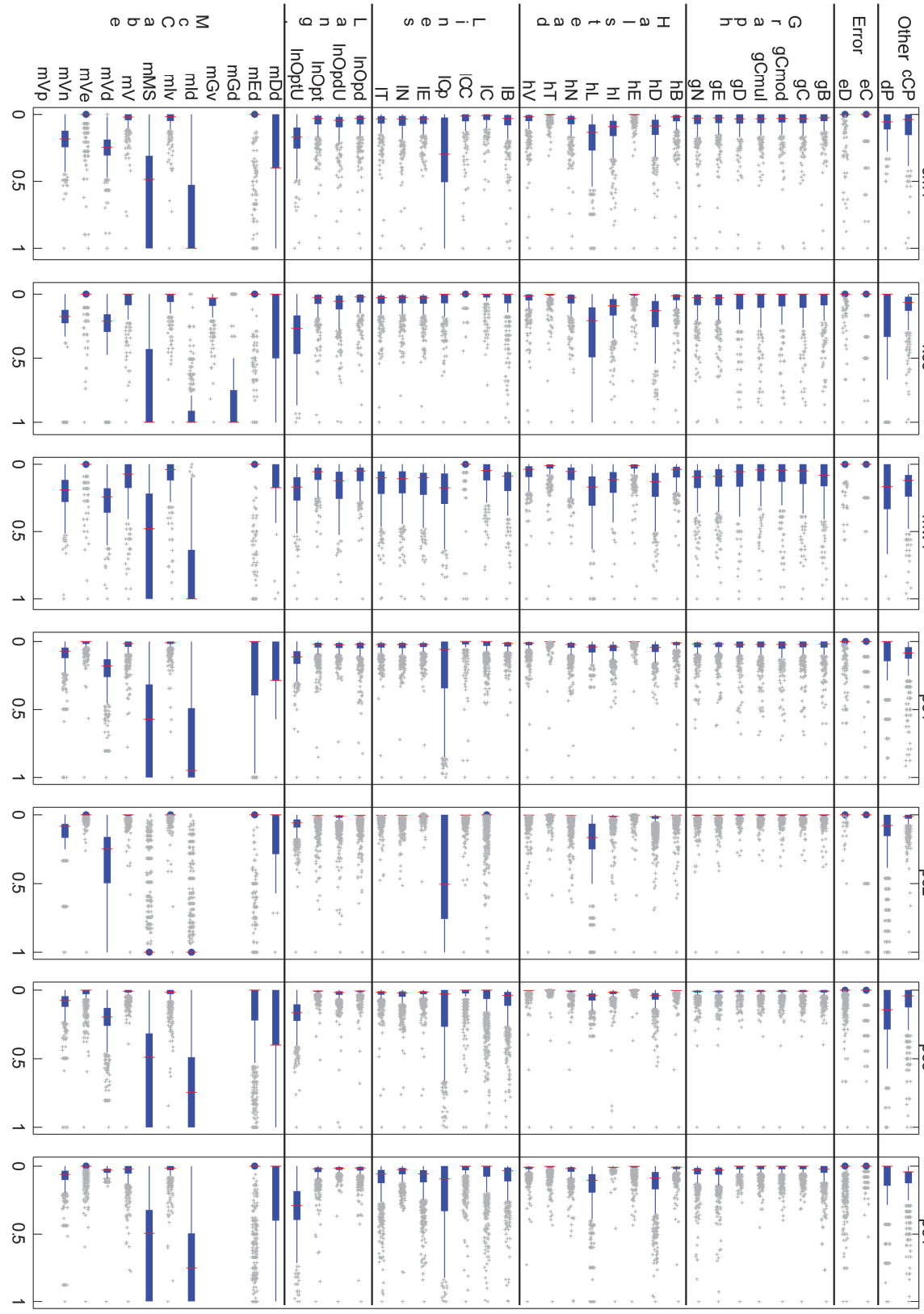


Figure 20 Feature distributions for the NASA data sets (excluding jm1 and kc1).

noted that between data sets there is some difference between feature distributions. For the

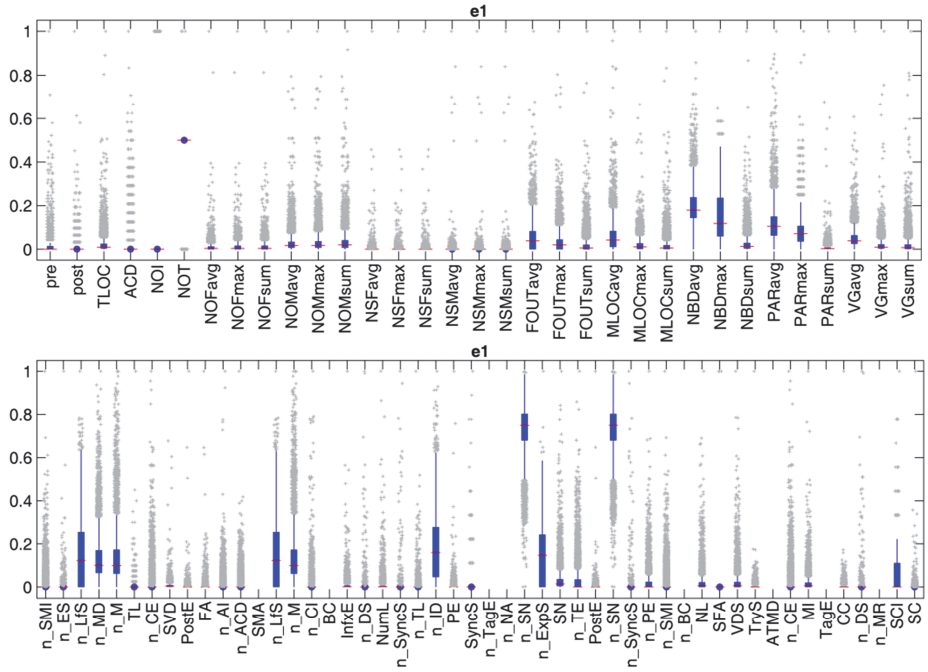


Figure 21 Feature distributions for the Eclipse data set e1.

line metrics lCp for example, the distribution for kc3 is much more skewed than the data set to its left, cm1. Given the many outliers in kc3, this is likely the result of their causing the centre of mass to appear close to zero rather than kc3 values actually being smaller than those of cm1.

Distributions for the Eclipse sets are very similar across data sets and so only those for the first release, e1, are shown in Figure 21. All non AST features are shown in the top plot, and only a random selection of AST features in the bottom as there would be too many otherwise to show. The pattern is very similar to that of the NASA sets, where most of the distributions are heavily positively skewed being centred close to zero and have long tails which contain many outliers relative to the median point.

5.9.2 Module Size

In terms of individual features of interest, one is module size, or total LOC. Module size has an effect on metric values, with smaller module size being associated with smaller metric values. It also has bearing on faults. Larger modules are more likely to contain faults as they tend to be more complex. To see how well this relationship holds, size distributions are shown, as well as the number of faults in each size bucket. Plots for the NASA set are shown in Figure 22. For each data set, bucket edges on which frequencies are based are spaced at

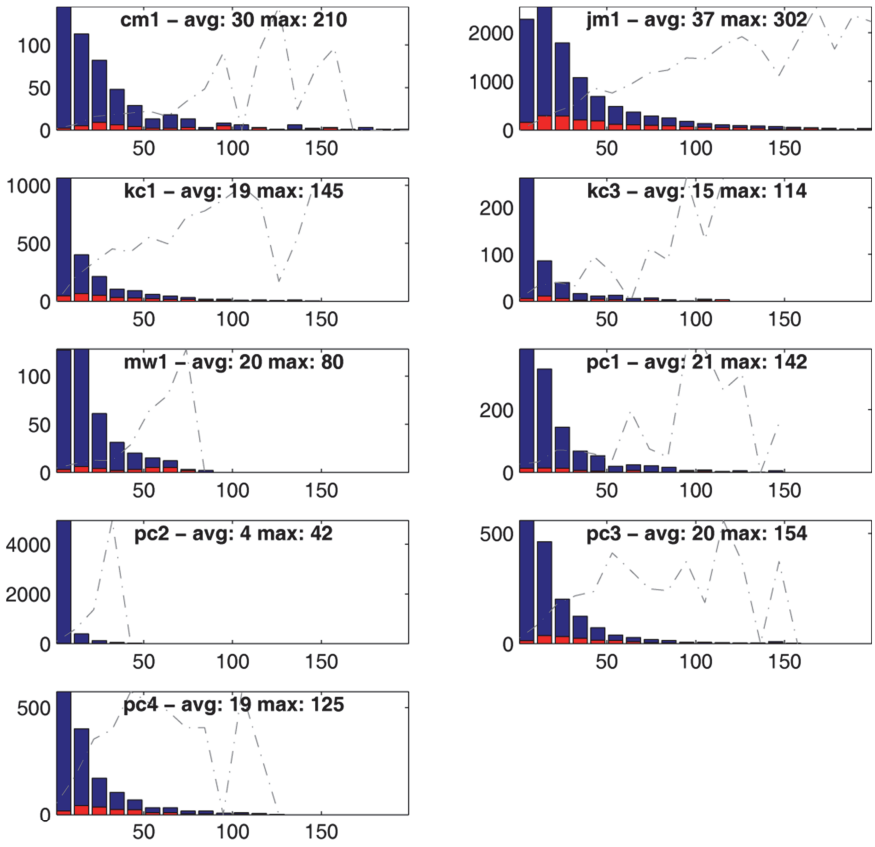


Figure 22 Module size distributions for the NASA data sets with the frequency of faulty modules indicated in red. The dashed line is the proportion of positives in each bin. There is a predominance of modules of smaller size, and a higher proportion of positives as module size increases.

intervals of 10, so that the first bar represents the count of modules with size < 10, up to a maximum of 200. The same buckets are used across data sets so distributions can be more readily compared. It should be noted that for some data sets there are outlying LOC values that extend beyond 200, which are indicated by the maximum value shown at the top of each plot. The average value is also shown as an additional factor to consider in making comparisons. It will be noticed there are two sets of bars, one showing frequency of all modules in each size bucket, and the other the frequency of faulty modules only (with faults > 0). Additionally, there is a dashed line which shows the proportion of faulty modules in each bucket rather than the total number of faulty instances as indicated by the bars. This reveals explicitly the relationships between module size and number of faulty modules.

At initial glance many of the size and error distributions are quite similar across data sets. However on closer inspection there are differences noticed. What is similar though to begin with is the positive skew as for other features. The largest number of modules lies in the first

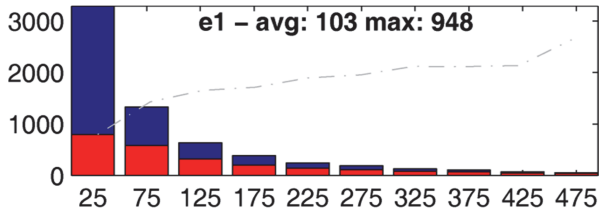


Figure 23 Module size distribution for Eclipse data set E1.

or second bin, and this gradually decreases as module size increases. The average module size tends to be around 20, except for cm1 and jm1 which have larger modules, and pc2 which has very small modules, with the average being only 4. In this latter case, other metric values may be of such limited range as to not provide enough variation by which to discriminate between the classes. Maximum module size varies. The size of 200 seems to be a good working limit to go by (or 150 if cm1 and jm1 are excluded), which covers bins in which there is visible frequency. Fitting less well to this range is jm1, whose maximum is significantly higher at 300, and pc2 whose maximum is significantly lower at 42. Faulty modules tend to have a fairly flat log normal distribution, peaking in the 2nd bar, and from there gradually decreasing. The highest levels are in jm1, corresponding to it having the highest overall proportion of faulty modules at 19% (from Table 2). As expected, the rate at which size frequency decreases is higher than the rate at which faulty module frequency does, and this results in a higher proportion of faulty modules as module size increases. This is more evident in the dashed curve showing the proportion of faulty modules.

A similar module size distribution plot for the Eclipse data set e1 is shown in Figure 23. It is only shown for this data set as the size distributions in the other two Eclipse sets are virtually identical. The only difference in the subsequent releases of Eclipse is the lower frequency of faults corresponding to lower fault levels. The size of modules in the Eclipse sets is much larger than the NASA ones, so the bucket size is increased from 10 to 50, up to a maximum of 500. As for the NASA sets, there is a strong positive skew, with most modules lying in the buckets of smaller size, with frequency decreasing as size increases. The notable difference is in average module size, which at about 100 is significantly larger than the averages of the NASA sets, the largest of which is 37 for jm1. The reason for this is that modules for the Eclipse set are measured on files rather than functions. The bars for error density are noticeably higher for the Eclipse set, and this is due to the total proportion of faulty modules being higher at 39% compared to the highest in the NASA set of 19% for jm1. There is

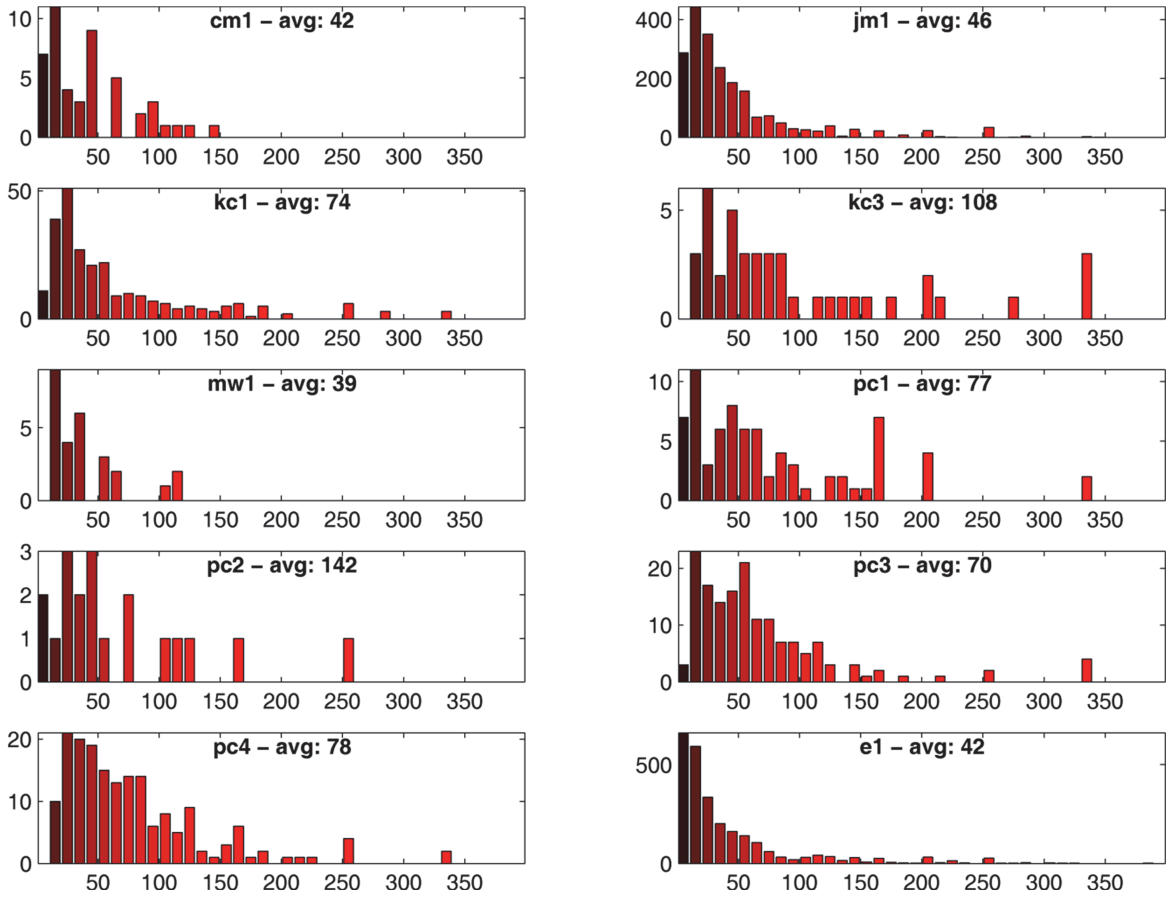


Figure 24 Error density distributions for faulty modules, for NASA data sets and Eclipse e1, which appear to conform to log normal shape consistently across data sets.

however the same increase in proportion of faulty modules as indicated by the curve, as module size increases.

5.9.3 Error Density

Another aspect of the data of interest is the distribution of error density. Previously, the distribution of faulty modules with respect to given module sizes was shown. However this was without regard to number of faults, or density of errors, other than being above zero - in any given bin the faulty modules could be of any fault density. The full distribution of error densities is of interest here, so it is possible to tell of the faulty modules which fault densities occur most often, and how this frequency is distributed.

The fault density distributions are shown for all of the NASA data sets and the first release of Eclipse, in Figure 24. There are 40 bins each of width 10 up to a maximum density of 400. Only modules with error density, denoted by eD, greater than 0 are included (as the number without any faults far exceeds the number of faulty modules and if plotted would overshadow

the frequencies of faulty modules). This covers most of the visible error density range. However there are outliers which lie beyond this, up to 750 in the case of jm1 and kc1. In order to plot error density for the Eclipse set, as it only contains error count, error density was computed as error_count × 1000 / TLOC. It can be seen from the plots that the error density distributions are positively skewed. Lower error densities are more frequent with frequency decreasing as density level increases. Averages are between 40 and 80, with some significantly higher, but these values are affected by extreme outliers.

5.9.4 Module Size and Error Density

There is a view that faults are correlated with LOC. Higher LOC creates greater complexity, and this consequently increases the chance of faults occurring. This relationship between size and faults was observed to an extent in the earlier module size plots, in that modules of some error density > 0, occurred in greater proportion the higher the module size. But the actual density values were unknown. Here the relationship between these two features is observed directly, with the purpose being to see whether there is in fact an obvious correlation between module size and error density.

A plot of these two features is shown in Figure 25. Contrary to expectation, there is no linear correlation between them, and in particular, as LOC increases, there is actually a decrease in error density rather than an increase. This can be explained by looking at the curve of points nearest the origin, which is generated from modules having error count of 1 over modules of

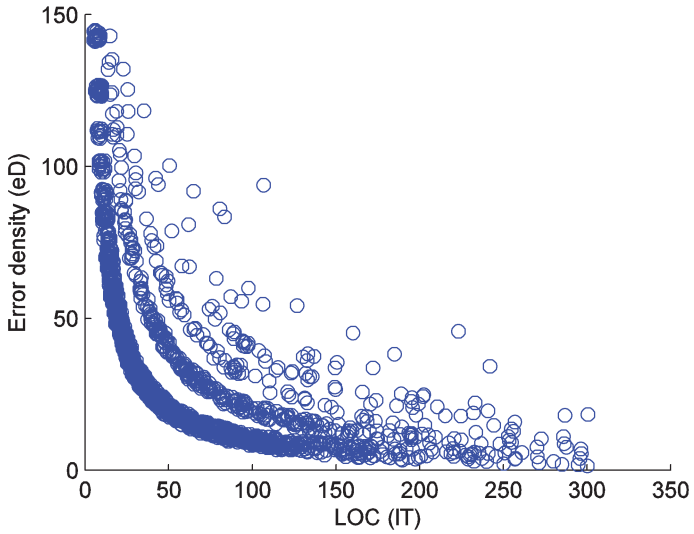


Figure 25 Scatterplot of LOC vs Error density, from jm1.

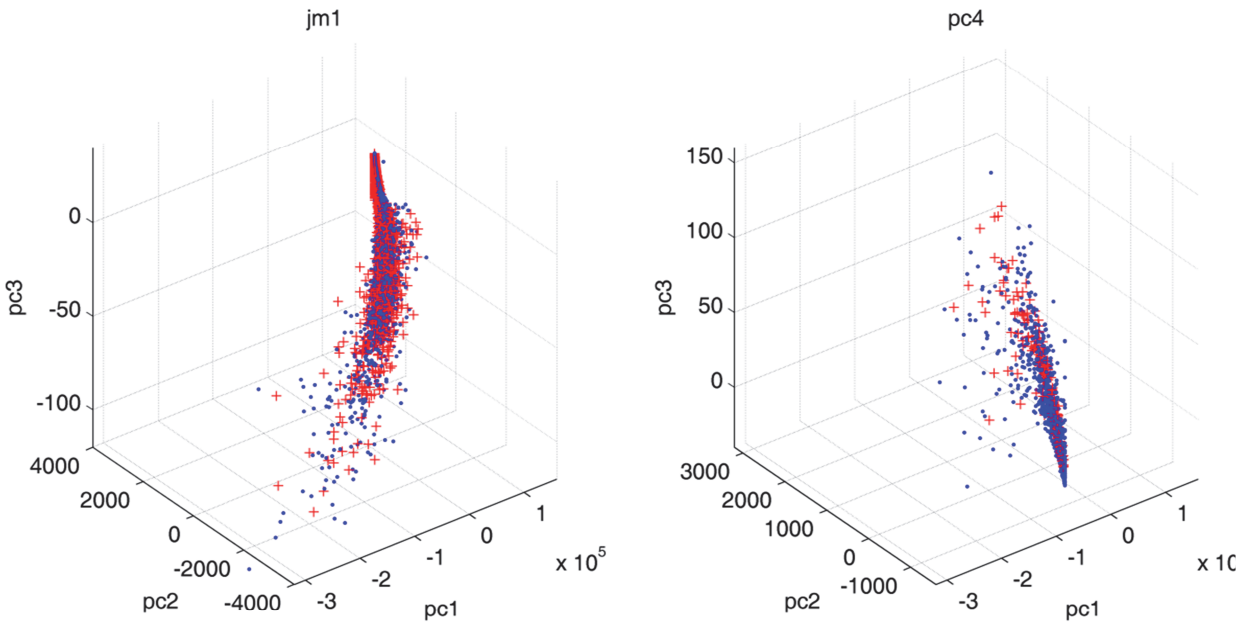


Figure 26 PCA scatterplots of jm1 and pc4 in which points of both classes are interspersed, particularly for jm1, appearing to offer little class separation that a learning algorithm could exploit.

varying size. When the module size is small, the error density is very large. As module size increases the error density drops. There is thus a trade-off between size and density for a particular error count, and this creates for a sweep of points from axis to axis rather than the expected diagonal. The counts of faulty modules shown previously in the module size plots are obtained by counting vertically the points above ranges along the x-axis. It can be seen therefore that the faulty modules with high LOC all have low error density.

5.9.5 PCA Visualisation of Data Sets

A means was sought to find a way to visualize the data with all its features. The method commonly used to do this is principle component analysis. Essentially this method transforms features into a reduced set, called principle components, which are uncorrelated. The first principle component accounts for most of the variation in the data, and second component, as much variation of the remaining data, and so on. In terms of visualisation, PCA provides a lower dimensional picture of the original data set. The first 3 components are used to produce a 3-dimensional visualisation. It was thought this might provide a useful visual representation for finding separation between the classes. However, as can be seen in Figure 26, the positive and negative points appear to be very much interspersed, with there

being little apparent evidence of separation. This is typical of PCA plots of any of the data sets. The PCA visualisations were thus found to be of little use.

It should be mentioned that the Mahalanobis distance was used to remove outliers prior to plotting so that the main point cloud would be of focus in the plot. Instances whose distance was greater than the 95th percentile were removed. The Mahalanobis distance is commonly used to detect outliers in multivariate data. It takes correlations between features into account and effectively measure distance of a point from the centre of mass of the data, as Euclidean distance divided by the width of the ellipsoid around the data in the direction of the point. A point that is further away from the centre of mass than another but which is closer to the ellipsoid thus, correctly, has a shorter distance. The method often works well even when data is not normally distributed. It may not work well if there are multiple clusters in the data, in which case a clustering outlier method may be more suitable. The NASA and Eclipse sets tend to be comprised of a single cloud of points, so multiple clusters are not an issue in this case.

5.9.6 Feature-feature Correlation

High correlation between features indicates a degree of redundancy between them, and to the extent this is so features may be removed without loss of classifier performance. Removal of highly correlated features in the case of Naive Bayes may improve performance due to the assumption of independence between features given the class. Correlation measures the strength of the linear relationship between two features. If two features are highly correlated then one offers little information beyond the other. The features are dependent in the sense that knowing one feature has bearing on the probability of the other.

Given that there are many features for which correlations are computed, correlation values are plotted as distributions for each data set rather than plotting each correlation individually. To do this, first multivariate outliers were removed using the Mahalanobis distance as extreme outliers can significantly alter feature means used in computing correlation. The correlation matrix was then computed which is a 2D matrix with rows and columns corresponding to features, and elements in the matrix being the correlations between them. The absolute value was applied to this matrix as correlations may be negative and the interest is only in their absolute value. After this, the matrix was flattened, which is to say the table of absolute correlation values was reshaped into a row vector. Finally, frequencies were obtained on these values in histogram fashion. The result is shown in the left of Figure 27 for two data

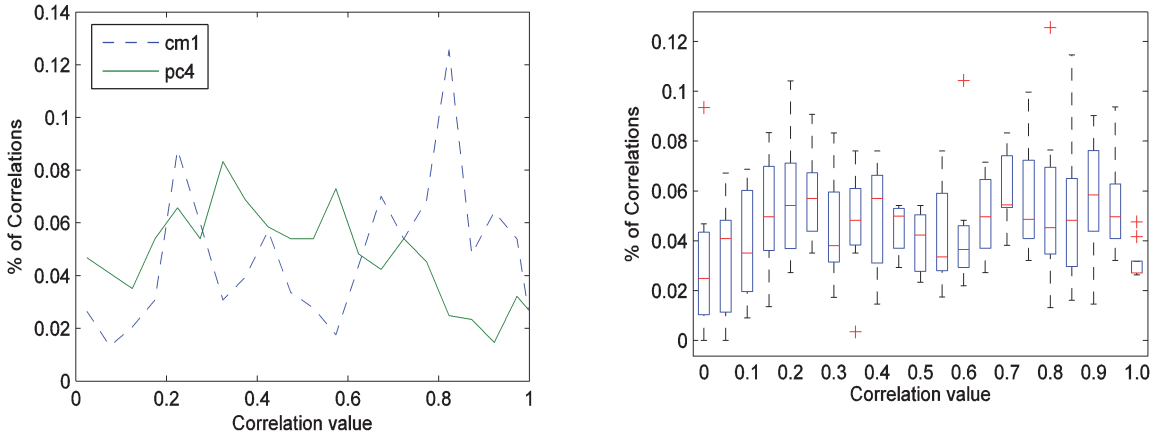


Figure 27 Correlation distributions for the NASA data sets. On the left are distributions of all correlation values between all features for two data sets. On the right, box plots summarize the correlation distributions across all data sets. Both plots show that correlations are evenly spread across the possible range of correlation values, and this suggests that higher correlations are at a significant level.

sets. With more data sets the individual curves become difficult to distinguish, so the distributions for all data sets are instead represented as box plots as shown on the right of the figure, each box representing the distribution of correlation percentages for a particular correlation value, across data sets. For the NASA sets single valued featres were removed prior to computing the Mahalanobis distance as these interfered with its results. These were mGv (global data complexity), mGd (global data density), and mVp (pathological complexity), which are all McCabe metrics. As well, the error count and density metrics were not included as only correlation between the input vector features is of interest here.

It will be noticed in the figure that correlations between features are fairly evenly spread across the correlation range. There are a couple of peaks than can be seen, at around 0.2 and 0.8. About 50% of the correlations are above 0.5. Based on this it could be said that there are many features which are clearly highly correlated and thus redundant, but equally, as many that are less so. Based on the higher correlations it would be expected that feature selection methods that dealt with redundant attributes would choose feature subsets significantly smaller than the set of features available.

The correlation distributions for the Eclipse set e1 are shown in Figure 28. There are two distributions shown, one for the non AST features only, and the other for all features, but

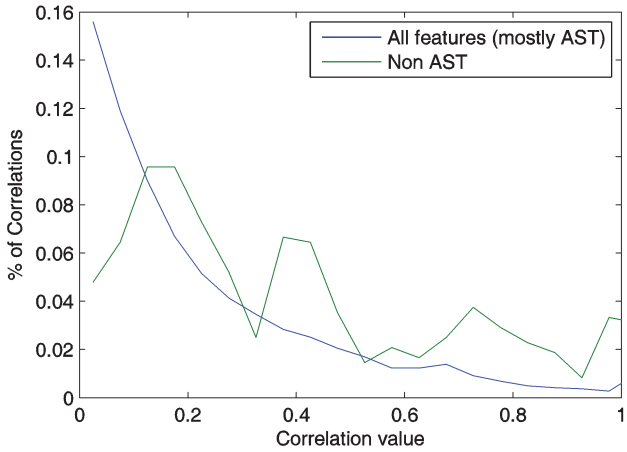


Figure 28 Correlation profiles for the Eclipse data set e1 revealing a different pattern of correlation to the NASA data sets in which there is a surprising predominance of lower correlations between features.

primarily the AST features which far outnumber the other. There is a different pattern here compared to the NASA distributions, in which there is significant right skew, with most correlations being low, with frequency decreasing in exponential fashion as correlation value increases. This might suggest that there could be many useful features amongst the AST ones, and more of the non AST features may be useful compared to those of the NASA sets. Independence alone however does not assure relevance to prediction of class labels, but it does mean that the feature has potential to contribute additional information not possesed by other features.

5.9.7 Feature-class Correlation

As well as the feature-feature correlations, the correlations with error density are also of interest, in providing some indication of information content of features that is predictive of errors. If all features were poorly related to error by correlation, that is they had no behavior in which there was some association with error, one might not expect a classifier to somehow extract from those features patterns that would be predictive. Although with features in combination, if the features are partially predictive over different areas of the input space, performance may be enhanced over that of individual features. Nevertheless, the individual predictiveness could provide a useful indicator as to the quality of the data and whether or not model performance is likely to be adequate. This analysis may also been seen a means of feature evaluation, as the features that are most predictive are good candidates for inclusion in

a feature set. From the feature-feature analysis, one would expect there would be correlation between the more highly predictive features found and other features. In terms of feature selection then, not all features found to be predictive would necessarily be included in an optimal feature set.

With many features in each data set with which to find correlations and limited presentation space, the amount of data this produces is best handled again using a box and whisker plot. As for feature-feature correlations, outliers are first removed using Mahalanobis distance to avoid adverse affects on correlations. The target variable (error density) correlations were found to be quite low as shown on the left of Figure 29. Each box represents the distribution of correlations with the target variable. Subsequently instead, correlations with class labels were computed, with 1 being assigned to an instance if its count was > 0, otherwise 0. These correlations were significantly higher as can seen on the right of Figure 29. It seems it is easier to find an assocation between features and the class label pattern as faulty or not fautly, than directly with error density. The latter plot with class labels as the target is probably more relevant to the models to be developed, as the output of the mapping to be learnt are class labels rather than actual error density.

Of the NASA sets, those with the most predictive features appear to be kc1, mw1, and pc4, based on the top of the boxes and also the top whiskers and outliers. The latter would indicate individual features that are markedly more predictive than others lying in the box region. For pc4, there is one feature which is far more predictive than any in any other data set. In order

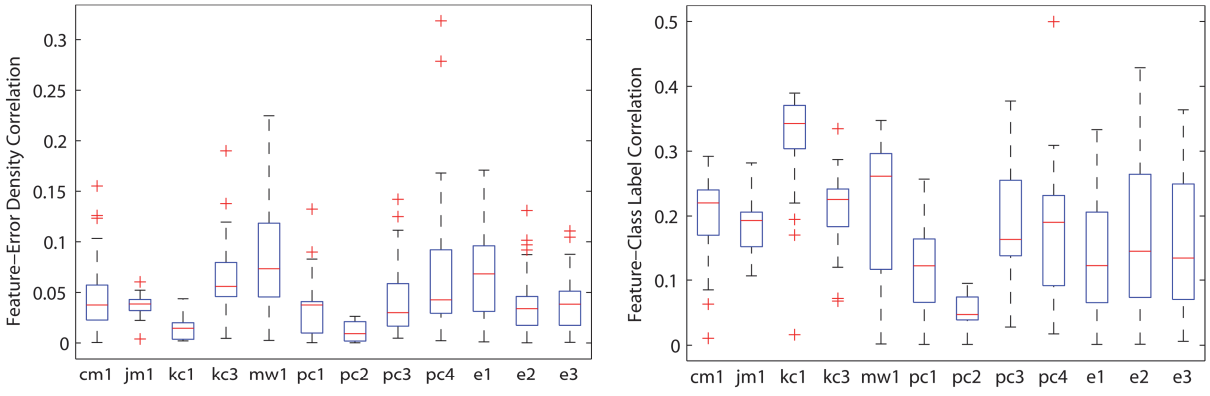


Figure 29 Correlation of features with error density (left) and class labels (right). The latter may be more relevant for binary prediction. It might be inferred from this plot that pc2 is likely to be one of the lesser performing data sets, for kc3 there may be a number of features that are highly predictive relatively speaking, and that an outlying feature for pc4 is virtually twice as predictive as most other features.

to identify this feature, and others that are predictive a table of the 5 most predictive features along with their correlations for each data set is shown in Table 4. This standout feature in the pc4 correlations, from the table, is lCC. lCC is also the most predictive for kc3 and pc1, and lC the most predictive for cm1. Comments therefore would seem to be quite useful for fault prediction, at least in these data sets. The dataset with the least predictive features is pc2. The correlations with class label for that data set are markedly lower than for the other data sets. It may be difficult therefore to obtain good performance with this data set. This would seem to agree with Menzies results from Table 1, in which the F value for pc2 was only 5 (despite surprsingly decent values for the TP and FP rates of 78% and 14%), which is extremely low and much lower than for the other data sets. The box for pc1 is also relatively low, but the top whisker suggests there is at least one feature that is as predictive as the better predictors of the other data sets.

The Eclipse set correlations are somewhat comparable with the NASA ones, although the boxplots for these sets represent many more correlations corresponding to the many more features in the Eclipse data sets. While the boxes extend to a low level, the top whiskers are relatively high, and this might suggest that performance on the Eclipse sets could be comparable with the NASA sets, if not the better.

Another view of the correlation distributions for the NASA sets is shown in Figure 30. Rather than the correlation values being represented by box plots, the values are plotted in descending order for each data set. Some of the data sets initially decrease gradually, such as kc1, others more rapidly, such as pc4. After the first few features there is a continuing descrease in correlations but there are sections where the curves level out where many features share the same correlation level.

	Rank 1		Rank 2		Rank 3		Rank 4		Rank 5	
cm1	lC	0.29	lE	0.29	lN	0.27	lT	0.27	lnOpdU	0.26
jm1	lT	0.28	mV	0.22	gB	0.22	mlv	0.22	lB	0.21
kc1	lnOpd	0.39	lnOpdU	0.38	hD	0.38	hN	0.38	hB	0.37
kc3	lCC	0.33	cCP	0.29	lnOpdU	0.27	hl	0.27	lnOptU	0.26
mw1	gN	0.35	cCP	0.34	lN	0.33	gE	0.33	hl	0.33
pc1	lCC	0.26	lN	0.23	lnOpdU	0.21	lB	0.20	lT	0.20
pc2	lnOpdU	0.09	hN	0.09	hV	0.09	lnOpd	0.09	lnOpt	0.09
pc3	lB	0.38	lN	0.34	lnOpdU	0.33	hl	0.32	lCC	0.29
pc4	lCC	0.50	mDd	0.31	lCp	0.30	lT	0.27	gD	0.25
e1	QN	0.33	ID	0.33	FOUTmax	0.31	SN	0.30	NBDmax	0.30
e2	QN	0.43	ID	0.40	SN	0.40	S	0.39	TLOC	0.38
e3	QN	0.36	S	0.36	SN	0.36	TLOC	0.36	B	0.36

Table 4 Features most highly correlated with class labels.

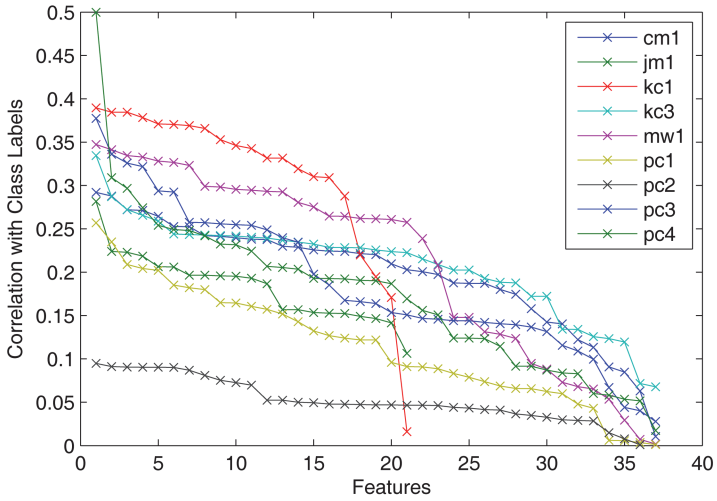


Figure 30 Correlations between features and class for each data set in descending order. This is an alternative representation to that in Figure 29 where for a particular data set, instead of a box and whisker plot summarizing the range of correlation values, the correlation values are plotted as a curve. Unlike Menzies' Information Gain plot in Figure 7, where many features were deemed predictive of class, here the curves drop more steeply suggesting that a smaller set of features may be most relevant, at least according to correlation.

5.9.8 Outliers

In the previous sections outliers were removed prior to computing correlations. It may be of some interest in terms of gaining a further understanding of the data, to see what level of outliers exist in each of the data sets. This could have some bearing on learning algorithms, depending on the methods used. However, Naive Bayes and Support Vector Machines are algorithms that are somewhat tolerant of noise, so the purpose would mainly be to appreciate better the nature of the data.

The method used previously to remove outliers did so if the Mahalanobis score was in the top 5%. The number of instances identified as outliers then by this method is a function of data set size. While this was adequate for avoiding adverse effects in computing correlations, in order to identify outliers more based on their outlyingness from most of the Mahalanobis values, rather than choosing a percentile threshold that selects a fixed proportion of the data set, a standard deviation threshold is used instead. Instances are considered outliers if they are further than 1 or 2 standard deviations from the mean of the Mahalanobis distances. This

method was applied to each data set and the results are presented in Table 5. As well as the number of outliers, their proportion is given, so values can be compared between data sets. With the higher standard deviation, the proportions are simply reduced by some factor as it turned out - there is not any markedly different behaviour in the distribution of proportions when the standard deviation threshold is increased. Plots are shown in Figure 31 of the two datasets with highest proportion of outliers from the table, cm1 and kc3. These are plots of the Mahalanobis distance for each instance sequentially in the data sets. Outliers as determined by the standard deviation threshold of 1 are indicated. It appears this threshold is effective in selecting only the most outlying instances without encroaching too much into the points more densely spaced along the axis which are closer to the point cloud.

5.9.9 Feature Class Distributions

The correlations between features and class labels (and error density) were obtained before for the purpose of evaluating the overall usefulness of features and to identify those which were the most useful. With the same purpose but by another means, class densities were plotted for each feature. A feature is good to the extent that the areas of the two distributions do not overlap. This also served to identify features that would likely give better separation in plots of two features, discussed in the next section - if two features individually have no divergence

	md > (mean+1std)		md > (mean+2std)	
	#	%data	#	%data
cm1	42	8	19	4
jm1	126	1	68	1
kc1	112	5	57	3
kc3	32	7	14	3
mw1	37	9	16	4
pc1	51	5	24	2
pc2	75	1	47	1
pc3	51	3	20	1
pc4	52	4	29	2
e1	297	4	156	2
e2	278	4	141	2
e3	350	3	191	2

Table 5 Frequency of outliers in data sets. md=Mahalanobis distance.

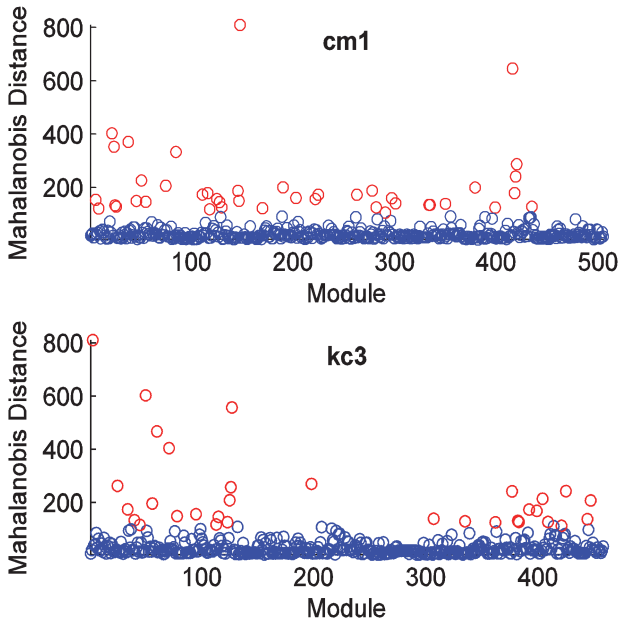


Figure 31 Mahalanobis distance plots with outliers marked (1 std. dev. threshold).

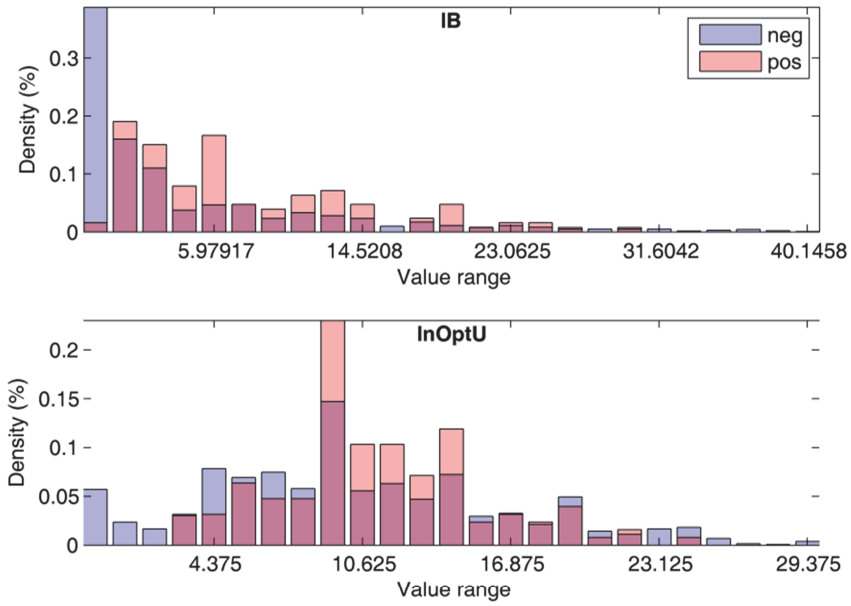


Figure 32 Feature density distributions by class for two pc4 features in which class separation is visible near the origin.

between classes, then they will not when plotted together either. Plotting for each feature resulted in many plots which cannot all be shown here. But two have been selected as examples possessing the desired class separation. These examples are taken from pc4, as it seemed to have better separation in its features than the other data sets. The features from this data set are plotted in Figure 32. Densities are computed as the number of instances in each bin over the range of values of the feature, divided by the total number of instances in the class.

It will be noticed that for lB, near the origin there are mostly negatives, and the negatives there are at high density relative to the positives. This feature alone therefore, based on a feature value threshold could be useful to discriminate between the classes. The second feature, lnOptU, again has some negatives near the origin, without positives, but the mass of negatives unlike for lB is more spread out and would thus likely be less useful.

5.9.10 Two-Feature Plots

Beyond the class distributions of single features, the extent to which separation can be found, can also been determined to some degree through use of pairs of features in so called XY plots. If the separation is good enough, the joint distribution may provide sufficient

information by which to derive a classifier. The difficulty with this approach is that the number of pairs of features is large and there can be too many to inspect. If there are n features, there are in the vicinity of $n^2/2$ pairings. To reduce the number, the features focused on were those identified in the work of the previous section that had significant difference between the class distributions. The XY plot of two features can be shown as a scatter plot, with each class of points distinguishable by colour. If the classes of points are separate, then the pair of features has some discriminative ability.

A problem that arises with this however is that the data sets have a large numbers of points, which sometimes are concentrated in small areas and it can be difficult to tell how many points are in those areas. This problem was resolved by creating two custom plots. In the first, points are plotted in scatter fashion but with contours superimposed to indicate class density. Density plots also lie off each axes which represent the class densities of points of each class projected onto each axes (effectively the class distributions for each feature). The second is a 3D surface plot.

One of the feature pairs showing better separation is shown in Figure 34 and Figure 33. The contours show that the area of highest density of positives is somewhat separate from the highest density of negatives along the x-axis. The contours are spaced at equal intervals apart over the density range, and the number of contours is chosen by the contour algorithm to be some number less than 20 and a multiple of 2 or 5. The surface plot view helps in visualising the densities. There are a number of other feature pairs which also provide some degree of separation, in pc4 and other data sets, but are not shown here. This would suggest that groups of features are going to be reasonably effective in finding separation between the classes, particularly for pc4 where clearly the classes are virtually separate using only two features. A possible practical use of these plots, apart from identifying useful pairs of features, is to create a number of 2D discriminative models combined by some model combination method such as voting. It should be mentioned that this XY plotting method was not applied to the Eclipse sets because of their large number features, although it could be if the feature set was first reduced by some means.

5.10 Discussion

The purpose of this chapter has mainly been to introduce and explore the NASA and Eclipse data sets from which models will be developed. The NASA sets may be higher in quality to the extent that NASA's procedures for detecting and correcting bugs was more rigorous and

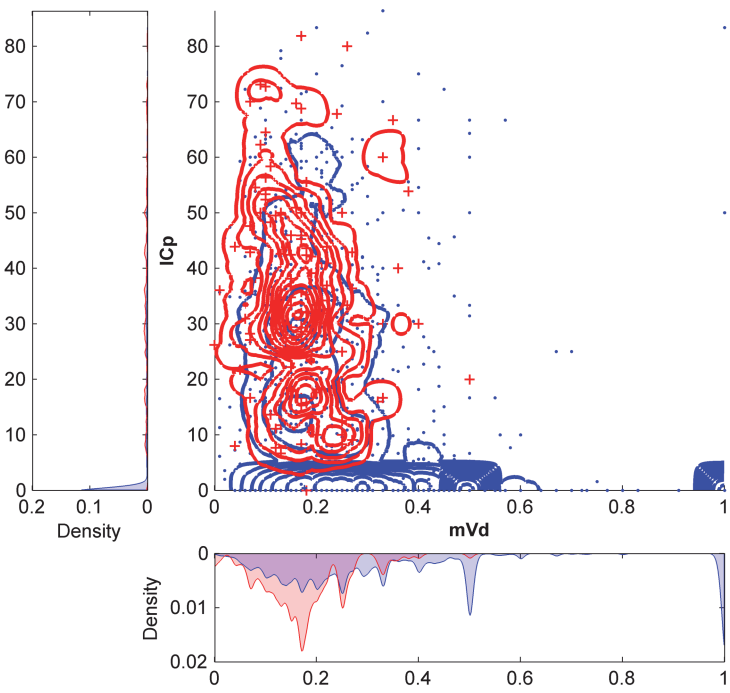


Figure 34 XY scatter plot of mVd and lCp from pc4 in which class separation is evident. The density of points for each class, not always readily apparent from the point cloud, is enhanced with the use of contours and side plots in which density is projected onto each axis.

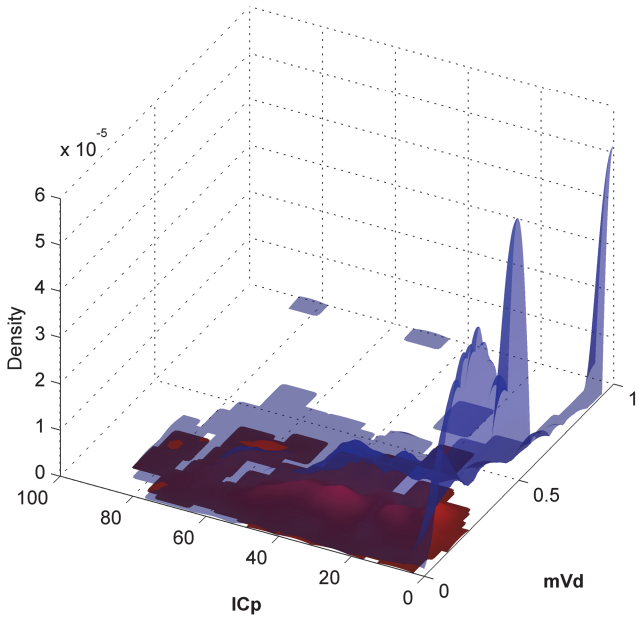


Figure 33 XY surface plot of the same two features in the previous figure, providing another view in which class densities are enhanced.

formalised. The Eclipse data was derived from a less reliable search of fault identifiers and strings in CVS comments. This is not to say though that the Eclipse data sets are not accurate

enough to produce useful models. Noise probably exist in both groups of data sets, and in any modelling exercise there is some awareness of this and an effort may be made if necessary to try to deal with it. It should be noted that the main source of noise is in error counts, and from these class labels, rather than module measurements, the latter of which is reliably obtained from metrics collection tools.

Both sources of data are of similar format in that they both contain a number of features describing software modules (which are all numeric variables without missing data which makes its handling easier), and a fault count feature. Perhaps an important difference, although it may not have much bearing on modelling, is that the Eclipse set modules on which measurements are taken are larger, being at the file level rather than function. This may be relevant in that there may be differences in the behaviour of the two groups of datasets, and the higher level of module granularity may partly explain them. Apart from this, and the Eclipse sets having many more descriptive features, the data sets are quite similar in terms of their makeup.

The chapter began with some background information. This will not be dwelt on at length, but some comments should be made. In relation to the types of metrics contained in the sourced data sets, the metrics they contained are all static code metrics. There is no inclusion of other types such as requirements specifications or 'code churn', the latter of which could serve as another feature by which class labels could be assigned. The noise issue has received some attention, in particular by Koshgovtaar who has studied the NASA data sets. His view is that treatment of noise is more important than the learning algorithm used (Khoshgoftaar & Seliya, 2004). But this focus on noise does not seem to have been shared in other studies that have attempted to develop quality models from the NASA data. This may be due to the difficulty in differentiating between noise and legitimate outliers. As well, some algorithms are tolerant of noise to some extent, as those used in this research are. Another issue with noise removal is that it represents an interference with the sample. This may render developed models less able to predict on new data drawn from the population, which may include instances like those removed.

The class imbalance issue was raised, and there has been an increasing research interest in this area in machine learning. It is still relatively new however, which makes any methods that have been proposed perhaps more experimental and less reliable in dealing with the imbalance problem. There is also doubt that imbalance alone causes deterioration in model performance, and that it may also be due to class overlap for example. Having said that, class

imbalance surely does affect some efforts to develop accurate models, and there are a number of strategies that were mentioned that could be tried in an attempt to deal with it. The most common at the data level is sampling, which is easily implemented and has been found to be effective, and though not used in the standard modelling work, it is Chapter 9. At the algorithm level there were many options mentioned, and if imbalance is seen as being worthy of significant pursuit, they could be explored. One algorithmic solution that will be tried is the "balanced ridge" option that was found to be available in one implementation of SVM that was obtained.

A few approaches were mentioned in dealing with data in the absence of class labels. These involved unsupervised and semi supervised learning approaches. These are not used in this research as there is sufficient data to work with that contains label information, and models are likely to be better learnt with labels than without.

Moving on to the exploratory analysis, it was quite apparent from the box and whisker plots of each feature across data sets, that most features were strongly positively skewed, with most values in the low end of the range of the feature, with a gradual decrease in frequency with higher feature values. This may have some bearing on modelling where normal distributions are assumed. This is the case with Naive Bayes, and perhaps is why log normalisation was found by Menzies to give a significant performance boost. It might also suggest looking for learning algorithms that are better suited to cases like this where this is such a predominance of skew among features.

The shape of module size and error density distributions were quite similar across all data sets, including Eclipse, all being positively skewed (as for most of the other features). Module size averages from about 20 to 40 LOC for the NASA sets, with a maximum of 300 for jm1, and 100 for Eclipse because of the file sized modules, with a maximum of close to 1000. There are not any significant differences to point out in the distributions of module size beyond the common skew and tapering to smaller frequencies as size increases. As expected, the proportion of modules with faults increased as module size increased. In terms of error density, the distributions are similar across all data sets, with peaks at around an error density of about 20, most densities being less than 200, although some being very high (eg 750 in jm1) when errors exist in small modules. The error density distributions will serve as a basis for selection of a threshold on error density by which to assign class labels before model training begins.

There was a fair bit of analysis on correlations, both between features and between features and class (and error density). On the former, the correlations were spread out fairly evenly accross the correlation range, meaning that there were many features that had high correlations and these would be causing a significant level of redundancy between the features. That this is so is probably reflected in the small feature selections that Menzies found to be optimal. It probably also adds support for Menzies' view of a performance ceiling having been reached, in that because of the high levels of correlation, there are fewer features in the data with potentially useful content not shared by other features. By contrast, the Ecilpse sets seemed to have a much lower level of correlation between features, with a strong and desired positive skewed in its distributions of correlations. There would be poentially more useful features in this data.

In relation to feature-class correlations, there was some indication given of the data sets with features that had higher correlations with class, and these data sets would probably turn out to be the better performing ones. These data sets were pc4 by some distance, greatly advantaged in having a single feature lCC with a high class label correlation of 0.5, kc1 and mw1. The lowest correlations were in pc2, and it is consequently expected that obtaining good performance on this data set may be difficult. In terms of the distribution of correlations of features with class, the correlations were sorted in descending order, and there was found to be a gradual decrease in correlation across features. Many features had simliar levels of correlation with class, and this may reflect high correlation between features. The Eclipse set features had class correlation levels that were roughly the same as those for the NASA data sets, having at least some features with higher correlations, so it might be expected that Eclipse model performance will be similar to NASA model performance.

Outliers were touched on briefly mainly to have another perspective on the nature of the data, rather than their identification being of any practical use at this stage (other than for removal prior to computing correlations). The data sets with highest proportions of outliers, between 7% and 9% , were kc3, cm1 and mw1. If outliers were to be removed prior to learning, of the two standard deviation thresholds for identification used, 1 and 2, the latter may be a better choice as it removes only the most extreme outliers and reduces the risk of removing potentially useful data.

The two feature plots were of interest, and as mentioned could serve as a basis for models that could be combined to arrive at a prediction. However, the better approach for now would

probably be to rely instead on learning algorithms to take advantage of any separation found but across more than two features.

Having explored some of the basic characterics of the data, in the next chapter, the focus becomes more practical, turning to preparation of the data prior its use for training models.

Chapter 6 - Data Preparation

Prior to running learning algorithms on the data, the data usually needs to undergo preparation of some kind. It might involve processing raw information such as source code to produce new attributes to be included in the data set. Or it might involve removing instances that are thought to be noisy, filling in missing values, assigning class labels to instances, transforming features, and of course, selecting features as discussed in Chapter 4 – Feature Selection. The NASA and Eclipse sets introduced in the previous chapter, are actually in a form that is fairly close to being useable without having to do that much preprocessing. There are however a couple of important aspects of the data that require some preparation, and these are assigning class labels to instances based on fault information already included in the data sets, and feature selection. Most of this chapter is devoted to the latter task. For this a number of experiments have been conducted using different feature selection methods to find selections of attributes that capture the essential information of the many attributes in the original data and are likely to perform well when it comes to developing models from the feature reduced data sets in the next chapter. The purpose of feature selection is to improve predictive performance as some classifiers such as Naive Bayes perform better when redundant highly correlated features are removed, or to reduce processing time involved in running learning algorithms on the data. It can also simplify models which in some cases may be useful if there is a desire to understand the relation between features and target concept.

The chapter begins with a section describing the intial steps taken in filtering instances and attributes from the data, and in creating normal and log normalised versions of the data prior to any other processing. Class labelling is then discussed, this being the most important preprocessing step involved, as labelled examples are required for supervised learning methods can be applied. The bulk of the chapter then covers a number of different methods used to select features, both manual and automatic using the Weka machine learning tool. Multiple methods are used as there is no single method that is guaranteed to give an optimal selection. Some methods may work better than others depending on the number of features, the nature of the data, and the learning algorithm subsequently applied to the feature reduced data. Finally, a couple of other issues that are relevant to the preprocessing stage are discussed. One is selection of instances rather than features according to module size, along the lines of the findings of the study by Koru & Liu (2005) mentioned in section 2.2 NASA Data Studies. It was mentioned that the many instances with small values can interfere with finding a satisfactory predictive model, their providing insufficient variation with which to

discriminate between the classes. Reducing instances to those of only larger size resulted in better predictive performance. The question is whether this could be of any practical use as an instance filtering preprocessing step.

An issue that is not discussed in the chapter in a devoted section is missing values, but which should be mentioned briefly. A few features in the NASA sets contained a large proportion of missing values, in some cases up to 90% or so of values. These were simply filled in with zeros, as it appeared that the absence of values was due to no count having been made. There was later support for this being an appropriate procedure when Menzies' data became publicly available, in which the same procedure had been applied.

6.1 Initial Filtering

Prior to assigning class labels, it was thought that some initial steps could be performed to covert the original data sets into a more useful form, that could then serve as inputs for other preprocessing. The first of these was the removal of features which contained only a single value, and which obviously have no predictive value. These are not named here as they are easily identified, but some of them can be spotted in the box plots of feature distributions in Figure 20 of Chapter 5, where the plot space is empty because the extent of the box is limited to a single value.

Another initial processing step performed was to remove outlying instances. In the previous chapter, outliers were deemed to be those whose Mahalanobis distance was outside 1 or 2 standard deviations from the mean. While this served to indicate the extent of outlying points, those thresholds are probably too low for the identification and removal of the most outlying instances, as it results on average of around 2% of points being outliers. Instead a higher threshold of 5 standard deviations was used, and this resulted in between 0.3% to 1.2% of instances being removed as outliers from the data sets, both NASA and Eclipse. There was a concern still that this could result in the loss of too much information which could be detrimental to model performance, even as outlying information. However, its use was justified in that if a model were to predict wrongly on all instances similar to these outliers, because of their omission in training, it would only result in a drop in accuracy of 1%. This is the worst case in which it is assumed that incorrect prediction is made on all similar outlying instances. More likely though, instances would be predicted the majority class, so of the 1% of outlying instances only 10% of these would be incorrectly classified, reducing the accuracy rate by only 0. 1%, which is hardly a concern. But by removing the extreme

outliers, the learning algorithm can focus on the patterns in the data where most of the data is, enabling to predict on that data more accurately when otherwise the algorithm may be distracted by outlying information. It will also help avoid adverse effects when it comes to computing correlations for removal of redundant features.

A final step in the inital preprocessing phase, was to apply log transformation, and from which normal and log transformed versions of each data set were created. A problem that arose in doing this is that the log of zero values is infinity, which is effectively undefined. To avoid this, Menzies' method of log tranformation was used in which for any value less than 0.000001, the log value assigned is the log of 0.000001 rather than the actual value (Menzies, Greenwald, & Frank, 2007). Another method that could have been used is Laplacian correction (Han & Micheline, 2006). An example of the distributions of normal and log transformed data is shown in Figure 35, in this case for the pc4 dataset. As can be seen the log transformed data tends to be spread out over a wider range, and this is potentially more useful to learning algorithms.

6.2 Class Labelling

Class labelling involves assigning a class category to each instance. For modelling fault

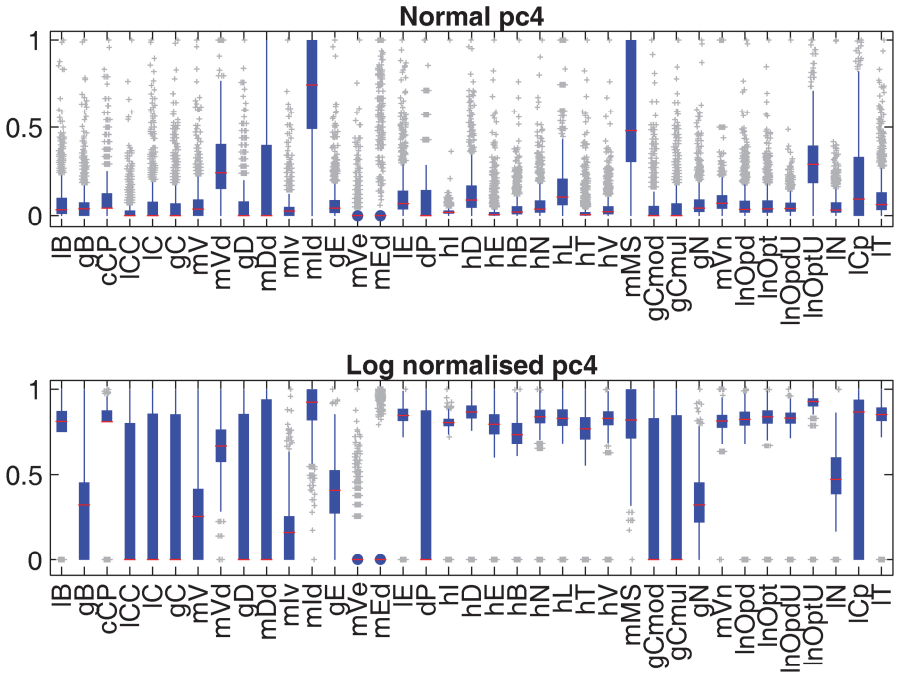


Figure 35 The effect of log normalisation on the data, which has removed most of the tails and values are spread over a wider range which could potentially be more useful to learning algorithms.

proneness there are two classes, fault prone and not fault prone, or low quality and high quality. The standard approach for labelling in this context reported in the literature (including Menzies) is to assign a label of fault prone to instances whose fault count is greater than 0, otherwise not fault prone. If an instance has any errors (at least 1) it is considered fault prone. The problem with this approach though is that it does not take into account the size of the module in terms of LOC. If a module has a fault count of 3 errors for example and is small in size, then the fault proneness or quality could reasonably be considered to be worse, than if the module were of larger size. The attribute of the module that this points to as being of better relevance to fault proneness is fault density. This measures the number of errors per some unit of code size, in this case taken to be 1000 TLOC. The density is calculated as error_count × 1000 / TLOC as mentioned in 5.9.3 Error Density. Apart from providing a better measure of fault proneness, it also has the advantage of providing more resolution on the degree of fault proneness. Rather than single count values, the densities are spread over a range of fractional values depending on fault counts and module sizes.

What remains to be determined is the threshold on error density to use to determine class label. Plots of error density distributions for instances where eD>0 were shown in Figure 24 of the previous chapter. The distributions appear to be log normally distributed. Where a threshold point is chosen is something of an arbitrary decision, and influenced by what degree of fault proness in modules there is interest in identifying in prediction. A threshold of 40 was chosen, with instances being assigned fault prone if greater than or equal to this value. This value is approximately just after the peak in the log normal error density distributions, and includes a significant proportion of instances with faults. In relation to module size, in the plot of LOC vs error density in Figure 25 in the previous chapter, Chapter 5, this threshold effectively includes in the fault prone set only those points above a line drawn horizontally through 40 on the y axis, which means that this excludes from the set of fault prone instances, instances of larger size which have lower densities because of their size. The reduction in the proportion of faulty instances as a result of using a threshold of 40 instead of 0 is shown in Figure 36. While choosing this higher threshold results in models targetting only the modules with higher fault density which is desirable, a downside is that it increases class imbalance, and this could make it more difficult for learning algorithms to predict accurately. Class imbalance was mentioned in the previous chapter as being a significant problem for many learning algorithms whose design is usually based on the assumption that classes are roughly balanced.

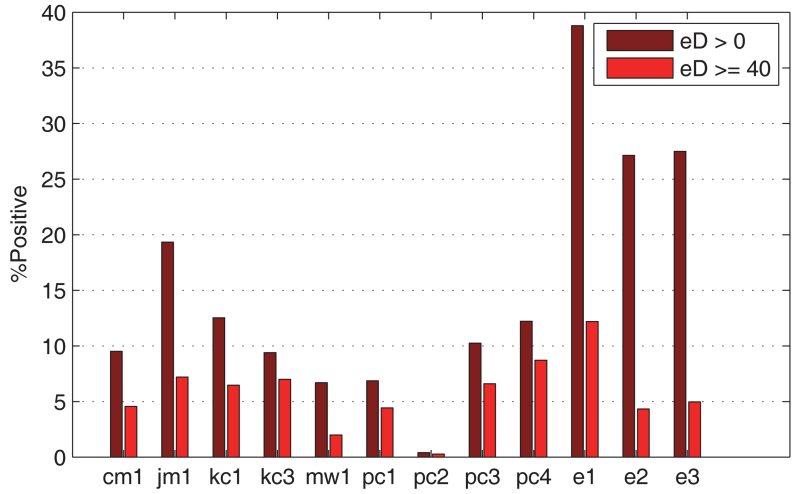


Figure 36 The proportion that positives make up of all instances (relevant to class imbalance) using two different labeling methods: an instance is positive if it has at least one error as used by Menzies and others, and the method used in this research, if error density >= 40. Using error density casts the positive class independently of size, and the high threshold value casts it as only those modules having a high density of faults as distinct from a high number of faults.

6.3 Correlation-Based Filtering of Attributes

Attribute selection is one of the main activities in preprocessing the data, and receives most attention in this chapter. While almost all of the methods used for attribute selection as discussed later are automatic, a simple 'manual' method is to remove attributes that are highly correlated with other attributes. Correlation measures the degree to which there is a linear relationship between two variables. If this is so then as the value of one variable becomes larger, so does that of the other. If values move together in this fashion, then both features offer similar information, and one can be said to be redundant in the presence of the other. As mentioned in Chapter 4 Feature Selection, it is desirable to remove such redundancy in arriving at an optimal set of features for input to a learning algorithm.

One automatic feature selection algorithm which does this explicitly, is the correlation based feature selection method (CFS) discussed in section 3.7.2 Correlation. It differs however from the manual method, in that correlation with class labels is also taken into account. Features are selected which have both high correlation with class, and low correlation with

other features. In the simple correlation based method considered here, only the latter criterion is applied.

Despite being a simple method, it should nonetheless be quite effective in reducing the size of the feature set and in so doing removing features that are unlikely to be useful. This is said based on the high levels of correlation found to exist between features in the NASA data earlier exploratory work described in section 4.9.6 Feature-feature Correlation. In Figure 27 of Chapter 5, it was seen that correlation values across the NASA data sets were equally distributed across the correlation range. This means for example, that 20% of features had a correlation with other features of 0.8 or greater. A significant reduction in the size of the feature set should thus be possible by removal of highly correlated features.

While this method should be effective to some degree as just described, there are many automatic selection methods, and there could be an argument for using them solely for selection. This is an option (bypassing this correlation method), however, using the correlation method does reduce the number of features and this can reduce the processing task of automatic selection methods, especially if exhaustive search is used.

The method used to select features for removal by correlation is dependent on a specified correlation threshold. Determining this is something of a matter of sensible guess and experimentation, and this lead to a value of 0.9 being chosen, the absolute value of correlation values ranging from 0 to 1. This is a high threshold which means attributes are only removed if highly similar, and it avoids removal of features even if the difference between them is small but still large enough to be considered significant.

With a chosen threshold, a simple algorithm has been devised that gives preference to removal of features with higher overall correlation before others, given that for any pair of highly correlated attributes either can be removed. The overall correlation of a feature is the sum of correlations with all other features. The effect of this is that in removing features with correlations above the threshold, as much other correlation below the threshold is removed as possible as well. In other words it removes features to produce the smallest area under the correlation profile curve as possible. The algorithm used is as follows:

```
obtain correlation matrix on all attributes
flist=sort attribs by column-sum of abs value of correlations in corr. matrix
set removedList to []
for each attribute f in flist
       s = identify attribs with corr. > threshold in col f of matrix
```

```
            s = s - removedList - f (ignore corr. with already removed attribs.)
            if s is not empty
                  add current feature f to removedList
            end
      end
end
```

After application of this method to remove highly correlated redundant attributes it was noticed that test models derived from the remaining features for one of the data sets was very low - lower than should be the case given Menzies' reported results. It appeared that features had been removed that obviously must have had predictive value. An experiment was then run to more exhaustively test predictive response obtained using different thresholds for correlation selection. A simple method was used to develop models from the selected attributes, the same method as used by Menzies. This involved ranking selected features first by information gain using Weka, and then creating Naive Bayes models on subsets of increasing size from the top of the ranking. Model performance was obtained using standard 10 x 10 cross validation. The mode with highest F value was chosen. The table of best performance obtained at different thresholds (and associated feature subset counts) on the non log normalised data sets is shown in Table 6. The details of the modelling method just described are not that important. What is important is that the table shows effectively what performance can be obtained on the correlation selected features for each data set at different

	1	0.95	0.9	0.85	0.8	0.75	0.7	0.65
cm1	24, 7	26, 6	25, 6	23, 6	23, 6	32, 7	32, 7	18, 8
jm1	24, 3	25, 3	24, 3	24, 3	24, 3	24, 4	24, 4	19, 4
kc1	27, 9	26, 10	3, 7	3, 7	4, 5	4, 5	5, 4	5, 4
kc3	20, 8	33, 7	36, 7	38, 7	38, 7	38, 7	38, 7	31, 4
mw1	10, 11	14, 6	14, 9	14, 8	13, 5	13, 9	19, 8	15, 8
pc1	19, 15	15, 10	12, 4	11, 8	10, 4	9, 4	12, 10	12, 10
pc2	7, 10	11, 7	9, 6	6, 8	5, 8	5, 7	5, 8	2, 6
pc3	18, 20	28, 22	24, 19	25, 17	26, 17	25, 13	24, 14	21, 13
pc4	41, 11	43, 8	43, 8	43, 8	43, 7	43, 7	45, 12	45, 12

Table 6 Naïve Bayes performance (F value and selected number of features) with feature filtering at different correlation threshold levels across data sets. For kc1 in the values highlighted, reducing the correlation threshold above which features are removed from 0.95 to 9, results in a significant drop in NB performance, and hence for this data set the higher threshold for feature filtering would be a better choice.

correlation selection thresholds.

It will be noticed that for kc1, when the correlation threshold is 0.9, the best F value is only 3. This increases to 26 when the threshold is raised to 0.95. Thus for kc1, a higher, more stringent correlation threshold must be used. At least this is according to the performance obtained with the modelling method used. Even by this method though, with IG identifying most relevant features, while performance may not be optimal, it should be better than 3. So this value would suggest that a valuable feature was removed. For the log normalised data, cm1 and jm1 had better F values when the correlation threshold was 1 i.e. no features were removed by the correlation method. It will be noted as well from the table that it is possible to lower the threshold and remove more features without loss of performance. Although with greater risk of losing potentially valuable information, lower thresholds were not used. What all of this amounts to is that while a correlation threshold of 0.9 seemed to work well for most data sets, for some higher thresholds were chosen. It is interesting to note from this that even when features correlate 0.9 or higher, there can still be valuable information not shared in what would otherwise be considered a redundant relationship. Another point is that reducing the correlation threshold to the lowest level possible without compromising performance based on the table for a given data set could be viewed as a simple attribute selection method for removing redundant features.

Correlation selections for each data set for the non log normalised data are shown in Table 7. Columns on the right show the total number of attributes prior to selection, the number after

	Selected attributes	Tot.	Sel.	%Red.
cm1	cCP,dP,hE,hL,lB,lC,lCC,lCp,lT,lnOptU,mDd,mEd,mId,mMS,mVd,mVe,mVn	37	17	54
jm1	hD,hE,hI,hL,lB,lC,lCC,lT,lnOpdU,lnOpt,lnOptU,mIv,mV,mVe	21	14	33
kc1	gB,hD,hE,hI,hL,hN,lB,lC,lCC,lnOptU,mVe	21	11	48
kc3	dP,gE,hD,hL,lC,lCC,lCp,mDd,mEd,mGd,mId,mMS,mVd,mVe	39	14	64
mw1	cCP,dP,gE,hD,hI,hL,lC,lCC,lCp,lnOptU,mDd,mEd,mId,mIv,mMS,mVe,mVn	37	17	54
pc1	cCP,dP,gE,hD,hI,hL,hT,lB,lC,lCC,lCp,lnOpt,lnOptU,mDd,mEd,mId,mIv,mMS,mVd,mVe	37	20	46
pc2	cCP,dP,gE,hD,hE,hI,hL,lC,lCp,lE,lT,lnOptU,mDd,mEd,mId,mIv,mMS,mVd,mVe,mVn	36	20	44
pc3	cCP,dP,gE,hD,hE,hL,lB,lC,lCC,lCp,lT,lnOpt,lnOptU,mDd,mEd,mId,mIv,mMS,mVd,mVe	37	20	46
pc4	cCP,dP,gC,gE,hD,hE,hI,hL,hN,lB,lC,lCC,lCp,lN,lT,lnOpdU,lnOptU,mDd,mEd,mId,mIv,mMS,mVd,mVe,mVn	37	25	32
e1	A,AA,AC,ACD2,AI,AT,BL,BS,CE,CI,CIC,CL,CS,CastE,DS,ES,FA,FD,FOUTavg,FOUTmax,FS,I,ID,IE,InfxE,J,JS,M,MD,N	154	121	21
e2	A,AA,AC,ACD2,AI,AT,BL,BS,CE,CI,CIC,CL,CS,CastE,DS,ES,FA,FD,FOUTavg,FOUTmax,I,ID,IE,InfxE,J,JS,M,MD,MLC	154	124	19
e3	A,AA,AC,ACD2,AI,AT,BL,BS,CE,CI,CIC,CL,CS,CastE,DS,ES,FD,FOUTavg,FOUTmax,FS,I,ID,IE,J,JS,M,MLOCavg,MLC	154	119	23

Table 7 Correlation selected features resulting in a substantial reduction in feature set size by on average 47% for the NASA sets and 21% for the Eclipse sets. The drop in correlation magnitude between features as a result of applying this filtering is shown in the next figure. (Eclipse set features cut off due to length.)

selection, and the percentage reduction. The percentage reduction is large for the NASA sets, at around 50%, but smaller for the Eclipse ones, at around 20%. The effect of this reduction in overall correlation for the NASA data can be seen in Figure 37, which shows the distributions of 'correlation profiles' across data sets before and after highly correlated features have been removed.

6.4 Selection Bias

Before proceeding further with attribute selection, consideration was given to the issue of selection bias. It is not uncommon for attributes to be selected using all data, and then models to be learnt as a separate step, again using all the data, usually with cross validation (CV). While this seems a reasonable approach, it can result in optimistic performances being obtained, which is undesirable as obtaining an accurate performance is important. The reason for the optimistic performance is that attribute selection in this scenario selects features to perform optimally on all the data. Thus when it comes to cross validation, also on all the data, the performance obtained on the test set is actually slighted biased, as the features were selected to perform well on that data. The selection bias problem has been described in (Singhi & Liu, 2006; Lecocke & Hess, 2004; McLachlan, Chevelu, & Zhu, 2008). To avoid this bias, the principle of complete separation of model development and model validation should be adhered to. This can be done by performing attribute selection within the cross validation loop on each train set, which is then applied to the data before the model is learnt. The attribute selection must also be applied naturally to the test data prior to obtaining model

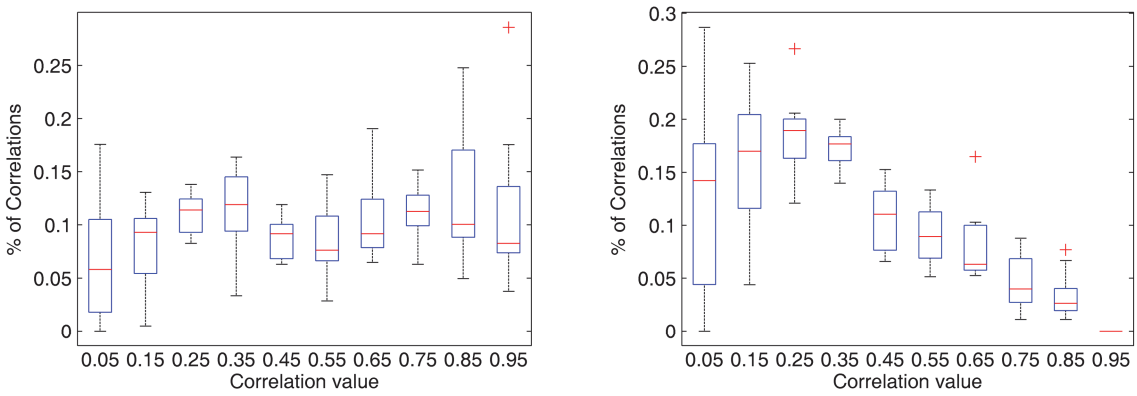


Figure 37 Effect of correlation based feature selection in reducing correlation between features across the NASA data sets – correlation profile ranges before feature selection are shown on the left, and after selection on the right.

predictions. In this way attribute selection has only seen the training data according to the principle of separation between the train and test data. Within the CV loop it is likely that different selections will be produced. A single attribute selection can be arrived at from a table of the number of times each feature was selected across cross validation iterations, and from the table choosing some subset of features based on hit rate. So there is a solution to the bias problem. However it does involve more computation, in that attribute selection is performed on each iteration of the CV loop rather than just once prior to CV. Note that the attribute selection and model training and validation steps can be performed separately, but they would need to use the same, possibly saved, train and test samples. As well, the issue of bias arises only in relation to validation in determining accurate performance. For the sake of attribute selection only, it is not an issue. But there is an interest in this research in determining model performance accurately, so the bias issue is relevant and it may affect how attribute selection is done.

Because of the extra computation involved in the unbiased method, there was an interest in determining whether or not, for the data sets in the present study, the common approach of performing attribute selection on all the data before model validation would produce biased performance. If so, the unbiased method described would have to used. To determine whether or not this was the case, an experiment was performed involving these two attribute selection and validation configurations. If the two configurations produced the same performance then the simpler 'biased' configuration could be used without having to be concerned about bias for the data sets under study. It is possible that there may not be much difference in performance if the data sets are large in terms of number of instances, but there is no known rule of thumb as to how large "large" is. This experiment determines with certainty one way or the other whether bias is a problem with these data sets. The two configurations are illustrated in Figure 38.

The attribute selection method used is the same as that used by Menzies as described in the previous section, with ranking by IG and then selection of the best top down subset based on Naive Bayes CV performance. In the biased configuration CV is performed over all the data. In the unbiased configuration, it is performed over the CV train set, effectively resulting in a nesting of cross validations. As before, the emphasis need not be so much on the details of the method, but on the configurations and whether or not there is any bias in performance with the biased method, as a basis for selection of configuration for subsequent model development.

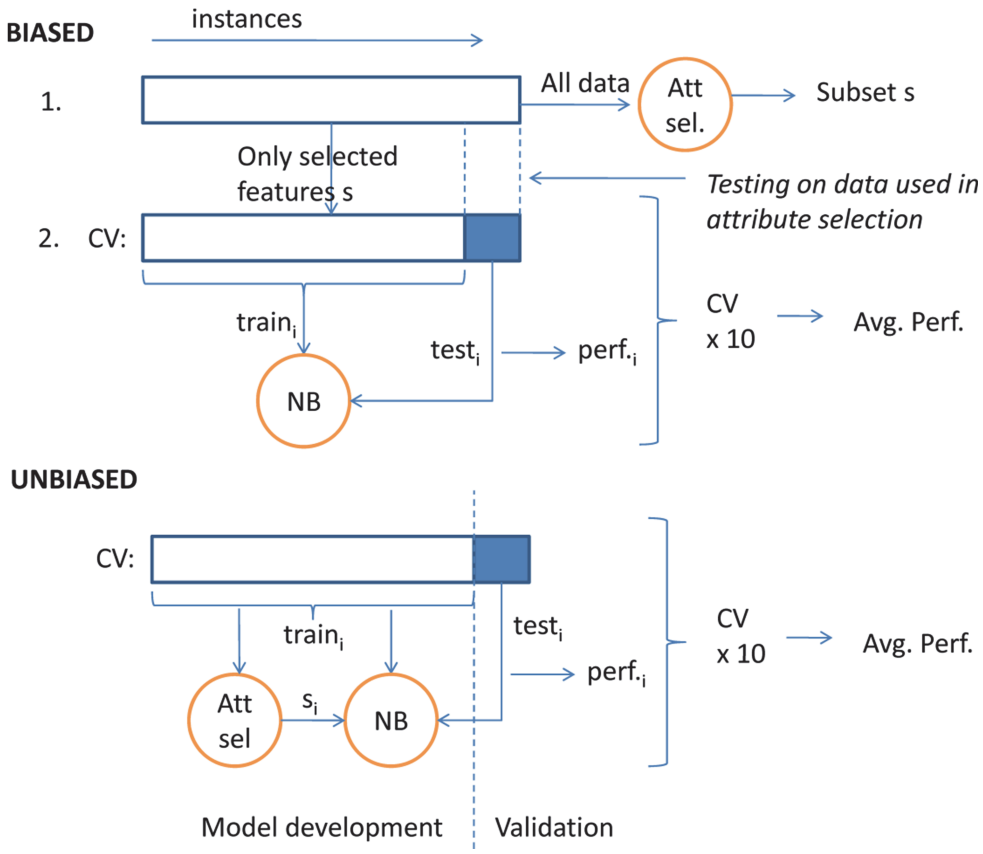


Figure 38 The customary and potentially biased method of obtaining a predictive performance measure shown at the top, and beneath, an unbiased method. Performance bias can occur when features are selected for optimal performance using all the data, including the data in the holdout test set (in cross validation). The unbiased method avoids this problem by selecting features only on the training set, and this is repeated for each CV iteration, however as such, it is potentially much more computationally intensive.

Performance of the two configurations for each data set is shown in Figure 39. They are virtually the same for each data set. The only notable difference is for mw1, which is the smallest data set, for which there is some bias evident. It has been decided therefore that the more common 'biased' approach can be used without it resulting in optimistic performance estimates for all data sets except mw1. An advantage of choosing this approach, apart from being simpler, is that it allows more direct comparison with Menzies' results who used the same approach. With this issue having been resolved, the work of attribute selection continues in the following sections.

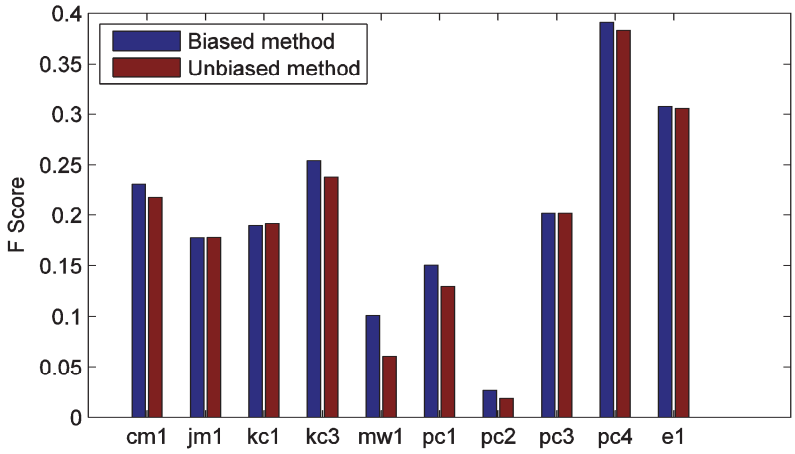


Figure 39 Comparison of performance obtained using the biased and unbiased validation methods of the previous figure, from which it appears the bias resulting from selecting features across the whole data set is negligible, and hence the less computationally intensive 'biased' method can be used on these data sets.

6.5 Relevant Attributes

Before applying automatic selection methods it was thought it might be worthwhile in order to get an idea of the relevance of features, to look at the distribution of feature relevance scores across features for each data set. The correlation analyses in the previous chapter gave some indication of feature relevance in feature-class correlations (see section 5.9.7 Feature-class Correlation). The relevance scores here complement this in providing selection perhaps more sophisticated measure of feature relevance using a feature ranking method.

Information gain (see section 4.6.1 Information Gain) was used by Menzies to obtain the most relevant features for his "iterative subsetting" modelling method. While this could be used here to measure relevance, the values of information gain are not standardized, and hence do not facilitate comparison of scores between data sets. As well there is the shortcoming with IG of it being biased in giving higher scores to features with more distinct values. For these reasons symmetrical uncertainty has been used instead to measure and rank by relevance (defined in section 4.6.1 Information Gain). It standardizes scores into the range 0..1, and it does not suffer the bias that information gain does. The ReliefF evaluation method was another option, but the distance values it computes appear to vary depending on density of points in the data set, which, as for the IG bias, does not facilitate comparison between data sets.

Attribute scores were obtained using cross validation, meaning that evaluation was repeated 10 times and the average score computed across these iterations. This provides a more reliable evaluation of the attributes. The values were computed only for the non log normalised data, but the resulting plots were similar. Only correlation selected features were included to avoid redundancy. The plots of the relevance scores for the NASA and Eclipse sets are shown in Figure 40.

In the NASA plot, the number of relevant features is quite low, other than for pc4, all features of which have some relevance by the symmetrical uncertainty measure. Menzies had mentioned in his rankings by information gain that there were many features with similar IG values at the top of the ranking (see Figure 7 in Chapter 2). This is not quite as evident here as the most redundant features have been removed. It will be noticed that pc4 has the highest scored attribute, significantly higher than any feature in the other data sets. This would suggest that pc4 is likely to give better performance than the other data sets. Similarly, there might be some rough correlation for the other data sets between the highest scores of relevance and the best achievable performance. The scores for mw1 are extremely low (and similarly for the log data), which would perhaps indicate that obtaining satisfactory performance on this data set may be problematic. However, Menzies was able to obtain quite reasonable performance with this data set, with a detection rate of 55%, false alarm rate of

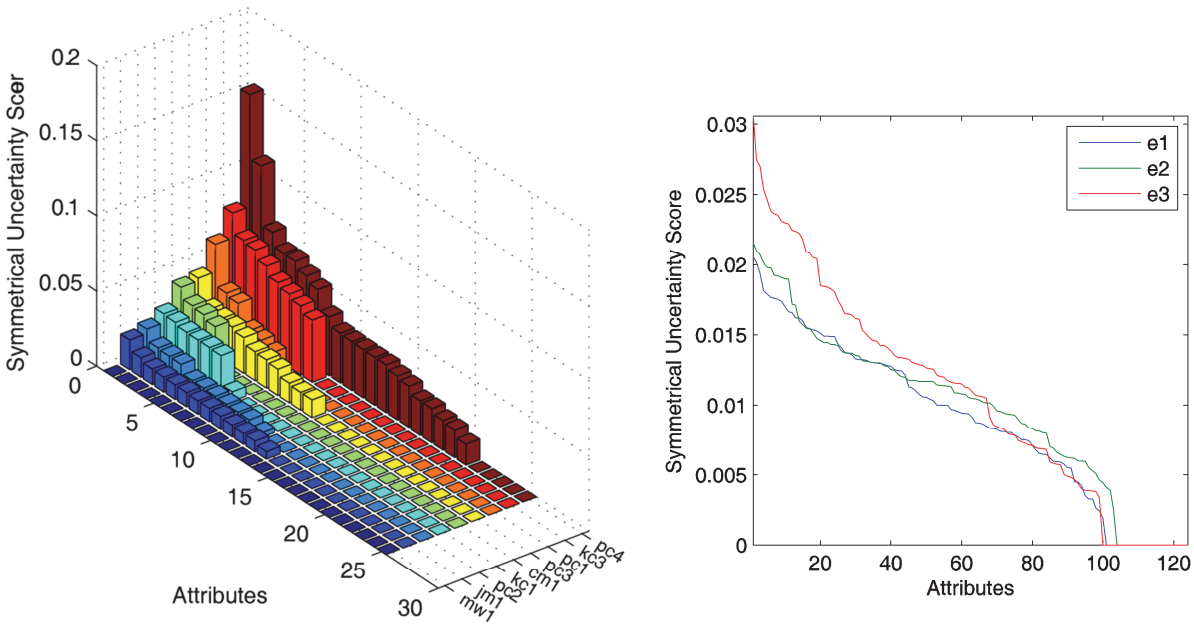


Figure 40 Relevance of individual features with respect to class, as measured by symmetrical uncertainty, in ranked order, for correlation filtered features across NASA data sets on the left, and Eclipse on the right.

14%, and F value of 32. This casts doubt as to how much can be read into these relevance scores in terms of their relation to model performance.

In the Eclipse plot, for sets e1, e2 and e3, almost all the features have some relevance, and overall, relative to the maximum relevance for each data set, the scores are quite high. The maximum scores though are considerably less than those for the NASA sets. For example, the best score of 0.03 compares with 0.15 for pc4, although relevance is comparable at the lower end of the NASA scores.

It was noticed that for the NASA sets quite a number of features had a score of 0. With apparent total lack of relevance this could be used as a means by which to safely further reduce the size of the feature set. Such a method is discussed next.

6.6 Removal by Intersecting Zero Relevance Scores

A number of the relevance scores in the previous section were zero. This would seem to suggest with some confidence that these attributes are unlikely to have much predictive value. Attributes could thus be removed by this means. There would be additional confidence in selection for removal if multiple relevance scores were obtained using different evaluation methods, and only removed were those features which scored zero across all of them. On browsing the attribute evaluation methods in Weka, only two were found to be suitable for this purpose - symmetrical uncertainty as mentioned in the previous section, and chi-squared. Relief gives non-zero scores for all features and so would not be useful for this method.

Applying this method produced the results for the NASA sets shown in Table 8. The rightmost columns show the number of selected features, the number removed, the percentage reduction, and F value. It will be noticed that for some data sets the reduction is substantial, such as cm1, for others, little. The amount of reduction is quite variable by this method. The F value in the last column is the performance by Naive Bayes on the removed attributes. This is produced as a means to confirm that the attributes removed with intersecting zero values do in fact have little predictive value. This is confirmed to be the case for all the data sets, suggesting reliability of the method in this regard, with the only exception being cm1, where there was still a significant F value obtained of 0.15 on the removed attributes. This suggests that some potentially useful information was removed by this method in this case. Consequently, for this data set there could be reason for doubt that this method would be a good means of reducing the size of the attribute set. It will be noticed that for mw1 all

	Selected attributes	#Sel.	#Rem.	%Red.	F
cm1	hL,lB,lnOptU,mVn	4	13	76	15
jm1	hD,hE,hI,hL,lB,lC,lT,lnOpdU,lnOpt,lnOptU,mIv,mV,mVe	13	1	7	0
kc1	hD,hE,hI,hL,hN,lB,lnOptU	7	4	36	4
kc3	gE,hD,hL,lCC,lCp,mDd,mMS,mVd	8	6	43	8
mw1	[]	0	17	100	9
pc1	cCP,dP,hI,lB,lCC,mVd	6	14	70	7
pc2	hD,hE,hI,hL,lC,lCp,lT,lnOptU,mId,mVd,mVn	11	9	45	0
pc3	cCP,gE,hE,lB,lC,lCC,lCp,lT,lnOpt,lnOptU,mMS	11	9	45	1
pc4	dP,gC,hD,hE,hI,hL,hN,lB,lC,lCC,lCp,lN,lT,lnOpdU,lnOptU,mDd,mMS,mVd,mVe,mVn	20	5	20	5

Table 8 Selected attributes after applying the 'zero intersection' reduction method in which features are removed that multiple ranking methods score as having zero relevance. The last column shows the F value obtained with Naïve Bayes on the removed features, confirming whether they do in fact have no predictive relevance, and the low values suggest this is so (except cm1).

features are removed, which draws into question as before, the accuracy of the methods that evaluate relevance.

For the Eclipse sets, the reduction in number of features by this method is around 20, in accordance with the earlier plot in Figure 40. Given the large size of the Eclipse sets, this not particularly large. For the Eclipse sets then, this method could be considered to be of limited use in attribute reduction.

An extension of this method could involve the removal of features with scores in the lowest specified percentage of scores that intersect across scoring methods. This would allow the use of the Relief method which gives non zero values. Another possibility would be to incorporate a classifier and increase the percentage threshold for as long as performance did not deteriorate on the reducing set of features.

6.7 Automatic selection

The previous sections were either exploratory or involved customized methods for feature selection. In this section the focus turns to attribute selection properly, with the use of automatic methods in Weka. Weka selection algorithms have two components, an evaluator and a search component. The search component directs the attribute selection process. It covers the space of possible subsets of attributes from which a selection can be made, either exhaustively or using a more efficient search strategy. The evaluator component evaluates subsets visited during the course of the search, although some known as attribute evaluators only evaluate individual attributes as encountered in previous sections. In the latter case an

attribute evaluator is used in conjunction with a special type of search called a Ranker, which conducts 'search' across individual features rather than subsets. The evaluator provides a score on each feature and the Ranker simply outputs a ranking of attributes rather than a best subset. Different evaluator and search components will be used together in the following sections in various combinations to produce different feature selections for the NASA and Eclipse data sets for use in modelling in the next chapter. The Eclipse sets are handled separately as they contain many more features which necessitates use of different selection methods. The task ultimately is to find which combinations of these components and any associated parameters are best suited to selecting features of these data sets given the number of features from which to select and processing constraints.

Much of the background for this work was presented in Chapter 4 – Feature Selection. This includes approaches to feature selection, namely subset, wrapper, and ranking, as well as detailed information on different selection methods and comments on search strategies. The primary subset selection method used, other than wrapper with Naïve Bayes evaluation, is Correlation-based Feature Selection (CFS) as described in section 4.7.1 Consistency, and the primary ranking method is Information Gain (IG) described in section 4.6.1 Information Gain. Other methods from Chatper 4 are employed however, such as Consistency based subset evaluation, and the Relief ranking method. Each of these methods are represented by an evaluator component in Weka. Selection of a suitable corresponding search component is guided largely by what is computationally feasible (although exhaustive search can lead to overfitting), except in the case of ranking evaluators which dictate the use of a special Ranker search.

An option in attribute selection is to use cross validation. This produces a selection for each CV training set, and a table of the number of times each feature was selected across CV iterations. From this table, a selection of features can be made based on hits. Attributes selected on few iterations can be excluded from selection to avoid overfitting of attributes to the data. The benefit of this approach is that it helps avoid overfitting. This is useful particularly for small data sets with fewer instances for which this type of overfitting is more likely. But given the sets under study are quite large, and the high ratio of instances to number of features, the CV approach is not seen as likely to be of much benefit and hence its use necessary. So attribute selection is performed over the whole data set once.

It might be mentioned that CV based feature selection has some similarity to the previous "selection bias" issue mentioned in section 6.4 Selection Bias. In both the aim is

fundamentally to avoid selecting features that fit the data too well. In the previous case however, this was purely to avoid an optimistic performance estimate. With CV based selection the aim is to select a single set of features (or perhaps a few using a different threshold for inclusion on hit count). And the idea may be to obtain *better* generalisation performance in later cross validation (the 'general' features may perform better in CV iterations, than 'overfitted' features derived once over the whole data set), although this may not necessarily be the result (a more realistic generalisation estimate might give lesser performance). Another distinction that can be made is that in the unbiased method in relation to earlier "selection bias", there was complete separation between feature selection and test data. In CV based feature selection, while selection is in a sense more independent of/less fitted to the data, as features are still derived from all the data (over CV iterations), in later cross validation to evaluate performance there is still a connection between feature selection and test data, and hence to some degree "selection bias" may still exist.

The initial set of attributes used for each data set are those produced by the correlation selection method mentioned earlier in the chapter (6.3 Correlation-Based Filtering of Attributes) - selections are not made on the original full set of attributes. All methods are applied to both normal and log normalised data, to produce separate attribute selections for each. Log data is included as Menzies obtained better performances from it with Naive Bayes. Attribute selections are made separately on this data rather than using those from normal data, as log normalising attributes changes the nature of the data. It would seem possible that because of this change attributes selected specifically for this data could produce better performance on models learnt from it.

6.7.1 NASA Sets

Distinguishing the NASA sets from Eclipse are the fewer features it has to begin with. This allows the use of different feature selection methods. The initial approach is to use an exhaustive search with a couple of different subset evaluation methods. This is only feasible because of relatively small number of features remaining after correlation selection.

A few other methods are also tried that could produce useful attribute selections beginning with more processor intensive subset methods. One is the wrapper approach in which subsets are evaluated using a classification algorithm. As subset evaluation then may be time consuming, a more efficient search strategy must be relied upon. Another approach tried is similar to Menzies' iterative subset method, except that features are rank using multiple

attribute evaluation methods rather than just information gain. Finally the iterative subsetting method is used in its original form for the sake of its performance to other methods in the next chapter, and because in doing so it may have merit over other methods.

6.7.1.1 Exhaustive CFS

A preference was given to exhaustive search to begin with because by its nature in trying every possible subset of features it may find a better performing subset. The problem with this search is the explosion in number of subsets as the number of features increases, which can quickly become computationally intractable. Other than this, there is a potential drawback with the exhaustive approach in that it can select a subset which is too highly fitted to the data, which can lead to optimistic performance estimates. Countering this however may be a degree of generalisation effected by the subset evaluation method used, which typically only provides an approximation of its predictive worth. Overfitting may also be less of a problem when there is more data, of which there is plenty in most of the NASA data sets.

Used in conjunction with exhaustive search is the CFS subset evaluation method (see section 4.7.2 Correlation). This evaluation method was chosen because it captures the goal of optimal subset selection: selection of attributes which are highly correlated to class, but have little correlation with each other.

An issue which arose before using this method, was the possibility that exhaustive search would not finish in a reasonable amount of time. To avoid this problem an upper limit to the number of features that the method could handle feasibly was sought based on the computer hardware used - a 2.4 GHz, quad core processor, although Weka uses only a single core at a time. This was found by running the method iteratively on an incrementing number of features. Execution time for each feature number was recorded. This posed a problem however in that at some point the execution time would inevitably become too long, and the experiment would become stuck. It would also be time consuming even for the iterations that completed assuming enough data points were obtained. These problems were dealt with by running each iteration in a separate thread with a time out in parallel. At the first time out all remaining scheduled iterations were abandoned and the experiment would finish. This experiment was run on the pc4 data set. The resulting time curve is shown in Figure 41. An exponential curve was fitted to the data points to have it extend beyond them to estimate processing time for larger numbers of attributes. As can be seen the increase is dramatic above about 30 attributes, at which point selection takes about 2.5 hrs. Ideally the number of

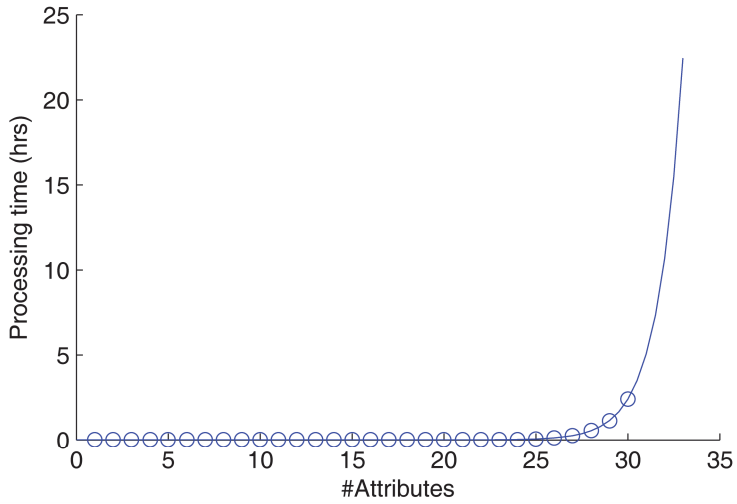


Figure 41 Evaluating the feasibility of using exhaustive search with the CFS feature subset evaluator by increasing the number of features input to this feature selection method and measuring execution time to completion. Execution time is exponential and hence is projected beyond the plotted points. It can be seen that a practical limit to the number of features that can be supplied to this method is around 30 (on the computer hardware used).

attributes would be less than this to avoid having to wait a long time for experiments to complete. With 25 or less features, the processing time is negligible. From the correlation selections in Table 7, for the non log normalised data, the maximum number of feature selected is 25 (for pc4), thus it is possible to run CFS exhaustively on all the NASA data sets without any significant computation time.

For the log normalised data sets, there is a timing issue with cm1, as the number of features after correlation selection for this data set is 37 (no features were removed by correlation selection due to a threshold of 1 being used). To drop the number of attributes by 10 to 27, the 10 worst or least useful features were identified. This was done using the attributes identified as removable by the zero intersection method. But the number of features with zero evaluations by the two attribute evaluation methods used in the zero intersection method, was 31. To select from these the 10 attributes with least predictive value, the attributes were ranked using a different attribute evaluation method, Relief. From this ranking the bottom 10 were selected for removal, to bring the number of attributes in cm1 down to the desired 27. An alternative to using Relief would be to use a wrapper evaluator with Naive Bayes or CFS

again, and a greedy stepwise search with the "generate ranking" option enabled. This might be seen as a selection trick that allows a ranking to be obtained from a subset evaluator.

The CFS evaluation method has a single parameter which is "locally predictive", which by default is set to true. This relates to how it computes correlations to evaluate attribute worth (Hall, 1999). In a more general scheme, CFS selects attributes according to overall correlations based on all the data. This tends to result in 'globalised' selections, which can overlook the contribution an attribute can make in being predictive on only part of its data. The locally predictive option takes this local predictiveness into account, and by so doing enables selection of features that are only partially predictive. The default setting for this parameter, in which it is enabled, was unchanged.

The results of exhaustive CFS selection are shown in Table 9, for both normal and log normalised data sets. They are quite small selections and look to be useful ones. The selections for mw1 with only a single attribute may perhaps be too small, in which case larger selections by other methods to be described may work better. Between the two versions of data sets, without and with log normalisation, the similarity between selections varies depending on the data set. For some data sets such as cm1, the selections are completely different, which would seem to support the approach of computing selections for each version of the data separately. Whether this results in better performances though remains to be seen.

6.7.1.2 Exhaustive Consistency

CFS seems to be one of the most often used and efficient subset selection methods. Another is the consistency subset evaluation method, which was described in 4.7.1 Consistency. It does take longer to run than CFS however, especially for larger data sets, and so the limit for

	Normal	Log normalised
cm1	lB,mVn,lnOptU	mEd,hN,hV
jm1	mVe,hE,lT	lB,mVe,lE,hL,lT
kc1	hI,hE,hN,hL	lE,hT,lT
kc3	lCC,mVd,gE,hD,lCp	lCC,mVd,hB,lnOpt
mw1	cCP	lB
pc1	lB,mVd,dP,hI	lB,cCP,lCC,mVn
pc2	lC,mVd,mId,hI,lCp	lC,mVd,mId,lnOpd
pc3	lB,cCP,lCC,lnOpt,lCp,lT	lB,hV,lCp
pc4	lCC,dP,mMS,lCp	lCC,dP,mMS,lCp

Table 9 Exhaustive CFS feature selections on the reduced feature sets (for exhaustive search to be feasible) from correlation-based filtering listed in Table 7.

the number of input features was reduced from 25 (from the previous section) to about 20. For the normal data, from the correlation selections, this mean reducing the pc4 selection by removing a small number of features using the zero intersection method. For the log normalised data, cm1 with 37 features was again a problem, and 17 of which were removed using the zero intersection method in conjunction with Relief, as described before. No options were available for the consistency method that could be modified. The results of this selection method are shown in Table 10. These selections do not look so useful. For four of the nine data sets, an empty selection set was returned. For two of the data sets, jm1 and pc3, the selections would seem to be too long. At least they are far in excess of the mere three that Menzies used. It was thought that perhaps the empty sets were related to the exhaustive search, so other search methods were tried, but empty selections were still produced. This may be a shortcoming with the consistency method.

6.7.1.3 Best First Wrapper

A feature selection approach that is often recommended for providing optimal classifier performance is the wrapper (see section 4.3 Filter and Wrapper Approaches). The wrapper approach uses a specified classifier to evaluate attribute subsets, usually the same one that is used to build the final model on the selected attributes. If the same classifier is used for evaluating subsets as for later classification it can be seen why this leads to selections with better performance. The drawback with the wrapper approach is that evaluation of each subset by training and evaluating a classifier is time consuming, depending on the classification method used and the size of the data set. Here, the classifier used is Naive Bayes which is a fast learning algorithm, which helps avoid longer processing times. As well,

	Normal	Log normalised
cm1	[]	[]
jm1	lB,lC,mV,mIv,mVe,hI,hD,hE,hL,lnOpt,lnOpdU, lnOptU,lT	lB,gB,lC,mV,mIv,mVe,lE,hI,hD,hE,hB,hN,hL,hV, lnOpd,lnOpt,lnOpdU,lnOptU,lT
kc1	lB,hD,hE	[]
kc3	lCC,mVd,hL,mMS,lCp	lB,cCP,lCC,mGv,lnOpt
mw1	[]	[]
pc1	hI	[]
pc2	[]	lCC,lC,hB,mVn
pc3	lB,cCP,lCC,lC,gE,hE,mMS,lnOpt,lnOptU,lCp	lB,cCP,lCC,lC,gE,hN,hT,mMS,gCmod,mVn,lnOptU, lN,lCp,lT
pc4	cCP,lCC,lC,gC,mIv,dP,hI,hD,hE,hN,hL,mVn	lB,lCC,lC,mVd,mVe,dP,mMS,mVn,lN,lCp

Table 10 Feature selections obtained using exhaustive search and the consistency subset evaluator. Many are empty which does not support the use of the consistency subset evaluator on these data sets.

rather than searching exhaustively, a best first search is used. Best first was chosen over greedy stepwise because it is more exhaustive but still efficient. An option with best first is search direction. Selections were produced using both forward and backward directions. The former adds most relevant features starting from an empty set, and the latter removes least relevant attributes starting with the full set of attributes.

An important modification was made to the wrapper subset evaluation method in Weka to find better selections given class imbalance in the data. The wrapper subset evaluator returns an evaluation of a subset using the specified classifier. As implemented in Weka this is the error rate, or 1 minus the accuracy rate. This performance evaluation measure is known to be biased in favour of the majority class. If used by the wrapper evaluation component it will therefore favour selections which give good performance on the majority class but not necessarily the minority. Therefore, a different evaluation method needs to be used by the wrapper evaluator. This was replaced with the F measure which gives emphasis to the minority class.

Results of this selection method are shown in Table 11. Curiously for some data sets with forward search no selections were returned. As expected the backward search selections are larger, but not excessively so.

6.7.1.4 Average Rank Iterative Subsetting

In the previous sections subset methods were used: CFS, consistency and wrapper. Attention now turns briefly to attempting to exploit attribute evaluation methods (see section 4.6 Filter Ranking Methods). These methods provide rankings of attributes. In order to arrive at a smaller selection, either a preset number of attributes can be specified, a threshold may be used on the scores computed by which the ranking is made, or as used by Menzies, the ranking may be iterated over to produce subsets of increasing size the performance of which is evaluated using a classifier in wrapper fashion, and the subset with the best performance is selected (see section 2.8 Menzies' Studies). The last approach gave good results in Menzies' research and would seem of the methods mentioned of selecting from ranked attributes one of the better ones.

The difference in the approach used here, is that rather than using only a single ranking, such as that produced by information gain as Menzies used, multiple rankings are combined to produce an averaged ranking. It is thought by using different ranking methods that a better final ranking of relevant attributes may be arrived at. The ranking methods included in this

Forward search		
	Normal	**Log normalised**
cm1	[]	lB,cCP,dP,hI,mVn,lCp
jm1	lB,hE,hL,lT	[]
kc1	lCC,lC,hL	[]
kc3	lCC,gE,mEd,mGd,hD,hL,mMS,lCp	cCP,lCC,mGv,mMS
mw1	dP,lCp	lB,hB,mVn,lT
pc1	lB,lCC,lC,mDd,mId,mVe,dP,hI,hD,hT,lnOptU,lCp	[]
pc2	[]	gB,cCP,lC,mVd,mId,mVe,dP,mMS,mVn,lN
pc3	lB,cCP,mDd,mIv,gE,mVe,dP,hD,hL,lnOptU,lCp,lT	lB,lCC,mIv,mId,mMS,gCmod,mVn,lT
pc4	lCC,lC,mDd,mVe,hN,mMS,mVn	lB,lCC,mIv,mVe,dP,gCmod,mVn,lN,lCp

Backward search		
cm1	cCP,lCC,mVd,mVn,lnOptU,lCp,lT	lC,mVd,mId,lE,hI,hL,mMS,mVn,lnOpdU,lN,lCp,lT
jm1	mIv,hD,hL,lnOpdU,lT	lB,gB,lCC,lC,mV,mIv,mVe,lE,hI,hD,hE,hB,hN,hL,hT,hV,lnOpd,lnOpt,lnOpdU,lnOptU,lT
kc1	lB,gB,lCC,mVe,hI,hD,hE,hN,hL,lnOptU	gB,mVe,lE,hB,hN,hT,lT
kc3	lCC,mGd,hD,hL,mMS,lCp	cCP,lC,dP,mGv,lnOpt
mw1	lC,hL,mVn,lCp	cCP,lC,mIv,mVe,hB,mMS,gCmod,mVn
pc1	lB,lCC,mVd,mDd,mIv,mId,gE,mVe,dP,hI,hD,hL,hT,mMS,lnOptU	lB,lCC,gE,mVe,dP,hL,hT,hV,mMS,mVn,lnOptU
pc2	lC,mId,lE,dP,hI,hD,hE,mMS,mVn,lCp,lT	lC,mId,hN,lnOpt,lN
pc3	lB,lC,mDd,mIv,gE,mVe,dP,hE,hL,mMS,lnOptU,lCp,lT	lCC,lC,mId,mVe,dP,hN,hL,hV,mMS,gCmod,mVn,lN
pc4	cCP,lCC,lC,mVd,mDd,mIv,gE,mEd,dP,hD,hN,lCp	lB,lCC,mIv,mVe,dP,gCmod,mVn,lCp

Table 11 Feature selections obtained using a Naïve Bayes wrapper subset evaluator (modified to return an F1.5 measure rather than accuracy) and best first search, both forwards (adding relevants to an empty set of features) and backwards (removing irrelevants from the full set).

method are two attribute evaluators, information gain and relief, and two subset evaluators, CFS and wrapper (with NB) using a greedy stepwise search with ranking output enabled.

A drawback in using only ranking methods is that they rank by relevance without regard to redundancy. It possible therefore that in the final averaged ranking produced by this method, there may be redundant attributes. Optionally these could be removed by quickly running CFS over them for example. This did not seem to be a problem however in Menzies' selections which relied on IG rankings. But with only 3 attributes selected maximum this left little room for redundancy.

In the iterative subsetting step following ranking, the best subset is chosen. This could simply be chosen according to the highest F value. However this may result in larger subsets being chosen, as there is no regard to subset size. With smaller subsets being preferred, both F value and subset size were considered in choosing the best subset. There is a trade-off between the two. A lower F value may be accepted if the number of attributes is significantly less. Taking both factors into account a formula is used to score a subset in which the F value

is multiplied by a scaled subset size. The number of attributes selected is dependent on this trade-off method.

Selections produced by this method are shown in Table 12. To provide an idea of the best subsets chosen by the trade-off method mentioned, plots of performances on subsets of increasing size for a few of the data sets are shown in Figure 42. The optimal subset chosen for each data set is marked with a vertical line based on the F measure and number of attributes. TP and FP rates are shown in blue and red, and the F value in green. The x value of the vertical lines corresponds to the number of attributes in the selections table for these data sets.

6.7.1.5 IG Iterative Subsetting

This is a simpler version of the above method and is similar to that used by Menzies. Except in this case more attributes are selected to try to obtain better model performance later. The best subset is chosen again using the trade-off selection method mentioned. Selections differ to those of Menzies, as apart from their being larger, class labels in the data are based on a error density threshold of 40 rather than 0. As well, correlation selection is applied beforehand on the original set of attributes. Selections produced using this method are shown in Table 13. The selection for log normalised jm1 is surprisingly large, given that the first few attributes are those with highest predictive value according to the IG measure, and it

	Normal	Log normalised
cm1	mVn,hL,lnOptU,lB,cCP,mId,lC,lCC	lB,mVn,cCP,lC,hI
jm1	lT,lB,hL,hE	lT,lB,mVe,lE,mIv,mV,hE,hT,gB,hL,lC,lnOpd, hB,lCC,hV,hI,hN
kc1	hL,hI,hE,lnOptU,hN,hD,gB,lB,lC,lCC,mVe	lE,hT,lT,hB,hN,gB
kc3	lCC,hL,hD,mMS,mVd,gE,lCp	hB,cCP,lCC,lnOpt,mGv
mw1	lC,mDd,lCC,mId,gE	lC,mIv,lB,gB,dP,gE,gCmod,mVn
pc1	dP,mVd,hI,lB,lC,lCC,mId,lCp	lB,cCP,lCC,mVn,dP,lC,mIv,gCmod,mVe,hL
pc2	lCp,mVd,lC,hL,mId,hI,mVn	lC,mVd,hB,lCC
pc3	lB,lCp,lT,lCC	lB,mMS,lCC,mVn,cCP,lCp,lT
pc4	lCC,lCp,mDd,mVn,lB,gC,mVd,hL	lCC,lCp,lB,gCmod,mVd,mVe

Table 12 Average rank iterative subsetting selections, which are obtained by ranking features according to the average of the rankings obtained by four different feature ranking methods, and then selecting the best subset from the top of the average ranking using 'iterative subsetting'. Naïve Bayes is used to evaluate the subsets, performance of which, along with selected optimal number of features, is shown in the next figure. For example, the four features selected for jm1 in the table, corresponds to the location of the vertical line for jm1 in the next figure.

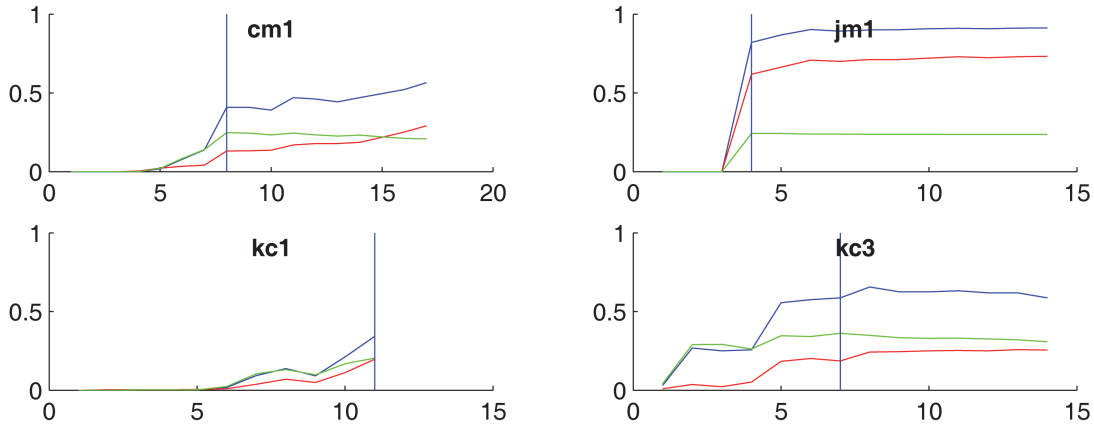


Figure 42 Examples of the iterative subsetting performance curves derived from the average feature ranking in obtaining the feature selections in the previous figure. The x-axis represents number of features from the IG ranking, and y-axis performance percentage. The vertical line marks the number of features selected at which performance was deemed optimal. (blue=TPR, red=FPR, green=F1.5)

would be expected that these initial attributes would provide most of whatever performance could be had. But according to these results, attributes much further down the IG ranking appear to have added significantly to the predictive performance of the subset.

6.7.2 Eclipse Sets

The Eclipse sets pose a different feature selection problem to the NASA sets. The main difference between them is the many more attributes present in the Eclipse sets from which to select. But there is also a larger proportion of relevant attributes as was seen in the plots of symmetrical uncertainty values in Figure 40. Due to the larger number of attributes, exhaustive search can be ruled out as an option. It would not be anywhere near feasible to search exhaustively on the 150 or so attributes in the Eclipse sets. And due to the large proportion of relevant attributes, an approach of ranking them and removing attributes at the bottom of the ranking becomes problematic. Even attributes in the lower half of the ranking may have significant relevance, and any of these could be of predictive value, if only partially.

The approach taken therefore with the Eclipse sets in order for processing time to be manageable is to use an efficient search to begin with, with a quick subset evaluation method. It turned out however that the resulting selections were too large. Subsequent efforts were then made to reduce selection size further. Two methods were tried in this regard. The first

	Normal	Log normalised
cm1	mVn,lB,lnOptU,hL,mVd	hI,mVn,lnOpdU
jm1	lT,lnOpt,hE	lT,lE,hN,lnOpd,lnOpt,hV,hB,hE,hT,hD,mV,lnOpdU,gB,lB,mIv,hL,lnOptU,hI,mVe,lC
kc1	hN,lnOptU,hL,hE,hD,hI,lB,gB,mVe,lCC	lE,lT,hB,hN
kc3	hD,gE,hL,lCp,mMS,mVd	lnOpt,hB,lT
mw1	mId,mIv,mVe,gE,lCC,cCP	mIv,lC,gE,mId,gB,lB
pc1	hI,lB,mVd,lCC,cCP,dP,gE,mId	mVn,lB,lCC,cCP,dP,lC,mId,mIv,lCp,gCmod,lnOptU,hL
pc2	hL,lCp,lC,hD,hE,hI,lT	lC,lnOpd,hT,lnOpt
pc3	lB,lnOpt,lT,lCp,lCC	lB,lCp,hV,lN
pc4	lCC,lCp,lB,mDd,gC	lCC,lCp,lB,gCmod,mVd,mVn

Table 13 Feature selections obtained from 'iterative subsetting' over features ranked by information gain. This is the same as Menzies' method except that the number of features is not limited to 3 and it is selected according to the F1.5 measure (and selection is also made on log normalized data).

selected the best attributes using the same best first wrapper method as used with the NASA sets. Because of the smaller size it becomes possible to use the more processor intensive wrapper method. The second was Menzies' IG iterative subsetting method.

6.7.2.1 Best First CFS

This is the same as first method tried for the NASA sets, in using the subset selection method CFS, but best first search is used instead of exhaustive. The results of this selection method are not shown as the selections are quite large. For both normal and log normalised data, the number of attributes selected ranges from approximately 15 to 30. While these selections could be useful in modelling, for example with SVM, it is possible that there may be more attributes in these selections than necessary which could harm predictive performance. The next sections thus turn to reducing the size of these selections further.

6.7.2.2 Second Step: Best First Wrapper

As mentioned in the NASA sections, wrapper is considered to be one of the best attribute selections methods if the purpose is to obtain higher classifier accuracy. It is not always possible to use it because of the longer time it takes to evaluate subsets, but in this case the attribute set has already been reduced by CFS, making its use feasible. As for the NASA sets, a modified wrapper is used in which the standard error rate that is returned as the evaluation of the subset is replaced with the F measure, and the classifier used is Naive Bayes. SVM could be an option as an alternative classifier, but the difficulty is knowing what parameters to

use that will enable it to provide a reliable evaluation of the attribute subsets. These parameters may vary depending on the data set. The only search direction tried here is forward. Selections produced by this method are shown in Table 14. There are some empty selections for the log normalised data. These may be avoided if a backward search were tried.

6.7.2.3 Second Step: IG Iterative Subsetting

With a selection to be made from a number of attributes similar to that in the NASA sets, Menzies' method of iterative subsetting is also applicable here. It worked well on the NASA sets in terms of producing well performing models, and so there could be reason to believe that it will do the same for the Eclipse sets. As before, the ranking method used is information gain, and subsets are evaluated using Naive Bayes. The results of this selection method are shown in Table 15.

This completes the work on attribute selection. A number of different approaches to feature selection were presented, tailored to the data sets under study. Not all of these were standard, such as Average Rank Iterative Subsetting, two step reductions, and customisation of the wrapper evaluation method to better handle class imbalance. The array of different feature selections produced for each data set will be applied in modelling experiments in the next chapter, Chapter 7, with the aim of obtaining better performance through feature selection. The results of these experiments will also indicate which of the feature selection methods applied here work best. The remaining sections cover other aspects of data preparation, namely module size and training set size.

	Normal	Log normalised
e1	A,FA,n_BS,n_ID,n_J,n_QN,n_ST,n_SVD	[]
e2	PARavg,A,CI,SC,n_B,n_J,n_PT,n_QN, n_RS,n_M	MLOCsum,NOFsum,TLOC,CS,PostE, RS,TD,n_QN,n_M
e3	A,CastE,n_J,n_PT,n_QN,n_ST,n_TryS	[]

Table 14 Best first wrapper feature selections (Eclipse) on the initially CFS reduced feature set.

	Normal	Log normalised
e1	A,NOFsum,InfxE,n_QN	A,NOFsum,InfxE,ExpS,n_QN,RS,FD,M,FA
e2	A,n_QN,NOFsum,n_PostE,n_VDE,n_B	TLOC,MLOCsum,RS,n_QN,PostE,NOFsum,FA,CE
e3	TLOC,A	TLOC,ExpS,A,RS,NumL,AA,FS,CastE,PreE

Table 15 IG iterative subsetting attribute selections (Eclipse).

6.8 Module Size

This section focuses on preparation from another perspective, that of sampling instances by size. In earlier exploratory analysis on the data sets, plots of module size distributions in Figure 22 showed that most modules were small, following a lognormal distribution. A study which focused on module size by Koru (Koru & Liu, 2005), discussed in 2.2 NASA Data Studies, saw this as a potential problem in obtaining good predictive performance. Small modules do not contain enough variation in attribute values to support discrimination. Further, larger modules which do have metric variation are too few relative to small modules for the classifier to recognize. Guided by this, Koru's finding was that training on only larger modules improved predictive performance. This position seems to have some validity, and sampling by module size seemed to be worth exploring. The principal experiment conducted by Koru involved sorting instances by LOC, partitioning them into 8 segments, and building models using the KStar classifier on each, which were validated on the same data. The trend of the performance curve for a number of the NASA sets was increasing as partition module size increased. Hence his observation that instances of larger module size were beneficial to performance.

A similar experiment was performed to see if the same result was obtained, in which Naive Bayes was used rather than KStar, on a single data set, pc4. The results of this experiment are shown in Figure 43. The box and whisker plots show the distribution of LOC from the training instances in each case or partition. The curve shows the corresponding F value obtained on each LOC based partition. The partitioning scheme of Koru's corresponds with the first 8 boxes. The other two will be discussed shortly. This result confirms Koru's, in that there is a significant increase in performance as module size in the training data increases. This seemed promising.

But the problem with the experiment, and with Koru's as well, in terms of the findings being of practical use (although Koru does not claim it be immediately so, but merely points to a potentially useful phenomenon), is that the performance is obtained from validation on data in the same partition, which is in same size range. It would seem likely that a model would not predict so well on instances outside of the partition as it has learnt only to predict on the patterns of modules whose module size is within a particular range, not to predict accurately on patterns of modules of different size. That performance diminishes when these models are applied to other data is shown in the last two columns in the figure. The second last column marks CV performance over all the data with a dot. This is the standard NB performance on

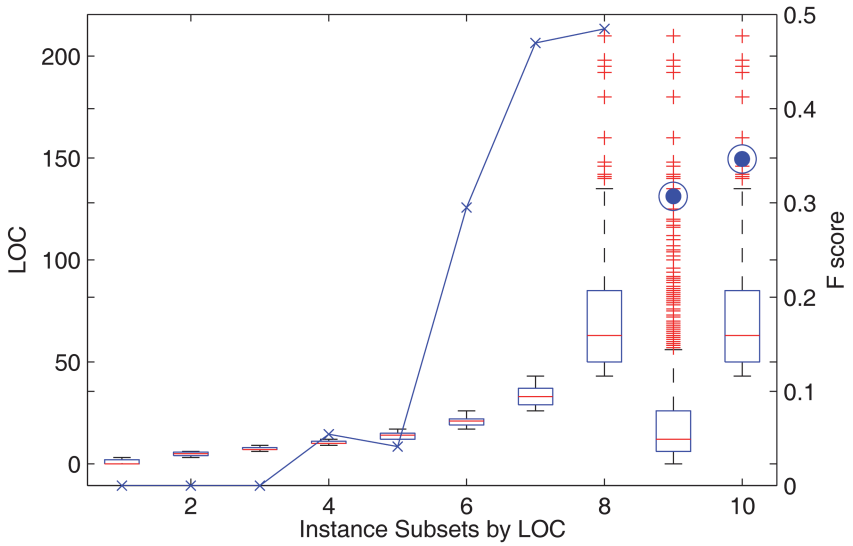


Figure 43 The effect of module size on classifier performance. As in Koru's study, modules are partitioned into 8 sets by LOC. The distribution of module size in each partition is shown in the (first 8) box and whisker plots, from smallest to largest. Naïve Bayes performance within each partition is shown in the curve. The second last entry shows 'standard' performance, marked with a dot, on the whole data set. The last entry shows performance on the whole data set on a model trained on the largest partition.

the data set. The last column marks performance on a model built from data in the 8th partition with larger module sizes, but is obtained by validating over all the remaining petitions. The 'larger modules' model in the last column offers little performance benefit over the standard NB model trained on all the data in the second last column. Because of this result, filtering out smaller modules as a preparatory step was not considered further.

6.9 Training Set Size

Mentioned in 2.8 Menzies' Studies was Menzies' finding that models can be learnt on as few as 100 instances (or 50 if sampled by class), without loss of performance. It was because of this that he concluded that there may not be much more information of predictive value in the data beyond what is contained in a small sample of the data. This finding of his was reproduced, for one of the data sets, pc4 again, based on a simple experiment. A range of increasing training set size values was first defined. For each training set size, a performance was obtained by averaging performance over multiple NB models created on different train

and test samples of the data. The training set was sampled according to the current size. The results of this experiment are shown in Figure 44. They agree with Menzies, in that a plateau in performance is reached at about 100 instances, marked with a vertical line in the figure. Note there could be some variation depending on the attributes selected. Also, this plot was obtained from Menzies' data whose class labels are based on an error density threshold of 0. The data in this study, which uses a threshold of 40, exhibits a similar plateauing effect, although it is not quite as well defined. Thus for this data the use of more training data would be warranted. As to whether training data size should be reduced as a preparatory step prior to model building, while this seems a reasonable approach to take given these results, it is possible that with the inclusion of a larger number of training instances, more predictive information could be extracted from the data to enhance performance. It is of no disadvantage at any rate to include more training data, other than requiring extra processing time, but this is minimal in the case of Naive Bayes. In the interests of pushing performance limits therefore, rather than downsample as a preprocessing step, the full data sets will be used for model development.

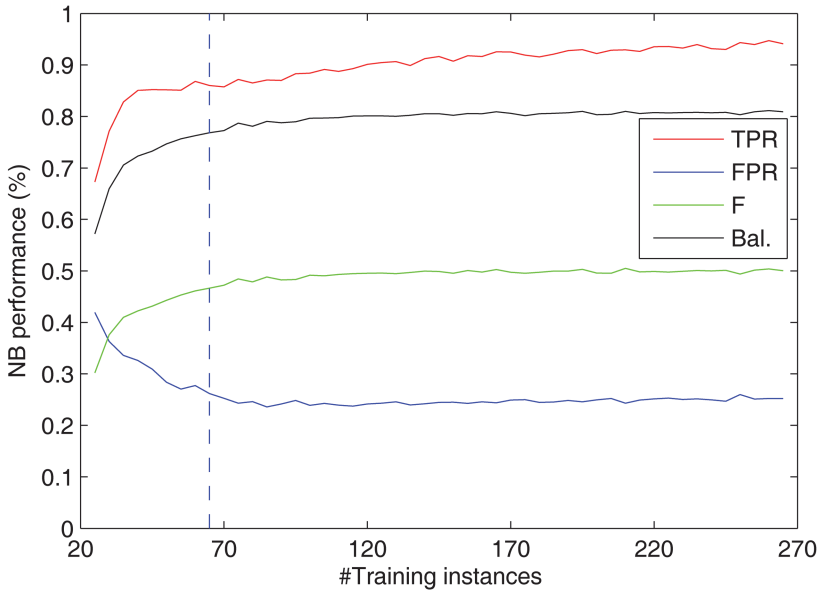


Figure 44 Naive Bayes performance as training set size is increased, confirming Menzies' finding that only a relatively small number of instances is required to reach optimal performance. This raises the question whether there is any useful information in the remaining data and how it might be exploited to improve performance.

6.10 Discussion

The main focus of this chapter has been on attribute selection. A comment could be made about class labelling and the use of an error density threshold of 40 instead 0, in that while it provides for more control over the severity of fault proneness that models detect, a drawback is that it exacerbates the problem of class imbalance. Imbalance is an issue however regardless of threshold, and from this point of view, it may not matter.

There were a range of feature selection experiments performed based on the methods available in Weka. It remains to be seen which methods and their selections give the best performance. It is worth mentioning that most of the experiments ran in a short time - with exhaustive consistency being of slight exception. This includes those for the Eclipse sets, which have a fairly large number of attributes. It seems then attribute selection need not be that time consuming a task. Nevertheless, all of the selection experiments for each data set were run in separate threads in parallel, to hasten their completion. But given short algorithm run times, not having done this would not have extended overall experimentation time by that much.

The difficulty in selecting attributes is being familiar with the different search and evaluation components, and deciding which combinations of them are most suited to the data and selection task. This is a considerable difficulty because there is little information immediately available on the various components and algorithms, and information is limited on how to interpret their parameters. Despite this, with the familiarity that was gained in working with these components in Weka, and with knowledge of the fundamentals derived from Chapter 4 on feature selection, it is felt that sensible choices were made choosing approaches to select features likely to result in useful feature sets. It must be said though that the reason there are many selection algorithms available is that none are necessarily optimal, and it is a matter to some extent of just trying different options.

Menzies used a simple selection approach, iterative subsetting. Presumably this method was chosen because various methods were tried and it was found to be among the best. If this is so, then again here, despite the various approaches used, it may also give the best performance. This could be a little surprising though, given that the method does not remove redundant attributes. This may be why Menzies chose and had success with only three attributes. Given this there may be a chance that using more attributes, with redundant ones removed, may give better results. At least this could be the case in theory. It must be said that based on some spot performance tests on CFS with Naive Bayes, that even though this

method has ideal qualities in both selecting for relevants and against redundants, performance seemed to be lacking compared to the simple ranking method of Menzies. A possible outcome, which may be likely, is that with more attributes selected than Menzies' few, as appeared in the tables presented, better performance can be obtained, but the improvement may not be that significant. Menzies may have found a practical sweet spot with his choice of three attributes for each data set. It is also possible that despite selections varying, according to the selection methods used, there may not be that much variability in performance.

One of the interesting aspects of the selection problem is that there are many features that are relevant, according to information gain or symmetrical uncertainty. This was reported by Menzies, and because of which he suggested that it was not important which particular attributes were selected, as long as that they were from taken from those most relevant. Menzies method takes only the first 3 and relies on their each providing some original information but without regard to redundancy. The tasks then is to find which of all the relevant features best complement each other in having information than enables prediction over different parts of the instance space. The methods largely relied upon to do this were the evaluators CFS, consistency and wrapper, which evaluate subsets of attributes together rather than individually as with ranking methods.

An issue not raised so far this chapter is class imbalance. It has received some attention in recent years in relation to classification, but there does not seem to have been much attention given to its impact on attribute selection. Apparently attribute selection methods can suffer the same problem as with classifiers in focusing on the majority class simply because there is much more data of that class. One selection method that does not suffer this problem is Odds Ratio, often used in text categorisation. But this is not available in Weka. The wrapper method used, with evaluation by the F measure, would be tolerant of imbalance as well. Recently new methods were added to Weka, CostSensitiveSubsetEval and also an attribute evaluator version, which make a base evaluator cost sensitive. A cost matrix is supplied specifying the costs for FP and FN errors. These cost sensitive methods could be useful in producing better selections when classes are imbalanced.

In relation to module size, though this issue has already been addressed, one possible application of the phenomenon that Koru identified of more accurate classification by training models on larger modules, is creating separate models for different ranges of module size. New instances would be classified by the appropriate model according to module size. It would then be possible to exploit the better performance from larger modules. But the

problem with this idea is that most instances are not large, so the higher accuracy would be limited to a relatively small proportion of the instances. The option of using one of the large module size models to predict on all instances of any size was also shown to be lacking, providing little performance benefit over the standard model trained on all the data.

In the next chapter, the attribute selections produced will be applied to the data sets, and models learnt using both Naive Bayes and SVM, with the aim of obtaining the best possible classification performance.

SECTION 3 - Modelling & Classification

Chapter 7 - Modelling Experiments

The aim of this research is to use machine learning methods to develop accurate predictive models of software quality, more specifically of fault proneness, from the available data sets. In the previous chapters, the two groups of data sets, NASA and Eclipse, were both explored and then prepared for model training, which is the focus of this chapter. Here, models are developed from these data sets using the supervised learning algorithms Naive Bayes and Support Vector Machine, and performance is evaluated as to their predictive accuracy. This is the crux of this line of the research, though the preparatory work covered, particularly class labelling and attribute selection, are important elements in the process of developing models.

The task of developing a model using a supervised learning algorithm is essentially straightforward. A data set is supplied to a learning algorithm, the data is processed and a model is produced, here either a probabilistic one from NB, or the discriminate hyperplane of the SVM. In order to determine how accurately the model can predict on subsequent instances that are supplied to it for classification, a validation method is used. Generalization performance may be evaluated using an entirely separate 'hold-out' set, decoupled from model parameter estimation so as to avoid bias. However, if there are limited amounts of data, or as in this case, limited examples of one particular class, the usual procedure for obtaining reliable estimates of predictive performance is cross validation (CV). This procedure was described in section 3.3 Performance Evaluation. Effectively the whole data set is used for testing, but in each loop of the procedure the model is always evaluated on data external to training. So in developing a model, what is happening actually is that a particular model configuration with associated parameters is used to develop multiple models in cross validation for the purpose of accurately approximating generalisation performance. The performance results reported in the chapter are from this cross validation procedure. Once a model setting has been identified as well performing, it may be used then to produce a single model from all the data deployed in some software quality assessment tool for example.

The above describes the basic task of developing and validating a model. One of the parameters to the task is the data set and its features from which the model is trained. It is usually desirable to first select features by some means, as was done in the previous chapter using a number of feature selection methods. The modelling experiments described in this chapter apply these various selections with the aim of obtaining better model performance. In addition to feature selection, extreme outliers are removed as was discussed in 6.1 Initial

Filtering to avoid their adversely affecting learning. As mentioned, their exclusion would have little bearing on performance, even if the outliers were legitimate instances from the population.

The performance Menzies obtained on the NASA sets with NB were listed in Table 1. It is of interest to see how the performance obtained in this chapter compares with his results. It may be possible to improve upon it through the use of more features or different feature sets, or the use of a different learning algorithm as with SVM. The results however it must be pointed out are not directly comparable due to the different error density thresholds used for class labelling, but comparison at a more general level is still valid. To confirm that differences in performance are due to this threshold difference, Menzies' approach is repeated in the course of the experiments and results similar to his were obtained. Menzies also stresses the importance of repeatability (Menzies, Ammar, Nikora, & Di Stefano, 2003). All of the experiments here are described in sufficient detail that they may be repeated and the same results obtained.

This chapter begins with a discussion on the choice of performance measures which are suitable in the case of data with imbalanced classes. Originally just the True Positive (TP) and False Positive (FP) rates were chosen, as used by Menzies. However it was recently reported that these may be inadequate when classes are imbalanced (Zhang & Zhang, 2007), and for this reason, the F measure has also been used which combines precision and recall into a single measure. Experiments are then described beginning with Naive Bayes. All the attribute selections of the previous chapter for each data set are applied to each data set. Performance is reported for all of these, and at the end of which details of the best model for each data set are presented. Following Naive Bayes, experiments with SVM are described. There are many more parameters combinations to explore with this algorithm. Again collections of results are presented, and then the best results for each data set, along with the parameters that produced them. Performance of each algorithm is compared in the discussion at the end of the chapter.

7.1 Choice of Performance Measures

An initial review of the literature on software fault prediction suggested the use of a couple of measures for performance evaluation, the TP and FP rates. These are what might be considered low level measures, or ones that are obtained directly from counts in the confusion matrix. The formulas for these measures were given in section 3.3 Performance Evaluation.

These two measures are quite useful. The TP rate Menzies calls the detection rate. That is, it is the proportion of actual positives that are correctly predicted as positive. This provides a measure of how effective a model is at recognizing faulty modules. The FP rate is the proportion of negatives that are incorrectly predicted as positive. Menzies calls this the false alarm rate. This provides a measure of error in prediction to complement the measure of correct predictions in the detection rate. In fault prediction, the false alarm rate may be equated with wasted inspection effort. The greater the proportion of the negative class incorrectly predicted positive, the more modules there are that have to be checked that do not contain faults.

Menzies has used the TP and FP rates only (sometimes with the addition of 'balance' (Menzies, Greenwald, & Frank, 2007)) in many of his papers to measure performance. However, these measures may not be adequate when classes are imbalanced, as they often are in fault prediction data. Faulty modules usually occur much less frequently than fault free modules. The core of the problem is that a low FP rate may suggest a good performance, when in fact, the number of FPs, which is the FP rate × size of the large negative class, may be relatively high compared to the number of correctly predicted positives. The problem is exacerbated with greater class imbalance.

The problem is significant only if the number of FPs relative to TPs is relevant. But it often would be. If a model predicts a much greater number of FPs than TPs, it would be difficult to regard the model as performing well, even though the rates may suggest it is. The performance measure that is also relevant here is precision, which is the proportion of FPs, to all predicted positives (TP + FP). Revealing the inadequacy of the TP and FP rates alone, it is possible to obtain the same TP and FP rates even though precision is completely different when classes are balanced differently.

In this research, we adopt the solution suggested by Zhang and Zhang (2007) and report results primarily through the use of the F measure, which is the harmonic mean of precision and recall. The standard F measure (where parameter β, mentioned beneath, is equal to 1) is given by $2pr/(p+r)$ where p is precision and r is recall. The focus of this measure is on the positive class: positive predictions relative to the positive class, and positive predictions that are incorrect relative to all positive predictions. It is thus better suited to imbalance and where the main class of interest is the minority positive one. It also has the advantage of combining two lower level measures into a single measure which facilitates easy comparison

of model performances. The F measure also incorporates a variable, β, as the weighting of recall versus precision, allowing control of the trade-off between them.

Visually, the F measure is shown in Figure 45, for β=1 and β=1.8, across the range of precision and recall. As can be seen, all three measures lie in [0..1], including the F measure. The F measure is perfect or maximal at 1, when both precision and recall are maximal. The red line marked shows the F value as precision decreases with fixed recall at 0.4. It will be noticed that when β is 1.8, the curve is more inflated on the recall side, so that for higher recall values, the F value is higher.

Another view of interest is how the F measure behaves as a function of recall and false alarm rates - the basic rates of interest. This is shown in Figure 46 for β=1 assuming a positive class size of 10%[2]. The same line as in Figure 55 is marked in this figure, to show how they correspond. An aspect of the F measure curve that will be noticed is the steep drop, nearly vertical, as the FP rate increases from 0. This is an important behavior to notice because it means that if the F measure is used to find the best performing models, it will favour models with very low FP rates, as these will have the higher F values. The reason for this steep drop

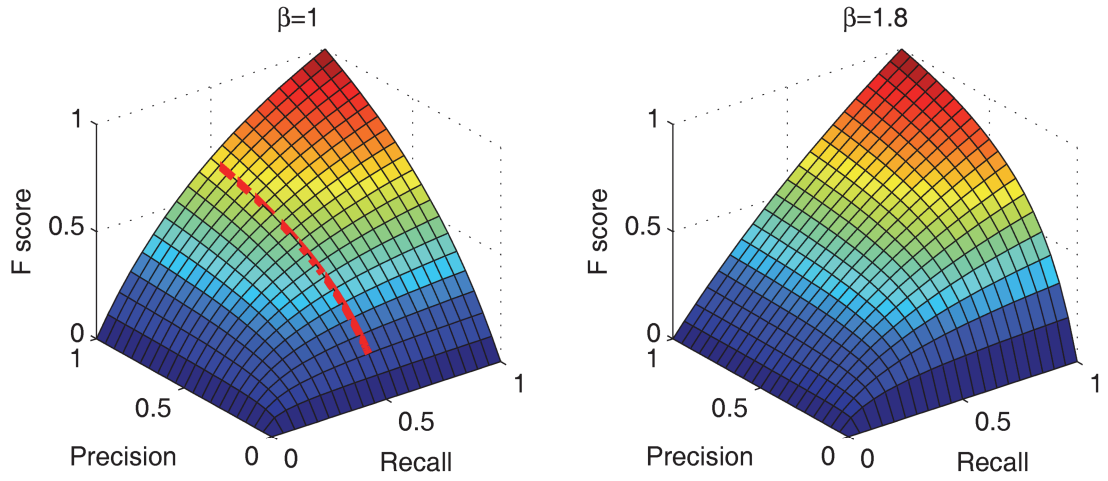


Figure 45 The F measure as a function of component measures recall and precision – the geometric mean of the two. Equal weighting between the components with β=1 is shown on the left. The F measure increases as precision or recall increase. The red line shows the F measure as precision increases from 0 to 1, with fixed recall at 0.5. Higher weighting is given to recall on the right with β=1.5, and this is reflected in the inflation of the surface with increasing recall values.

[2] The class size needs to be set in order to calculate confusion matrix values and derived measures. Otherwise, for a given TP and FP rate, any number of matching confusion matrix values and derived measures could be found.

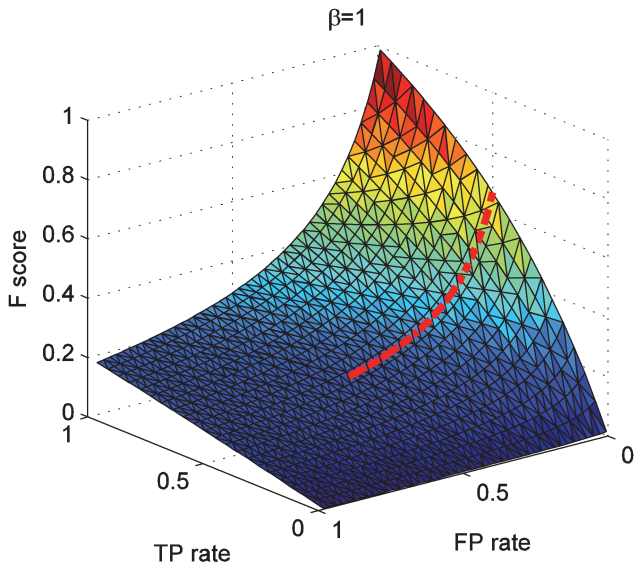


Figure 46 In the previous figure the F measure was shown as a function of recall and precision. But the measures used in this study (and others) is recall and false alarm rate. The F measure is thus shown as a function of these two measures (assuming positive class size of 10%, and with β=1). The red line of the previous feature is mapped here in terms of these measures. Notable is the rapid drop in the F measure as the FP rate increases from 0, which is due to large effect this has on precision when positive class size is small. The F measure with β=1 therefore highly penalizes higher FP rates, which motivates the use of a higher weighting on recall.

is that as the FP rate increases, even by only a small amount, the number of FPs correspondingly, relative to the positive class, is quite large (in the case of imbalanced classes), and this contributes to a rapid decrease in precision which affects the F value. This suggests the use of a higher weighting on recall when classes are imbalanced in order that less penalty is incurred with higher FP rates and correspondingly lower precision values.

To give a concrete example, consider two TPR/FPR value pairs: 90/10 and 70/7 and assume that in the context of fault prediction, any FP rate up to 10% is tolerable. FP rates above this would only be tolerable if there was a significant gain in the detection rate gained. This being the case, of the two pairs of rates, 90/10 should easily win by the F measure. But actually, the F1 values (where the suffix indicates the value for β) assuming a positive class size of 10% again, are similar, at 64 and 65. In fact, the second pair 70/7 wins, contrary to the desired outcome. If β is 1.5, the F values are instead 72 and 67, which gives the correct outcome, and the first pair scores better than the second by a margin of 5%. Taking another pair, 90/15, suggests a β value of 1.5 being the more suitable. This is because this pair gives an F1.5 score of 65, approximately the same as for 70/7, and it would seem reasonable for there to be

equivalence in merit between them. With a 20% increase in detection rate and roughly 10% in false alarm rate between the pairs, the latter is tolerable if the former is doubled. A β of 2 is also feasible, which in the first pair of rates, would further increase the difference in F values which may be desirable if higher recalls are to be preferred at the expense of lower precision and higher FP rates.

To summarise, the F measure is used to take precision into account, to avoid the problem of TP and FP rates indicating a model is well performing when in fact, due to class imbalance, precision may be unacceptably low. Various β values may be used to provide different evaluations in the trade-off between recall and precision, although 1.5 would seem to be a good choice given the level of class imbalance. It also serves as a single measure by which to compare models. The TP and FP rates however are still of interest as the lower level measures, assuming precision is at a tolerable level as guided by the F measure.

7.2 Naive Bayes Models

The Naive Bayes (NB) algorithm (see 3.1.1 Naive Bayes) is the simpler of the two learning algorithms from which models will be built. For the experiments to be performed, there are no parameters other than choices made in modifying the input data, which here are the attribute selections of the previous chapter, and log transformation. Naive Bayes is a very quick learning algorithm so execution time is not an issue for which any special treatment was required.

Naïve Bayes is the preferred algorithm for modelling on the NASA data of Menzies', one of the leading researchers in the area, whose work was covered in detail in section 2.8 Menzies' Studies. Menzies chose NB over the decision tree for its better results, and he believes NB offers an advantage over other algorithms in that it better handles brittleness of the model space that can arise, for example, due to minor changes in the data in sampling. It achieves this through multiple polling of Gaussians across features (Menzies, Greenwald, & Frank, 2007). An important finding of Menzies in relation to this algorithm was the benefit to performance in log normalising the data, and this is applied in the experiments to be described in this section. Naïve Bayes was also found to be among the best performing algorithms in 3 of 5 NASA data sets in a study on the random forest algorithm by Guo et al. (2004).

Comparison with the Menzies' NB results (Table 1) is of interest. There may be difference in performance obtained from those of Menzies due to the different class labels used. As well,

Menzies applied attribute selections from normal data to log transformed data, whereas in this study, the data was split into two 'strands' -normal (untransformed) and log - and attributes were selected separately on each. The selections are kept with the data from which they were derived. Thus whenever a log derived attribute selection is applied in these experiments, the data is also log transformed. Similarly, if the selection is derived from normal data, normal data is used for learning. Also of interest apart from the performance obtained is identification of the attribute selection methods discussed in the previous chapter that give the best performance, and whether of these any particular method occurs more often which might be recommended for use in similar situations. As well, as mentioned in the discussion of the previous chapter, it will be of interest to see whether the feature selection methods applied offer any significant advantage in performance over Menzies' iterative subsetting method, or whether, in the few features this methods selects, a practical size 'sweet spot' has been found.

Models were generated and performance obtained by iterating through the data sets, and for each data set, iterating through the set of available feature selections. For each feature selected data set, 10x10 cross validation was used to accurately estimate generalisation performance. Available feature selections identified by feature selection method and associated identifier are listed in Table 16. All wrapper evaluations were performed with Naive Bayes. An asterisk preceding feature selection identifier indicates selection on log

Abbrev.	Description	Ch. 6 Section
NASA		
C	Correlation-based selection	6.3
E.CFS	Exhaustive search, CFS eval.	6.7.1.1
E.C	Exhaustive search, Consistency eval.	6.7.1.2
BFf.W	Best First forward search, Wrapper eval.	6.7.1.3
BFb.W	Best First backward search, Wrapper eval.	6.7.1.3
Ravg.W	Iterative subsetting on avg. ranks (avg ranks, Wrapper eval. on subsets)	6.7.1.4
IG.W	Iterative subsetting (IG rank, Wrapper eval. on subsets)	6.7.1.5
Eclipse		
C	Correlation-based selection	6.3
BF.CFS	Best First search, CFS eval.	6.7.2.1
rBF.W	Reduction with Best First search, Wrapper eval.	6.7.2.2
rIG.W	Reduction with iterative subsetting (IG rank, Wrapper eval. on subsets).	6.7.2.3
IG.W	Iterative subsetting (as above but on all attributes).	(6.7.1.5)

Table 16 Feature selection methods from Chapter 6 applied to both normal and log normalized data sets to see which methods produce better performance in modelling experiments reported later in the chapter.

transformed data. Results for the NASA set are shown first in Figure 47.

It will be noticed that some bars are missing. This is usually due to empty feature selections, as some methods did not select any features. It could also be due to the prediction of all instances in validation as negatives, although this is less common. The results shown are F1.5 values with the highest values indicated in red. The bottom bar in each plot may be regarded as a baseline measure. This bar is the performance on the original feature set after removal of highly correlated features, i.e. 'manual' correlation-based selection (see 6.3 Correlation-Based Filtering of Attributes). The full attribute set could have been used, but due to the adverse effect of correlated features on Naive Bayes, it was thought the correlation based selection would make a more appropriate baseline.

There are two obvious observations that can be made about these results: best performance varies considerably between data sets, and performance across feature selection methods is generally quite similar. In the latter case, there are exceptions however, such as for mw1, where the F measures are spread over a substantial range. The pc4 data set performs the best by far, with a value at around 50%, and the worst performing is pc2, the value barely reaching 10%. Most of the highest values are situated around 30%, which is the average, and is evidently not particularly high. The poor result for pc2 is perhaps not surprising given its very low average module size of 4 LOC indicated in Figure 22 of Chapter 5, and the very low correlations found between its features and class labels as indicated in Figure 29 of the same chapter.

A central question in these experiments is whether the various feature selection methods applied improve performance over that of Menzies' iterative subsetting method, and as well how these compare to the performance of the baseline correlation-based feature selection method. To more easily address these issues, the relevant performance measures from Figure 47 are summarized in Figure 48. Looking at the baseline scores first, it would appear generally the gains by other feature selection methods can be significant. Gain varies between that for jm1, which benefits very little from feature selection, and mw1, whose performance improves substantially from it. For the other data sets, gains lie between these extremes. But mostly, there is a gain over the baseline that may be significant, perhaps around 5 to 10%.

In relation to Menzies' iterative subsetting method, denoted by IG.W, there was a suspicion that this approach might be overly simplistic as ranking by relevance does not exclude irrelevant features. Increasing size of rank subsets by this method, might just be including features with similar information. It does not provide for inclusion of subsets that might

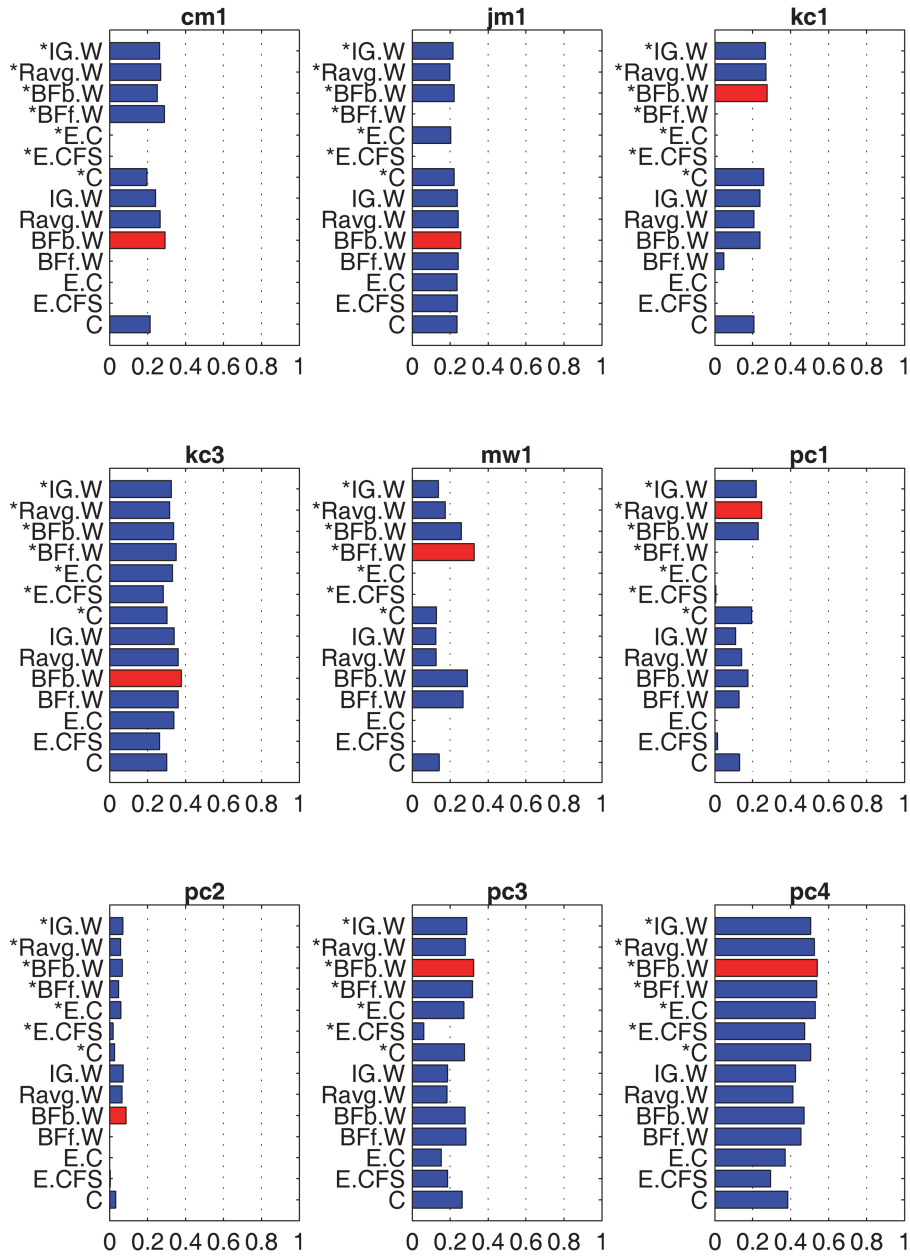


Figure 47 Naive Bayes performance (F1.5 measure) on the NASA data sets, for each of the feature selection methods listed in the previous table, Table 16 (the features selected by which were tabled in the previous chapter, Chapter 5 – Data Preparation). Selections on log normalized data are preceded with an asterisk. Best performance is highlighted red. Empty bars occur for empty feature sets. A summary of these results is shown in the next figure.

perform better with specific feature complementation. Despite this however, compared to best performance across data sets, it faired well – the various feature selection methods applied only provide a marginal if consistent improvement over Menzies' method. The only data set for which the gain over Menzies' method is significant is mw1, and though this may

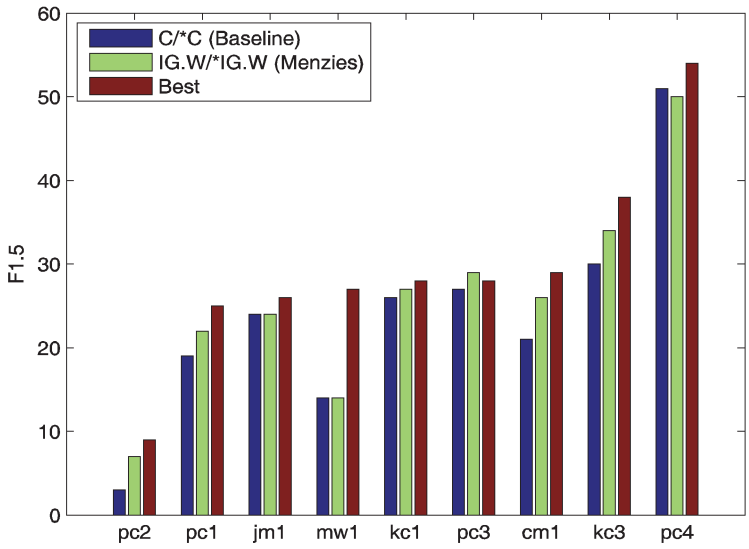


Figure 48 Summary of the previous figure comparing best performance with those obtained for the baseline (C) and Menzies' (IG.W) feature selection methods.

be explained by the different feature selections (essentially just different data), it may also be due to the higher FP rate obtained for Menzies' method which the F measure more heavily penalises when classes are imbalanced, due to reduced precision.

Turning to the attribute selection methods with reference back to Figure 47, a distinction can be made between those based on log transformed data and those based on the untransformed normal data. As mentioned, log selections were applied to log data. Menzies reported that applying log normal transformation improved performance (Menzies, Greenwald, & Frank, 2007). But the results here do not support this, at least not consistently. If log data were better, then all the best performing red bars would lie in the top half of the plots, on the log end of the range of selection methods (with asterisks). Specifically, 6 of the 9 best results were obtained from normal data and selections. One explanation for this difference in finding is the different experiment approach used by Menzies, which as described before, entails different class labels and lacks log and normal 'strands'. But it also relates to the performance measure used. When the F1 measure was used instead (not shown), bars were different to how they appear in the last figure, and almost all of the best scores were on the log end of the plots. So perhaps the interpretation of this could be that if higher precision is preferable at the expense of recall, then log normalisation gives consistently better results. Otherwise, if

higher recalls are desirable, log transformation may sometimes improve performance, although this is depends on the data set.

Looking further at what can be gleaned from the plots in relation to attribute selection methods, it will be noticed that all the best results were obtained from feature selection methods that used wrapper evaluation. This result might be expected, as the selection is optimized for better performance given that the same classifier is used for validation as in wrapper in evaluation. This tends to support suggestions in the literature (John, Kohavi, & Pfleger, 1994; Foreman, Guyon, & Elisseeff, 2003), that wrapper methods be used if feasible if attributes are being selected for optimal performance rather than purely for dimension reduction. The best methods more specifically, irrespective of whether log transformed, were BFb.W, which was selected 6 times, and BFf.W, which was selected twice. Again, this agrees with the literature with backward search being more conservative, producing larger selections that have potential for better performance. There is only one exception to this, and that is for pc1 for which the average iterative subsetting method, Ravg.W, gave the best performance, but this was only slightly better than BFb.W.

Detailed information about the best results in the plots is listed in Table 17. Apparent is the repetition of the BF.W method as the selection method producing the best performance. The number of features is usually larger than the 3 chosen by Menzies, but from the last figure this results in only marginal improvement in performance for most data sets. Not shown in the last figure are the TP and FP rates. It can be seen from the table that these vary dramatically. For jm1, the rates are 74 and 50, and for mw1, 25 and 1. This suggests substantial variability in data characteristics. The trade-off that Menzies reported between TP rate and FP rate is

	Method	Attribute Selection	#	TPR	FPR	F1	F1.5	F2
cm1	BFb.W	cCP,lCC,mVd,mVn,lnOptU,lCp,lT	7	38	8	25	29	32
jm1	BFb.W	mlv,hD,hL,lnOpdU,lT	5	74	50	18	26	33
kc1	*BFb.W	gB,mVe,lE,hB,hN,hT,lT	7	72	38	20	28	35
kc3	BFb.W	lCC,mGd,hD,hL,mMS,lCp	6	47	10	34	38	40
mw1	BFf.W	dP,lCp	2	25	1	28	27	26
pc1	*Ravg.W	lB,cCP,lCC,mVn,dP,lC,mlv,gCmod,mVe,hL	10	34	8	21	25	27
pc2	BFb.W	lC,mId,lE,dP,hI,hD,hE,mMS,mVn,lCp,lT	11	43	4	6	9	12
pc3	BFf.W	lB,cCP,mDd,mlv,gE,mVe,dP,hD,hL,lnOptU,lCp,lT	12	37	11	25	28	31
pc4	*BFb.W	lB,lCC,mlv,mVe,dP,gCmod,mVn,lCp	8	95	24	43	54	64

Table 17 Best Naive Bayes performance (from the previous two figures) on the NASA data sets. For all data sets except pc1, the feature selection method that produced best performance was best first search with wrapper evaluation (BFb.W or BFf.W).

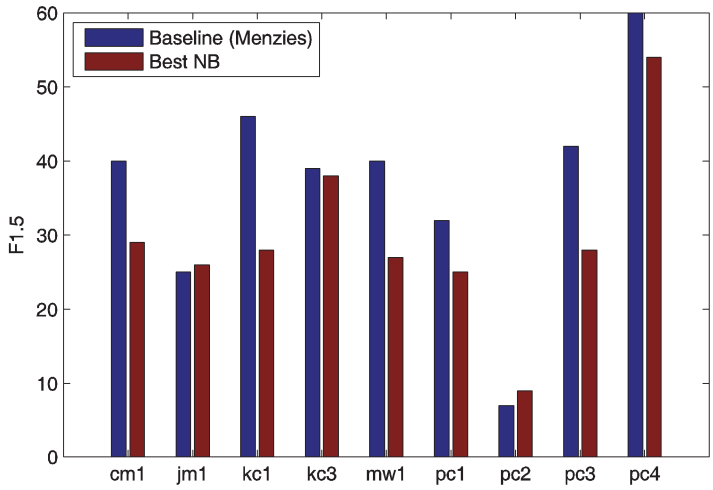


Figure 49 The best NB results of the previous table, Table 17, are compared here with the benchmark (NB) results of Menzies from Table 1 to see if they have been improved upon. But the graph shows the latter are markedly better for 6 of the 9 data sets, and this would be due the different class labeling method Menzies used, based on error count rather than error density, which evidently makes for an easier prediction task for Naïve Bayes.

also clearly evident. When FP rate is low at 1% for example, the detection rate is also low at 25% (referring to the mw1 result). The increase in recall occurs only at the expense of increased false alarm rate, as evident in the jm1 result. Because of this trade-off the F values tend to be fairly low. There is no opportunity, where detection rate increases without penalty, for the F value to increase. For pc4, the F value is higher, with a near perfect detection rate.

With the best results having been listed in the previous table for Naive Bayes, it is of interest to compare these with the baseline Naive Bayes results of Menzies. As noted earlier, the IG.W results may not be identical to those of Menzies due to the different approach taken in developing models. Comparison is made based on his reported results, listed in Table 1 of Chapter 2, as shown in Figure 49. The Menzies results are clearly better in almost all cases, and sometimes by a significant margin (cm1, kc1, mw1, pc1, pc3). The main reason for this is due to the different experiment setup in which data is labelled using a higher error density threshold of 40 rather than 0. It appears that predicting on a 0 threshold is a far easier prediction task for Naïve Bayes. To verify this is the case, that this is in fact the reason for the difference, the NASA experiments above are repeated, but using only the IG.W method, which is Menzies selection method, but with 0 threshold data. This is virtually the same as

Menzies setup, except for the partitioning of data into log and normal strands (and also automatic selection of the best subset size rather than defaulting to 3).

The best results table is shown in Table 18. It will be noticed that using Menzies' IG.W method, the features selected at which optimal performance was found are smaller in size, closer to Menzies' chosen value of 3. In the case of the largest selection, for pc1, it happens that this is the data set singled out by Menzies for special treatment in which exhaustive subsetting was used instead of iterative - perhaps to find a better performing smaller subset. Also, log data and selections win most of the time over the unmodified normal data, agreeing with Menzies' findings. A comparison between these results and again Menzies' baseline results is shown in Figure 51. Now, with the 0 threshold data, results are more comparable, as should be the case. Actually there is an improvement over Menzies' figures, significantly so for jm1 and kc3 whose F values are increased by 13% and 8% (and this could perhaps be taken further using other attribute selection methods on the 0 threshold data, not just IG.W). This is probably due to stranding and the different feature selections arising from this, the larger feature selections for 5 of the 9 data sets, and the selective application of log transformation depending on whether it was found to give better results, rather than blanket application as used by Menzies. That there is this improvement on Menzies' results is not really the issue however. The point to be made from these results is that they confirm that the poorer earlier results with Naive Bayes compared to Menzies in Figure 49 are largely due to

	Method	Attribute Selection	#	TPR	FPR	F1	F1.5	F2
cm1	*IG.W	lnOptU,lC,hE,lT	4	70	27	33	41	48
jm1	*IG.W	lT,lnOpdU,lnOpt,mV,hE,mIv	6	38	15	38	38	38
kc1	*IG.W	hN,hL,hE	3	78	34	43	52	59
kc3	*IG.W	hD,gE,hL,lCp	4	77	24	38	47	54
mw1	IG.W	cCP,gE,mIv	3	52	7	43	46	48
pc1	IG.W	lB,lC,hI,mVd,lCC,cCP,lnOptU,lnOpt	8	32	8	27	29	30
pc2	IG.W	lC,lCp,hI,hE	4	17	2	7	9	11
pc3	*IG.W	lB,lCC,lnOpt	3	65	20	38	45	51
pc4	*IG.W	lCC,lCp,lB	3	98	29	48	60	69

Table 18 That NB results were underperforming compared to the Menzies benchmarks (previous figure) is due to the different class labeling method used is confirmed here by running the same NB experiments but on Menzies labeled data (positive if error count > 0) and using only Menzies' 'iterative subsetting' feature selection method for brevity. Using the same class labeling method, the results shown are at least as good as the benchmark figures. This is more easily seen in the graph comparing the two in the next figure.

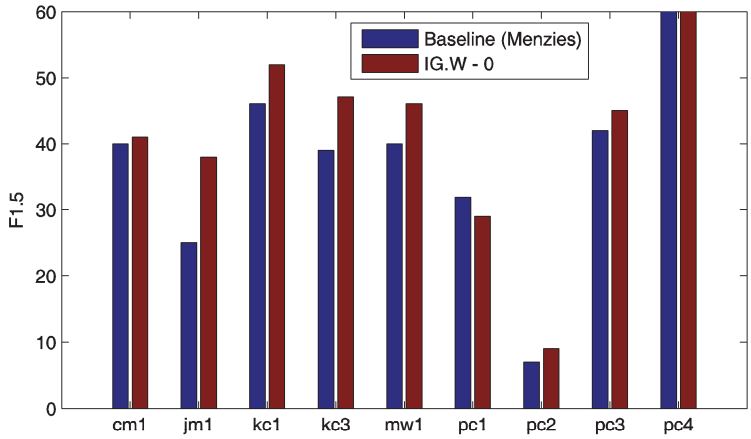


Figure 51 Confirming that class labeling method is the cause of diminished performance, the results of the previous table (shown above as IG.W – 0) are compared to the baseline, and performance is now comparable (and even significantly better for 4 of the data sets).

the different error density threshold used for labelling instances. Labelling instances using the threshold of 40 rather than 0 as they are in this research, it would seem that it can be expected that Naive Bayes performance will be lower.

Turning to the Eclipse data sets, performance with Naive Bayes is somewhat similar. The feature selection methods used differ though. Plots of F1.5 scores for each data set are shown in Figure 50. The pattern immediately apparent is that performance is virtually the same regardless of feature selection. This might suggest the nature of the data is different to that of the NASA sets. Or it may be that with so many features, many of which had similar relevance (according to Figure 40 of the previous chapter), reducing them down to a small set

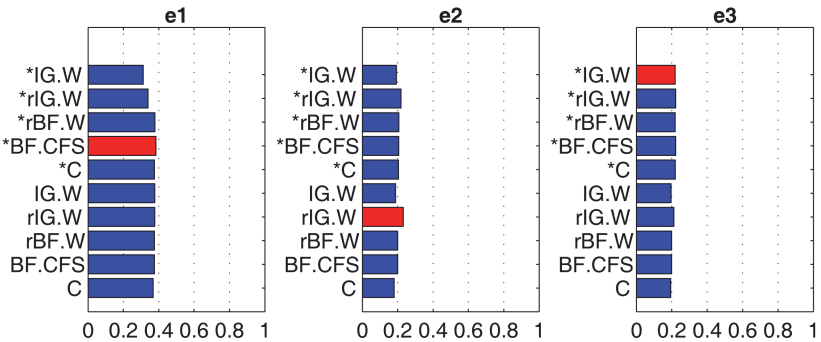


Figure 50 Naive Bayes performance (F1.5 measure) on the Eclipse data sets, for each of the feature selection methods listed for Eclipse in Table 16.

	Method	Attribute Selection	#	TPR	FPR	F1	F1.5	F2
e1	*BF.CFS	NOFsum,A,CL,FA,FS,InfxE,PostE,TE,TD,IE, M,n_A,n_BS,n_CE,n_DS,n_FD,n_ID,n_InfxE, n_J,n_NumL,n_QN,n_ST,n_SVD,n_VDF	24	68	39	30	38	45
e2	rIG.W	A,n_QN,NOFsum,n_PostE,n_VDE,n_B	6	69	31	16	23	30
e3	*IG.W	TLOC,MLOCsum,ExpS,InfxE,FOUTsum, A,RS,VDS,NumL,LfS,VDF,ST	12	46	22	16	22	27

Table 19 Best Naive Bayes performance for the Eclipse data sets.

results in much the same information. Details of the best performance are listed in Table 19. The best performing data set by far is e1, with an F1.5 value close to 40%. The other two are roughly half this, near 20%. Compared with the NASA sets, the average score for those was 29%. The average Eclipse score is 28%. So on the whole, model performances between the NASA and Eclipse sets are similar. Note that for e1 and e3 the number of features is not small, and it may be possible to choose smaller selections which give virtually the same performance. Also, because of the similarity of performances, there is little point in comparing best selection methods, and the use of 'manual' correlation-based selection could be considered.

7.3 SVM Models

The SVM algorithm, introduced in section 3.1.2 Support Vector Machines, is based on the statistical learning theory of Vapnik, and serves as a very different way by which to produce predictive models from the data compared to Naïve Bayes. While Naive Bayes is a simple and elegant solution to the predictive problem, SVM also has features that make it of appeal. It is an algorithm that was developed more recently, and has become a popular choice due to its recognized ability to perform well in various applications. A distinguishing feature is its ability to automatically find the optimal balance between model capacity and training set error to maximise generalisation performance. The algorithm has few parameters, and there is a conceptual simplicity in it, in that learning is reduced to finding a linear decision boundary from training points mapped into different spaces where separation between the classes might be more easily found. As another algorithm to complement the probabilistic approach of Naive Bayes, SVMs would seem a sensible choice.

In the context of the literature surveyed, Elish & Elish (2008) chose SVM to develop predictive quality models on the NASA data for a number of reasons: it is independent of feature space dimensionality (both number of features and instances), it finds a global

optimum solution, it is robust to outliers by varying the cost parameter C, and it is able to model nonlinear functional relationships that can be difficult to model with other techniques. SVM performance was compared to that of eight other classifiers, and the overall conclusion reached was that it is at least as good if not better than other classifiers. Also considered a suitable algorithm for software quality modelling, a study by Xing et al. (2005) applied SVM to a small medical imaging system data set, and achieved relatively good performance.

Its suitability could also be considered in light of the class imbalance present in the data. The literature suggests than when imbalance is moderate, SVM will in general perform favourably compared to other classifiers. In the more extreme case of imbalance, there have been contradicting findings, or at least not consistent ones. The imbalance in the data sets under consideration is verging on the extreme with approximately 7% positives on average. In (Foreman, Guyon, & Elisseeff, 2003), in the domain of text categorization, as class imbalance increased there was more variability in the F measure. This suggests SVM can still perform well on highly imbalance data. Robustness of SVM to imbalance was found more conclusively in (Japkowicz & Stephen, 2002). However, the more common view is that SVM does suffer performance loss when imbalance is extreme. There are a few reasons given for this (Akbani, Kwek, & Japkowicz, 2004) which are mentioned briefly. The first is that the ideal decision boundary is some distance away from the positives, which SVM by default lacks the bias to find. The boundary tends to be closer to the positives which results in a lower detection rate. The use of a soft margin may exacerbate performance loss because it gives more importance to margin maximization than penalization of error (particularly positive error) and this tends to result in a large margin in which most instances are classified negative. Imbalance has also been found to cause imbalance in the support vectors, although this might be offset to some degree in positive α_i having higher values perhaps explaining SVM's tolerance to moderate imbalance. Methods have been proposed that may assist SVM in handling imbalance. These fall into two approaches, data pre-processing approaches, namely over-sampling and under-sampling (also multi-classifier and bagging), and algorithmic approaches. An example of the latter approach is to have separate cost constants for each class, with the cost for the positive class set at a higher value (see for example (Xing, Guo, & Lyu, 2005)). Neither of these approaches are used here, as SVM was found to perform satisfactorily without them.

There are a couple of variables in building models with SVM: algorithm parameters and selected features. There is the cost penalty parameter C for the main SVM algorithm. This is

a weighting on the errors that arise from points lying on the wrong side of hyperplane corridor boundaries. If the penalty is large, then there is less tolerance for error, and the resulting decision boundaries tend to produce a so called hard margin classifier. If the penalty is reduced, more error is permitted, allowing closer fitting to the data, and the boundaries tend to be more generalised, this being referred to as soft margin classification. The other main parameter of the SVM algorithm is the kernel. The kernel encapsulates the dot product of φ-mapped feature vectors, $\varphi(x_i).\varphi(x_j)$. It transforms these vectors from input space to feature space in which learning and linear classification is made. The transformation applied allows for non-liner decision boundaries, in the mapping of the linear hyperplane in feature space back to instance space, and also can allow linear separation to be more easily found as feature space is often of higher dimensionality than instance space. Various kernels provide for different mappings, some enabling better separation and classification performance than others, depending on the data set. In this work, three standard kernels are used:

- Linear,
- Quadratic (or polynomial of degree 2), and
- Radial Basis Function (RBF).

Fortunately, easing the problem of the number of parameter combinations to explore, the only kernel parameter to deal with use RBF's γ.

The other variable to model construction is feature selection. For Naive Bayes many selections were used because the algorithm takes little time to run, and Naive Bayes often benefits from reduction due to its requirement that features be conditionally independent given class. In the case of SVM, there is an issue of algorithm processing time, which is often much longer than for NB. As well, SVM is known to perform well when there are many features, not suffering the same dependence problem that NB does. For these reasons, only a subset of selections are used:

- All attributes: normal and log normalised data
- Correlation selection: normal and log normalised data
- Best selection from Naive Bayes results

This amounts to 5 feature selections altogether. It might seem unnecessary to include an all attributes selection given that correlation selection would seem to be a better choice with redundants having been removed, but as it turned out in the experiments, the use of all features surprisingly often gave superior performance to other selections.

Because of the longer running time of SVM, an effort was made to have the experiments take advantage of multiple cores and multiple machines. Weka by default only runs algorithms in a single thread. For these experiments, in order to achieve parallel distributed execution, each experiment, involving 5 cross-fold evaluation on a particular parameter combination, was run in a separate job in Matlab (with calls to Weka) with a specified timeout of 30 minutes. At the end of execution of a job, results were saved to file. Upon completion of all jobs, files were processed to combine them into a format suitable for presentation of results. Even with execution in parallel, the experiments were time consuming, particularly the larger data sets, with total time running into many days. Part of the reason for this was the exhaustive method used to explore parameter combinations. An alternative that could have lead to some reduction in processing time would have been the use of a grid search, which relies on heuristics to guide the search over the space of parameter combinations into better performing areas avoiding those that perform less well. This approach has though more advantage when there are more parameters, and with only C and in the case of RBF, γ, the benefit in reduced processing time by grid search may not be that great. As well, most of the NASA data sets are not that large. The ones that were by far the most time consuming were the largest, jm1 with roughly 11,000 instances and the Eclipse sets with roughly 7000-11000 instances.

Results are presented for each of the kernels in the sections below, and then in the discussion section the best performances are listed, across the kernels, for each data set. Performances of SVM with NB are also compared in the discussion.

7.3.1 Linear Kernel

Beginning with the linear kernel, the only parameter other than the linear kernel itself is C, to which 7 values were assigned: 10000, 100, 10, 5, 1, 0.5 and 0.05. These values allow the determination of the effect on classification of both hard margin and soft margins. Models were generated with this kernel by iterating through data sets, and then through the 5 feature sets, and for each data set so obtained, applying SVM with 5 cross fold validation to obtain generalisation performance. The total number of models generated by this method is 12 data sets * 5 feature selections * 7 C values * 5 CV, which gives 2100 models.

Results of the linear kernel experiments are shown in Figure 52. These are the F1.5 scores on the average TP and FP rates from cross validation for each parameter set. Parameter variables are shown along the X and Y axes. The X axis represents attribute selections beginning with

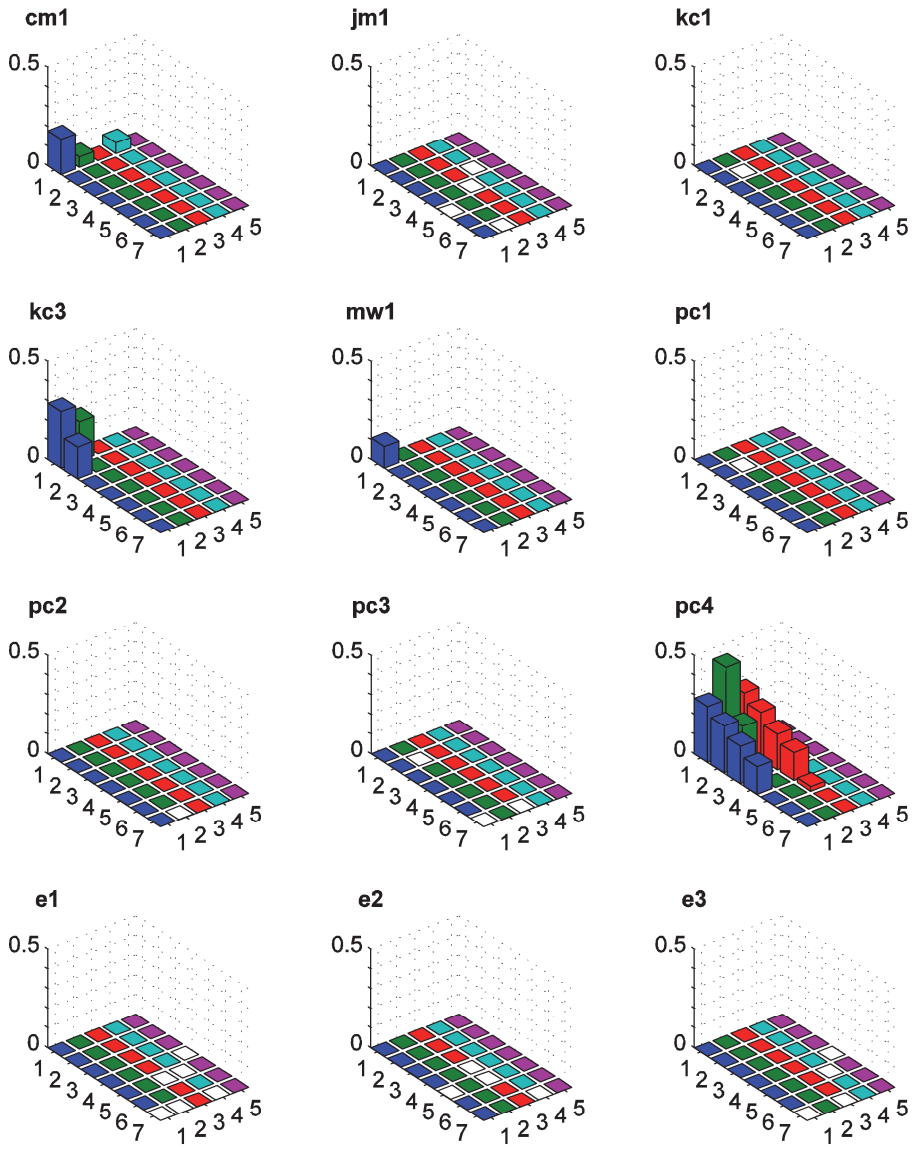


Figure 52 SVM performance (F1.5) with the linear kernel. Axis values 1..7 refer to C values, and 1..5 to features selections. White boxes indicate that SVM timed out. These results are poor with most models giving a 0 F measure, and this is due to SVM predicting all instances as negatives.

all features, and the Y axis the 7 C values mentioned above, where 1 is 10000. Where cross validation timed out, bars are coloured white.

Most of the F values are 0, as indicated by the many bars without any height. This indicates that SVM was unable to find any separation at all for many of the C values. This is probably occurring because all values are predicted negative, which is not surprising given high class

	Attribs.	C	TPR	FPR	F1	F1.5	F2
cm1	all norm	10000	17	4	18	18	18
kc3	all norm	10000	25	2	32	29	27
mw1	all norm	10000	13	3	10	11	11
pc4	all log	10000	39	2	49	45	43

Table 20 Best results for SVM linear, which are only available for only a 4 of the 9 data sets, but for these, hard margin on all features performs best, and false alarm rates are consistently very low.

imbalance. Clearly the best performances are given for pc4. Some of these are quite good, approaching an F value of 0.5. The only other data set that appears reasonably well performing with the linear kernel is kc3, which is at about 0.3. Overall, F values appear to be better with a hard margin. In terms of feature selections, better values were obtained on all features rather than the smaller subsets. While log normalisation of features did not appear to be of benefit for most of the data sets, for pc4, it improved performance significantly.

Detailed information on the best performance for data sets for which a non-zero result was obtained, are listed in Table 20. TP and FP rates are lower than those obtained with Naive Bayes. These are not necessarily worse though, as reflected in the F values which are comparable. They are just at a different point in the trade-off between the two rates, where detection rates are lower but precision is higher. This could be a useful characteristic of the SVM models if they were to be combined for example with the NB models.

7.3.2 Quadratic Kernel

Experiments with the quadratic kernel are very similar to those for the linear kernel. The results with this kernel are shown in Figure 53. It will be noticed at a glance that there are many more tall bars, indicating that the quadratic transformation had more success in finding separation between the classes. Again pc4 and kc3 have performed relatively well, as has cm1. But the surprise is pc1, for which the values are quite high, comparable with those of pc4. Looking at the C values, generally the hard margin or higher C values perform best, and performance tends to drop off as the margin is relaxed. This was the case with the linear kernel as well. Looking at the attribute selections, performance seem to be reasonably consistent between 'all features' and 'correlation selection'. The smallest selections represented at the back row of the plots (i.e. where X=5) (those found to perform best with

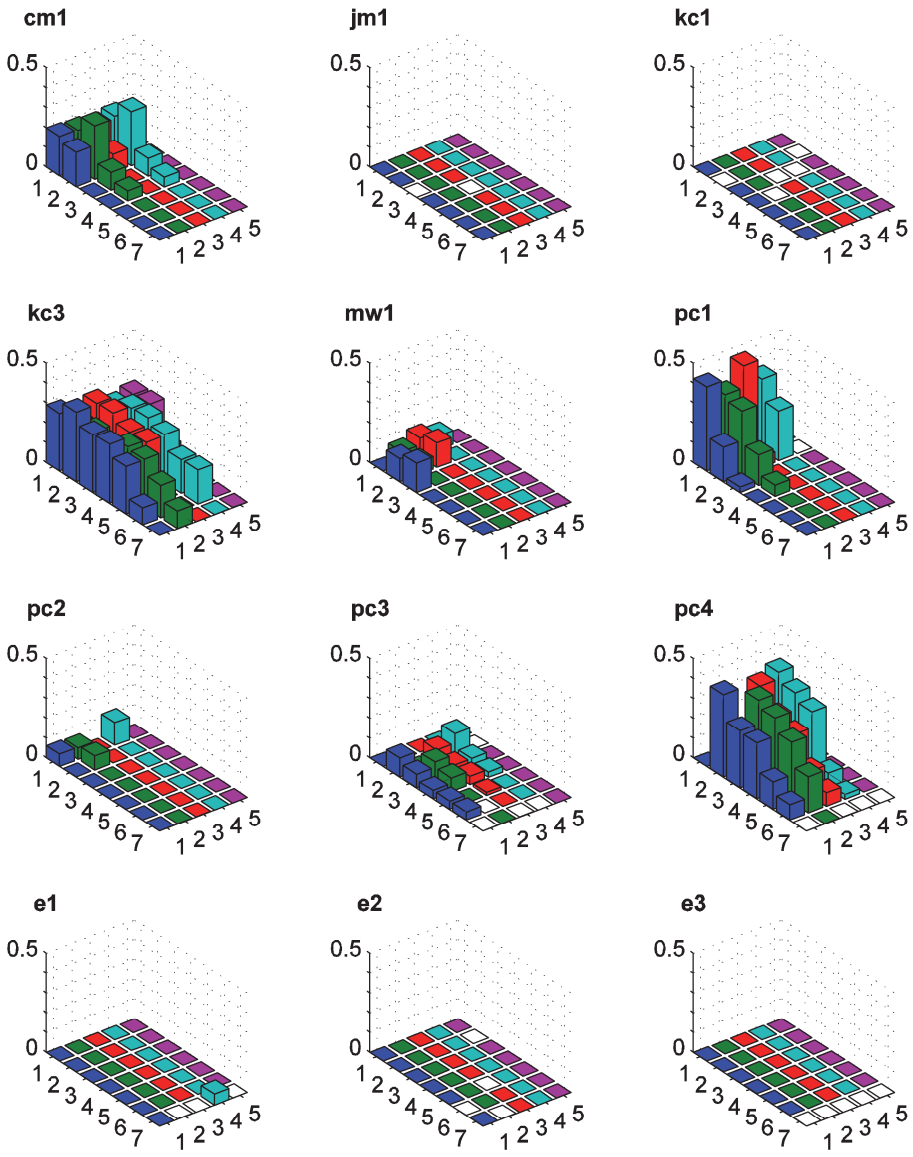


Figure 53 SVM quadratic results in which compared to the linear kernel results, there are more non-zero performing models, although SVM still has difficulty with a number of the data sets with this kernel.

Naive Bayes), have performance that would appear mostly to be obscured by and less than that of other feature selections.

Detailed information on models with the best F1.5 values for each data set are listed in Table 21. This is again only for those data sets for which at least one F value exceeds 0. As with the linear kernel, larger selections are preferred. There really is not much difference between

	Attribs.	C	TPR	FPR	F1	F1.5	F2
cm1	all log	100	30	5	25	27	28
kc3	all norm	100	34	5	35	35	35
mw1	all norm	10	13	1	17	15	14
pc1	csel norm	10000	53	5	42	46	48
pc2	csel log	10000	13	0	11	11	12
pc3	csel log	100	9	1	14	11	10
pc4	all norm	100	37	2	45	42	40
e1	csel log	0.05	4	0	7	6	5

Table 21 Best results for SVM quadratic, in which unlike the linear results, half of the data sets performed best with correlated selected features rather than all features (although looking at the previous figure performance across feature selections is similar), and soft margins are preferred here in most cases. Low false alarm rates remain.

'all' or 'correlation', or between normal or log data going from the plots, so there is not much to be read into best attribute selections. Larger C values are again preferred, and again the TP and FP rates are low.

7.3.3 RBF Kernel

With the addition of the RBF kernel parameter γ, the number of parameter combinations increases considerably compared to those for the previous kernels. Ranges used for γ in the literature include 0.5 to 2 (Yu, Debenham, Jan, & Simoff, 2006), and 0.01 to 10 (Srinivasa, Venugopal, & Patnaik, 2006). Based on this and other reported ranges used, and some other work done with the RBF kernel, the range used here is similar, from 0.05 to 3. Over this range 9 values were selected, 0.05, 0.1, 0.2, 0.3, 0.5, 0.7, 1, 1.5 and 3. This would seem to be a reasonable selection to cover any variations possible in the models produced by the RBF kernel. With these 9 values, the number of models for the previous kernel is multiplied by this figure to give 2100*9 = 18,900. Consequently these experiments took much longer to run and there were many more results to compile and present. Results are shown in a similar fashion to the preceding bar plots, except that rather than a single X value for each attribute selection, there is an X value for each γ value within each attribute selection. This stretches the plots along the X axis. Because of the larger size of the plots only a few are shown in Figure 54 to give an idea of the results obtained.

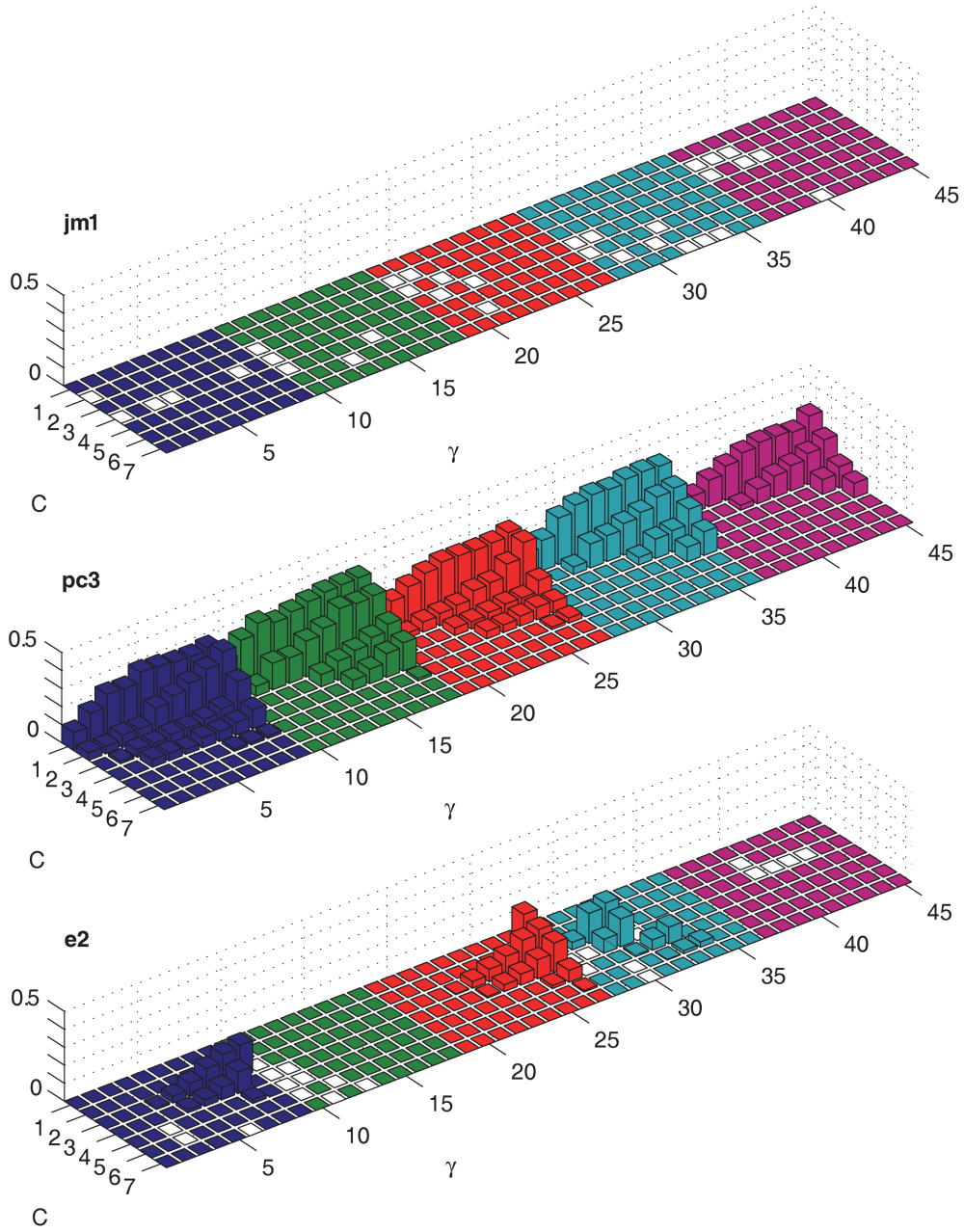


Figure 54 SVM performance with the RBF kernel for a small selection of the data sets. Unlike the plots for the other kernels in previous figures, while the C axis remains, the feature selection axis is replaced here with the kernel parameter γ. The five different feature selections used are incorporated into the plot by repeating (C × γ) along the γ axis, each coded by colour. In these results two extremes are seen in jm1 and pc3, where in the former, again no separation between the classes is found. Feature selection would not appear to make much difference for pc3 but it does for e2.

For jm1, still no separation is found, this data set seemingly proving intractable for SVM. Results for pc3 are better than those obtained using either of the previous two kernels. For e2

	Attribs.	C	γ	TPR	FPR	F1	F1.5	F2
cm1	all log	100	1	30	3	32	31	31
kc1	all log	100	3	6	1	9	8	7
kc3	all norm	10000	0.05	44	5	42	43	43
mw1	csel norm	10000	0.05	25	2	22	23	24
pc1	all norm	10000	0.3	47	4	42	44	45
pc2	all log	10000	0.2	13	0	14	14	14
pc3	all log	10000	1.5	37	6	33	35	36
pc4	all log	10000	0.3	58	5	54	56	57
e1	all norm	5	1	10	1	18	14	12
e2	csel norm	100	3	19	2	23	21	21
e3	nbsel	1	1	1	0	1	1	1

Table 22 Best results for SVM RBF, for all data sets but jm1 for which all models gave a zero F measure. Variation exists across data sets for attribute selection, C, and γ, although choice of one parameter value over another might be due to only slight differences in performance.

a result is registering when previously one was not. Again with the RBF kernel there are many zeros in performance, and each data set has a different pattern of results.

Details of the best performance for each data set are listed in Table 22. This table is larger than previous ones because there were more data sets with non-zero scores. It also contains an additional column for γ. Many of the best F values were from log selections and data, although as before, there usually is not much difference between log results and the nearest normal ones. There are a couple of lower C values for e1 and 3. Values for γ are mostly at the upper end of the range, except for kc3 and mw1, whose values are at the minimum. Based on this variation it is not possible to identify a value for γ which works well for all data sets. As for the previous SVM results with the other kernels, the TP and FP rates are again low.

What is especially notable about this table, is that the performances for each data set are as good as any other kernel, and are usually better. It serves then as a table of best performances obtained with SVM. There is one exception however, for pc1, for which RBF gives a value of 0.42 and quadratic 0.46, but this is a small difference and would probably not warrant making a special exception for it, in using the quadratic kernel instead.

Excluded from the table is jm1, for which no non-zero result was obtained. This is also almost the case for e3, which produced an F value of only 1%, which means the model is effectively useless.

7.4 Discussion

There were two sets of experiments carried out, on the NASA and Eclipse sets, using two classifiers Naive Bayes and SVM, and a number of different features selections without and with log normalisation. In the case of SVM there were also algorithm parameters, C, kernel, and kernel parameters in the case of polynomial and RBF kernels. In the case of the Naive Bayes experiments, processing time was very manageable, being a fast learning algorithm, and with the only variable being feature selection. While for SVM, the number of models generated was quite large and the task was a time-consuming one. There is something of a contrast in scale of the experiments between the two classifiers. There were however many more attribute selections tried for Naive Bayes.

There was a fair amount of information produced and discussed, and it would be worthwhile to summarise. For the NASA sets and Naive Bayes, the main points were:

- The average F1.5 values across all NASA sets was approximately 30%. The maximum was 54% for pc4, and minimum 6% for pc2. The latter is very low due to the exceedingly small positive class size of 0.2% (down from 4% before applying the error density threshold of 40, which reduces the number of positives). The average of 30% would not seem to be very favorable in terms of the predictive ability of the models, and this is for an F value that is more tolerant of higher FP rates with β at 1.5-.
- Compared to Menzies' results, the Naive Bayes results reported here were lower. This was shown to be due to the class labelling threshold of 40 used instead of 0. The latter when used to replicate Menzies' experiments gave results at least as good as his, and in a few cases significantly better performance. The improved results over Menzies when using the 0 threshold could be due to selective use of log transformation, feature selection made on log data, larger feature selections and possibly removal of extreme outliers from the data sets.
- The attribute selection method that consistently performed better than others across data sets, was found to be wrapper evaluation with best first search. Backward search was found to be better in most cases. This is not a surprising result, as wrapper evaluation selects features, not according to some heuristic or rule, but by evaluating subsets according to the performance of derived models. Based on this result it may be possible to provide a recommendation for the use of wrapper evaluation in feature selection. Although against this is more time consuming subset evaluation, but in these experiments with efficient search strategies employed, running time was quick.

However, performance obtained using wrapper selection methods was often, but not always, only marginally better than those produced using Menzies selection method IG.W. Menzies method may not be recommended though because it may be simplistic in ignoring redundancy, and for not being as consistently good as wrapper methods.

- Further to the previous finding of wrapper methods performing usually only marginally better than Menzies' IG.W method, the larger sized selections from wrapper selection were usually only marginally better than the smaller selections of Menzies. That is, for the NASA data sets, the use of a larger number of features beyond the few used by Menzies does not usually provide a significant improvement in performance (at least with the 40 threshold data).
- There was often much similarity between the results of different feature selections, although BF.W often had the edge over other methods and were consistently good.
- The baseline performance obtained from correlation selected features was usually not greatly improved upon through used of other selection methods and corresponding further reduction in feature set size. However, there were some larger gaps in these results, and overall, the benefit obtained by using other selection methods with further size reduction was substantial enough to warrant the use of other feature selection methods.
- Menzies reported that better performance was obtained by log transforming the data before training. Log data in these experiments was not found to provide consistently better performance according to the F1.5 measure. This may be due to the different class labelling error density threshold used. It seems also to depend on the model performance measure used, with the F1 measure preferring log models more often.

In relation to the Eclipse sets with Naive Bayes:

- Average model performance was similar to that for the NASA sets, being close to 30%. e1 was by far the best performing at 40% with the other two sets being at only roughly half this.
- All attribute selections were almost identically performing, with less variation than for the NASA set. An exception was IG.W for e1 which was lower than other methods by a surprisingly large margin. Similarity in performance might relate to the observation in the previous chapter on feature selection, that many of the features in the Eclipse sets have similar relevance by the Symmetrical Uncertainty score. Features selected

might have the same information content. This could perhaps suggest the use of other feature selection methods.

- Log data again did not perform consistently better than normal data.

The findings on SVM were rather fewer than for Naive Bayes, partly because there were no existing results in other research to compare them with. These may be summarised as follows, for both NASA and Eclipse sets:

- The RBF kernel produced better results than either the linear or quadratic kernels. The best results with the RBF kernel were generally obtained with higher values for C, with harder margin. There was no particular value or sub-range of γ that occurred most often, with best values occurring across the range of γ values tried depending on the data set. The average RBF F1.5 value was approximately 30% (excluding e1 which had a value of 1%), which is the same as for Naive Bayes for both NASA and Eclipse sets.
- There were many zero results across parameter combinations, regardless of kernel. In these cases the model is defaulting to negative classification for all or most instances. This reflects difficulty had by SVM in placing the decision boundary so as to find some separation between the classes.
- When all features were used, without any features selected, the performance was usually better than with only subsets of features. This is in line with the view that feature selection is less important for SVM compared to NB. It may not be necessary at all in the case of the data sets under study.

Returning to Naive Bayes, the average F1.5 performance was only 30%, which is not a satisfying result. One might reasonably question whether models with these performance levels would have much practical value. However, Menzies reported seemingly better results than these, an average detection rate of 71% and false alarm rate of 25% (Menzies, Greenwald, & Frank, 2007). This result was based on NB models across all data sets except jm1 and kc1 which are of a different format. It was noticed that these two data sets had much higher false alarm rates in the NB results, perhaps because they are noisier higher level data sets, and this might have been more the reason for their exclusion. So it was thought worthwhile to repeat the Menzies experiments on the selected data sets, and compare this result with the NB results obtained in the experiments reported here. The table of relevant results is shown in Table 23. Only performance for selected data sets is shown. Average performance is shown at the bottom of the table. Note there is one data set, kc4, that Menzies

	Baseline (Menzies) (0 threshold)			Best Naïve Bayes (40 threshold)		
	TPR	**FPR**	**F1.5**	**TPR**	**FPR**	**F1.5**
cm1	68	28	40	38	8	29
kc3	68	28	39	47	10	38
mw1	55	14	40	25	1	27
pc1	51	18	32	34	8	25
pc2	78	14	7	43	4	9
pc3	80	35	42	37	11	28
pc4	98	29	60	95	24	54
Mean:	**71**	**24**	**37**	**46**	**9**	**30**

Table 23 Comparison of Menzies' mean "80-20" baseline result (on selected data sets) with the mean of best NB performance. NB performance lacking probably due to labelling threshold.

used that was not available for this research, but being only a single data set it should not interfere much with the comparison.

The roughly 80-20 result can be seen in the averages of the first two performance columns. The mean F1.5 value from Menzies' results on these data sets is 37% (an improvement on the current experiment average over all data sets, but with a different labeling threshold). For the NB experiments here, the TP and FP rates are lower compared to Menzies, with higher precision/lower false alarm rate being gained at the expense of detection rate, and the average F1.5 value on the selected data sets is still 30%. The removal of jm1 and kc1 therefore had no positive effect on the average. The conclusion from this is that neither Menzies' results with 0 threshold data, or the results here with 40 threshold data, when taken across only the selected subsets, are particularly good. Menzies' view of his results becomes less tenable when classes are highly imbalanced, due to the lack of precision at such a large FP rate.

So far in relation to comparison of learning algorithms NB and SVM, only the averages have been compared, and these were found to be the same at approximately 30% (excluding jm1 and e3 for SVM which scored 0 and 1 respectively with F1.5). The individual results for each data set, for each algorithm are compared in Figure 55, by F1.5 values. Naive Bayes performs markedly better than SVM for jm1, kc1, e1 and e3. Conversely, SVM performs markedly better than Naive Bayes only for pc1. For most of the remaining data sets, SVM performs slightly better than NB, although NB wins a couple of times.

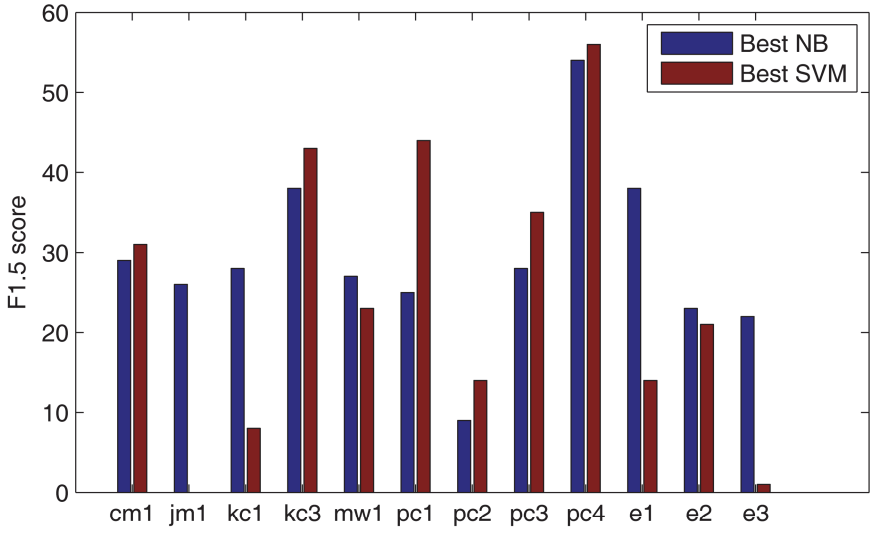


Figure 55 Comparison of the best NB and SVM performance for each data set, in which the better performer of the two varies depending on data set and neither could be said to be the better performer overall.

Another comparison can be made of the TP and FP rates of the two algorithms. This is shown in Figure 56. Apparent are the very low FP rates of SVM, and correspondingly low TP rates. By comparison, while there are also some NB results in this same area of trade-off where the FP rates are low, about half of the data sets have much higher FP rates, accompanied by higher TP rates allowed by the trade-off. Overall across all NB points, the FP rates are higher, and this could be of use if models of the two algorithms were to be combined, to include models at different places in the trade-off spectrum.

The most likely reason for this difference in performance between the two learning algorithms is SVM's tendency when classes are imbalanced to position the decision boundary close to the positives as discussed in 7.3 SVM Models. This occurs with the higher values of C that gave the best performance, rather than lower values which resulted in poorer performance due to the higher importance given to a wider margin leading to negative classifications predominating. The effect of this tight decision boundary near the positives is to limit the positive prediction region which lowers the detection rate, and at the same time, lowers the rate at which negatives are predicted positive, the FP rate, as is evident in the plot.

Curiously, this contradicts the findings of the study by Elish and Elish (2008), in which in applying SVM to the NASA data sets, SVM was found to produce higher detection rates than other algorithms. This may be due to the data sets in that study containing more positives

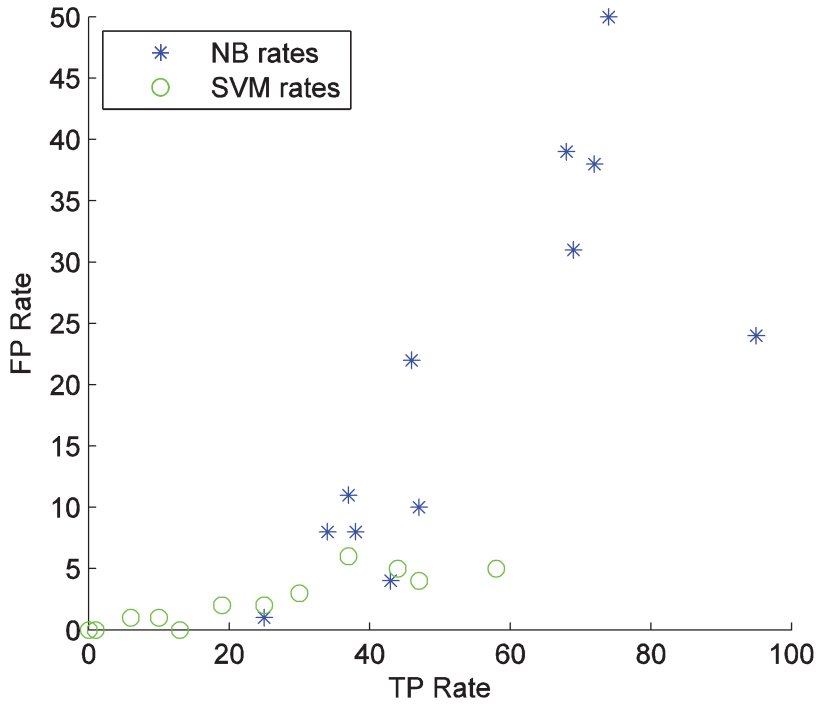


Figure 56 Comparison of performance between NB and SVM across data sets as for the previous figure, except here in terms of TP and FP rates rather than F measure. Apparent are the very low FP rates of SVM, all under the ideal threshold of 10%, which is desirable if wasted inspection effort is to be avoided. NB points are generally further to the right having a higher detection rate, but this is at the expense of in many cases a dramatically higher false alarm rate.

because of a different threshold for class labeling, and also performance reported on an RBF kernel with different parameters chosen for its higher detection rates.

Finally, with performance having been reported for each algorithm separately, a tally of best results across both algorithms can be presented for each data set and compared with the baseline. These are shown in Table 24 for the data sets Menzies focused on (excluding jm1 and kc1). While different labeling threshold seemed to disadvantage NB performance, the mean F1.5 value here, of 36, is virtually the same as for the baseline. With SVM providing the better performance for all data sets but one, it seems to have countered the effect of labeling with NB. It is notable as alluded to previously that SVM outperformed NB on virtually all these data sets (although it must be said, usually only by a modest margin).

A comparison with Menzies' baseline results beyond mean F1.5 values is shown for each data set in Figure 57 (including jm1 and kc1). Bars above the horizontal line indicate performance

	Baseline (Menzies) (0 threshold)			Best NB/SVM (40 threshold)			
	TPR	**FPR**	**F1.5**	**TPR**	**FPR**	**F1.5**	**Best**
cm1	68	28	40	30	3	31	SVM
kc3	68	28	39	44	5	43	SVM
mw1	55	14	40	25	1	27	NB
pc1	51	18	32	47	4	44	SVM
pc2	78	14	7	13	0	14	SVM
pc3	80	35	42	37	6	35	SVM
pc4	98	29	60	58	5	56	SVM
Mean:	**71**	**24**	**37**	**36**	**9**	**36**	

Table 24 Comparison of best NB/SVM performance with baseline (for all NASA data sets except jm1 and kc1). NB/SVM performance is virtually the same as baseline by mean.

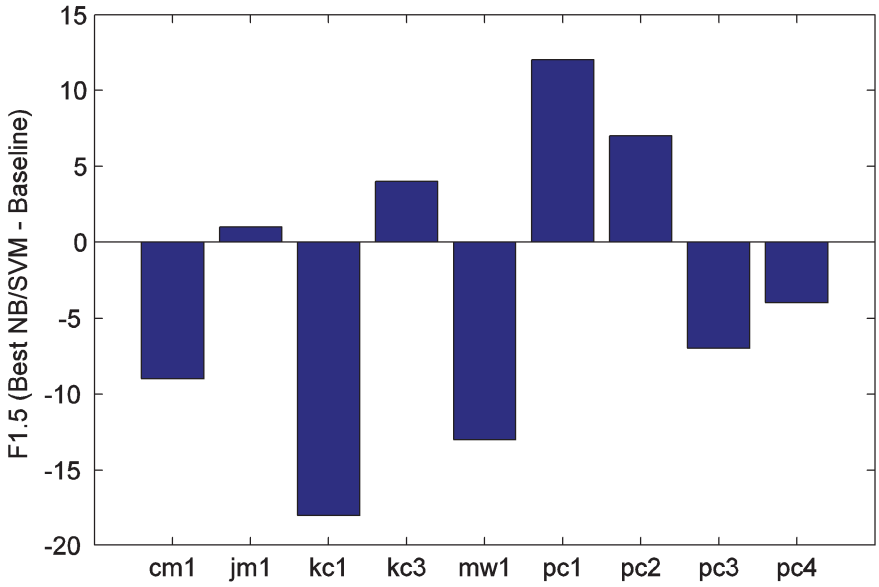


Figure 57 Best NB/SVM performance compared with baseline, in terms of their difference. Bars above the horizontal line indicate improvement over baseline.

improvement over the baseline. The net result, as suggested by the mean F values, is that ground lost in performance is almost matched by performance gained. That there is a rough net equivalence with baseline performance could be considered a solid result. However, with the extensive range of experiments conducted better results might have been expected. That they were not obtained may be due to the use of a different labeling threshold.

Chapter 8 – Rank Sum Classification

A starting point for exploration of the NASA data was to look at the data of each feature individually. This lead us to obtain a graphical representation of this data, and it seemed the most natural and useful way to do this was with a histogram, for each class. There are other ways this may be done, such as plotting the values in ascending order for example, but this did not seem to capture the nature of the data as well, largely because a sense of density of points is lost. From this graphics representation arose the question of whether it could be used to say whether a particular instance was more likely to be of one class than another? One answer follows Naive Bayes in the use of data to arrive at a probabilistic estimate of class membership. Here we look at the density curve for each class (introduced in 5.9.9 Feature Class Distributions) and if a value is in a region where density is high for a particular class this would add to the likelihood that it was of that class. Taking multiple features into consideration, an instance is likely to be of a class to the degree that across features, values of the instance lie in the higher density regions of the class distributions for that class. An operative element here is the degree to which a value lies in the higher density region and the necessary partitioning of the domain. The rank sum method uses a joint ranking as a way to measure this, and as such imposes an abstraction of ranked density levels over the raw density information.

The method is described formally in the next section. Following this, some initial performance results are reported, and their deficiencies analysed. It is found that many of the computed ranks for an instance offer little difference by which to discriminate. A modification on the method as a means to resolve this problem is then described in which bins are not spaced equally, but instead have variable widths according to the change in slope of the class density curves. Experiments using this variable bin width method are then described in an effort to see how well the rank sum method can perform. Finally the performance obtained is compared to that of Naive Bayes which was found to be the better performing method compared with SVM in the previous chapter, and amongst the best performing methods on the NASA data in other research.

8.1 Rank Sum Method

The task of supervised learning is to find a model than can discriminate between different classes of objects from examples of each class. The simplest and most common use of this is

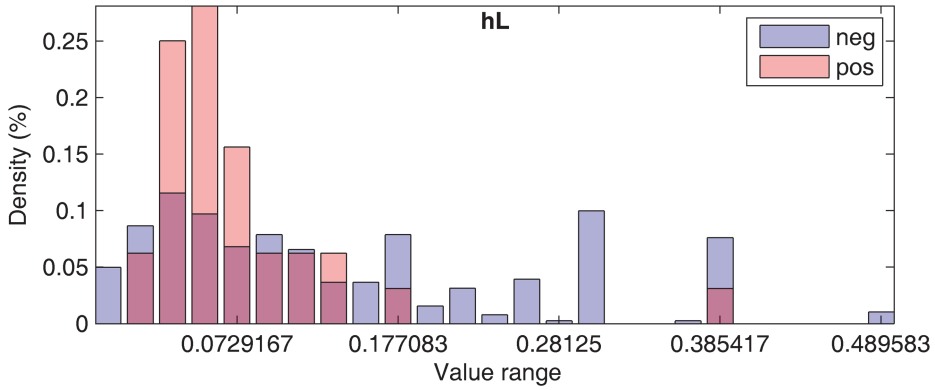


Figure 58 Probability distribution of values for each class (feature hL from kc3) which motivates and underlies the rank sum method.

discrimination between two classes, and may be described as $f(\mathbf{x}) \rightarrow \{1,0\}$ where $\mathbf{x}$ is an unseen instance. The classes may be referred to as positive and negative. The training examples $\{(\mathbf{x},l)\}$ are each comprised of a vector of attributes, $\mathbf{x} = [x_1, x_2, x_3,...]$ and an associated label l. The attributes are selected to contain information that is likely to be useful in discriminating between the classes.

Looking at the composition of an attribute from the training data, it is comprised of a set of values, and associated with each is a class label l. From the subsets of values for each class so defined, a probability distribution for each class may also be defined. This may be represented for the binary case as shown in Figure 58.

These distributions can be discretized by partitioning the range of the attribute into bins, usually of equal width. That is, bin edges are placed from the minimum value to the maximum value of the attribute at a uniform distance apart of $(a_{max} - a_{min}) / b$ where b is the number of bins. A bin exists between each pair of adjacent edges. For each bin a probability value is calculated for each class. This probability is then relative to the class rather than to the total number of instances in the training data. For an attribute value x_i, the bin probability for bin B_j given class C, is calculated as:

$$p_i = p(x_i \epsilon B_j | C) = \frac{\#\{(x_i \in C) \text{ ^} (x_i \in B_j)\}}{\#\{x_i \in C\}}$$

The probability is then the fraction of the values of the class that lie in B_j, divided by the number of values of the class.

With the binning over each class distribution and probabilities defined for each bin for a single attribute, the scenario may now be thought of in terms of two dice. There is a die for each class. The sides of each die correspond to bin numbers, and each has probability p_i. In rolling the dice, because they are biased, the sides with the larger probabilities will occur more often. While this may have some utility, the problem here is posed as one of trying to determine the true underlying class which generated the outcomes. For this purpose, it is more convenient if the die outcomes are sorted according to probability. Thus, if the outcome from one of the dice is 1, now signifying a large fraction of the probability mass, in the absence of a similar outcome from the other die it is likely that the example is of the former class. The sorting of sides or bins according to probability effectively imposes a rank ordering, B':

$$p\left(x_i \in B'_j \middle| C\right) = b_j, \; where \; b_1 \geq b_2 \geq b_3 \ldots \geq b_n$$

where the probability for an attribute value x_i is redefined for the new labelling of bins B'_j. B'_1 then has probability b_1 which is the largest bin probability for that class. Throwing a die for each feature leads to a multinomial distribution over the ranked bin numbers 1..b, where the probability of each 'face' is r_j.

For the purpose of classifying an instance, a rank function is defined which gives the rank for particular value x_i of a feature,

$$r(x_i) = \left\{ j \middle| x \in B'_j \right\}$$

The rank sum for a given class for instance vector **x** is then given by:

$$RankSum = \sum_{i=1}^{n} r(x_i)$$

The predicted class is that corresponding to the larger rank sum.

The method may also be thought of in terms of a transposition of density data, into an incremental grading of the densities for each class, which are the ranks. While the approach as described has some attractive properties and would seem to have potential to work in classification, it remains to be seen whether or not this is so, and the experiments carried out and reported on here seek to answer this question.

The rank sum method has been implemented using the objects shown in Figure 59. This provides a concrete perspective to the theory. The implementation is straightforward. For a given feature, bins and densities are computed by DensityDistribution. The subclass

RankDistribution computes the ranks across bins. Example distributions for each are shown on the right of the figure. With multiple features, there are a set of RankDistributions as shown in Figure 60, and for a given instance to be classified rank pairs from each are summed, with rank sums thus obtained for each class.

8.2 Initial performances and Failure

The rank sum method was tried initially on a few data sets. Evaluation at this stage was done

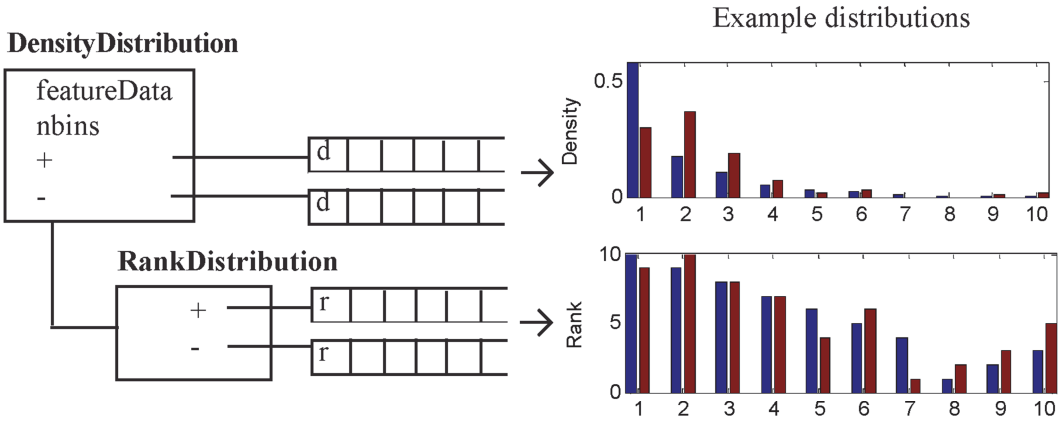


Figure 59 Rank sum implementation is composed of a density distribution and derived from that a rank sum distribution.

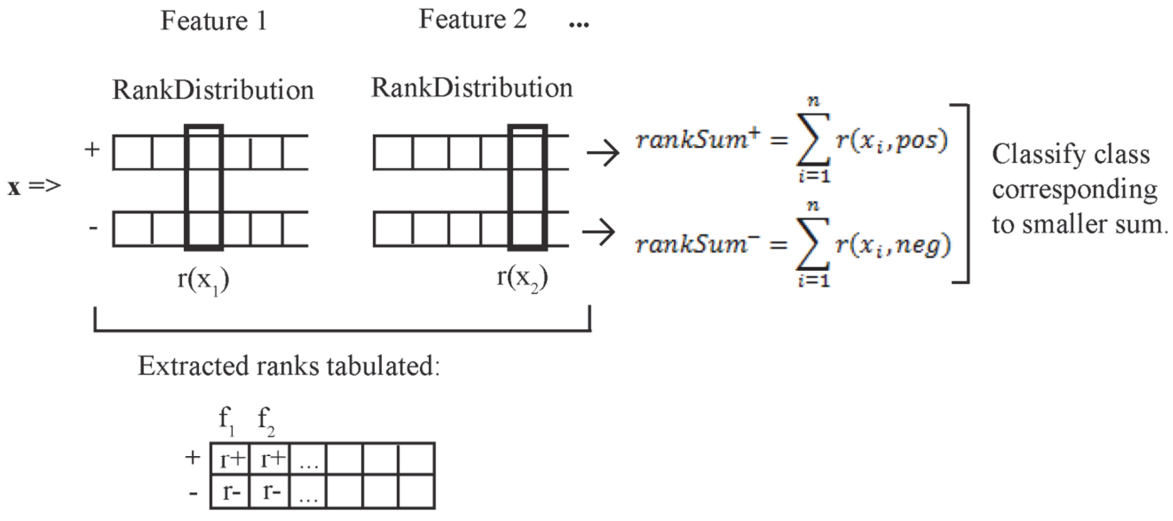


Figure 60 Classification of an instance with rank sum. An instance is classified by summing the ranks, for individual feature values, across features for each class.

Data set	Feature selection	TP Rate	FP Rate	F
jm1	RF8	58	34	16
pc4	IG9	52	16	34
kc3	RF3	58	30	20

Table 25 Performance obtained from the first application of rank sum to a selection of data sets using a somewhat arbitrarily chosen number of bins and the indicated number of features from either Relief or Information Gain rankings.

using a 10% holdout set repeated 30 times for averages. Bin count was set to either 30 or 60, and initially this was the same for all features. Results are shown in Table 25. A caveat that must be mentioned is that only the F1 value has been used and is reported in this part of the research. However at the end of the chapter, F values for other β values are given for comparison with earlier results with Naïve Bayes. Because of the selection of models based on the F1 measure, F measures by other β values may not be optimal. From the F values in the table, rank sum performs best on pc4. For jm1 the FP rate is high, but this is a difficult set on which to predict accurately and kc3 is similar. It should be mentioned that while only a single test set is used here, rigor of evaluation increases in the course of this work with cross fold validation later being used instead.

Results were visualized using a scatter plot of rank sum pairs calculated on each test set instances. An example is shown in Figure 61. A decision boundary is marked diagonally. Points are indicated by their actual class, but are classified according to their position relative to the decision boundary. As desired, many of the positives are above or near the decision boundary, and negatives tend to be below it, although there is an excess of negatives spilling into the positive region contributing to a higher FP rate, and lower F value. While this representation of the predictions of the model was interesting, and for this data set were encouraging in terms of rank sum's ability to discriminate, the patterns were not very consistent, and nor did there seem to be any means of usefully exploiting them through for example the use of clustering.

8.3 Similarity of Rank Pairs

The results discussed above were lackluster and to determine why this was the case, the rank information generated by the method, was inspected. There are two sets of rank data: the rank distributions for each class, and the rank information obtained from these rank distributions to

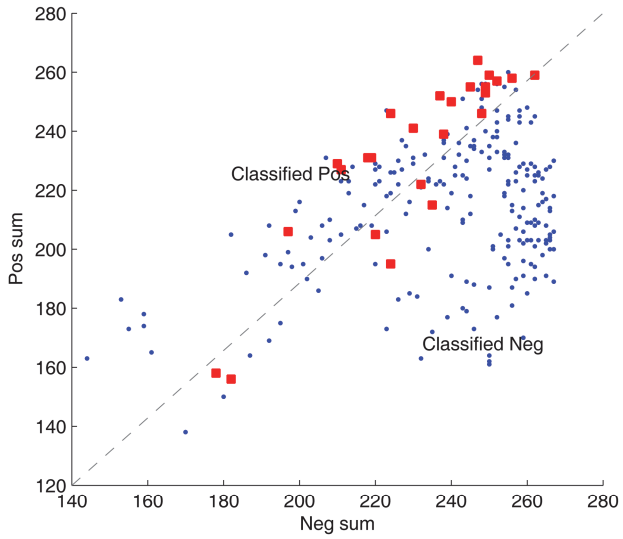


Figure 61 Rank sum pairs computed on test instances (randomly sampled from the data) encouragingly show some degree of separation, with most of the positives falling on the positive side of the diagonal decision boundary and negatives below.

classify instances. First the latter was inspected, for the kc3 data set. It was observed that most values in the table of ranks were the same, and were of the highest rank. The explanation for this was that for any given feature, most values of the feature lie in the highest density bins, which correspond to the highest ranks. This is due to the nature of the distributions of features in the NASA data which tend to be exponential. This poses as a potential problem because similar values do not facilitate discrimination, but it is not one of the method so much as the nature of the data. This problem can also be seen in the rank distributions, which are shown in Table 26 for 3 Relief selected features of kc3. Higher ranks have higher numbers, the maximum being 30 in this case, rather than starting from 1. The highlighted cells contain highest ranks and which correspond to bins with highest densities in the class distributions. These rank pairs are thus the ones most often used in obtaining rank pairs for a feature in classifying instances. It will be noticed that the rank values in these cells are similar, reflecting overlapping peaks in the class distributions. This does not lend to increased difference in rank sums for each class by which to discriminate.

In further analysis of the rank data used in classifying instances for kc3, a histogram was produced of all rank values, regardless of class, and its shape was virtually the same as the distributions of the raw data, exponential. For kc3, 60% of rank values were 29 or at the

Bin:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
hL																														
Rank -	20	24	29	25	26	22	21	19	27	16	17	14	18	13	28	1	2	23	3	4	5	6	15	7	8	9	10	11	12	30
Rank +	28	29	30	23	26	27	20	1	24	25	2	3	21	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	22
dP																														
Rank -	30	1	2	3	4	5	6	7	8	9	29	10	11	12	13	14	15	16	17	18	28	19	20	21	22	23	24	25	26	27
Rank +	30	1	2	3	4	5	6	7	8	9	29	10	11	12	13	14	15	16	17	18	28	19	20	21	22	23	24	25	26	27
mMS																														
Rank -	24	22	20	26	23	27	1	2	28	9	10	21	13	29	3	16	19	17	18	14	15	25	4	11	12	8	5	6	7	30
Rank +	20	21	1	2	26	28	3	4	29	5	6	22	7	27	8	23	9	24	25	10	11	12	13	14	15	16	17	18	19	30

Table 26 Against achieving better performance with rank sum, from the rank tables for three features above, positive and negative rank values in the bins among the highest in rank, as highlighted, show little difference. Most instances lie in these bins (highest density giving highest rank), and therefore similarity in rank value in these bins contributes to similarity in rank sums, which does not facilitate discrimination. (Tables are of 3 Relief selected features from kc3 with 30 bins. Highest rank is 30.)

maximum of 30. This serves to emphasize the problem of similarity of rank values used in classification due to the nature of the data. However looking at these individual values gives only some of the picture. A better indication of the lack of difference in ranks can be obtained in looking at rank pairs. A histogram of all pairs of ranks used in classification for kc3 is shown in Figure 62. Along the diagonal marked there is no difference. The further pairs are away from the diagonal the more difference there is, potentially contributing more to the difference in sums of ranks (potentially, because even though rank pairs can have different values, the sum can still be the same). As can be seen, most pairs are clustered at the maximum rank, and lie on or close to the diagonal. This makes classification a more difficult task and is an issue that may need attention for method performance to be improved.

It has been assumed that a larger difference in rank sum values for each class results in better performance. This assumption is here confirmed, and the relationship between the two is shown to be linear. Measurements of rank difference are first defined. One is the mean difference in rank pairs across all instances. The other is the mean difference of the rank sums.

To see whether rank difference correlates with performance, a few different models were created on pc4 with different feature selections (selections independent of those of earlier chapters). The models were evaluated over the training set from which performance measures were obtained. Rank measures were also obtained on each model. If there is a correlation between performance and rank difference, then that should be apparent in these figures. The

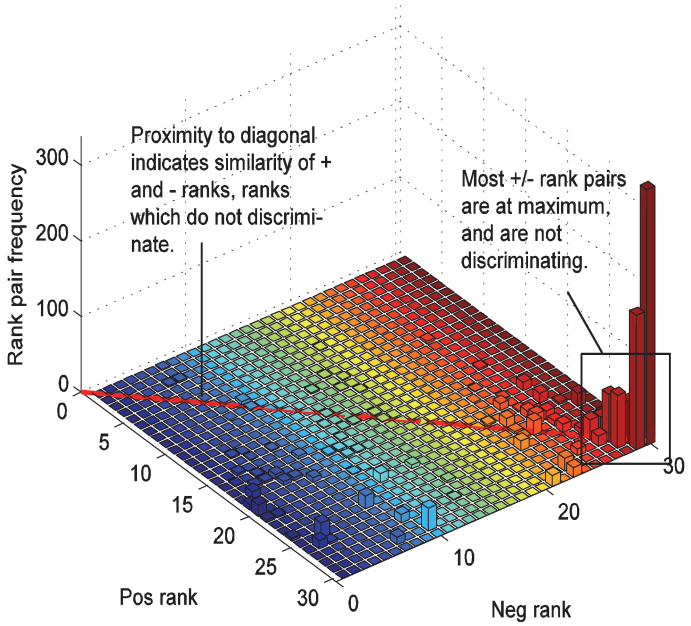


Figure 62 This histogram shows the actual distribution of computed rank sum pairs from the rank tables in the previous figure for all instances in the data set. Confirmed is the problem that most rank pairs have similar value with most bars appearing on or near the diagonal, and also apparent is that most rank pairs have rank values close to the highest rank with the tall bars at the end of the diagonal.

Feature selection	Mean rank pair diff.	Mean rank sum diff.	TP	FP	F
CFS 4	4.5	16.3	66	20	35
IG 9	4.1	29.4	60	21	40
RF 2	8.4	16.0	68	23	33

Table 27 Calculating the mean difference between computed rank sum values for each class across instances (3rd column) for three different models on pc4 with different feature selections, it is clear there is a strong relationship between this and the performance of those models (last column) – if greater difference can be found between rank sum values, performance can be improved.

figures for each model are shown in Table 27, and indeed, between mean rank pair difference and the F value (3rd and last columns), a correlation is observable.

Further evidence of this was found in looking at the accuracy in classifications over subsets of instances derived using an increasing threshold on rank sum difference. With each

subsequent subset the threshold was higher and the set contained only instances with greater rank sum difference. The accuracy on each of these subsets is plotted in Figure 63. It can be seen that as rank difference increases there is approximately a linear increase in accuracy.

The simple explanation for performance improving with rank sum difference is that the larger the difference, the greater the extent to which instance values lie in higher ranked bins of one class than the other. The stronger this effect, the more likely it is that the class with the higher sum is the actual class.

8.4 Variable Width Binning

In the previous section we have seen insufficient separation in rank sums, leading to weak classification performance. Many instance values were found to lie in the bin of highest rank, and frequently this was because the ranks for positive and negative curves were found to coincide.

Underlying these problems was the selection of bins from which the ranks were obtained. It is their configuration which is seen to exacerbate the above problems. In the existing binning method, there are a particular number of bins and all have equal width. Two shortcomings

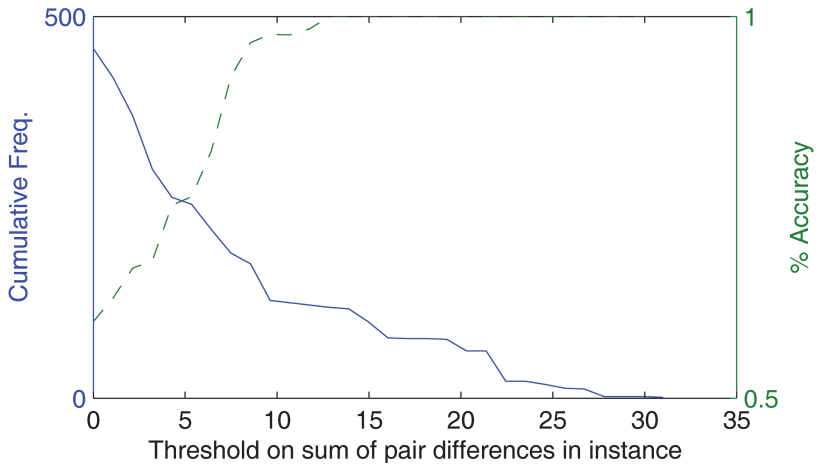


Figure 63 The previous table showed a strong relationship between mean rank sum difference and performance. Here, a similar relationship is shown, but in terms of the increase in the percentage of correctly predicted instances (dashed line) as the set of instances is reduced (blue line) by gradually excluding instances with a lower sum on *rank pair* differences (the sum of difference across features, rather than the difference in *rank sum* values as for the previous figure).

with this have been identified as illustrated in Figure 64:

- Two different values may have the same ranks even though densities may be markedly different.
- Two different values may have different ranks even though densities may be very similar.

In the first case, the values share the same rank because they lie in the same bin, despite their having different densities. This is the more significant problem of the two – density changes tend occur where peaks are and this affects more points i.e. points which are given the same rank. The argument here is that seeing as there is an abundance of information and differing levels of density across the bin, more difference in ranks could be brought about by partitioning the bin further, so that it has in a sense more resolution on the data. This would result in more rank difference where otherwise many values would have the same rank.

The other shortcoming is the opposite, in which there is difference in rank, where rank represents a level of density, simply because the value is further along the x-axis, when there is little change in density. Having additional bins over the x-range creates additional rank levels, when there is actually no change in density. This probably would work against achieving better performance with rank sum as it would fill the rank distribution with superfluous ranks.

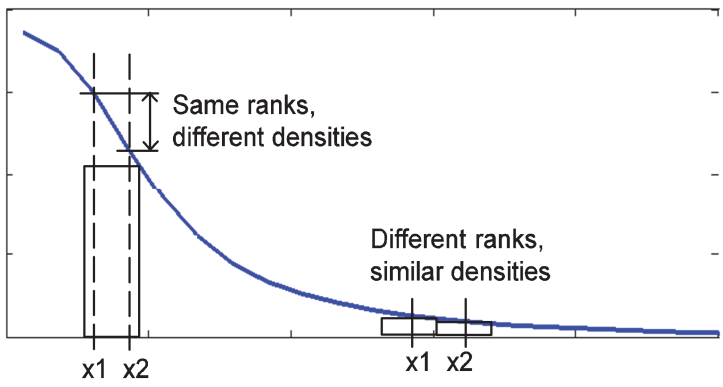


Figure 64 With the need for difference in rank pair values to improve performance having been established, a potential problem that arises with fixed width bins is that in high density areas of the feature probability distribution even though change in density can be dramatic within a bin, rank remains the same, as shown on the left. A related problem that may diminish performance is the formation of ranks across flat areas of the curve, shown on the right.

A solution was found to these problems in varying the width of bins according to the rate of change in the density curve. If there is a rapid change in slope more bins are created to capture the different levels of density over it. If the curve flattens out, fewer bins are created.

An initial approach to implement this solution involved the use of contour lines. The feature range is first partitioned into a specified number of equal width bins from which a density distribution is obtained. Contouring is then applied to this density distribution, and bin edges formed where contours touch the x-axis. The desired number of contours is specified as a second parameter, which determines the number of variable width bins. However, while the density levels at which contours are formed can be specified in the contour function, the actual number of contours that result cannot. Thus, the number of contour density levels is increased iteratively until the target number of contour lines is approximately reached. Once the variable width bin edges have been obtained densities are recomputed. This approach is illustrated in Figure 65.

A point that should be made is that densities are recomputed relative to bin width rather than

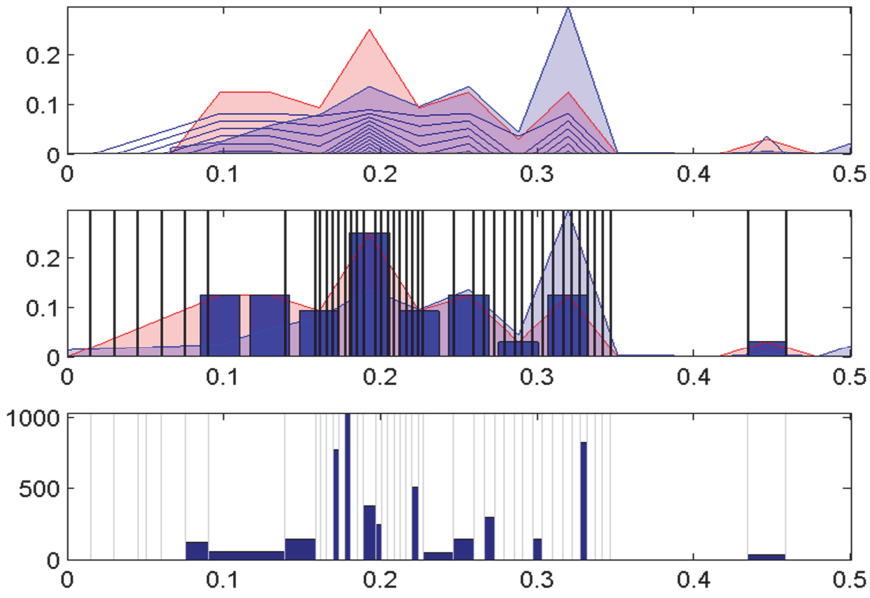


Figure 65 Contour-based method for producing variable width bins. The first graph shows contours formed on the positive class distribution. The second shows vertical lines where contours in the first graph touch the x-axis, which become bin edges (histogram of positives also shown). In the third graph the probability distribution is recalculated, but this is done relative to bin width rather than class size the reason for which is explained in the next figure.

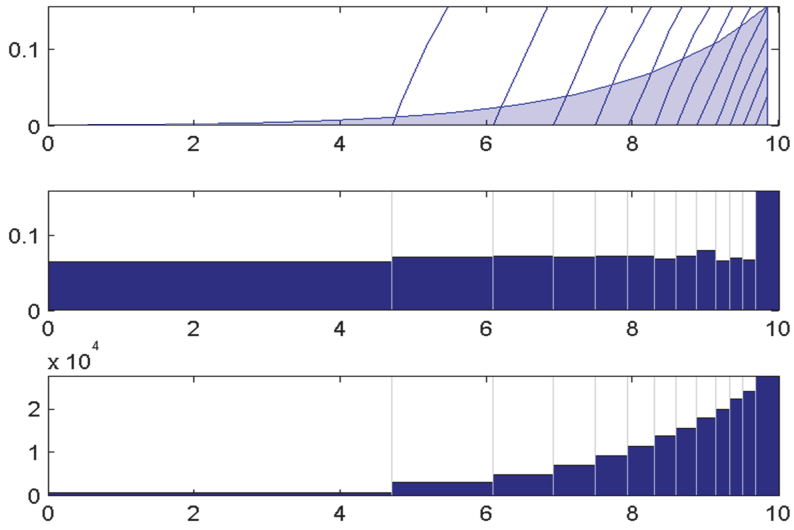


Figure 66 Why variable bin width densities are computed relative to bin width rather than class size. The exponential distribution at the top is partitioned into variable width bins, and densities computed by class size and then bin width in the graphs below. Clearly, the flat distribution is undesirable, as by this it would not be possible to assign ranks to bins, and the desired effect is beneath it.

total number of instances in the class, the reason for which is illustrated in Figure 66. With a step added later in which narrow bins were merged while contour density levels were increased to meet the target number of bins, a pathological situation developed an example of which is shown in Figure 67. Because of this, and the computational overhead of running the contour function repeatedly, a different approach was found, described next.

The intersection based method is straightforward. It involves running horizontal lines spaced an equal distance apart over the y range of the density curve, through the curve, finding intersection points, and then projecting those onto the x-axis to obtain bin edges. This is illustrated in Figure 68. Between each pair of successive intersection points the curve remains within the upper and lower bounding density levels. For variable bins that span multiple equal width bins, the resulting variable bin density is approximately the average of the component densities within it. The formal statement of the algorithm is given in Figure 69.

The approach suffers at times from an excessive number of narrow bins in regions in which the curve is steep. This is more noticeable in relation to the spacing of bins in the flatter regions of the curve. The contrast arises because of the imbalance of densities in the curve.

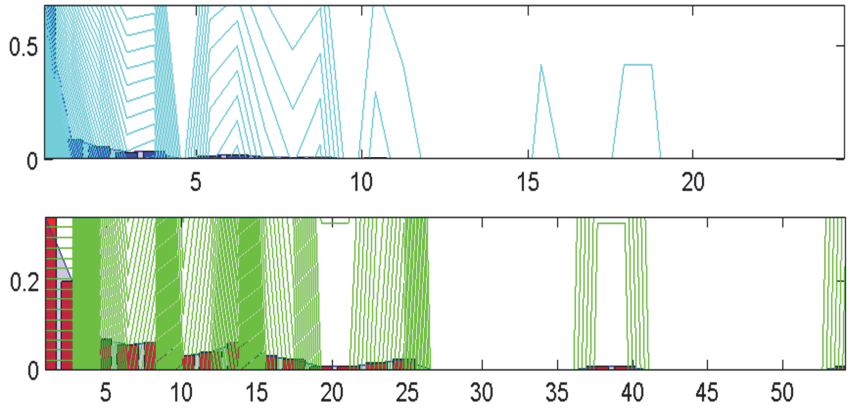


Figure 67 The definition of variable width bins with an existing contour function is convenient, however when trying to reach a target number of variable width bins of 60, an explosion of contours occurred. This lead to the use instead of an intersection-based method for defining bins, illustrated in the next figure.

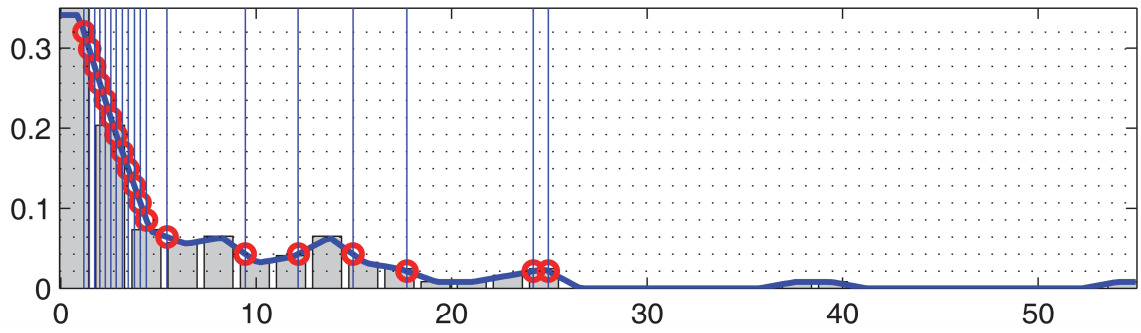


Figure 68 Intersection-based method for defining variable width bins. Horizontal lines (dotted) are run across the probability distribution curve, and points of intersection (circles) are projected onto the x-axis as bin edges.

To control this there is a minimum bin width proportion parameter. This is used in a post processing step over the variable width bin edges, in which runs of bins that are less that the $proportion \times range(X)$, are replaced with bins spaced at the minimum width. A potential problem with the use of this parameter is that large values can result in significant ranges being effectively spaced with equal width bins, which might defeat the purpose of having variable width bins.

An issue that arose in the development of this method was the appearance of a large bin near the origin. This is evident in variable width binning of Figure 68. This occurs because the

```
1.fDist = create fixed bin width distribution for class (X,N bins)

        remove extreme outliers from X (X > 99.5th percentile)
        binWidth = (max(X)-min(X)) / (N-1)
        create bin edges from min(x)-binWidth/2 to max(x)+binWidth/2
        replace end edges with -Inf and + Inf, to include outliers
        compute density for each bin as (count(X in bin)/count(X))

2. vDist=create variable bin width distribution (fDist, #dlevels)

        % create interpolated density curve
        x = bin centres (from fDist)
        y = bin densities (from fDist)
        yI = run linear interpolation on points (x,y) at defined xI from min(X) to max(X)

        % run horz lines through y range to find intersections with interpolated curve
        dInterval= range(y) / (count(y)+1);
        dLevels = space #dlevels from min(y)+dInterval to max(y)-dInterval
        initialise variable bin width edges binEdges to []
        for each dLevel
                line=create horizontal line at dLevel
                intersectionPoints = intersect(line, interpolated curve (ie xI,yI))
                add x coords of intersectionPoints to binEdges
        end
        sort binEdges in ascending order

3. fix variable width bins (vDist, minBinWidthProp)
        % apply minimum bin width criterion

        minBinWidth = minBinWidthProp * range(X)
        for any sequence of bins where width(bin) < minBinWidth
                if length(seq)=1
                        leave as is
                else
                        replace with bins spaced at approx minBinWidth
                end
        end

4. compute densities on variable width bins

        compute density for each bin as (count(X in bin)/binWidth)
```

Figure 69 Algorithm for the intersection-based method for defining variable width bins illustrated in the previous figure.

density curve starts at the first bin centres value, which is not at the origin but some distance to the right of it. For the curve to be drawn properly and intersections properly obtained, the first density information needs to be at the origin. So a modification was made to the method in which the first bin starts left of the origin with centre at the origin. In later computation of variable width densities, the area over which density is computed for the first bin is only half its width.

Some comments need to be made about the variable width binning approach. Methods have been described to produce a variable width binning based on a density distribution. It was not specified whether this was for one class or for both. In the former case, each class has different binnings where the number of bins and bin edges are not the same. In the latter case they are both shared. The approach taken has been to customise bin selection to each class

distribution. The reason for this is because the ranking abstraction does not compare densities in a common bin to decide class or to contribute to an overall probability. Ranking is simply a measure of distance on an incremental scale over variable width chunks of density from the chunk with highest density within the class. There is no 'a priori' reason for the chunks of density over which the ranking is defined to coincide. The abstraction is one which determines an instance to be of a class by the degree to which the feature values lie in highest density bins. The relation of interest in this abstraction is that within the class. An advantage of having the bin selection separate for each class is that they can be tailored to fit each distribution better. Clearly there would be problems if many bins were being created for one distributions because of rapid change in slope, when for the other class the curve in that region were relatively flat. A consequence of this approach is that the number of bins for each class may be different. And this necessitated converting ranks to rank fractions between 0..1 before being added to the rank sum. If the number of bins for each class is the same, this conversion to fraction produces the same classifications as when the actual ranks are used.

8.5 Experiments

The aim of this section is to see what performance the rank sum method is capable of producing with the variable bin width enhancement. A number of different experiments are performed starting with simpler ones progressing to more computationally intensive ones. These are focused on the pc4 data set. The approach found to work best is then applied to another data set, kc3.

Rank sum performance for both data sets is compared with that of Naïve Bayes.

8.5.1 Initial

The effect of the variable width binning method was assessed using data set pc4, identified as the best performing in the previous chapter. Models were created for the two binning methods, with 4 CFS features. Training set model evaluation was used, as the interest was simply to compare representations. Rank differences were also measured as the average difference between all rank pairs, to provide another dimension by which to compare the two binning methods. Default parameters, considered to be roughly optimal based on exploratory observation, were used for both binning methods, and were applied to both classes. Results are shown in Table 28. Parenthesized values in the rank difference column are the fractional averages converted to a scale as though there were 30 bins in each class.

Configuration	Avg. Rank Diff.	TPR	FPR	F
Equal width bins (#bins 30)	0.15 (5.34)	66	20	35
Variable width bins - contour (30,30)	0.18 (6.17)	65	16	39
Variable width bins - intersection (30,30,0.005)	0.18 (6.18)	86	18	45
Variable width bins - intersection (30,10,0.005)	0.23 (7.56)	68	13	45

Table 28 Improvement in performance using the intersection-based method for variable width bins (which also gives the highest average difference in rank pair values), compared to both fixed width bins and the contour-based method. The comparison is based on the creation of a rank sum model for each bin configuration using the pc4 data set with 4 CFS selected features.

It can be seen that variable width has improved performance over equal width. The increase is 4% with the contour based method, and a better 10% with the intersection based method. This would suggest that variable width binning does provide an advantage over fixed bins for the rank sum method. However, a systematic analysis of the parameters underlying the method failed to yield settings suitable for broad application, leading to a focus on maximizing performance for each individual feature. This is considered in the next section.

8.5.2 Single Feature

Initial attempts to improve single feature performance are focused on class-specific bin selection. This would allow rank distributions to be more customised to each class, and this may provide for rank distributions that better represent the data for the purpose of classification. It is not possible to determine optimal parameter settings by considering each class individually. Optimality is determined by performance, and for this both classes are required. Thus, the strategy chosen to find optimal parameters for each class was a relatively simple exhaustive one.

Parameter ranges and the values over them were based on the previous experiments with the number of fixed bins ranging from 30 to 200 at intervals of 20. This gave a total number of parameter sets of 320 (8 * 8 * 5, the number of values for each parameter). In combination over 2 classes, there are 320*320 possible pairs. For each a model is obtained, and this is repeated over 5 samples, giving a total of 512,000 models. The algorithm to obtain performance on each parameter pair is shown in Figure 70. The average of performances across samples is taken.

```
for each parameter set, posSet
      for each parameter set, negSet
            for each sample
                  posDist = createRankDist(sample.train, posSet)
                  negDist = createRankDist(sample.train, negSet)
                  classifier.train(posDist, negDist)
                  pred = classifier.classify(sample.test)
                  performance = compare(sample.test.actualY, pred)
            end
      end
end
```

Figure 70 Algorithm used to try every possible combination of parameter sets over pos and neg distributions for a single feature, in order to find the best performing pair of parameter sets.

An efficiency in implementation was made by caching the distributions at the outset. This avoided repeating the creation of distributions for different combinations which contained the same parameter set. The data structure used to contain these distributions is shown in Table 29. An example of the distributions that would be picked out for a given pair of parameter sets is highlighted.

First this experiment was tried with the pc4 feature lCp. This is percent of comments in the code, and was one of the features chosen in the CFS selection on the pc4 data set. This feature is real valued, rather than integer, and so may provide more of a spread of information with which the variable binning method can work. The experiment was then tried with lCC, which is the number of lines of code and comments, also chosen by CFS. The two features are among the better discriminators for the data set. Their class distributions are shown in Figure 71, which shows some opportunity for discrimination near the origin, where there's a high density of negatives.

	Parameter sets				Sample 1		Sample 2		...
#	p1	p2	p3		Pos	Neg	Pos	Neg	
1	val.	val.	val.	→	RDist.	RDist.	RDist.	RDist.	
2	...			→	RDist.	RDist.	RDist.	RDist.	
..					..	..	..	..	
320					RDist.	RDist.	RDist.	RDist.	

Table 29 Data structure used to cache distributions to avoid having to recompute them on the fly when corresponding parameter sets reappear in the course of exhaustive search. Highlights show the pairs of distributions used across samples for a given pair of parameter sets (pos class = param set 1, neg class = param set 2).

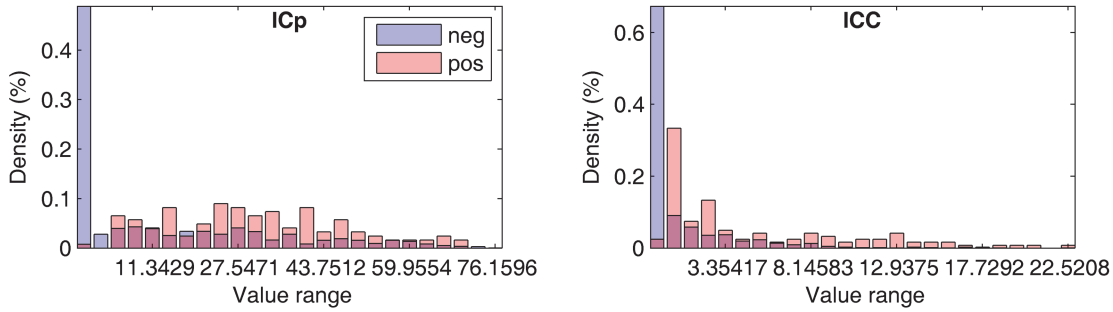


Figure 71 Class distributions for two of the better features of pc4, selected by CFS, and on which different parameter combinations were tried for each class exhaustively.

	Pos class			Neg class			Perf.
Feature	**#Bins (eq.)**	**#Levels**	**MinWidth**	**#Bins (eq.)**	**#Levels**	**MinWidth**	**F**
lCp	90	130	0.0063	30	30	0.0025	32
lCC	30	30	0.00437	70	130	0.00625	37

Table 30 An exhaustive search through bin parameter sets across classes for two features lCp and lCC yielded the best parameters and performance shown.

The results for each feature are shown in Table 30. In relation to performance, the only relevant comparison to make is with equal width bins, also on multiple features, the best score of which was 35 (Table 28). Compared to this performance, the single feature of lCC has performed reasonably well at 37 with class specific variable width binning. It must be emphasised this is only a single feature, and its performance is comparable with the complete set.

To compare performance using equal width bins, to see how much improvement variable width bins was making, equal bin-width rank sum performances (with CV) were obtained for lCC with the number of bins again increasing from 30 to 200 by 20. Different bin numbers were used so that its performance was fairly evaluated. A plot of performance is not shown, but the F value curve at 30 bins was 24, and for all remaining bin numbers was around 33. The optimal number of fixed bins would seem to be higher than 30, perhaps around 50, at least for this feature. Compared with fixed bins, variable bins provide a 5% improvement in performance with this single feature. Therefore not only does class specific binning seem to fare well, or is comparable with, the same binning parameters for each class, with merit indicated for the former, but class specific *variable width* binning exceeds fixed performance.

A conclusion drawn from this experiment is that class specific binning seems to offer potential for better performance than the same parameters for each class. Also there is an advantage to be had with both variable width widths and class specific parameters over fixed bins. Lower parameter values may also be worth exploring.

A further idea to improve individual feature performance was to partition the class distributions and apply variable width binning separately to each. This is similar to the idea of class specific binning, but is more specific in that binning is applied to different sections of the data within each class. This is discussed in the next section.

8.5.3 Distribution Partitioning

In looking at some of the bin selections for different features, it appeared that there may be too many bins in the parts of the curve where slope is steepest, and too few bins elsewhere. This was evident to some degree in the earlier Figure 68. The problem seems to relate to one which has been identified in other contexts, which is within-class imbalance. Many features in the NASA data sets follow the lognormal distribution, in which there is a pronounced peak near the origin which drops and levels out over some range. This results a large number of bins in the peak, and many fewer in the tail.

Some balance in bin selection was found by dealing with peak and tail regions of the curve separately. In a sense this tackles the within class imbalance problem head on. A partition point is chosen, which provides a good separation point between the two different regions of the curve. This point applies to both classes. This split point creates two partitions of data for each feature based on class. For each of these a separate rank bin distribution is created. That is, each partition is effectively treated as a separate feature. An issue that arises is what impact this has on computing the rank sum. Having partition-based features is simply handled (if a little inelegantly), in that if a feature value belonging to the instance to be classified does not lie in a partition, then the contribution of that partition/feature is zero to the sum. So there is no difficulty in splitting the data into separate features.

An example of partitioning is shown in Figure 72 using the method just described. It can be seen that prior to partitioning, on the left of the figure, there are many bins near the origin at the steepest part of the curve and relatively few elsewhere. This distribution is partitioned at 0.5, just to the right of the tallest bar, and the right partition is shown in the right side. This data is treated as a separate feature, to which variable width binning is applied, and it can be

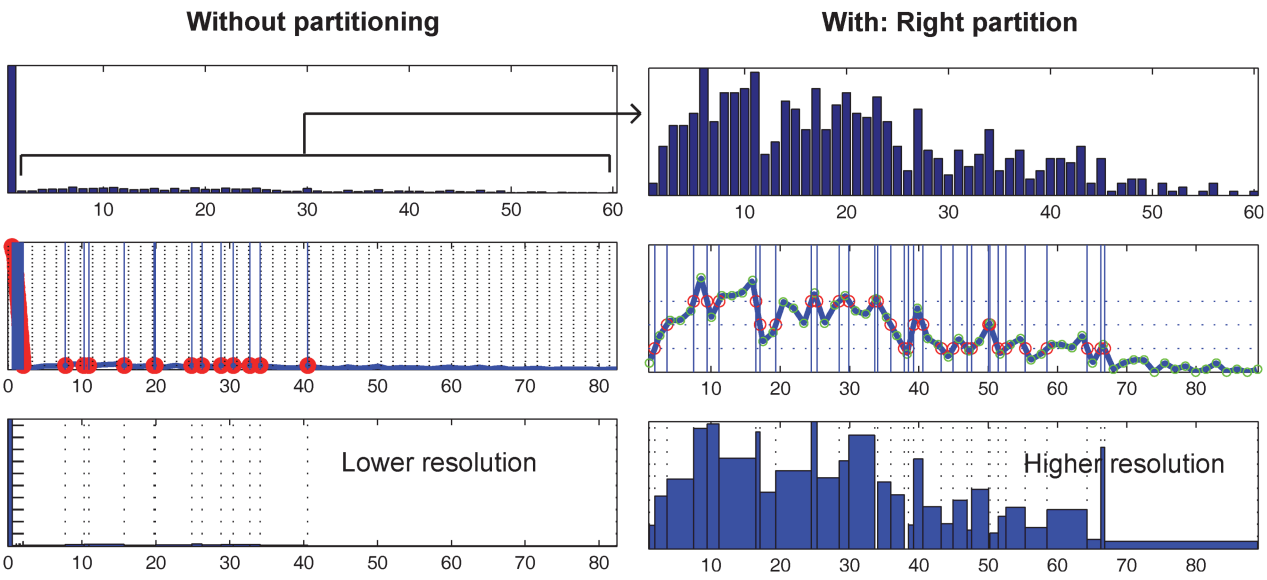


Figure 72 A non-partitioned distribution on the left (neg. lCp from PC4), and a partitioned version on the right in which it can be seen that for flatter part of the original curve there is more resolution, or in other words, the variable width binning of the distribution is less coarse.

seen in the two bottom-most plots that the separate treatment of the right partition has produced more resolution on the distribution than on the left side. Partitioning does achieve what is desired, but it remains to be seen whether this translates into better performance.

A special case of this partition method is when the partition point is close enough to the edge of the distribution range, that there is only a single bin in the partition nearest the end of the range. Without special treatment for this partition, the rank for each class would be the maximum, resulting in equality, which would not help in discrimination, even though there is information in the bin as to the densities of each class. If there was only a single feature, a value lying in this bin would always predict positive, as the default prediction when rank sums for each class are equal is positive. To avoid this, and to use the information in the single bin to assist with discrimination, the rank fraction for each class is the proportion of instances of the class over the total number of instances of both classes in the partition:

$$rankFraction_c = \frac{\#\{(x_i \in C) \text{ ^} (x_i \in \mathrm{P})\}}{\#\{x_i \in P\}}$$

where P = partition and C = class.

Attention now turns to experiments to see if the partitioning method offers any performance improvement on a single feature. The feature chosen here is lCC and the partition point chosen was 0.5 as in the last figure. In the first experiment, a representative set of parameters

was chosen and applied to both classes (30 and 20 for equal bins and density levels respectively). Class specific parameters as used before were not used here because what may be optimal for the original distributions, may not be for the partitioned distributions. So the focus was on choosing reasonably good parameters and seeing what improvement could be had in performance with partitioning. Returning to the experiment, the same binning parameters were applied to each partition. These binning parameters only had effect on the right partition for each class, as the left partitions contained only a single bin because the partition point chosen was close to the origin.

Surprisingly, the performance obtained with this configuration actually decreased from the previous result without partitioning. The F value dropped marginally from 37 to 34. In looking at the classification data, it was found that almost all instances in the left partition were classified correctly, and this is because almost all the points left of the partition were negative. The classification in this partition was based on the proportion of positives and negatives according to the handling of the special case when there is only a single bin. Where the misclassifications were occurring was in the right partition. And that there was a drop in performance there (compared to without partitioning) was unexpected given that through partitioning more resolution there was obtained. Efforts then subsequently were focused on the right partition and seeing if some optimal binning pattern could be found, that together with the high accuracy on the left side, would give a better result overall. However the improvements gained with this were only marginal.

Partitioning was the final attempt at improving performance with a single feature. The main other approach used as described was class-specific binning which was found to benefit performance. The other finding of some significance was the density levels much less than 30, even as low as 3, have potential to provide better performance than higher numbers. The attention turns next to maximising performance with multiple features.

8.5.4 Multiple features

Having explored possibilities with a single feature, attention turns to the use of multiple features. The best performance so far is 37 (Table 30), obtained with a single feature lCC and class-specific bin parameters, with which multiple feature performance will be compared. As mentioned in the outline, individually optimal parameters are first applied to two CFS features, and then to all CFS features, before turning to exhaustive search for optimal parameters across two features. With this exhaustive parameter search being found not to

help much performance wise, and performance with CFS features also lacking, the focus turns finally to individually optimal parameter selection and finding optimal selections of features.

The first step into multiple feature territory was to use two features, and see whether or not the combined performance could exceed both individual performances. In selecting parameters for the pair of features, rather than find parameters that were optimal in combination, instead parameters were determined for each feature individually as those that were optimal for the feature based on its predictive performance alone. This was thought to be a sound approach due to the sum nature of rank sum which simply compounds the contribution from each feature. This would save the expensive search involved through all the parameter combinations in considering the features together. So the first step was parameter selection for each feature. This was done for 4 CFS features, from which the two best performing features were subsequently selected for modelling.

The method used for single feature parameter selection is the same as was described earlier, in which models are trained on every pair of parameter sets for each class and cross validated. This time, however, smaller density level values were used. Rather than 20 to 200 by 20, levels used were [2 3 5 7 10 20 30 50], and to save computation time only a single minimum proportion width of 0.005 was used. The results of this parameter selection for each CFS feature are shown in Table 31. These results are noteworthy in themselves in that lCC performance alone is 41. The performance of the other features drops considerably from this. It will be noted too that some of these best performing parameter selections have very low density levels, at 3 and even 2.

The two best performing features from parameter selection above, lCC and lCp, were used to develop a rank sum model to see whether the combined performance could improve on individual. While there were hopes that this was possible, the result obtained in combination was 35 which is roughly in the middle of the individual performances. The rank contribution

	Neg			Pos			Performance		
	b	d	m	b	d	m	TPR	FPR	F
lCp	60	30	0.005	60	20	0.005	79	32	31
lCC	30	30	0.005	60	2	0.005	76	19	41
dP	30	3	0.005	45	3	0.005	70	52	20
mMS	90	3	0.005	30	30	0.005	95	86	18

Table 31 Selection of parameters for each pc4 CFS feature, for each class, through exhaustive search of parameter set pairs. b=#equal width bins. d=#density levels, m=minimum width proportion.

of lCp to that of lCC was causing instances previously classified correctly with lCC alone, to be misclassified. Details of classification were looked at for the last fold of cross validation, and in that, of the total number of false predictions of 60, 53 of them were false positives, and only 7 of them false negatives. It seemed that with two features there was a strong bias towards positives based on the classifications made in this last sample. The reason for the positive bias could be attributed to their being no prior involved, unlike Naive Bayes, to dampen the strength on the positive side. Positive ranks are given just as much weighting on the sum as negative. This is just a hypothesis though, and it could instead be due simply to an unfortunate choice of features.

Following this discouraging result with two features, each with optimal parameters individually, all four CFS features were then combined. The result was worse with the F value dropping further from 35 to 28. Similarly with 9 IG features, performance was weak at 29, again with false positives being excessive. It perhaps would not hurt at this point to bring to attention the result obtained previously on lCC alone of 41. Clearly, the sets of multiple features tried were not improving on that.

It was thought that a reason for the lack of performance might be because binning parameters were being selected for each feature individually and in combination they might not be optimal. So attention turned next to searching for the best parameter sets determined by performance across both features, lCC and lCp. This is the first instance in which exhaustive search has been applied across multiple features. The reason this had been avoided was because of the explosion in number of possible combinations. This has been ameliorated to some extent however by reducing the number of parameter values included in the search space. The algorithm used to perform the search is shown in Figure 73. Loops were executed in parallel for speed. The nesting of loops effect a search over all permutations of parameter sets. The total number of models created in this search was 21 parameter sets to the power of 4 for the number of places in the permutation space, multiplied by the number of cross fold samples which was 6, to make a total of 1,166,886 models. The best F value was 40. Given the number of models evaluated, this is not an impressive result, but at least performance has improved by 5% over the F value of 35 obtained previously using the same two features but with parameters optimal for each individually only.

The only obvious avenue to pursue further was to use more features. The only feasible method for parameter selection was to use those that were optimal for each feature individually. The problem remained then of which selection of features would work best,

```
cache distributions (as before except for 2 features instead of 1)
for each neg param set for feature 1, n1
       for each pos param set for feature 1, p1
              for each neg param set for feature 2, n2
                     for each pos param set for feature 2, p2
                            for each sample,s
                                   pD1 = cacheDist1(sNo, p1)
                                   nD1 = cacheDist1(sNo, n1)
                                   pD2 = cacheDist2(sNo, n2)
                                          nD2 = cacheDist2(sNo, p1)
                                   classifier.train(pD1,pD2,nD1,nD2)
                                   pred = classifier.classify(s.test)
                                   sPerfs(sNo)
                                          =compare(s.test.actualY,pred)
                            end
                            meanPerfs(n1,p1,n2,p2)=mean(sPerfs)
                     end
              end
       end
end
```

Figure 73 Algorithm used to try every possible permutation of parameter sets over pos and neg distributions across two features, in order find the best performing set of parameter sets.

given that performance on the CFS selection previously was lacking. Relied upon was a ranking of features based on the individual performance of each. With this, first an "iterative subsetting" approach was tried, and then a best first search approach, the latter of which produced the best results of all the rank sum experiments.

For the first approach, features were first to ranked in descending order by performance. Models were then created and evaluated on subsets of increasing size from the top of the ranking. This approach is sometimes used in finding an optimal subset of features from a ranking feature selection algorithm. Combining features that individually were among the top performing ones (not only lCC and lCp) could provide an opportunity for performance improvement. Individual performances using 10*10 CV for each feature sorted by F value are shown in Figure 74. The best performing feature as previously identified is lCC, followed by lCp. There is essentially a steady decrease in the F value from 40 to 18 from best to worst performing features. The best TP and FP rates are at the start where there is a wide gap between them. These progressively become closer, reflected in the poorer F value. The performance results of top down subsets of these features is shown in Figure 75. In this figure the F value is plotted on the same scale as the TP and FP rates, so the difference in F value is not as noticeable. But the F value progressively decreases from 40 to 30 as more features are added to the set. The behaviour of curves overall is surprisingly consistent. The main issue

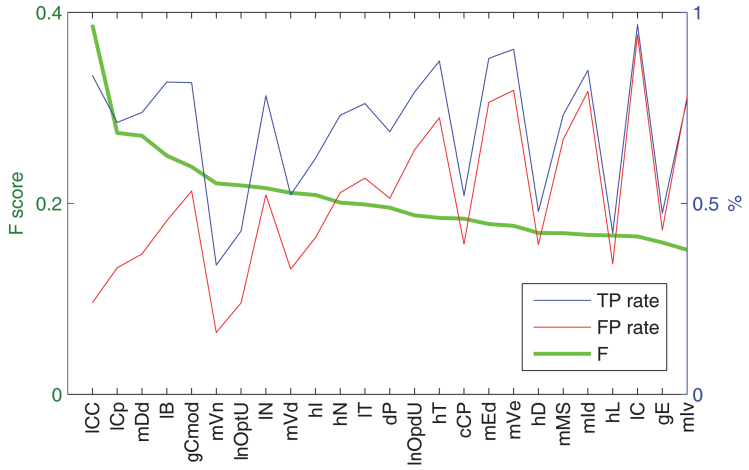


Figure 74 Rank sum performance on individual pc4 features, each with individually optimal bin parameters, in descending order by F measure.

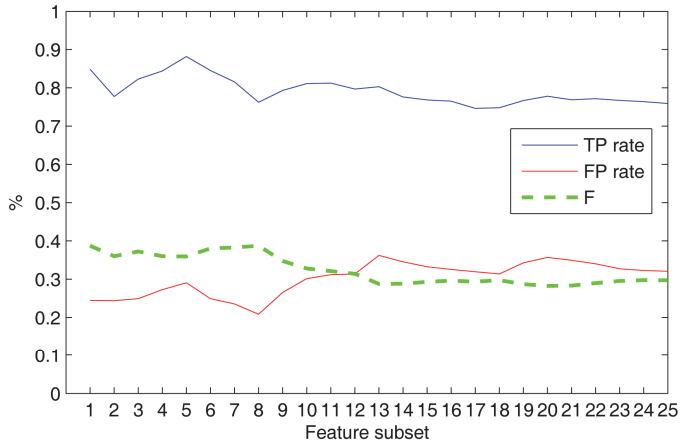


Figure 75 Rank sum performance on top down subsets of features from the ranking of features in the previous figure. No subset of features performs better than the first feature alone, with a gradual decrease in F measure of 10% as subset size increases.

in relation to the objective of higher performance, is that additional features in this top down approach do not increase performance above that obtained on lCC alone at around 40.

Combining features in the top down subset approach just described was unsuccessful. It was assumed that this must have been because of simply non optimal combinations of features. To find which sets were, again a more exhaustive best first search approach was resorted to.

Combinations of features of size three were searched first. Features are searched in order of merit according to ranking by individual performance. For each subset of features models are

generated in cross validation fashion to obtain a performance. The subset with best performance is chosen. Better performances tend to be obtained at the outset of the search because better features are used first in the combinations generated. lCC was set always as the first of the set with the assumption that this feature in performing so well by itself would have to be one of a best performing feature set. This reduced the number of combinations to only those of the remaining two features. This made for only about 300 different feature set combinations. The best F value obtained from these feature triplets was 43. Increasing the number of features to 7, the F value rose to 48 with features {lCC, mEd, mVe, lB, dP, cCP, mMS}. The corresponding TP and FP rates were 69 and 11, and the F1.5 score was 55. There is thus an increase of only about 5% by increasing the number of features from 3 to 7, but which is perhaps worthwhile.

8.6 Summary

There are many figures and experiments mentioned in the preceding sections, so it would seem worthwhile at this point to summarise what has been done. Assisting with this, the main elements of the work and performance that was obtained along the way are shown in

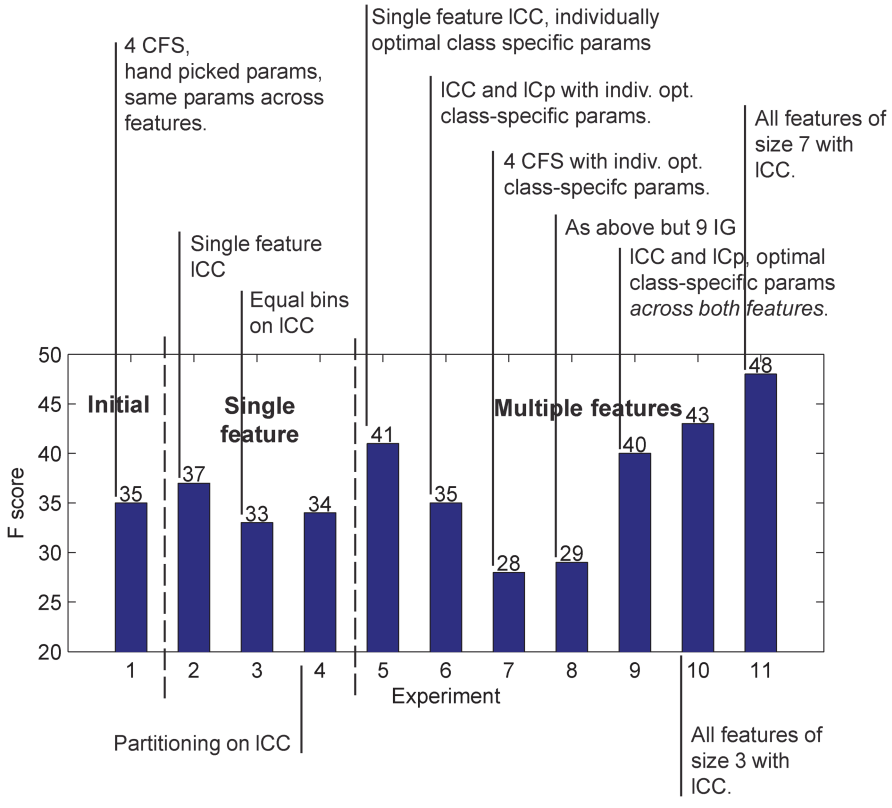


Figure 76 Summary of the work described aimed at obtaining the best performance from rank sum classification with pc4.

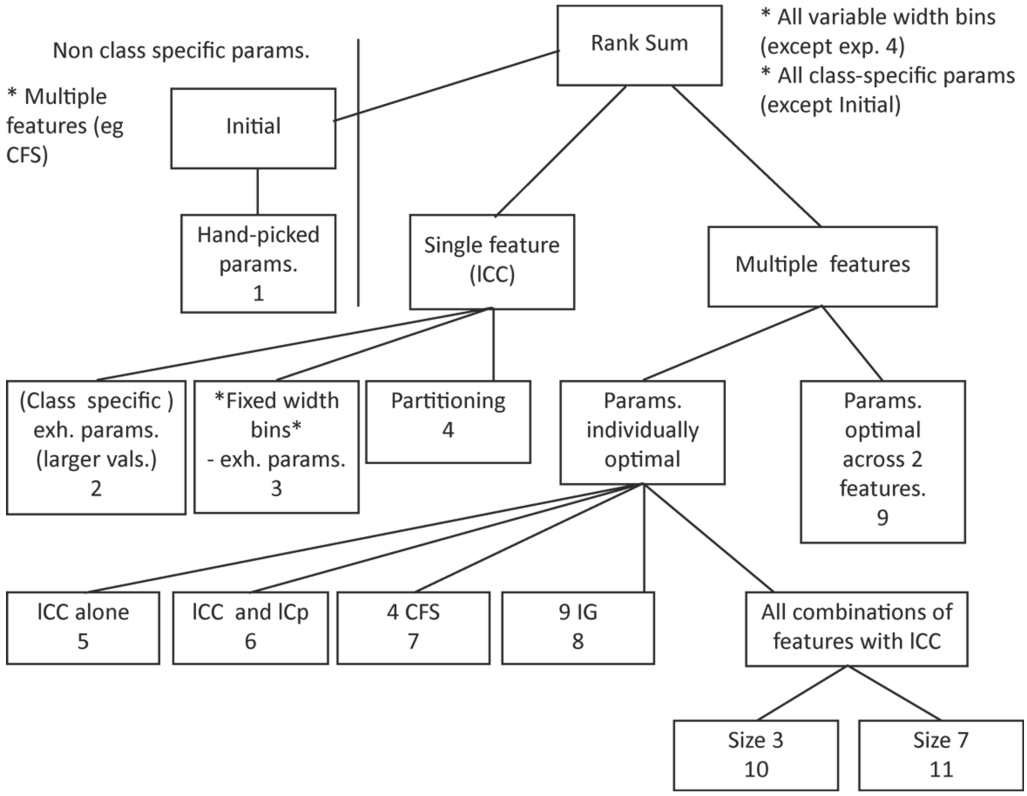


Figure 77 Structured view of the previous figure of rank sum experiments on pc4.

Figure 76 and Figure 77. The summary will trace through that for each bar in the former figure.

At the outset, there were some experiments done to test the water as it were and from which to determine a plan by which to achieve the aim of maximising performance. This involved creating a couple of models with multiple features, using rough binning parameter selections. The results from which being somewhat low relative to NB performance, it was decided to concentrate on trying to get the most performance-wise out of a single feature, which may then serve as a guide for obtaining better performance with multiple features. Having only one feature allows for experiments which might otherwise, over multiple features, be infeasible or at least more protracted. The feature this work focused on was lCC, which is undoubtedly the single feature best able to discriminate between the classes in pc4. To see what sort of performance it could give, the main variable being parameters to produce binnings by which rankings were obtained, a large number of different parameter set combinations were tried, with different sets applied to each class. This gave the F value shown of 37. The reason this was not higher was because at this stage it was thought that optimal parameters would be 30 or higher for density levels for example, when in fact it was

later found that relatively low values performed better. For comparison, performance was obtained on lCC alone with equal width bins, and this gave an F value of 33 as shown for experiment 3. So variable width binning was giving a marginal improvement only. The final attempt on lCC was with partitioning of the class distributions.

With focus turning to multiple features, before combining them in models, parameters specific to each class were found for each feature, that were optimal for each feature alone. These optimal parameters for a feature were found by evaluating models created on the single feature across a range of parameter values. This gave the result shown for experiment 5 of 41 for lCC. Two features were then combined, the better ones of the CFS selection, lCC and lCp, and this resulted in a drop in performance from that obtained by lCC alone, to 35. This drop in performance continued with additional features, with 4 CFS features and then 9 IG in experiments 7 and 8. The initial picture then from these F values in using multiple features was not overly encouraging. In an attempt to lift performance, two features were used gain, lCC and lCp, but with parameter sets tried exhaustively across both classes of both features, rather than using the parameter sets that were optimal individually. In parameter selection, this takes into account the effect of summing ranks which is ignored when features are evaluated individually, and thus these parameter sets are likely to produce better performing models. And this was the result, with the F value increasing by 5% over the previous multiple feature results to 40. This was not better than lCC alone, but at least performance was not being adversely affected noticeably with the addition of other features.

What was sought though, was an improvement in performance with additional features above that of lCC alone. The final experiments, 10 and 11, in which the best performance was obtained, explored all possible combinations of 3 and 7 features using individually optimal parameters for each feature rather than the default ones of the previous experiment. This increased the F value further to 48, the best result obtained with rank sum on pc4.

8.7 Other Data Sets

To have a better sense of how the rank sum method performs on the data sets, it is here applied to two other data sets, the next best performing data set, kc3, and the worst performing data set, pc2. For the first of these, kc3, a detailed description of the application of the rank sum method will be given, at least in its developmental form. The method used is the one that gave the best result above. It involves first finding the optimal parameters for each feature of the data set. This is done by generating a list of parameter sets, and then

applying them to each class in all combinations, and obtaining rank sum performance on each. The parameter set which produces the highest performance is the optimal set for the feature. The features are ranked according to performance. Many combinations of features are then generated of a specified length, starting with those at the top of the ranking. Rank sum performance is obtained on each feature subset combination, and a subset chosen according to the best performance.

The first step in finding optimal parameters for each feature is the most time consuming, because there are many parameter sets and combinations in which they can be applied to both classes. This might be avoided though by some method of approximation, by determining a parameter set that performs reasonably well across all features and applying that to all features. The actual parameter values from which sets were generated were [30 60 90] for #bins, [1 2 3 5 7 10 15 20 30 40] for #density levels, and only a single value 0.0005 for minimum bin width proportion. 1x10 cross fold validation is used to obtain performance on each parameter combination. The second step in trying combinations of features, which is effectively incorporating feature selection into the model building task, turns out to be quicker than it might be thought because a best first search is used in which the best performances are obtained amongst the first combinations. 10x10 cross validation is used in this step.

For kc3, the optimal parameters obtained from the first step are listed in Table 32. Just those for the top 7 performing features are shown. (Features appear in this list that were not included in previous feature selections for kc3, as the optimal parameter algorithm was applied to all features of the data set, not the correlation-reduced data set.)

Applying the second step to kc3 to find an optimally performing feature set, in which a best first search of feature sets is performed, with a subset length of 4 specified, the best F value obtained was 33, with corresponding TP and FP rates, and F1.5 score of 50, 12 and 37

	Neg class params.			Pos class params.		
	b	d	m	b	d	m
mV	60	10	0.0005	30	20	0.0005
lnOpdU	30	10	0.0005	60	2	0.0005
hT	30	30	0.0005	30	30	0.0005
mVd	60	2	0.0005	30	3	0.0005
hN	90	40	0.0005	30	5	0.0005
mIv	90	30	0.0005	30	5	0.0005
bB	90	40	0.0005	30	5	0.0005

Table 32 Individually optimal parameters for a selection of features from kc3.

respectively, and features { mVd, mGd, hL, mDd }. With 7 features, {mVd, mGd, mEd, lCC, mVe, hD, mMS}, the score increased marginally to 35 (F1.5 38).

Moving to the more difficult data set, pc2, it was of interest to find out whether rank sum could at least match performance of NB as with the other two data sets (pc4 and kc3). The same method as just described was applied to pc2. Surprisingly, in the final step of best first search of feature combinations, the best performing subset contained only a single feature, lCp. This was the feature at the top of the ranking by individual performance (as in Table 32 for kc3). The F value obtained on this single feature was 5. This appears to be a particularly poor result, but the value reflects the low precision that arises due to extreme class imbalance in this data set. The result looks better according to the TP and FP rates, which were 41% and 5% respectively.

8.8 Discussion

The ranks sum method is based on a simple idea: the closer a point is to the bins of highest density for a given class, the more likely it is that the point is of that class. Closeness is evaluated using a ranking abstraction. The ranks obtained for an instance to be classified are summed across features for a combined sum as the predictor of class.

A problem discovered with the original method was that many of these ranks were the same and were of the highest rank. If they are the same then there is no difference by which to discriminate between the classes. While this phenomenon is to be expected to some extent, as it cannot be avoided that many points will fall in the highest density bin or bins, an obvious solution which came to mind was to use more bins, providing finer resolution on density of the class distributions. This lead to an enhancement being made to the original rank sum method in which bins were made of variable width, varying in width according to density. With this, the rank sum method was finalized. Later, partitioning was considered as another enhancement to the method, but it was concluded that the additional resolution it provided did not provide much performance benefit, and was thus abandoned. In terms of properties of the rank sum method, the main one identified was that classification accuracy improved with increasing difference in rank sum. This motivated the variable width binning enhancement for higher resolution where the curve changed more rapidly, and lower resolution elsewhere.

The initial experiments were carried out somewhat ‘in the dark’ so to speak, in that there was little idea of which parameters would work best - the parameters being the variable width

binning parameters for each class, and also the selection of features. Consequently the initial results were substandard. This motivated the rather lengthy series of experiments described to try to better performance. The approach taken was a gradual one, with the aim of avoiding larger and more complex computational experiments if possible, and beginning with simpler ones. Experiments began with non class specific parameters and on a single feature, later turning to multiple features. Multiple feature experiments began with parameter selection optimal for features individually, for some standard feature subsets, then turned to the large computational exercise of finding optimal binning parameters across two features, and so on. Essentially there was a careful treading through a space of possible experimentation to cover areas that might provide some performance return, and to as efficiently as possible achieve the goal of finding best possible performance. As it turned out, the best results were not obtainable early on in this exploration strategy, and there were many experiments and eventually much computation was involved.

The best results of rank sum for the three data sets pc4, kc3 and kc2 are shown in Table 33. Best results for NB from the previous chapter are also listed for comparison. Performance between the methods is virtually identical. Rank sum performs as well as Naïve Bayes on the NASA data, but no better. This is a good result for the rank sum method, to match the performance of NB. It indicates that the rank sum method has some merit as a classification algorithm. That results were not better, for NB as well, may be due to the fact that there simply is no more discriminative content in the data to exploit. This is Menzies' view, at least in relation to existing algorithms, none of which have been able to go beyond the 'glass ceiling' of the performances obtained.

Looking at the table again, while the TP and FP rates are similar for kc3 and pc2, for pc4, rank sum gives a lower false alarm rate at 11%, at the cost of a lower detection rate. While this could be due to a bias in the rank sum method, it is more likely due to the selection of

		Features	TPR	FRP	F1.5
pc4	**Rank Sum**	lCC, mEd, mVe, lB, dP, cCP, mMS	69	11	55
	Naive Bayes	lB,lCC,mIv,mVe,dP,gCmod,mVn,lCp	95	24	54
kc3	**Rank Sum**	mVd, mGd, mEd, lCC, mVe, hD, mMS	50	11	38
	Naïve Bayes	lCC,mGd,hD,hL,mMS,lCp	47	10	38
pc2	**Rank Sum**	lCp	41	5	7
	Naïve Bayes	lC,mId,lE,dP,hI,hD,hE,mMS,mVn,lCp,lT	43	4	9

Table 33 Best rank sum results for pc4 and kc3 compared with those of Naïve Bayes from Table 17.

best model according to the F1 measure which prefers higher precision. In relation to best feature sets, there is some similarity between selections, but perhaps not as much as might be expected. For pc4, there are only 3 features in common. This may not be that significant though, as many of the features have similar relevance. The same result can be obtained with a number of different combinations of features providing similar information. For kc3, there are only 2 features in common.

While it is pleasing that rank sum was found to perform as well as Naïve Bayes, a criticism could be made of rank sum, and because of which it could be seen as less favourable in comparison. In its current form, there is a fair amount of search involved, both for parameters and features, in order to find a best performing model. One option to reduce computation could be to reduce the number of values in the parameter search space. However it is possible to obtain good results applying the same parameters to all features, which can greatly simplify the parameter selection task. For pc4, a result only 2% less than the best performance of 55 was obtained using 30 and 110 bins for negative and positive classes respectively, and corresponding density levels of 10 and 5. A similar near optimal result was obtained for kc3, using bin numbers 30 and 60 and density levels 2 and 30. Common to both is a higher number of bins for the positive class.

Some comments can be made in relation to the partitioning approached. While the partitioning idea seemed to have potential, the results did not show that to be the case. In reflection, while it is clearly the case that more resolution was gained on the less variable part of the curve as was desired, this might not actually be helpful to the rank sum method, which seems to prefer (or performs as well with), at least in some instances, fewer bins. If the data is thought of as a cloud of points, the tail that is seen in Figure 72 say - the resolution of which is increased by partitioning - is the area around the centre of mass of the points where point density is not high, and where minor fluctuations over the range of the feature are probably not significant. That is, the increased resolution that is provided through partitioning over this region probably contributes little to assist with discrimination between the classes. It may be that the original binning without partitioning, referring to Figure 72 again, where the tail is treated rather coarsely in binning, is just as good if not better than the higher resolution ones with partitioning.

On a final point, Menzies found that log normalizing the data improved classification performance significantly with Naïve Bayes. It was thought this might be the case with rank sum as well, and that with log normalization it would be possible not just to match NB, but to

outperform it. However on applying log normalization to the best performing feature set for pc4, the F1.5 score dropped by around 5%. That the effect of normalization was having the opposite effect to that with Naïve Bayes, lead to some investigation on the issue. Looking at the rank sum bin configurations on a single feature for untransformed and log data, did not reveal anything that would explain the drop in performance on the single feature. The log transformed model looked satisfactory. A different avenue was then pursued in seeing if customizing parameters to the log data could turn around the effect on performance. After a search for optimal parameters and features on the log data, an improvement was found, with the F1.5 score increasing to a new best score for pc4, 57%, which is 3% higher than Naïve Bayes. While this is a good result, it might suggest that the rank sum method is somewhat parameter dependent, at least in relation to log normalization. Looking into why some features performed better than others with log normalized data revealed that while some individual features benefitted more than others through log normalization, they weren't necessarily the ones in the optimal feature set.

In the next chapter, a novel use of rank sum was found, in which training instances are mapped into two dimensional rank sum space according to their rank sum values for each class, in which some class separation was evident. Being two dimensional, the mapping provides a simplification of the data and the classification problem. By itself points in this form do not constitute a classifier, but classification may be achieved by applying SVM to find a decision boundary separating the classes.

Chapter 9 – Trade-off Models on Rank Sum Data

In the previous chapter, it was established that in applying the rank sum method to test instances, a rank sum value is produced for each class for each instance. A scatter plot of these values was shown in Figure 61. There was insufficient consistency though in their distribution for this information to be of any value beyond direct classification based on the sums. However, these observations of rank sum points were made at the outset of the development of the rank sum method. As the method had been improved since, the output of rank sum values was revisited. Rather than looking at the rank sum values on test points, the idea arose, what if the rank sum model was applied to the training data? This would reduce the original N dimensional data to only 2 dimensions. The transformation was applied to the whole data set, rather than on a sample, and what was observed in this instance was an apparent polarization of the classes. Negatives were more heavily populated on one side of the point cloud, and positives on the other. An example of this pattern for pc4 is shown in Figure 78. A PCA plot of the original data is also shown for comparison to see an apparent improvement in class separation with rank sum transformation. It would appear that separation is enhanced with the rank sum transformation.

This finding was considered to be of value, as with separation brought about by any other means, and immediately it seemed that this could be exploited with the use of SVM which could find an optimal decision boundary between the classes. But this alone would not be sufficient to warrant the extra step involved in learning a model from the rank sum data, assuming that prediction performance obtainable was a good as that with rank sum alone. There is an advantage though that an SVM model might offer, and that is the ability to adjust the model parameter-wise to effect a trade-off in performance between the detection and false alarm rates. As observed by Menzies and in this work, different models produced by any learning method perform according to a trade-off curve between these performance values. A higher detection rate is only had at the cost of a higher false alarm rate. A lower false alarm rate at the cost of a lower detection rate. This might be seen as the curse of this fault prone modelling problem. Different rates are usually obtained depending on different feature selections for example. With the use of SVM, rather than obtain a particular performance trade-off somewhat arbitrarily, it might be possible to obtain a particular trade-off by choosing particular SVM parameters. If by this method performances were as good as by

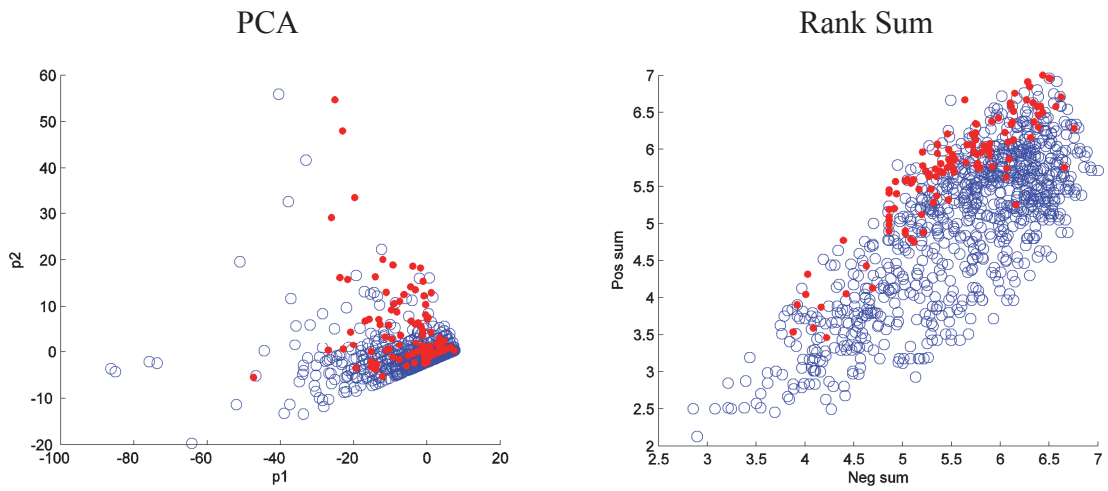


Figure 78 Rank sum points for pc4, and compared with same instances in 2D PCA space.

other methods, then this SVM approach in offering the ability to additionally control trade-off would have merit over standard methods, and in practice its models would have greater utility.

The chapter therefore focuses on applying SVM and varying its parameters on the rank sum transformed data to define a trade-off curve in detection and false alarm rates. The user of the model would then simply refer to this trade-off curve and choose the values corresponding to the desired trade-off. Included in the parameters is the kernel, of which a number are tried including linear, polynomial of varying degrees and radial basis function (RBF). While the RBF kernel gave the best results in the modelling chapter, Chapter 7, it cannot be assumed that it will also work best in this application. The transformed data is of a different nature and may be considered a different problem, and for this reason other kernels are tried as well. Most of the chapter is devoted again to the pc4 data set because it has shown to be the one that gives best performance of all the data sets. But as in the previous chapter, an additional data set is included for completeness, and this is the same as before, kc3, because it would appear to be from results obtained earlier the next best performing data set after pc4.

Throughout the chapter there are plots of data points with superimposed SVM decision boundaries. Usually these points are training points with the derived boundary, which indicates goodness of fit of the boundary found by SVM to the training data. The performance reported however is based on associated test data. An example of test data for an obtained quadratic decision boundary is shown in Figure 79. From this, classification counts can be obtained from which performance rates are reported. These counts or outcomes are

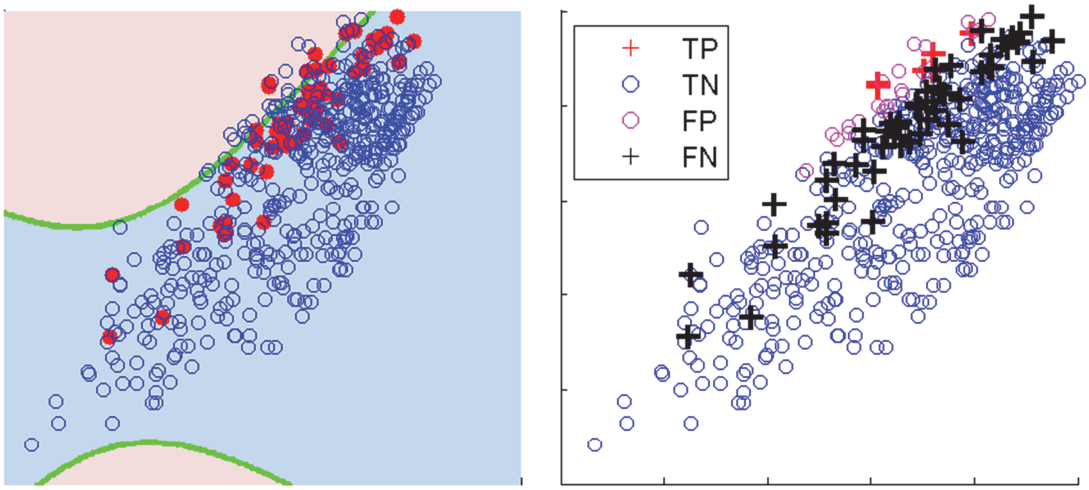


Figure 79 An SVM decision boundary is shown on the left by which a set of test points is classified, the positive side of the boundaries shaded red, and on the right, another view marks the test points according to category of predictive outcome (positives as pluses, and correct classifications in the standard red and blue). In this figure and all other SVM decision boundary plots, the x-axis represents the negative rank sum, and the y-axis, the positive rank sum. The decision boundary is shown in green, and not shown here, blue and red lines show the hyperplanes wx+b=+-1. All subsequent SVM plots show the training data from which the decision boundary was derived rather than the test data.

illustrated in different ways in the figure. On the left, points are marked only according to class and the classifications into the different categories of TP, TN, FP and FN are inferred from the decision boundary. Any points lying on the positive side of the decision boundary, the region shaded red, are classified positive, else negative. Points are marked more explicitly according to classification in the plot on the right. All actual positives are indicated with a + sign, and all actual negatives with an o. Predictions are indicated by colour as shown in the legend. Positives predicted positive are red; false positives magenta. Negatives predicted negative are blue; false negatives black. The two plots show the same information but present it differently.

On an implementation note, while modelling experiments before, including SVM, were conducted using the Weka machine learning tool, for the SVM experiments reported in this chapter, the Spider Machine Learning toolbox (The Spider) for Matlab was used instead, which relies on the well known LIBSVM library. This tool was used as it provides plotting functions through Matlab on the SVM decision boundaries learnt, which can be of help in

evaluating the goodness of learnt models. Also it should be mentioned that the F values reported in this chapter, rather than being derived from the average performance rates from cross validation, are instead the average of F values, but these different methods used would provide values that are comparable.

9.1 Linear Kernel

The linear kernel is the simplest of the three kernels tried. The decision boundary it finds is simply a straight hyperplane, or in this case 2D line. The kernel does not perform any mapping of the data in instance space into some other transformed feature space. However, given the nature of the dispersion of points with positives and negatives being polarized according to a roughly linear diagonal, this kernel has potential to find a good decision boundary.

Initially the kernel was applied using a range of C values: [Inf 100 10 5 1 0.5 0.05]. The first of these represents a hard margin, and the remaining values soft margins. These 7 values are used in many of the subsequent experiments where the C value is to be varied. In terms of sampling, given the size of the pc4 data set, a 50/50 split would likely be satisfactory, in particular in representing the positive class in the training set, and this has been used in some subsequent experiments. However here, 5*10CV has been used. Applying the kernel with the C values mentioned gave the performance results shown in Table 34. Evident is the problem as mentioned in the modelling chapter, of class imbalance causing the decision boundary to lie too close to the positives resulting in low detection and false alarm rates. With a softer margin there is an obvious tendency to default to negative classification, as indicated by the zero values in the table. Clearly, there was a need to overcome this imbalance problem, and this was done using oversampling, described next.

C	TPR	FPR	F1	F1.5
Inf	14	2	21	18
100	0	0	0	0
10	0	0	0	0
5	0	0	0	0
1	0	0	0	0
0.5	0	0	0	0
0.05	0	0	0	0

Table 34 Linear kernel results over a range of C values.

The oversampling experiment with linear kernel was straightforward. The C value was set to hard margin, as this gave the best result before without oversampling, and varied was the training set, to which oversampling was applied on the positives at increasing levels from 0% to 100%. The oversampling percentage of positives is relative to the size of the negative class, so at 0.5, there are half as many positives as negative, and at 1 the classes are balanced. Random sampling was employed[3]. Employing this method and the simplest of kernels, the results were surprisingly good. The highest F1.5 score at an oversampling level of 50% was 55%, one of the training sample decision boundaries for which is shown in Figure 80. It can be seen that the boundary is well placed in separating the classes. As well, a trade-off behavior was evident in the F values as oversampling level increased from 0 to 40%. Computing performance at finer steps of oversampling within this rage, increasing by 5% instead of 10%, produced the trade-off curve shown in Figure 82.

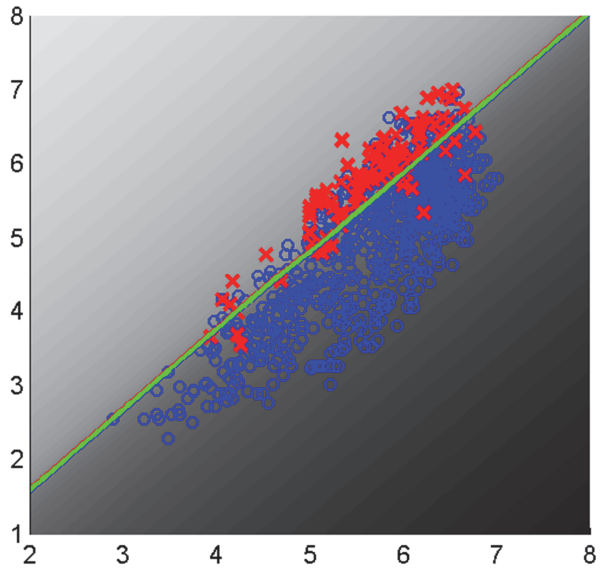


Figure 80 Linear decision boundary with 50% oversampling on the positives providing what appears to be an ideally positioned linear boundary between the two classes of points.

[3] There are more sophisticated sampling techniques than random, but it has been reported in the literature that often random works as well as them. Apart from this, the purpose of oversampling is not for such a fine effect on prediction, it is merely to push the decision boundary lower to better classifying existing points, and by this justification the random approach should be adequate.

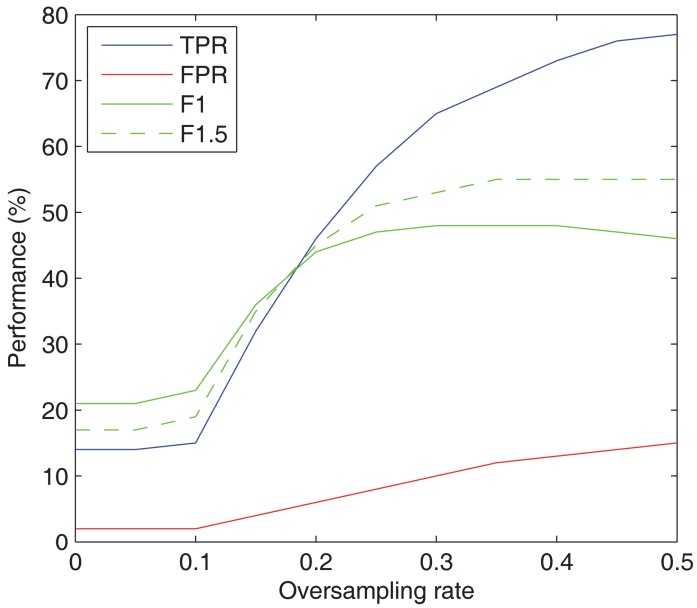


Figure 82 Performance on the linear kernel with increasing levels of oversampling, effecting desired trade-off curve.

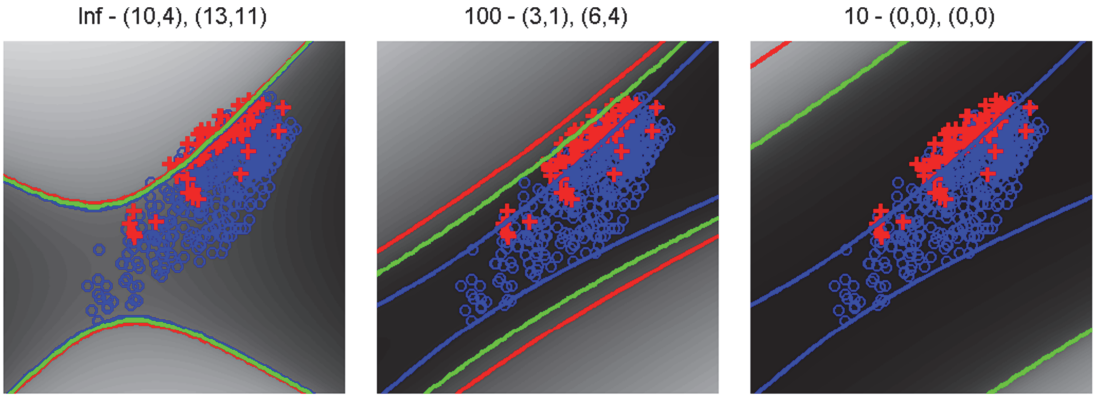


Figure 81 Quadratic kernel decision boundaries with an increasingly softer and less predictively accurate margin on a 50% training sample. Numbers at the top are "C – (TPR, FPR) (F, F1.5)".

9.2 Polynomial Kernel

A polynomial kernel of degree 2, or quadratic was first applied to a single 50/50 train and test sampling of the pc4 data and over the same range of C values as used with the linear kernel. A few of the resulting decision boundaries are shown in Figure 81. The first on the left is with hard margin, and to the right the margin becomes progressively softer. C value is given as the first number in the title of each plot. The numbers following it are the TP and FP rates

as a parenthesised pair, and then the F1 and F1.5 scores, also parenthesised. The best performance is obtained with the hard margin, with 10% detection and 4% false alarm rates. As the margin becomes softer, the decision boundary shies further away from the relatively small cloud of positive points, until both positive and negatives lie completely on the negative side of the boundary and no points are classified positive at all. Repeating this over 10 train and test samples gave the same result, with F values being similar to those for the linear kernel in the previous table, Table 34. As with the linear kernel, but shown graphically in the previous figure, the decision boundary is too close to the positives and the softer margins cause even less inclusiveness of positives, resulting in lower detection rates. The same approach of oversampling to deal with imbalance and to increase the detection rate, but with the quadratic kernel is applied in the next section.

Other polynomial kernels of different degree were tried as shown in Figure 83 for different C values. With degree 3, the curve is virtually linear, and while this resulted in a boundary more inclusive of positives and a higher detection rate, softer C values had the same effect of reducing it. Softening the margin allowed no room for improvement, and its effect on the

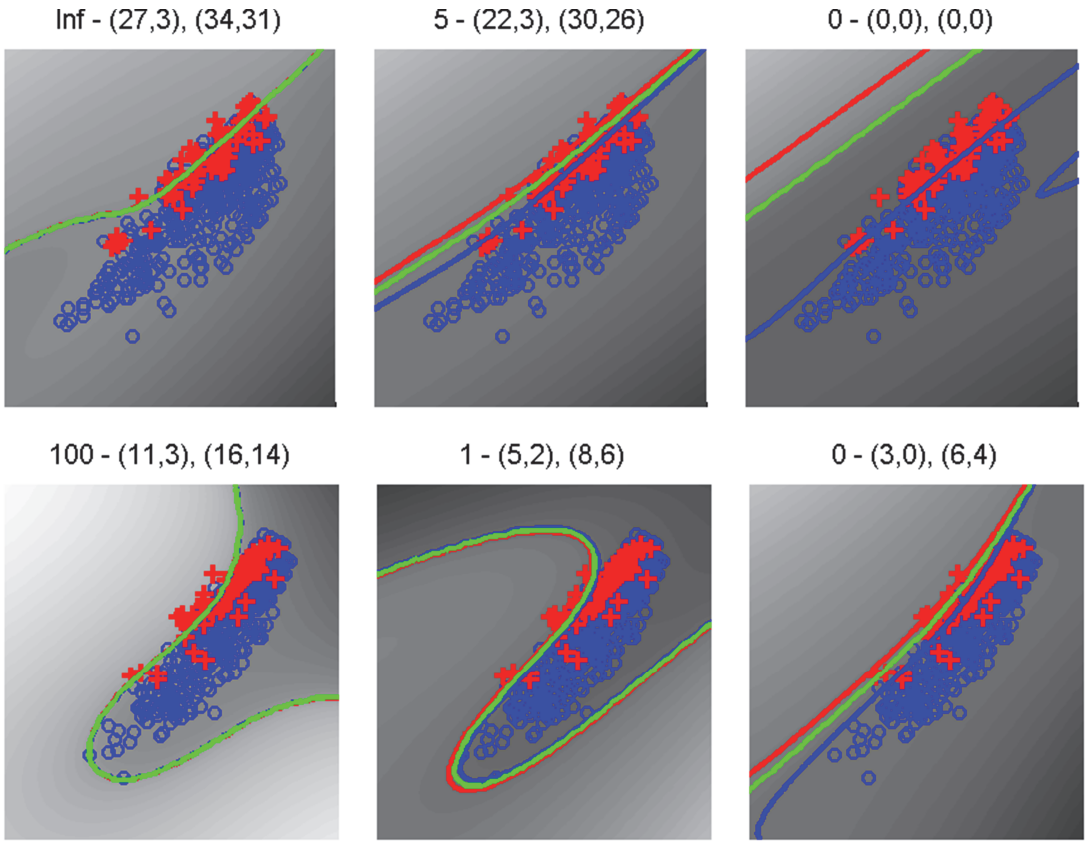


Figure 83 Decision boundaries for polynomial kernels of degrees 3 (row 1) and 4 (row 2) none of which appear to offer any advantage over the linear kernel.

decision boundary was virtually the same as for the quadratic kernel. Additional curviness with degree 4 seemed not to offer any advantage.

9.2.1 Quadratic with Oversampling of Positives

Oversampling is used here as a means by which to increase the representation of positives so that they may be better recognized by the SVM algorithm, and through which the detection rate maybe increased – a prerequisite to finding a trade-off curve. An initial spot test was done to see whether oversampling increased the detection rate (above the previous best detection rate with the quadratic kernel of 10%). This is similar to that in the previous section in applying the quadratic kernel to a single sample, but in this case with a single oversampling rate. The spot test was done with C=Inf only, and oversampling involved random repetition of positives until the desired percentage of positives was obtained, in this case 20%. The decision boundary and performance is shown in Figure 84. With a detection rate reaching 50%, this is a much better result, with oversampling enabling SVM to better recognize the positives.

Following this spot test, oversampling levels were explored more thoroughly, over the range 0% to 100% by 5%, with 0% being no oversampling. With a higher detection rate achievable as the last the result showed, the interest in running this more exhaustive experiment was not only to see whether better performance could be obtained, but also whether there was any trade-off pattern evident in the performance figures with the varying oversampling level. A

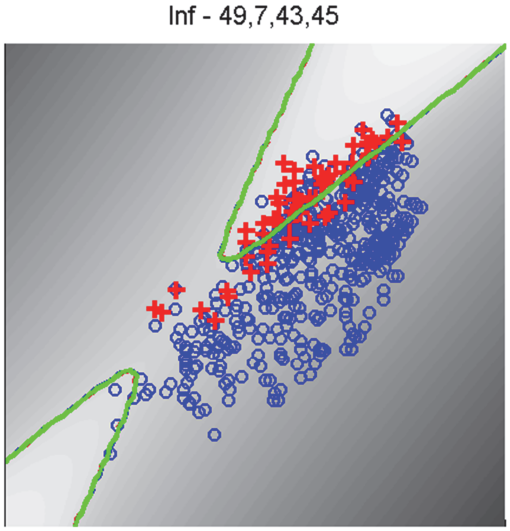


Figure 84 Spot test with quadratic kernel and oversampling at 20%.

hard margin was used as before, as softer margins only diminished performance. Also as before 10 samples were taken and the average of these obtained for reported performance. On each sample a model was generated for each oversampling level. The number of models generated in total was 21 oversampling levels * 10 samples = 210 models.

The performance obtained at each oversampling level, produced a desired trade-off curve, as shown in Figure 85. Detection rate increases as oversampling level increases, at the cost of an increasing false alarm rate. The best F1.5 value is 49 at oversampling level 0.6. This is 6% below that obtained with the linear kernel (Figure 82). This is due to higher false alarm rates which creep up to 25% (versus 15% with linear).

As a matter of interest, the effect of increasing the oversampling level on the decision boundary relative to the positives can be seen graphically in Figure 86. These plots were taken from one of the samples in the above experiment. As found with the spot test and indicated by the trade-off plot, inclusiveness of positives increases as oversampling level increases.

Another means by which to effect a trade-off given that a higher detection level was possible with oversampling, specifically to contract the positive region to reduce false alarms and with that the detection rate, was thought to be in softening the margin. This had the effect in previous results of causing the decision boundary to be shy of the cloud of positives which is the desired effect in this case. But surprisingly, varying C with softer margins had little effect

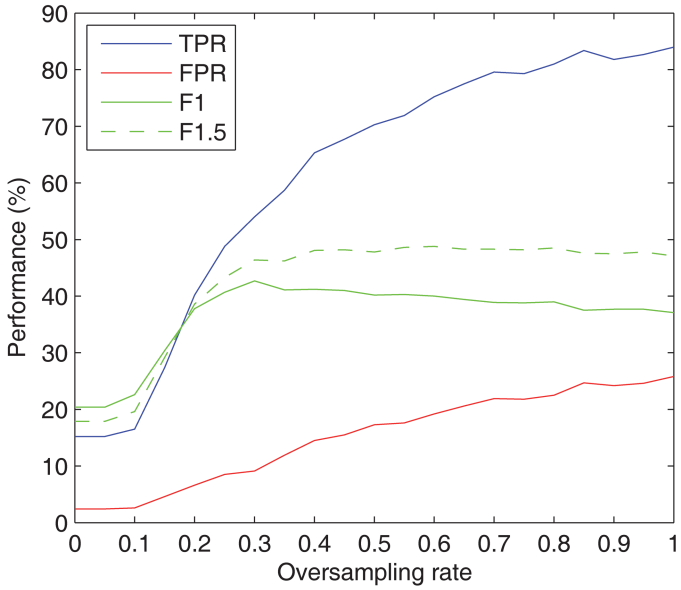


Figure 85 Performance on quadratic kernel with increasing levels of oversampling, effecting desired trade-off curve.

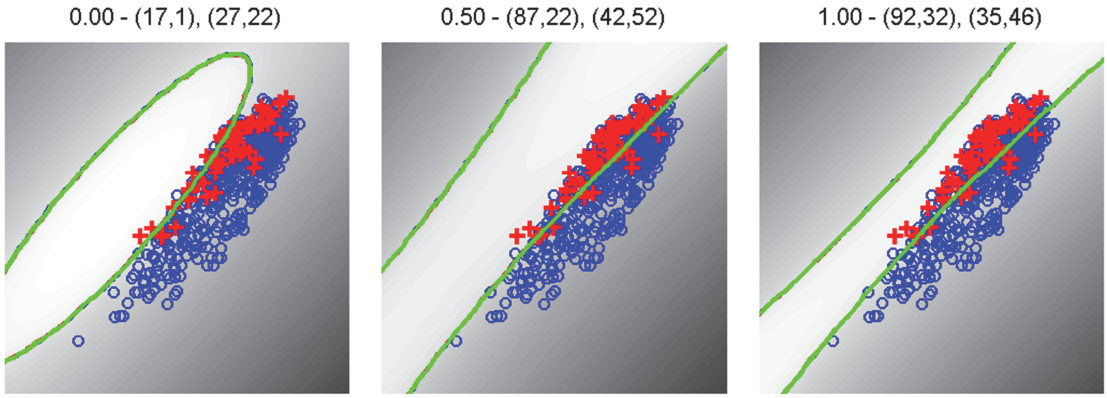


Figure 86 A few of the decision boundaries generated in producing the performance curve in the previous figure with quadratic kernel, hard margin and an increasing oversampling level.

on performance or the decision boundary when oversampling had been applied.

Yet another means of achieving a trade-off is discussed in the next section, which involves the use of a so called 'balanced-ridge' parameter to SVM.

9.2.2 Quadratic with Balanced Ridge Parameter

A parameter of the Spider implementation of (or interface to) SVM is a so called 'balanced ridge'. It is suggested this might be used to better handle class imbalance. The balanced ridge parameter is a regularization parameter and works the same as a ridge parameter. The effect is to assign more importance to the correct prediction of positives. This might have a similar effect to oversampling and increase the detection rate as desired. If it does then by varying this parameter or C, the desired trade-off behavior might be obtained.

With C=Inf the ridge parameter had no effect on prediction performance. Thus this was

Bal. Ridge	C	TPR	FPR	F1
3	**Inf**	86	19	45
3	**100**	86	19	45
3	**10**	86	19	45
3	**5**	84	19	44
3	**1**	60	1	45
3	**0.5**	22	4	27
3	**0.05**	0	0	0

Table 35 Selected balanced ridge performance in which trade-off behavior is evident with varying C.

excluded from the set of applied C values. Balanced ridge values used were [0.05 0.1 0.5 1 2 3 4 5 10 20]. With these parameter sets defined, SVM models were generated for every pair of parameter values. Because of the large number of pairs, the number of samples to which each pair was applied was reduced from 10 to 5, for this experiment. The total number of models generated was thus 7 C values * 10 balance ridge values * 5 samples = 350 models. The performance generated for each pair was perused looking for a suitable trade-off pattern, the best of which was found when balanced ridge was equal to 3. The trade-off was occurring as C varied over this balanced ridge value from values 0.05 to 5. This can be seen in Table 35, which shows just the performance values when balanced ridge is 3.

In order to obtain a trade-off curve, the above experiment was repeated, but for just the balance ridge value identified and smaller increments of C. This produced the curve shown in Figure 87. As for the oversampling trade-off curve, there is a steep increase in detection rate which flattens out, accompanied by a gradual low lying rise in the false alarm rate. The best F1.5 value is 51 when C is 2.25. This is slightly better than the oversampling result, but 4% less than the linear kernel result. This completes the work on the quadratic kernel for pc4. The next sections focus on obtaining a similar trade-off curve using the RBF kernel.

9.3 RBF Kernel

The RBF kernel adds another parameter to the modelling task, which is γ. The aim is to find

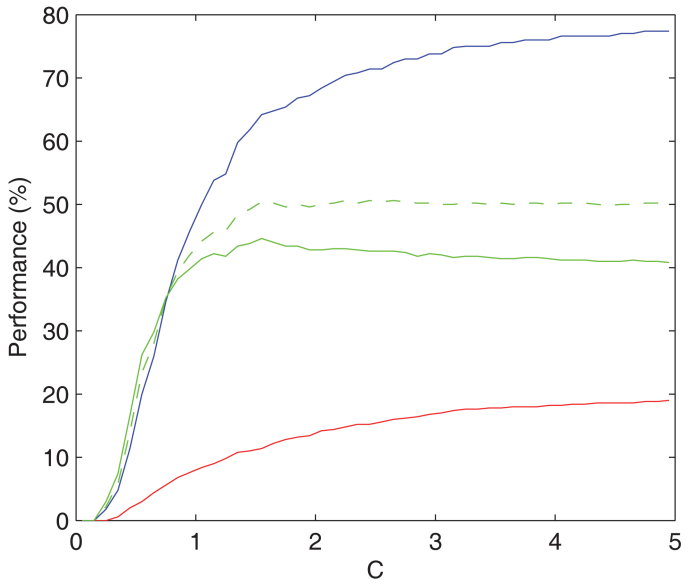


Figure 87 Performance on quadratic kernel as C is varied over a balanced ridge value of 3, exhibiting the desired trade-off.

the value for γ, C and later oversampling level, which give the best performance and on which a trade-off curve can be based. As with the quadratic kernel, first tried are the hard margin with different γ levels. Following this, softer margins were tried, also with different γ levels. Both hard and soft margins yielded better performance than those with the quadratic kernel. However they were not as high as desired, and so focus then turned to oversampling. What was observed though from the hard and soft margin experiments were a couple of behaviors in the RBF decision boundary as values for γ and C, were varied which will be discussed shortly.

9.3.1 Hard Margin

Beginning with the hard margin, the only parameter that changes given that C is fixed is γ. The values chosen for γ were similar to those in the modelling chapter (Chapter 7) being just one value less (without value 1.5), namely [0.05 0.1 0.2 0.3 0.5 0.7 1 3]. As previously, models were generated across 10 samples to obtain an average performance that was more stable. The results of this are shown in Table 36. Performance for $\gamma \leq 0.3$ are quite reasonable with the F1 value reaching 32 and detection rates that were not trifling, These were certainly better than the initial results with the quadratic kernel (without oversampling) in which the F1 values did not exceed 15.

Without any picture at this stage of what the RBF decision boundary looked like with different γ values and fixed C at Inf, these were plotted in case some pattern might be observed that would be useful in guiding the present effort to find a decent trade-off behavior. The plots were obtained from a single sample from the previous table. Shown in Figure 88, it can be seen that as γ increases the decision boundary tends to be less tightly fitting to the training data, and it expands in area to create a bubble effect which becomes quite extreme at the maximum value of 3. It perhaps could be said that larger γ expands positive regions to

γ	TPR	FPR	F1	F1.5
0.05	29	5	31	30
0.1	33	7	32	33
0.2	34	8	32	32
0.3	30	7	30	30
0.5	18	4	22	20
0.7	15	3	20	18
1	13	2	18	16
3	7	1	11	9

Table 36 RBF performance with hard margin and varying γ.

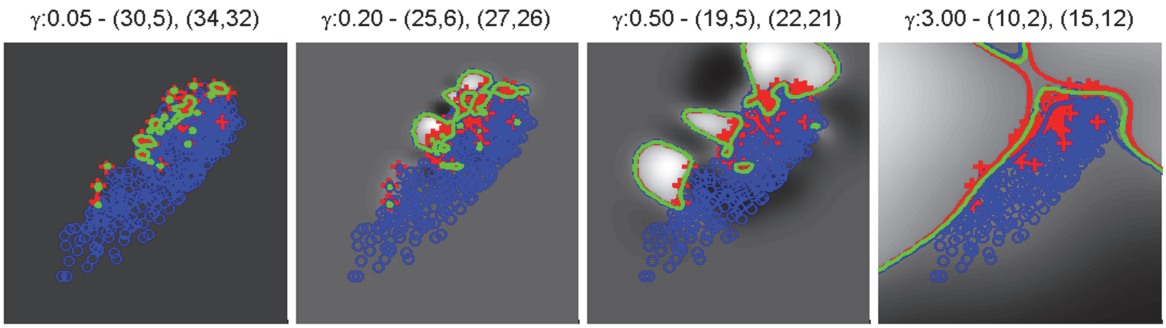


Figure 88 RBF decision boundaries for hard margin and increasing γ from the previous table (Table 36) (but only a single SVM model from each CV set) causing a bubble effect on the positive regions.

increase generalization of the model. Although, that it is not to say necessarily that it is more inclusive of positives. From the previous table the higher γ values with larger bubbles actually resulted in a lower detection rate. Its usefulness therefore is questionable, at least at the higher end of the γ value range. It would seem based on the plots that the optimal setting for γ is somewhere between 0.2 and 0.5, the middle two plots – not too tightly fitting, nor too loosely fitting.

9.3.2 Soft Margin

While the hard margin performance is better than hard margin with the quadratic kernel, it is still lacking. So attention turns now to soft margins. With the quadratic kernel, the soft margin only caused the decision boundary to shy away from the positives and the detection rate to lessen. This might be reason to doubt whether softening C will be of any benefit with the RBF kernel, but being a different kernel, its effect on the decision may not necessarily not be useful. First a spot test was done to have some idea of what C was doing to the decision boundary for a given γ level. Different C values were tried for γ at 0.5, the results of which are shown in Figure 89. It can be seen that as the margin softens the bubbles of positive regions contract. This may be due to the softer margin being more tolerant of error, allowing more positives to lie outside the positive regions. This contraction effect might be seen as a counter force to the expansion of position regions with larger γ.

With two effects in play, of expansion of the positive region with larger γ, and contraction with softer C, it seemed possible that an optimal balance might be found between the two that would give the desired performance. To see if this was the case, models were generated for every pair of C and γ in the ranges already defined. This was only done on a single sample

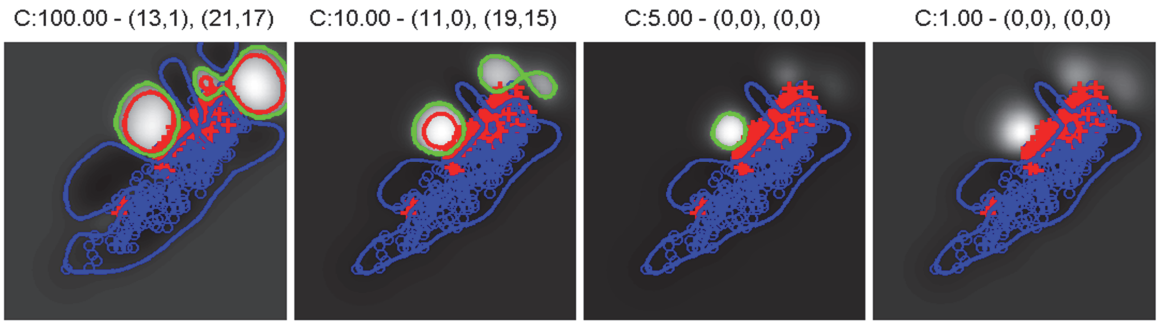


Figure 89 RBF decision boundaries for softening margin and fixed γ at 0.5, contracting the positive regions.

due to the large number of parameter pairs (7 C * 8 γ =56 of them). A table of results is not presented as the performance was not as high as it needed to be with the F1 value increasing only from 32 reported above with hard margin, to 37. It should be mentioned that being only the results on a single sample, better results might be obtained on other samples, but the sense was that the value would unlikely be lifted much and that effort would be better directed at oversampling which worked well with the quadratic kernel.

9.3.3 Oversampling

As for the quadratic kernel, oversampling involves repeating positive instances until the desired number of positives is reached, as a proportion of negatives. With the quadratic kernel this was only done with hard margin, and softer margin caused the positive decision region to be less inclusive of positives. Here, the hard margin will be tried, but also the soft margin, as it will not necessarily have a negative effect as it did with the quadratic kernel.

For the hard margin, C was set to Inf and γ to 0.1, and oversampling was varied as before from 0% to 100%. A surprising result was obtained here in that oversampling had no effect on performance. The table of results need not be presented, suffice to say that the figures were all very similar across oversampling levels. This lack of any effect on performance was also found with other γ values, so attention turned next to oversampling with soft margin.

With the soft margin there was a chance that as with hard, oversampling would not make any difference to the decision boundary. But as it turned out, it did make a difference. A spot test was done again, this time testing the effect of oversampling with soft margin. A model was created without oversampling, and another with oversampling at 30%, both with C at 10 and γ at 0.4. Plots of these are shown in Figure 90. It can be seen that the higher oversampling

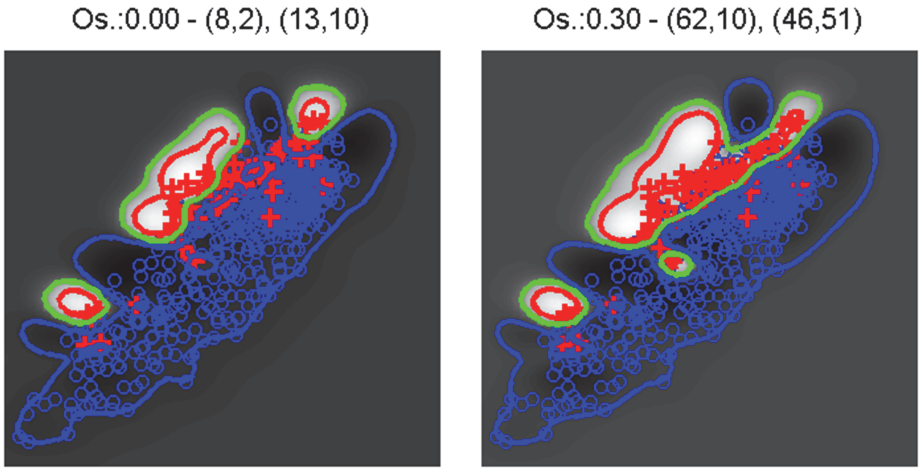


Figure 90 The effect of oversampling with soft margin, improving the positive decision region.

value on the right enlarges the positive region and gives much better performance, the F1.5 score rising from 10 to 51. This showed that using the soft margin can play a part in improving performance. This lead next to exploring oversampling with soft margin more thoroughly with different parameter values.

The more thorough exploration of oversampling with soft margin involved trying all combinations of oversampling and C values, to produce a 2 dimensional table of models, with rows and columns corresponding to each parameter respectively. This was done for γ values 0.1, 0.3, 0.4 and 0.5. Additionally, this was repeated 5 times on different 50/50 splits of the data. This effectively adds a third dimension to each table, each sample making for an additional 'sheet'. For each sheet, oversampling at different levels is made on the same sample. The number of models generated for each γ is 7 C * 8 oversamples * 5 samples = 385 models. The maximum oversampling level has been reduced to 0.7, as it was found above this performance did not improve.

The best results from the above were obtained with $\gamma = 0.4$. The table of results at this γ level is shown in Table 37. Plots of decision boundaries for models in one sheet of the table are shown in Figure 91. These models do not correspond directly to performances in the table, but contribute to them, and may be regarded as representative of the variation in pattern of the boundary as parameter values vary.

Looking at the plot grid first, it will be noticed that the boundaries at a glance are quite similar, for the γ value chosen. Correspondingly many performance values in the table are

	Inf	100	10	5	1	0.5	0.05
0	24 5 24 24	15 3 19 17	13 2 17 15	12 2 17 15	6 1 10 8	3 0 5 4	0 0 0 0
0.1	24 5 24 24	16 4 19 18	13 2 18 15	12 2 17 15	7 1 11 9	4 0 6 5	0 0 0 0
0.2	28 6 26 27	32 6 32 31	35 6 35 35	35 6 34 34	33 6 33 33	31 5 31 31	3 0 4 3
0.3	40 9 33 35	46 10 37 40	53 10 41 45	53 9 42 46	51 9 41 44	49 9 40 43	25 4 24 24
0.4	43 10 35 38	61 13 42 47	62 12 43 49	61 12 42 48	60 11 42 47	62 12 43 49	51 10 38 42
0.5	17 4 13 15	63 14 41 47	67 13 44 51	67 13 44 51	66 14 43 50	67 13 43 50	61 13 41 47
0.6	26 6 21 23	64 14 41 47	69 15 42 49	71 15 43 50	70 15 43 51	70 15 43 51	70 14 44 51
0.7	26 6 21 23	66 15 41 48	75 16 43 51	75 16 43 51	75 17 43 51	74 17 43 51	73 16 43 51

Table 37 RBF performance with varying oversampling level (row) and margin (col), with γ fixed at 0.4.

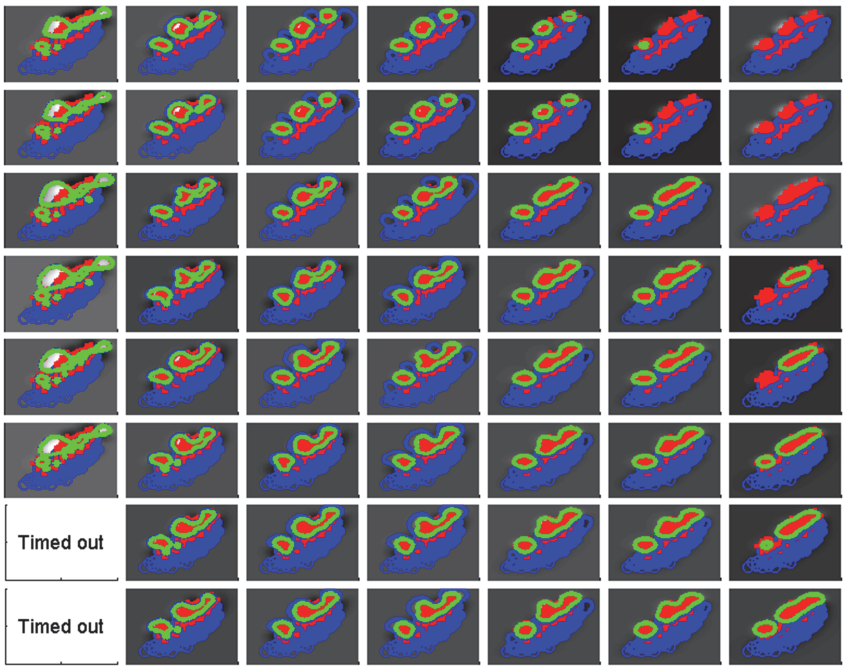


Figure 91 RBF decision boundaries for one slice (or sample) of Table 37.

similar. Despite this, it can be seen in the plots that there is a bubble effect with hard margin, caused by the higher γ value of 0.4. This bubbling decreases, or positive regions contract, moving right across the grid of plots, as the margin becomes softer. This is according to the behavior identified earlier of softer margins contracting the positive region (see Figure 89). The other behavior at play also observed previously, is the enlarging of the positive region as oversampling level increases (see Figure 90), moving down the grid. This is more noticeable when positive regions are smallest to begin with which occurs in the far right column of the

table when C is softest. The contraction effect is working right to left, and countering this an enlargement top to bottom.

Looking at the table, the best results are at the bottom right corner. One may simply look at the bottom row, where performance is virtually at optimum level. The F1.5 value is at 51, which is a good result, the same as that obtained with quadratic kernel with balanced ridge. With this result being good enough on which to base a trade-off curve, the focus next is on obtaining performance values which describe a trade-off behavior. This is found not horizontally across the table but vertically, down columns, with low detection rates at the top rows, and higher rates at the bottom ones. The column that appeared to provide a transition of values that would describe a smooth trade-off curve was C=10. This begins at 13,2 for TP and FP rates, and increases to 75,16.

From the above it was possible to produce a trade-off curve by setting C to 10, γ to 0.4, and varying the oversampling rate from 0 to 0.7. Increasing oversampling level in fine increments for a smoother curve resulted in the trade-off curve shown in Figure 92. The best F1 value from this curve is 41 at oversampling rate 0.34, and F1.5 value 49 at oversampling rate 0.66. This is a bit less than the result obtained before with quadratic kernel and balanced ridge.

This concludes the work on pc4. A minor point that might be mentioned is that the balanced ridge parameter could be applied with RBF kernel as it was with the quadratic. This was

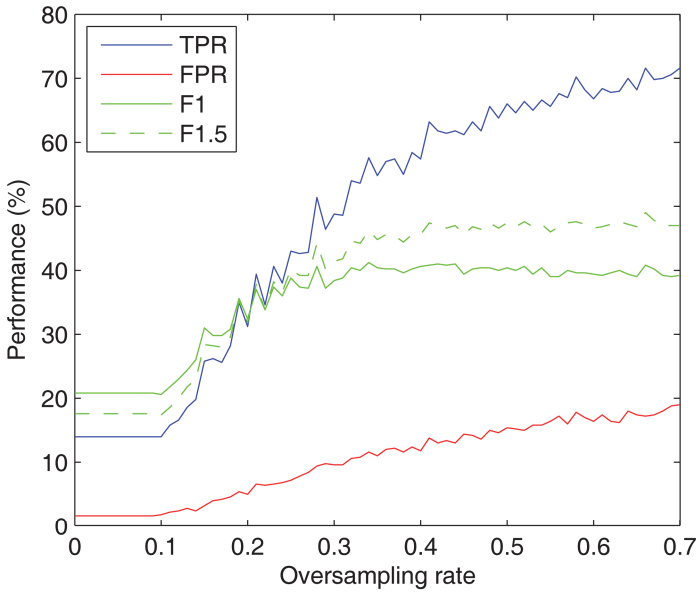


Figure 92 Trade-off curve obtained with the RBF kernel by increasing the oversampling rate with fixed soft margin = 10 and γ = 0.4.

tried, however the performance it produced was slightly less than with oversampling, so it has not been covered here.

9.4 Another Data Set

The data set used in previous sections, pc4, is a convenient one to work on, as it was found previously to be the best performing. Of any data set it would be most likely to provide separation in rank transformed data, and be amenable to the task of finding the trade-off behavior sought. The approach taken with pc4 is further validated in this section by applying it to another data set. As for the previous chapter, the data set chosen for this is kc3, partly for consistency, but also because it was found to provide relatively good performance in the modelling experiments.

The first part of the analysis is to see if there is some separation between the classes in the rank transformed training points as there was for pc4. For this the rank transformed points are shown in Figure 93, and there would appear again to be some separation, enabling the work to proceed to find a suitable trade-off curve.

In the next sections, each of the three kernels are applied, and only with oversampling as this was found to improve performance on pc4.

9.4.1 Linear Kernel

Use of the linear kernel with kc3 is the same as for pc4. Oversampling level is increased from

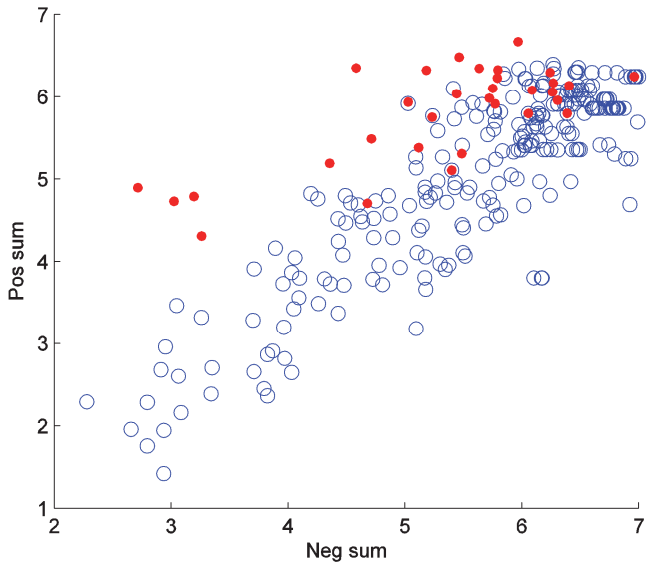


Figure 93 Class separation evident in plot of rank sum transformed kc3.

0 to 100% and performance was obtained using 10*10 CV. Rather than present the table of performance results for all oversampling levels, just a subrange is plotted as found suitable for a trade-off curve, this range being 0 to 30% as shown in Figure 94. The highest F1.5 value is 38% at an oversampling rate of 25%.

9.4.2 Quadratic Kernel with Oversampling

There are two parameters in this case with the quadratic kernel, C and oversampling level. For the previous data set, pc4, C=Inf worked best without oversampling, so it was assumed that would still be the case when oversampling was applied. While this assumption could be made here, it seemed a reasonable approach to explore other values of C, reusing code from the RBF analysis but with quadratic kernel. The RBF analysis above produced models in tabular form with each a parameter represented by the rows and cols of the table (and samples in the 3rd dimension) and for the same variables under investigation here. The number of samples or sheets was set to 10.

The results from this were fairly poor, with the maximum F1.5 value being 29, far below the NB F1.5 value of 55. There would seem little point in showing the performance table in this case. Curiously while better results were thought to be obtained when C=Inf (for quadratic analysis of pc4), in this table better performance occurred with smaller C values. This might indicate that other C values in the earlier analysis could have been explored and may provide

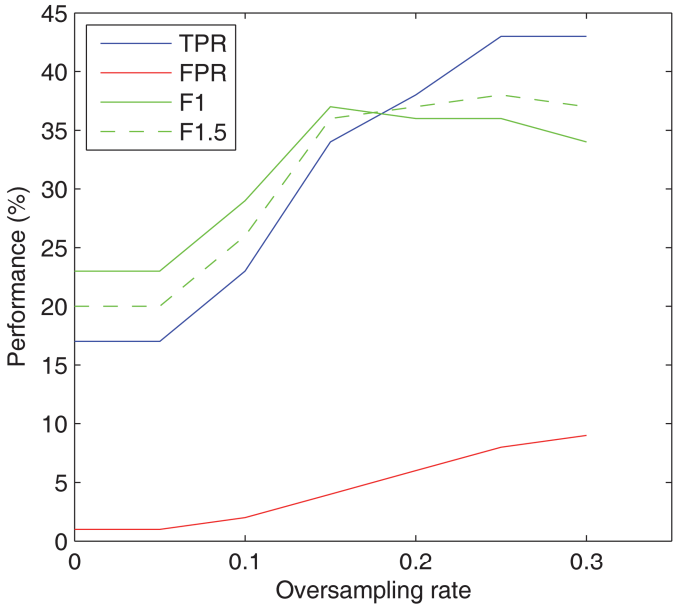


Figure 94 Linear kernel with oversampling trade-off curve for kc3.

better performance than that obtained considering only C=Inf. With the quadratic kernel results being poor, focus turns to the RBF kernel.

9.4.3 RBF Kernel with Oversampling

With RBF, compared to quadratic of the previous section, there is the additional parameter γ. The experimental task is very similar to the above, the only difference being that there is a table (C × oversampling level × samples) for each chosen γ. The performances in the tables produced for each γ value were underperforming, considering the separation evident in the rank sum transformation plot for kc3 (Figure 93), and performance obtained earlier in the modelling chapter on the non-transformed data with NB and SVM. There seemed to be something amiss for the results to be as they were. The first point of inquiry in finding out why this was happening was to produce plots of the decision boundary and test points. The graphical picture may quickly point to what the problem might be, which it did.

Plots for two models from the table are shown in Figure 95. On the left is the training data and decision boundary derived from it, and on the right the same decision boundary but with test points. The performances naturally are derived from the test cases in the plots on the right. What is immediately obvious is that the test points for both models lie below the

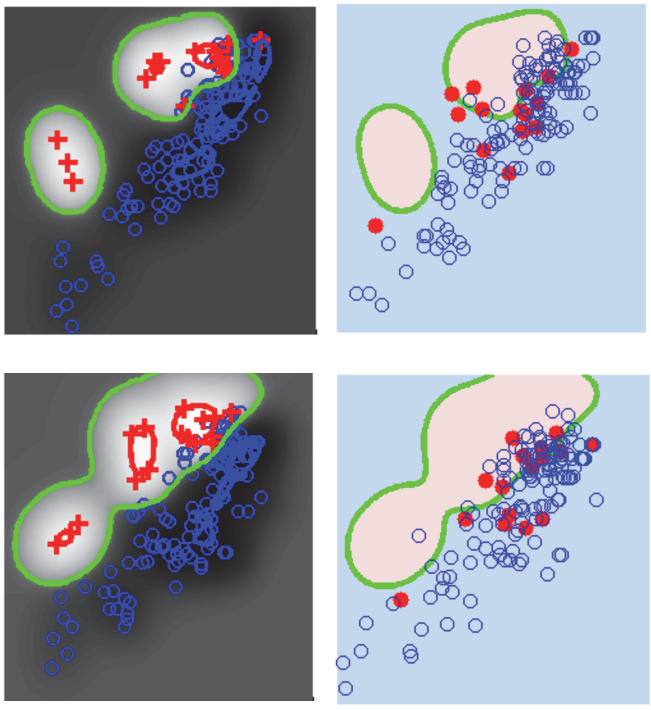


Figure 95 A curious problem in which the decision boundary created on the training data on left did not even roughly coincide with the test data on the right.

position decision regions. This would clearly be diminishing performance as most of these test instances would be incorrectly classified negative. The positives in train and test sets, seeing as they are selected at random, should be randomly interspersed rather than appearing separate. More test positives should therefore lie in the region based on the training positives. It should be noted that this problem seems only to occur with the positives. The negative test points appear roughly in the same location as the negative training points as they should.

One possible cause of the positive test points appearing below the positive training points is that the original train and test samples from the untransformed data were not properly random. To determine whether this was so, for 4 models (based on different samples), the original train and test samples were plotted, but just the positives, and only selecting two features so the points could be plotted. These plots are shown at the top of Figure 96. The result is as expected, with the positive points for train and test set being randomly interspersed. The problem then is not with original sampling, but in the rank sum transformation of points. To confirm this, the rank sum transformed positives of the previous figure are plotted at the bottom of Figure 96. Here the bias affecting test points is quite apparent, with greens tending to lie below the reds. It will be noticed too that the bias or separation is occurring vertically which is the dimension of the positive sum.

The explanation for this problematic behaviour is that due to the small number of positives in kc3, there were not enough positive points in the training set for the positive region to be properly captured in modelling. The training set was capturing some part of the positive

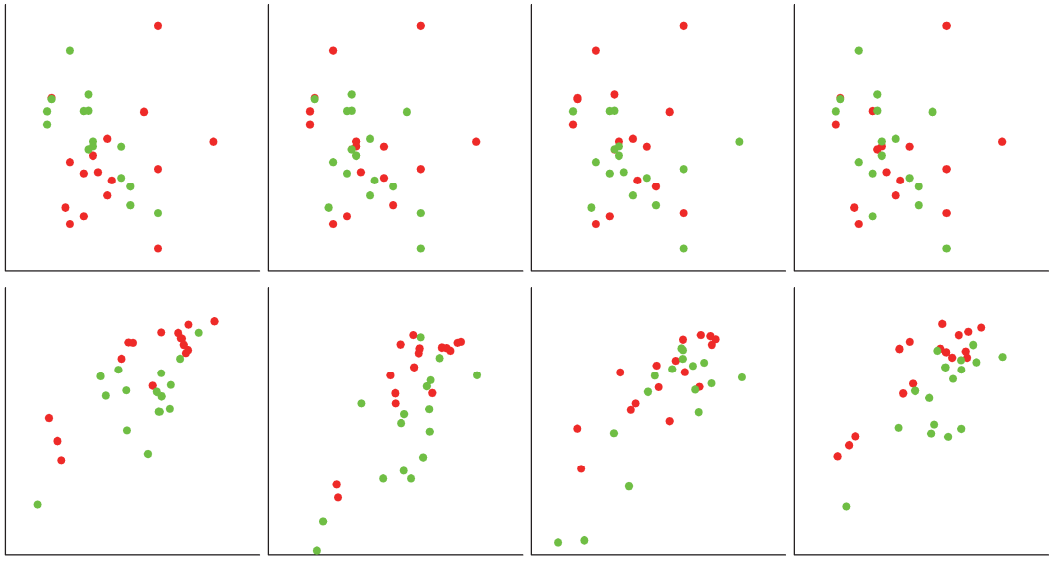


Figure 96 Train and test positives, the above of the original data randomly interspersed, and below the rank transformed data with separation bias.

cloud, and test set another part, to varying degree depending on the samples. When train and test positives cover different regions, naturally the rank sum model is going to give higher pos rank sums to the points it was trained on, in the training region, to test points in another region. The more separate the regions of train and test points the greater the difference or bias. It might be mentioned that this phenomenon of lower lying test points was found not to exist for the larger pc4 data set.

The solution to this problem was to change the train and test split from 50/50 to 90/10, which is the standard proportion for 10 CV, thus providing more positives in the training set with which to capture the positive region. Further, while using this larger proportion for the training set increased performance, there was more variability in results than was the case for pc4. Again this was due to the smaller size of the kc3 data set, and because of which 10*10 CV was used. This increased the number of models generated, increasing the number of sheets in the models table from 10 to 100.

With the above problem resolved, experiments were performed as described at the outset of the section, producing tables of models for each γ value. A total of 6 C x 8 oversample x 100 = 4800 models were generated for each γ value. Of the γ values tried, 0.3, 0.4, 0.5 and 0.6, best performance was obtained with 0.5. These are listed in Table 38. The best F1.5 value is 36. The column for which a trade-off behavior was evident is highlighted, for C=0.5. Values of 1 or 5 could also have been chosen. Based on this selection, a trade-off curve was produced by generating models and performance at finer levels of oversampling from 0 to 0.5 as shown in Figure 97. The best F1.5 value is 37 at oversampling level 0.35.

Use of balanced ridge for the RBF kernel involves a similar experiment setup to the above, except that ridge parameter replaces oversampling level. But as the results were no better

	100	10	5	1	0.5	0.05
0	21 2 23	19 1 22	19 1 23	12 1 14	4 0 5	0 0 0
0.1	21 2 23	22 2 25	21 2 24	17 1 20	10 1 12	0 0 0
0.2	30 4 30	31 4 31	32 4 32	33 4 33	31 4 32	0 0 0
0.3	33 6 30	36 6 34	37 6 34	39 6 36	39 6 36	5 1 6
0.4	37 9 30	41 9 34	40 9 34	40 8 34	42 8 36	27 5 26
0.5	38 12 28	45 12 33	45 12 34	46 11 36	46 11 36	40 8 34
0.6	43 13 31	50 15 34	52 15 36	50 14 35	49 14 35	44 11 35
0.7	43 15 30	51 16 33	51 17 33	53 17 34	52 17 34	47 13 34

Table 38 RBF performances for γ = 0.5, and varying oversampling level (rows) and margin (cols).

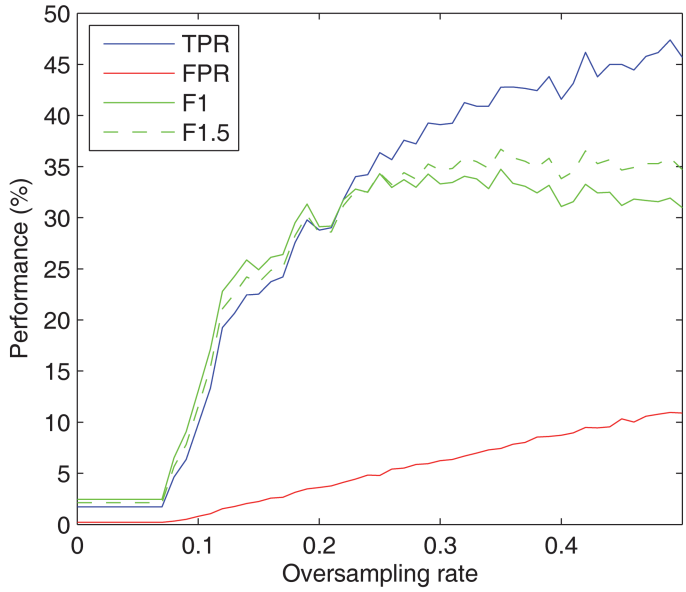


Figure 97 Trade-off curve for kc3 based on varying oversampling rate, with fixed C and γ, both 0.5.

than oversampling, they are not reported here.

9.5 Discussion

In the experiments described it will have been noticed that they lead to the use of methods to deal with imbalance, by oversampling or using the balanced ridge, to obtain better performance. This is not surprising given that imbalance in the data sets is extreme. But it is a surprise considering that relatively good performance was obtained in the modelling chapter, Chapter 7, with SVM, without oversampling or using the balanced ridge parameter. Not being necessary on the raw data, this might suggest that there is room perhaps for obtaining better results on the transformed data without applying methods to deal with imbalance. The work described though would seem to have covered those bases. The explanation for the need for oversampling may be that imbalance is less well handled when the data is reduced in dimensionality.

The main objective of this chapter has been to produce not so much individual models derived from the rank sum transformed data that perform well, although this has been necessary, but to take advantage of the parameters of SVM and see if they can be set according to a parameter based trade-off curve to achieve a particular desired performance trade-off. If this is possible, and the same performance levels as those of standard SVM and Naïve Bayes

models can be achieved, then the approach offers an advantage in providing the flexibility to choose desired trade-off without compromising performance.

There were a number of different trade-off curves produced using different methods. The main criteria for their evaluation is what best performance each was able to give, relative to the F values of standard SVM and NB models. This comparison is shown in Figure 98 for the best trade-off model for each data set. For pc4, the quadratic kernel with ridge parameter and RBF oversampling produced fairly good results with F1.5 values of 49 and 51 respectively (not shown in plot). However, the best performance was obtained with the linear kernel with an F1.5 value of 55. In the figure, NB and SVM results are plotted in the first two bars, and the best trade-off value of 55 in the third. It can be seen that the performance for each approach is similar, which is a good result. There is no loss in performance in using the trade-off model on the rank sum transformed data. For kc3, a less well performing data set in general compared to pc4, the best performance was obtained with the RBF kernel with oversampling which gave an F1.5 value of 37, and linear kernel with oversampling which gave a result 1% higher at 38. This latter result is shown in the figure and it can be seen that it is equal with NB performance, but both are less than SVM. This result is not quite as good as for pc4, but the rank sum based trade-off model is competitive with NB, and both are within 5% of the SVM score.

At the outset of the chapter, a comparison was made between PCA and rank sum

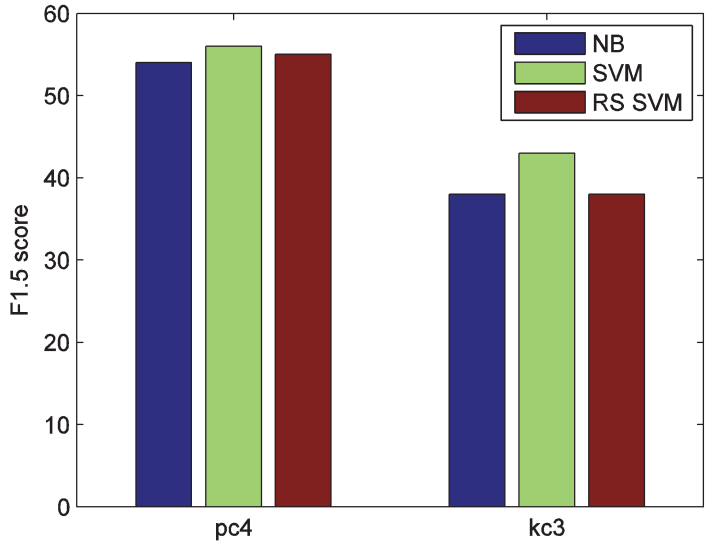


Figure 98 Comparison of best SVM performances obtained on rank sum data with those of SVM and NB on raw data.

transformations for pc4, in Figure 78. Exploring the PCA-based feature sets further, it was found that while the rank sum transformation offers some advantages, the advantages are marginal for some data sets.

It could be mentioned in relation the problem of positive test point bias for the kc3 data set, that it brings to attention the need to take more samples of the data, with more thorough validation, with which to obtain performances, when data sets are smaller. When this problem was encountered, as unusual as it was, that the number of samples was too small was not immediately thought of as a possible cause. Initially it was thought sampling implementation might be at fault in not producing properly random samples.

SECTION 4 - Conclusions

Chapter 10 – Conclusions

The purpose of this research overall has been to explore techniques by which software quality might be improved. More specifically, with quality having been measured in terms of fault counts, the purpose has been to develop quality models for the detection of faulty software modules. These models may be useful in software development practice to identify faulty modules automatically and as early as possible in the development process. Automatic detection of fault prone modules can focus testing and save some of the considerable cost involved in this effort. The approach taken here to this quality modelling or fault detection task has been to develop models using machine learning algorithms which learn to classify based on examples from past data of instances of each class, fault prone or not fault prone. The primary elements involved here are the training data, sourced from NASA MDP and the Eclipse project, and the machine learning algorithms used, Naive Bayes and SVM. A new rank sum based classification approach was also described. The primary interest in developing these models has been to achieve a high rate of predictive performance, sufficiently so to be of practical use. In this regard, performances have been compared with efforts of Menzies who has to date appeared to lead the way in quality modelling efforts on the NASA data. The main contributions of this work, which will be discussed in more detail in the following sections are:

- Exploratory analysis of the 12 data sets.
- Application of an extensive range of feature selection methods to the data sets, to find the best performing approaches, as well as actual features selections for subsequent modelling.
- Systematic generation of classification models on each data set using Naive Bayes and SVM classifiers with various feature selections produced previously, to find optimal performance and with comparison to the performance obtained by Menzies.
- Development of a new classification method, rank sum, which classifies an instance according to the sum of ranks across features for each class, where ranking is an abstraction over the variable width density distribution that measures proximity to the bin of highest density.
- Development of enhanced classification models based on SVM and rank sum data that allows parameterised control of the desired performance trade-off between detection and false alarm rates.

Exploratory analysis was performed on all the data sets, with analysis covering various aspects of the data. At the outset, at the most basic level, with box and whisker plots of values for each feature, a right skew was apparent in most of the features. That is, most of the values were low, often near 0. Considering the distribution of classes separately for each feature, often there was a shadowing of the class curves, with density of positives following that of negatives but at a lower density. This reflects the pattern observed in PCA scatter plots of the data in which points of both classes tended to be mixed. The right skew found in most features was found to include module size and also error density (for instances with fault count ≥ 1). The former case was explored later in filtering training data by module size on the supposition that features with low values provide little range by which to discriminate between the classes. Based on this skewed error density curve, a particular threshold for class labelling was selected at a point roughly in the middle of it, at the value 40. It seemed that error density was largely a function of module size, as number of errors tended to be low, regardless of size. High error densities occurred when modules were small, and low error densities when modules were large. But most modules were in the middle, having moderate size and moderate error density. One of the final aspects of the data explored was correlation between features, and also with error density and class label. Between features, based on distribution of correlation values, it was found for the NASA sets that 50% of correlations had an absolute correlation coefficient above 0.5, which reflects a fairly substantial dependence between features, or similarity in information content. This suggested that it might be possible to remove many features in feature selection without loss of performance. This is desirable in particular for Naive Bayes which is sensitive to feature dependence. A different correlation pattern was found for Eclipse in which the correlation distribution was right skewed with most correlations being low. This did not translate however into better model performance on the Eclipse sets compared to the NASA ones. Correlations with error density were found to be fairly low, but with class labels higher, although the latter did not seem to be a good indicator of which data sets were likely to perform better, and as such this information was of questionable value.

In the data preparation phase of the research, in which the main focus was on feature selection, an initial investigation was made into how feature selection should be done with regard to the potential problem of selection bias which can result in predictive performance being overly optimistic. Perhaps of surprise, there was no mention made about this problem in the papers reviewed in relation to modelling on the NASA data. The selection bias problem

can arise when features are selected on the same data that models are tested on, as features will be chosen to maximise performance on the data from which they were selected. A solution is to perform selection on each training set within the loops of cross validation. This however involves more processing time. To see if bias was in fact affecting results and whether the more costly bias free method should be used instead, tests were performed across data sets using feature selection with bias on the whole data set, and without bias within a cross validation loop. For all but the smallest data set (mw1), performance was found to be the same, and hence it was concluded that feature selection could be done in the usual way on the whole data set. Bias was not adversely affecting results in this case most likely because of the relatively large size of the data sets and the relative few features selected compared to the number of instances.

Having decided on a feature selection method, without cross validation being necessary, and with instances having been labelled using a selected error density threshold of 40 rather than the usual 0 used in other research, notably Menzies', the main effort in data preparation was in finding suitable feature selections for later modelling. The purpose of feature selection is to remove redundant and irrelevant features from the training data. Leaving these in the training data can result in loss of performance, as they can interfere with the learning algorithm being able to find the hypothesis that best discriminates between the classes. More features increased training time, which can be a factor for larger data sets such as jm1. Thus features may be selected to improve performance or reduce the size of the training set without performance loss. In this research the objective of feature selection has been the former, with optimal performance being the main consideration. A total of seven different selection methods were applied to each NASA data set and five methods to each Eclipse set. A different approach was warranted for the latter due to the much larger number of attributes (200 Eclipse versus 43 NASA). All feature sets were applied with each learning algorithm as described subsequently when efforts turned to modelling on the prepared data sets.

It is of interest to note that while an extensive range of feature selection methods were tried, Menzies reported the use of only one that he found to give optimal performance, a so called iterative subsetting method. This involves obtaining the information gain (IG) score for each feature, ranking features by this score, obtaining performance on top down subsets using a chosen classifier, in his case Naive Bayes, and choosing the subset whose performance was highest – actually he defaulted to a size of 3, a slight variation of the approach. If Menzies' method was sound and able to give optimal performance then use of other methods would not

be necessary. But there appears to be a lack of theoretical underpinning to his approach that provides any support for an optimal feature selection and performance – although this is not uncommon in feature selection. Moreover, upon analysis, it is apparent that there could be a flaw in the approach. It ignores redundancy and complimentarity of features. The features that score highest may well be relevant, but they may also be redundant. This redundancy would not be excluded by taking top down subsets of the ranked features. The top down subset approach would also quite obviously miss particular compliments of features that may enhance performance. What Menzies found, perhaps, was a method by which a set of good features could be found and which in combination gave good performance but not necessarily optimal performance. Using other selection methods in this research, and a larger number of features apart from the default of 3, provided an opportunity to find whether better performing feature subsets and associated selection methods existed, the outcome of which is discussed shortly.

There were a couple of less substantial explorations in data preparation in relation to module size and minimum training set size. In relation to module size, Koru & Liu (2005) reported that models trained on larger modules performed better. The reasoning behind this was that there were too many instances with values close to zero providing no range by which to discriminate, and relatively too few instances with larger values which would thus tend to be misled by the learning algorithm. This seemed a worthwhile avenue to explore but Koru was reporting a phenomenon that occurred when the model trained on larger modules was tested on the same large modules. In experiments performed in this research, when tested over all instances in the data set, the performance gain was not significant. In relation to minimum training set size with which optimal performance can be obtained, Menzies' finding of that number being around 100 instances was confirmed in this research (see Figure 55). Performance plateaus at approximately this number. However with class labels assigned according to a larger error density value of 40, this number was a little higher. While this could be justification for reducing training set size in later modelling experiments, all instances were used to see if it might be possible to exploit information contained beyond the 100 or so found otherwise to be sufficient, and because there is no loss in predictive performance to be had in using more instances.

The modelling phase of the research applied two machine learning algorithms Naive Bayes and SVM to each of the data sets and with various feature selections. There were two main interests here, the first being comparison of performance with Menzies to see if his

performance could be bettered, and the second being which feature selection method worked best. There was also an interest in seeing which of the two learning algorithms performed better. To compare models a little investigative work was done initially on performance measures in which a single value is used to represent the performance of a model. This avoids having to compare multiple lower level performance values, such as detection and false alarm rates. The F measure is one commonly used, but it was found that this measure, due to class imbalance, heavily penalises performance upon even slight false alarm rates. The reason for this is that in the imbalanced situation, small false alarm rates cause high levels of imprecision, to which the F measure is sensitive. Thus an F measure with a β value weighting on recall of 1.5 was found to be more suitable in the context of this work, where more tolerance of false alarm rate was permissible. Increases in false alarm rate were allowed in the sense that performance was not heavily penalised, only if there were considerable gains in the detection rate. This measure might be used in future work for the justification given. It is also possible, as in this work, to consider use of the F measure with other β values to provide other perspectives on performance, in relation to the trade-off between recall and precision.

Jumping to the results of the modelling work, in terms of best performance in comparison with Menzies, performance was less. This is due to the error density threshold on which class labels are determined being 40 rather than 0. This seems to make the classification task more difficult, despite the fact that in using error density it is possible to have control over the severity of fault proneness in the modules detected as fault prone. This use of a different threshold makes comparison with Menzies' work generally more difficult, but where possible comparisons will be made. Apart from performance on the whole being less than Menzies, there is the issue of which learning algorithm performed best, over both the NASA and Eclipse sets. The better classifier was found to be Naive Bayes. Of the 12 data sets, it performed significantly better on 4 of them, while SVM on only 1 of them, while the rest performed similarly. This supports Menzies' finding that Naive Bayes performs as well as, if not better, than any other algorithm. These results were obtained using the standard algorithms and their parameters, along with feature selection. There was no special treatment applied to the data, involved sampling for example to deal with class imbalance.

In discussing model performance, as noted by Menzies, there is a trade-off in play in the performance results between the detection rate and false alarm rate, as is common in classification performance generally. Any increase in detection rate is accompanied by an increase in false alarm rate. Graphically this may be explained in terms of a cloud of points

where the classes of points are interspersed. In order to obtain a higher detection rate, the positive region must be enlarged, but in doing so, negatives amongst the positives are mistakenly classified positive, thus increasing the false alarm rate.

Having developed models on the feature selections found earlier for each of the data sets, the modelling experiments section provided the opportunity to find which gave the best performance. Overall, the best feature selection method, which gave the best performance (F1.5 value) was found to be wrapper subset evaluation with backward best first search (BFb.W). This would agree with advice given in the literature that if optimal performance is sought, over maximal reduction in features for example, then wrapper subset selection is the best option. In relation to Menzies' work, while wrapper gave the best results overall, they were usually only marginally better than the equivalent of Menzies' iterative subset method, referred to in this work as IG.W. As to which method should be used, there is no strong reason to choose one over the other. Wrapper would likely be the preferred choice however, for the marginal improvement in performance it provides. The consequence of this result is that Menzies' simple method works as well as more elaborate ones, and as well, despite the shortcomings of Menzies' method in overlooking redundancy and complimentarity, there is actually little to be gained performance wise in methods that take these factors into account. Menzies' method for feature selection for these data sets, happens to be able to capture most of the useful information that exists in the data sets.

A result worth mentioning in relation to model performance was that when error density threshold for determining class labels was set to 0, which was done to confirm why performance was less than that obtained by Menzies, the performance obtained on a couple of the data sets was significantly better than Menzies'. This result was obtained using only selections produced by IG.W. This shows that it is possible to improve on Menzies' results. If other feature selection methods had been tried apart from IG.W, for example wrapper methods, there might be further performance gains to be had on other data sets. This was not explored however, as the focus in this research has been on the use of an error density threshold of 40 only. This threshold was chosen in order that resulting quality models would only identify modules at a higher level of fault proneness.

Another variable in the modelling experiments apart from feature selection was log normalisation. Both feature selection and modelling were divided into strands, unnormalised and log normalised. Log normalisation had been included as it had been reported by Menzies that this improved performance. As mentioned, it is difficult to make direct comparisons with

Menzies work, but it can be said that in the experiments in this work, models based on unnormalised strands performed better, contrary to Menzies' finding. But it was found that using a different performance evaluation measure, F1 instead of F1.5, that models based on log normalised strands performed better. That is, log normalisation would appear to give better performance only if lower detection rates are desirable, as are favoured by the F1 measure. This only applies however in the current setting with the threshold of 40. In the case of 0 threshold data, as might normally be used, it may be that log normalisation provides better performance regardless of which F measure is used.

An assumption made at the outset of this research was that static code metrics captured sufficient information about the quality of software that they could be used to effectively discriminate between modules on the basis of quality. It was argued in the introductory chapter that this assumption was fair to make as the approach has been used in other research. A number of studies were then cited in the previous studies chapter, Chapter 2, which demonstrated that it was possible to obtain good predictive results, further supporting the assumption. Having come to the conclusion of this research it may be asked whether the results obtained here support this assumption. Is quality modelling from static code metrics useful?

An appropriate baseline for assessing this question may be obtained if one compares model performance obtained in this work, developed from static code metrics, with that of a model that has no predictive ability. The latter is one which predicts randomly as either fault prone or not. If static code metrics are effective in discriminating, and hence useful, then these models should perform markedly better than a randomly predicting model. Performance of a randomly predicting model is represented at the top of Figure 99. Fairly obviously the detection rate would be 50%, as well as the false alarm rate, as reflected in the red positive prediction window which lies over half the positives and half the negatives. Based on a positive class proportion of 5%, which is the mean for NASA data sets, the F1.5 value is calculated to be 13. Not surprisingly this is quite low relative to many of the performance values reported in this work. The question now then is, does the mean model performance in this research improve on that of the random model? Mean model performance is represented in the second diagram in Figure 99. The mean TPR and FPR is 36% and 9% respectively based on best NB and SVM performance as listed in Table 24 in Chapter 7. The positive prediction window is again placed in accordance with these rates. The point to be made in answer to the posed question is that there is a dramatic difference between random model

performance and mean NASA model performance. The positive prediction window covers much less of the negatives, meaning a dramatic reduction in false positives. Looking at the performance numbers themselves, false alarm rate has dropped, but more tellingly perhaps, the F1.5 score has increased from 13 for the random model to 36 for the NASA models – it has nearly tripled. Clearly, NASA model performance has improved dramatically over that of the random model. The models in this research at least to this degree therefore support the assumption that static code metrics are useful in predicting software quality. As a matter of interest, the mean performance of Menzies' baseline models (from the same table as before, Table 24) is represented in the bottom diagram of Figure 99. The positive class size is a bit larger due to the different threshold used for class labeling. It also shows improvement over the random model, but the higher detection rate (positive prediction window overlapping more of the positive region), almost double that obtained in this work, is penalized with a substantially larger false alarm rate (positive window extending in the opposite direction over the negatives). The rather pronounced difference in performance behavior between the models in this work, and those of Menzies, is due to difference in labeling threshold. Menzies detects a wider range of faulty modules, some of which might have more similarity with negatives, which could explain the prediction of more negatives as positive.

The above though only shows that static code metric models improve substantially on random, not necessarily that the improvement is to the degree that they are practically useful (although visually it looks as though they would be). This is addressed by Menzies in two ways (Menzies, Greenwald, & Frank, 2007). First, the mean TPR and FPR of his models, 71% and 25% (Table 24), is close to that of standard binary predictors from the University of California Irvine machine learning repository, of 80% and 20%. Second, his detection rate of 71% is much better than that of existing industrial inspection methods, such as 60% TPR reported in the 2002 IEEE Metrics panel. Though the models obtained in this research have different performance characteristics to those of Menzies, they are comparable by the F measure, and so it could be argued their performance is sufficient also be of practical use. A more objective assessment of model usefulness was discussed in 2.4 Performance Evaluation. Arisholm et al. (Arisholm, Briand, & Fuglerud, 2007) introduced a measure of 'cost effectiveness', which is the benefit a model provides in identifying as many faulty modules as possible for a given limit on code to inspect, over a baseline of selection of modules by size (as a coarse means of identifying faulty modules). Only recently proposed, this measure could be applied in future work, as a better means of assessing their usefulness.

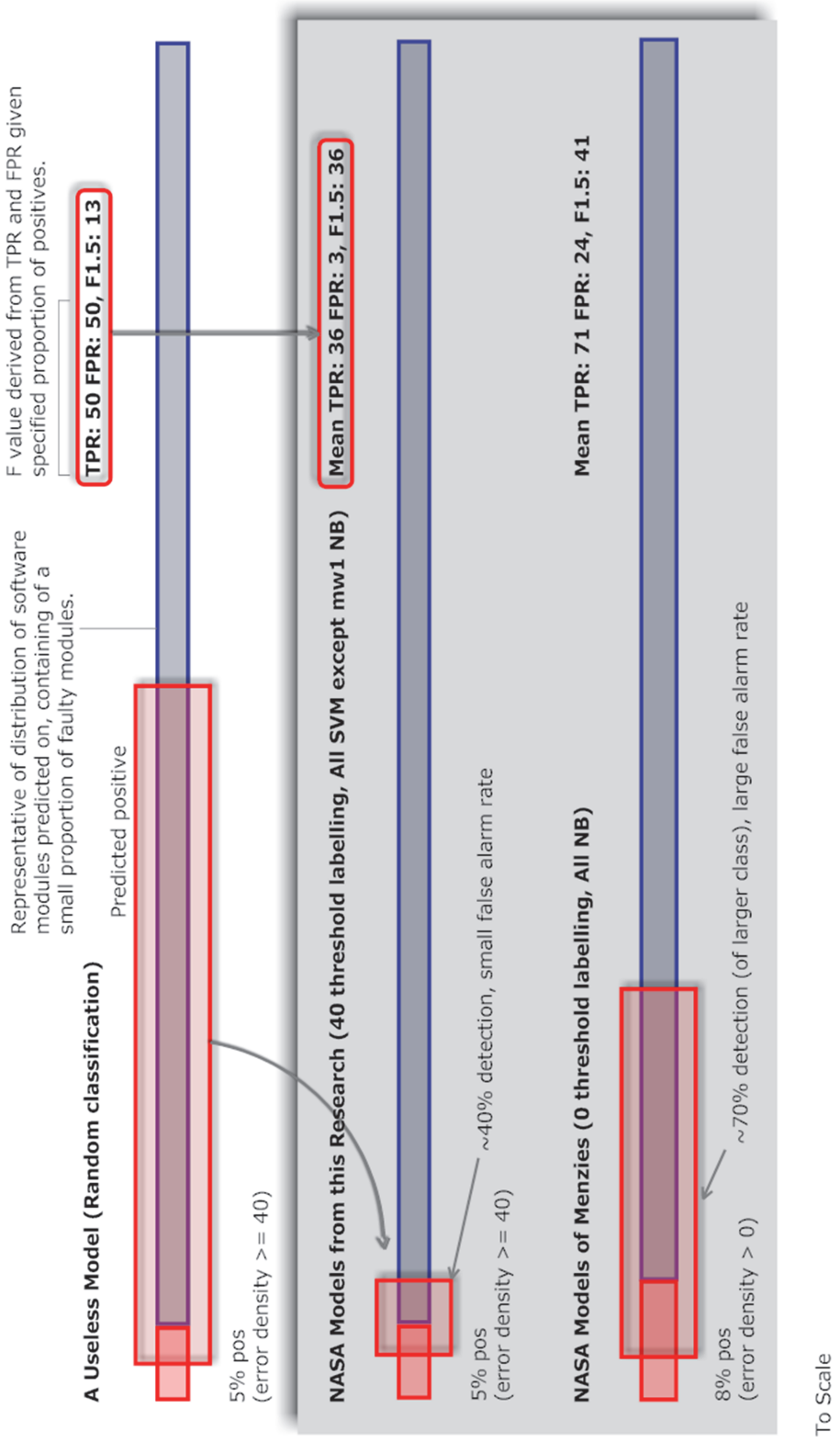


Figure 99 Improvement in performance of the NASA models in this research and of Menzies over a model that predicts randomly, as evidence that static code metrics can be used to develop effective software quality models.

Another aspect of the research involved the development of a new classification method, in which a classification is made by summing ranks for each class across features, where rank

measures proximity to the bin of the highest rank. The simple intuitive idea underlying this is that the more consistently a point lies in the highest density areas of a class, the more likely it is to be of that class. To improve performance of the method, rankings are based on variable width bin densities rather than fixed width bins. Experiments focused on obtaining the best performance possible with this method, and for which more exhaustive searches were resorted to. For example, the best set of features is obtained by finding parameters that are optimal for each feature individually. Using these more exhaustive methods gave good results. They were as good as Naive Bayes for the two data sets included in the experiments. This attests to the method having some discriminative power. For practical use, a more efficient means of determining parameters was required. In this regard, it was found that with default settings found for each data set, applied to each feature, near optimal performance can be obtained. The default set of parameters needs to be found still for the data set, but in applying the same parameters to all features, this greatly reduces search space. In one instance, rank sum performance exceeded Naive Bayes performance using a default parameter for number of bins, and individually optimal density levels for each feature.

Finally, the rank sum method was used to produce a two dimensional representation of the data to serve as a basis for SVM models with parameters that could be adjusted to choose the trade-off desired between detection and false alarm rates. Having the ability to choose the desired trade-off is an advantage, as in some cases it may be more important to find all defective modules allowing a higher false alarm rate. In other cases there might be more emphasis on avoiding wasted inspection effort in which case a lower false alarm rate would be desirable. The two dimensional representation of the data is convenient for modelling, allowing visualisation of SVM decision boundaries. Important however is the ability of these trade-off models to perform as well as standard SVM models. With this in mind, various kernels were applied to the rank sum data. The simplest linear kernels performed the best of a range of kernels tried. Oversampling was necessary to counter the adverse effect of imbalance on SVM. The results with the linear kernel and oversampling were virtually as good as SVM and Naive Bayes for pc4, which is a pleasing result. It means that the model is of practical use, and provides function in terms of performance trade-off beyond what the ordinary model provides. For the other data sets tried, performance was on par with Naïve Bayes, but marginally below SVM. This result is not quite as good, but it is possible that performance could be improved by better selection of rank sum parameters which would provide data in the first place with better class separation on which the SVM model is based.

In regard to future work that may be done to improve model performance there are a few avenues that may be worth exploring. The first is methods that deal with class imbalance as the data is highly imbalanced, as machine learning algorithms tend to suffer performance wise on such data (Guo, Yin, Dong, Yang, & Zhou, 2008). There are many means of dealing with balance, which exist at both the algorithm and data level. Some of these methods were touched on in the course of the earlier chapters. Using such methods, individual classifier performance may be improved. Little attention has been given to this approach in other research. Consideration might also be given to class imbalance in feature selection. Another avenue that might be explored is in improving differentiation between the classes where most of the points lie which is at the peak of the log normal distributions of features. Other transformations to the data apart from log normalisation that might also improve performance could be explored as recommended by Menzies (Menzies, Greenwald, & Frank, 2007). However, a recent study applied a few transformations to the NASA data and concluded that they did not improve performance (Jiang, Cukic, & Menzies, 2008). There is also the option in the case of the Eclipse sets, which associates instances with source modules in the public repository of the Eclipse project, of collecting and possibly defining new metrics to add to the data set with which to model. There are also ensemble methods such as boosting and bagging for combining classifiers that may improve performance by taking into account the predictions of a number of classifiers rather than an individual one.

While these avenues exist and they are potential means by which to improve performance, there has been a notable opinion expressed by Menzies, one of the leading researchers in modelling with the NASA data, that a "performance ceiling" has been reached. While some marginal improvements in performance may be made, he believes largely that the discriminative content of static code metrics has been fully exploited, by the many learning algorithms that have been applied to the NASA data in various studies, and that the only way that more substantial improvements in performance may be achieved is with the inclusion of other types of information in the data set, such those more related to business. In light of this opinion, this study might be seen as one which provides support for it, and if so, other types of data would be the way forward as Menzies has suggested if models are to be developed that provide software practitioners greater accuracy in fault detection.

Appendix A – Software Metrics

Commonly used software product metrics are described in this section. Some subset of them appears in each of the data sets used in this research. Descriptions are provided as they may be useful, in that rather than treating each metric as a generic variable, it can be known precisely what each metric measures and how it is calculated.

There are four groups of metrics described, namely:

- The lines of code metrics include metrics that count lines in different ways depending on line content. Some metrics include comments for example, others not.
- The Halstead metrics are a suite of metrics based on a theory of software called Halstead's Software Science. It was developed in the 1970's but the metrics are still commonly used today. The metrics primarily measure program size and complexity. Metrics in this suite include Volume, Error Estimate and Program Level.
- The McCabe metrics, the best known of which is cyclomatic complexity, measure structural complexity of a module in various ways. The measure is derived from a representation of a module as a control flowgraph.
- The Chidamber and Kemerer metrics are a more recent suite of 6 metrics that were developed in the 90's in response to the increasing popularity of object oriented programming. They measure special features of this programming style, that are not captured by traditional procedural metrics.

A.1 Lines of Code Metrics

The Lines of Code metrics quantify the size of a software module by counting physical lines of code. There are a number of LOC metrics that count LOC differently depending on what the lines contain. A line may contain only code, only comments, both code and comments, or it may be blank. Corresponding LOC measures are listed in Table 39.

While comments might not seem to be that relevant to quality, it is quite possible that there might a correlation between amount of comments and the number of faults. More comments would suggest a more rigorous approach to coding, and a better understanding of what the code does. They would also help to reduce the introduction of errors in later modifications to the code. Based on this reasoning, it could be expected that higher comment levels will be associated with a lower number of faults.

Metric name	Lines included
LOC_EXECUTABLE	Code
LOC_COMMENTS	Comment
LOC_CODE_AND_COMMENT	Code and comment
LOC_BLANK	Blank
LOC_TOTAL	All lines (code, comment, code & comment, blank)
PERCENT_COMMENTS	Comment

Table 39 Lines of Code (LOC) metrics, as used in the NASA MDP data sets.

While LOC is a commonly used metric and it is easy to visualize what it represents, it is often criticized as a size measure for being dependent on language and coding style. The influence of style on LOC is more an issue for high level languages which allow for greater flexibility in the way code is written than older languages such as assembly and FORTRAN. A FOR loop for example could be written using 4 lines of code:

```
For(…)
{
      Statement;
}
```

or it could be written using only a single line of code:

```
For(…) { Statement };
```

If different programmers on the same software project use different coding styles, the LOC measures are not really comparable and a form of noise is introduced. The noise is not only present in the training data, it can also be present in new instances that a model predicts on. It has been suggested therefore that other size measures that are less language dependent be used instead, such as Halstead Length and logical LOC.

In relation to quality, a large value for LOC may be an indication of poor module design. It may also indicate low functional cohesion in that the larger the size the more likely it is that the tasks performed within the module are unrelated. A larger value would increase complexity. Thus one would tend to associate larger module size with a higher number of faults and lower quality.

A.2 Halstead Metrics

The Halstead metrics are a suite of metrics based on a common theory of software known as Halstead's Software Science (Halstead, 1977) which measure various aspects of software included size, effort and programming time. The theory, published in 1977, was considered at the time to be of profound consequence. Software was considered to be governed by a unifying set of physics like laws. Although these metrics have been shown to be useful in numerous studies, criticisms of them have been made. The theory however is still considered to be a significant contribution to the field of software metrics.

In the theory, a software module is seen as a stream of tokens composed of operands and operators. Measures are either basic, as raw counts of these elements, or derived, in which the basic measures are combined in various ways to measure more complex properties of the software. The basic measures are listed in Table 40.

Unique counts are denoted with lower case n, and totals by upper case N. The union of unique operators and operands is referred to as the vocabulary. The union of all operators and operands is referred to as the length. From these basic measures the Halstead metrics can be summarised as in Table 41. Each of these measures is described in detail in the following sections.

A.2.1 Program Length

Program length $\hat{N}$ is an estimate of module length in terms of the total number of operands and operators, N. The estimate is obtained using only the vocabulary, the unique number of operands and operators. As such this measure captures the relationship between vocabulary and length. Halstead noted that that such a relationship should exist is unexpected, but in his own research and in study after study subsequently, program length measures were found to

Basic Measure		=
Unique	Operator count	n_1
	Operand count	n_2
	Vocabulary	$n_1 + n_2$
Total	Operator count	N_1
	Operand count	N_2
	Length	$N_1 + N_2$

Table 40 Basic measures of Halstead's.

Metric	Abbr.	Definition
Program Length	N	The estimated length of a module based on vocabulary. It is an estimate of N from n.
Program Volume	V	The size of a module in bits calculated as the number of bits required to represent an item from the vocabulary multiplied by the number of occurrences of items in the module.
Potential Volume	$\hat{V}$	The volume of the smallest possible implementation of an algorithm in which a call is made to an already existing implementation of the algorithm.
Program Level	L	The level of abstraction of a module, as a function of unique operator count (fewer increase the level) and unique operand count relative to total operand count (less repetition increases the level).
Intelligence Content	I	The information content of a module, a language independent measure obtained by adjusting volume (which is language dependent) through multiplication by level of abstraction
Difficulty	D	The inverse of level of abstraction. Low abstraction using low level concepts makes for greater difficulty.
Effort	E	The amount of mental effort required to implement a module, in terms of the number of elementary mental discriminations, which is equivalent to the Volume of the module multiplied by the Difficulty.
Programming Time	T	The estimated length of time in seconds required for implementation based on effort (mental discriminations / number of mental discriminations per second).

Table 41 Halstead metrics.

be surprisingly close to actual program length. The formula for Program Length is:

$$\hat{N} = n_1 \log_2 n_1 + n_2 \log_2 n_2$$

The total number of operands and operators in an implementation, $N_1 + N_2$, is estimated by summing the operator count multiplied by its binary logarithm and the operand count multiplied its binary logarithm. In other words, each unique count n_i is mapped to a total count N_i, by multiplying n_i by $\log_2 n_i$.

To derive the formula, an implementation is considered to be composed of substrings each of which contains *n* items from the vocabulary. If the vocabulary is {A, B, a, b } for example, where A is an operator, and a an operand, the total number of different substrings possible then is n^n assuming no two are identical eg { AAAA, AAAa, AAAB,…}. This can be reduced taking into account that operators and operands alternate to: ${n_1}^{n1} \times {n_2}^{n2}$. This is an upper limit to *N*, as *N* cannot be larger than the total number of possible unique substrings formed from the vocabulary. The number is also an upper limit to all possible subsets of *N*, ie 2^N. Thus:

$$2^N = {n_1}^{n1} \times {n_2}^{n2}$$

And solving for N,

$$N = \log_2({n_1}^{n1} \times {n_2}^{n2})$$

The right hand side expands to give the formula for Program Length.

A.2.2 Program Volume

Program Volume is another measure of size of an implementation but in terms of number of bits required to represent it as a function of both *n* and *N*, vocabulary and totals. This is in contrast with Program Length which measures size in terms of the total number of operands and operators, *N*. The minimum number of bits required to represent any item from a vocabulary of *n* items is $log_2 n$. Eg if the vocabulary size is 8, 3 bits are required for each item. The total number of bits then for the implementation is given by *N* multiplied by the total number of bits required to represent each:

$$V = N log_2 n$$

Program Volume is an alternative to LOC as a size measure. It may be a better one because it is a logical measure of size which is independent of implementation text. It is thus less affected by developer coding style or the length of names of operators and operands, as is LOC. However, volume may change from one language to another. A Java implementation for instance will have fewer operators and operands than the assembly language translation of it.

A.2.3 Potential/Minimal Volume

Potential volume, also called minimal volume, is a measure of the smallest possible implementation size of an algorithm. The smallest size occurs when the algorithm is implemented simply with a call to an existing, possibly inbuilt, function in which the

algorithm has already been implemented. Minimum volume is calculated using the Program Length formula mentioned, which expanded is:

$$V = (N_1 + N_2)log_2(n_1 + n_2)$$

However, with the implementation being of minimal form, variables are substituted as follows:

- Total operators $N_1 = 2$ as there are only 2 operators, function call and assignment statement.
- Total operands $N_2 = n_2$ as operands need only to appear once, being passed as parameters rather than occurring multiply as would ordinarily be the case if an external implementation were not being relied upon.
- Unique operators $n_1 = N_1 = 2$.
- Unique operands n_2 does not change.

This gives minimal volume as:

$$\hat{V} = (2 + n_2)log_2(2 + n_2)$$

It is the same as Volume $V = Nlog_2n$, but with substitutions for N and n for the minimal implementation case. This metric is only used in the calculation of other metrics.

A.2.4 Program Level

Program level is a measure of the level of an implementation, where level is similar to what is meant when talking of 3rd and 4th generation languages. The lowest level language is one in which there are no inbuilt functions and everything has to be coded, resulting in implementation of high volume. The highest level language is one in which all functions are inbuilt, minimal coding is required, and volume is low. Languages tend to fall between these two extremes, with varying degrees of inbuilt function support. Program length quantifies this level as applied to module implementation. It is considered to be an important measure because level affects implementation in terms of effort, likelihood of error, and understandability. A higher level implementation which is shorter and expressed in higher level concepts will take less effort, be less error prone, and more understandable, assuming there is fluency in the language.

Program level may be calculated in a couple of ways. The first assumes that a potential volume measure is available and is calculated as:

$$L = \frac{\hat{V}}{V}$$

Level is essentially a function of V, as the minimal volume of an implementation is fairly constant (only affected by number of unique operands). When V is minimal, the program level is maximal at 1. As V increases above the minimal volume, level decreases from 1 towards 0. Thus level is inversely proportional to volume.

The alternative method for calculating level is used when a measure for $\hat{V}$ is not available. It provides an estimate of L directly from the implementation, although it may be thought of as an alternative definition to program level. The formula is based on two observations, about operators and operands respectively and their effect on program level. The larger the number of unique operators, relative to the minimum, the lower the level. This can be represented as:

$$L \approx \frac{\hat{n}_1}{n_1}$$

In addition, the larger the number of unique operands the lower the level. However as it is unknown what the minimum number of unique operands is (as this depends on the algorithm), the total number of operands is used instead:

$$L \approx \frac{n_2}{N_2}$$

Combining these gives an estimate for L:

$$\hat{L} = \frac{2}{n_1} \times \frac{n_2}{N_2}$$

A higher level is associated with a lower unique operator count, and lower repetition of operands.

A.2.5 Intelligence Content

Intelligence content is a language independent measure of inherent information content in an implementation (information content may be a more appropriate name, but it is commonly associated with Shannon's Information Theory, hence the use of Intelligence Content instead). The previous measures described, volume and level, are both language dependent. However, if these measures are multiplied together, the result is a value that is the same regardless of implementation language. The formula is expressed as:

$$I = \hat{L} \times V$$

$$= \frac{2}{n_1}\frac{n_2}{N_2} \times (N_1 + N_2)\log_2(n_1 + n_2)$$

The measure is approximately equal to and highly correlated with $\hat{V}$, and since $\hat{V}$ is language independent, intelligence content is then to. These two measures are closely coupled and may be used interchangeably.

A.2.6 Difficulty

Difficulty is a measure of difficulty in implementation based on program level. Assuming the implementer is familiar with the language, a high level program is implemented using high level concepts and is thus easier to implement and understand. In addition, from the formula for level, a high level program has less volume, which makes it less complex. Difficulty is therefore inversely proportional to program level and is given by:

$$D = \frac{1}{L}$$

A.2.7 Programming Effort

Programming effort is a measure of the amount of mental activity required to generate a given program by an implementer fluent in the language. It is based on the idea that implementation involves selection of N elements from the vocabulary n, and assuming an efficient binary search is carried out mentally for each, the number of comparisons required to select each element is $\log_2 n$, thus requiring $N \log_2 n$ comparisons in total, which happens to be the same formula as volume V. For each comparison, Halstead difficulty gives the number of elementary discriminations required for each comparison. Thus the total number of discriminations or effort E is:

$$E = V \times D = V \times \frac{1}{L} = \frac{V^2}{\hat{V}} \quad as\ L = \frac{\hat{V}}{V}$$

In the initial form, $E=VD$, effort may be thought of as volume adjusted according to the level of abstraction. The last equation states that effort is essentially the square of the volume. It could be said therefore that from one language to another, effort varies with the square of the volume, given the language independent potential volume.

This measure has been used in a number of metric studies and was found to be a good predictor of amount of time required for implementation, and number of errors introduced during implementation. However, it has been criticised for the assumption made that an

implementer selects an element from the vocabulary using a binary search in the same way as a computer.

A.2.8 Programming Time

Programming time is a measure of the estimated length of time in seconds required for implementation and is proportional to effort. It is calculated by dividing the total number of mental discriminations required for implementation as defined by effort *E*, by the number of number of mental discriminations carried out per second by the implementor, *S*, and is given by:

$$\hat{T} = \frac{E}{S} = \frac{V^2}{S\hat{V}} = \frac{n_1 N_2 N \log_2 n}{2Sn_2}$$

It is thought that ordinarily the number of elementary mental discriminations that the brain performs per second, S, is between 5 and 20. (For the NASA data sets the value used for S is 18.)

A.2.9 Error Estimate

Error estimate gives an estimate of the number of bugs in the implementation. Given that volume is equal to the total number of mental discriminations required for implementation, and assuming that an error is made every *X* discriminations, error estimate is expressed as:

$$B = \frac{V}{X}$$

A.3 McCabe Metrics

One of the most fundamental properties of software is its complexity. Complexity here relates to the difficulty in implementing or understanding a program rather than computational complexity. Originally complexity was measured in terms of lines of code, larger modules being seen as more complex. However, the well-recognized shortcomings of that metric lead to the development of improved complexity measures less dependent on language and coding style. One of the commonly used of these is McCabe's Cyclomatic measure introduced in the 1970s (McCabe, 1976). It is often referred to as just complexity or program complexity. Extensions were later made to cyclomatic complexity to create new metrics for specialized applications, two of which are Essentially Complexity and Design Complexity. All of these complexity metrics are based on a control flowgraph representation of the software. This representation is described first, and then each of the three complexity metrics.

McCabe's complexity measures are based upon the observation that a software module can be represented by a control flowgraph. This graph shows the different paths in which execution may take through a module. It essentially reflects the decision structure of the logic within the module. Each type of decision structure has an associated graph form as shown in Figure 100. These are instantiated and combined according to the sequence of decision structures in the module. The graph is composed of nodes, which represent either decisions, or non decision related expressions, and edges which represent transfer of control.

The calculation of the complexity metrics from the control flow graph is based on graph theory which assumes that the graph is strongly connected i.e. a node is reachable from any other node via the edges. In order to make a software control flow graph strongly connected, a start and end node is added (referred to as null nodes), as well as an edge from the end node back to the start node. The returning edge also serves to represent return back to the point from which the module was called. Often the returning edge is assumed and not explicitly shown. An example of module source and corresponding flowgraph is shown in Figure 101.

A.3.1 Cyclomatic Complexity

The cyclomatic complexity measure is a count of the number of different paths through the control flow graph of a module, or more generally it is a measure of the amount of decision logic. This measure in graph theory is called the cyclomatic number because it is a count of the number of basic cycles through the graph, and is where the metric gets its name from. More precisely it is the minimum number of paths that can in linear combination generate all possible paths through the module. As a function of the graph it is often referred to as V(g), or just v. It is assumed in this measure that there is a relationship between the number of control paths within a module and its complexity.

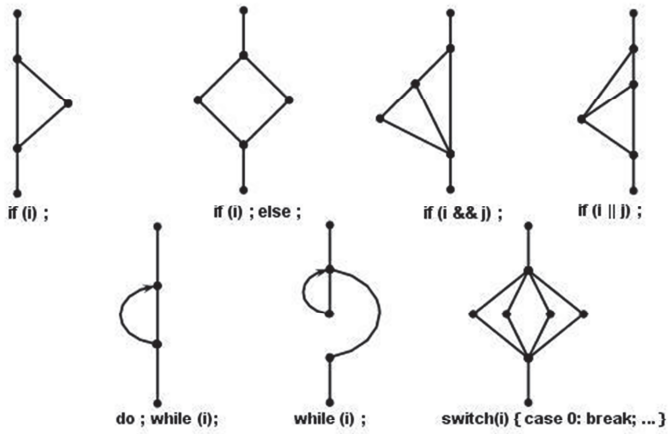


Figure 100 Flowgraph notation for different logic constructs.

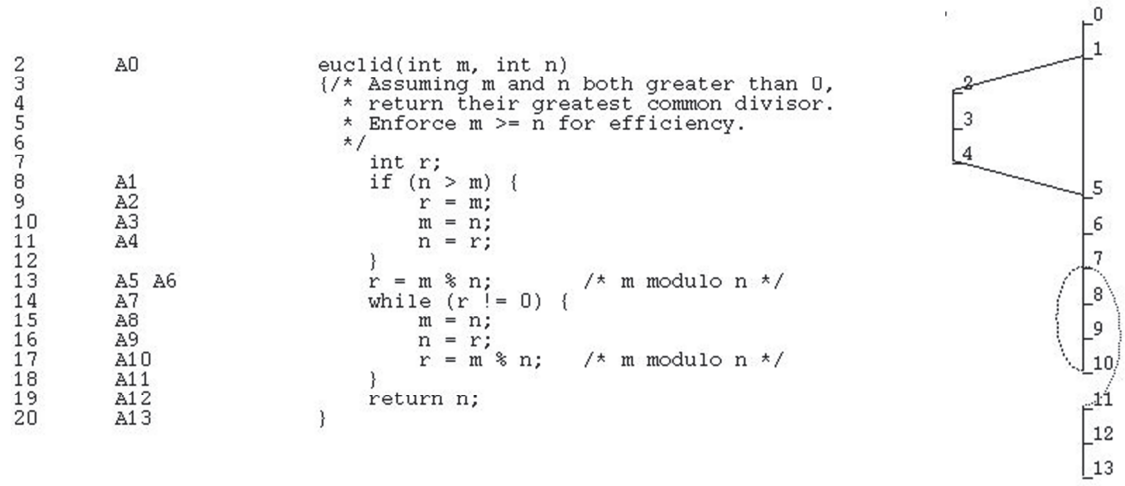


Figure 101 A control flow graph derived from source code. Nodes are indicated in source as Ai.

The basic formula used to calculate the cyclomatic number is taken from graph theory and is the number of edges e minus the number of nodes n + 1. In the case of a control flowgraph in which the loopback edge is not explicitly included, the number is calculated in one of the following ways:

- Nodes and edges: $v = n - e + 2$ (1 is added for the loopback edge)
- Decision nodes: $v = n_d + 1$
- Regions: $v = r$

In practice, for structured source, to avoid having to construct a control flow graph, v is calculated by counting decision statements in the source code, which is equivalent to counting decision nodes.

A.3.2 Essential Complexity

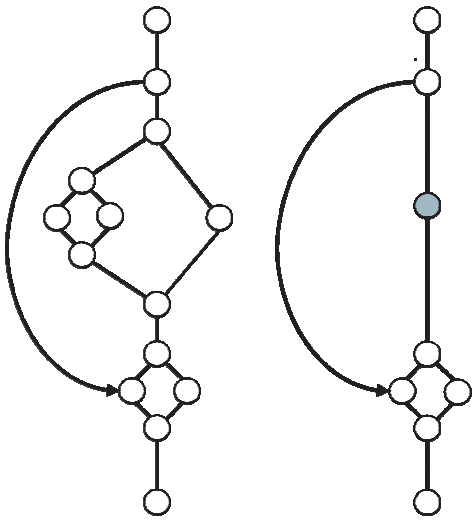
Essential Complexity measures the amount of unstructured decision logic in a module. Structured logic is described in terms of IF, WHILE and REPEAT statements, while unstructured in terms of GOTOs. Use of the latter can makes code much more difficult to follow, and for this reason is singled out for measurement. The metric is often denoted by EV(g), or just ev, and is calculated in the same way as cyclomatic complexity but on a reduced version of the graph in which structured logic is first removed as shown in Figure 102. A module with no unstructured logic reduces to only the start and end nodes which has a cyclomatic complexity of 1.

A.3.3 Design Complexity

Design complexity measures the amount of decision logic that has bearing on the calling of other modules. Complexity of module design is therefore measured in terms of the interaction between decision structure and subroutine calls. It is calculated in a way similar to essential complexity, in that the control flowgraph is reduced prior to calculation of cyclomatic complexity, but the reduced graph includes only decision structures which are needed to reach code blocks containing subroutine calls, as shown in Figure 103. A higher value is obtained the larger the amount of control structure, v, and the larger the number of blocks of code within it in which subroutine calls are made. Design complexity is often denoted by IV(g), or iv. This measure does not reflect directly on the number of different modules that are called, nor on the number of call statements. It reflects only on the number of different code blocks in which calls are made within the control structure, and more directly the number of paths formed from the control structures that are used to reach them.

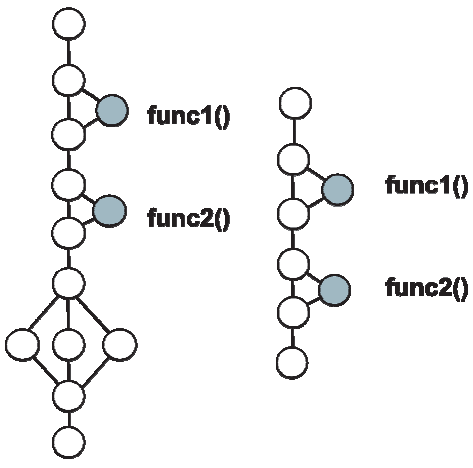
A.4 Chidamber & Kemerer Metrics

Traditionally software has been developed in the procedural style. This style is based upon the concept of the procedure call. Variables are defined and procedures contain code which perform a procedure, as a series of steps, that access and change the variables. However, as software became larger and more complex, shortcomings in this style became apparent.



Original, v = 5 Reduced, v = 3 (ev)

Figure 102 Essential complexity - structured logic is removed.



Original, v = 5 Reduced, v = 3 (iv)

Figure 103 Design complexity - logic which has no impact on subroutine calls is removed.

When the software crisis occurred in the late 1960's, the object oriented style was proposed as an alternative as a means to improving software quality, and it has since proliferated. It is based on the on the concept of the object rather than procedure. An object combines data and behaviour into a single coherent entity. A system is composed of many such interacting objects. Special characteristics of object oriented systems are inheritance, coupling, polymorphism. They are also highly modular. Given the difference in these approaches, the metrics that had been defined for procedural software were found to be inadequate for capturing the characteristics of object oriented software, and so for the latter new metrics were developed. The best known of these are the six Chidamber and Kemerer or CK metrics that were first introduced in 1991 (Chidamber & Kemerer) and later revised in 1994 (Chidamber & Kemerer).

The CK metrics have a strong theoretical basis the roots of which lie in the theories of Bunge in which there 'substantial individuals' each of which has properties. These correspond to an object and its methods. This theory mathematically formalizes concepts such as cohesion and coupling. The theory shows that the metrics relate to aspects of quality.

Given that the class is the most significant element in object oriented design, the CK metrics measure the class rather than function or module. The metrics are also measures of class design rather than implementation although they may be calculated from either. Altogether 6 metrics are defined which are:

- Weighted Methods Per Class
- Depth of Inheritance
- Number of Children
- Coupling Between Objects
- Response for Class
- Lack of Cohesion of Methods

Each is described in detail in the following sections.

A.4.1 Weighted Methods Per Class

Weighted Methods Per Class (WMC) is a measure of complexity of a class. It is calculated as the sum of the complexities of its methods:

$$WMC = \sum_{i=1}^{n} c_i$$

As there is no universally accepted method for measuring complexity, choice of measure is left as an implementation decision. However, given that methods may not have been implemented, often each is just assigned a complexity of 1. They are considered to be equally complex. In this case, WMC is effectively just a count of the number of methods in the class.

$$WMC = n$$

In this form the metric might not seem to be a good measure of class complexity, but it is in accordance with Bunges view that complexity is the cardinality of properties of a thing, the number of methods and attributes in the class (Chidamber & Kemerer, 1994). Attributes are ignored in this case however as they are not seen to contribute significantly to class complexity.

WMC is seen as an indicator of development and time and effort. It is also believed to be an indicator of application specificity. A class is more application specific and less reusable if it has a large number of methods. Chidamber and Kemerer observed that most classes have a small number of methods, 0..10 and concluded therefore that most are simple and have specific abstraction and functionality.

A.4.2 Depth of Inheritance

Depth of Inheritance (DIT) is a measure of the depth of a class in the inheritance hierarchy (ie the length of the maximal path from root node to class node). The inheritance hierarchy is the result of design choices about restricting or expanding the scope of properties of classes. DIT may be seen as a measure of the scope of a class in terms of the number of classes whose properties are propagated down the hierarchy to it. Or, similarly, as the number of scopes in super nodes within which the class lies and is potentially affected by. Number of Children described next is another measure of scope, and together these two metrics are thought to represent what is referred to as the genealogy of a class.

Deeper classes are thought to be more complex because they inherit more methods and their behaviour is less predictable. Chidamber and Kemerer noted that most classes have a low DIT, and that this might be due to developers sacrificing reuse through inheritance for simplicity of understanding.

A.4.3 Number of Children

Number of Children (NOC) measures of the number of immediate subclasses subordinate to a class in the inheritance hierarchy. NOC complements DIT in terms of the genealogy of a

class. It measures the number of classes that lie within the scope of the class and which are potentially affected by the its properties. While DIT looks at what effects a class, what scopes it lies within, NOC looks at what classes it affects, which classes lie within its scope.

NOC is seen to relate to reuse and abstraction. Higher values could be associated with more errors, especially for classes with a large number of methods due to difficulty there is in providing services to a range of subclasses each having their own context. Chidamber and Kemerer observed that most classes have few children.

A.4.4 Coupling Between Objects

Coupling Between Objects (CBO) is a measure of the number of classes to which a class is coupled. A class is coupled with another if it uses the methods or attributes of the other class, or vice versa. The coupling can be in either direction. It includes coupling through inheritance.

Higher coupling is seen to reduce modularity, reusability and maintainability. It may also increase error proneness as a highly coupled class is likely to be more sensitive to changes in the system. Chidamber and Kememer noted that higher coupling occurs in:

- languages which rely more heavily on message passing
- systems with more classes in them
- systems in which more inheritance is employed.
- interface classes which are coupled to all the classes that implement them.

They found that approximately 50% of classes do not refer to any other classes (including superclasses).

A.4.5 Response for Class

Response for Class (RFC) is a measure of the size of the response set for a class. It may also be regarded as a measure of potential communication as it counts the number of calls to other classes. A class can be viewed as containing responses to possible messages. The response set includes the methods of the class, as well as methods of other classes invoked by these methods that can also be invoked in response to a received message. Membership to the response set is usually defined only up to the first level of nesting of method calls due to practical considerations involved in calculating the metric. Formally, the response set is defined as:

$$RS = \{M\} \cup_{\forall i} \{R_i\}$$

Where $\{R_i\}$ is the set of methods called by method Ii and {M} is set of all methods in the class.

A larger response set is seen to increase class complexity. Chidamber and Kemerer observed that as with CBO, most classes have low values for this metric, and suggests that languages that rely more on message passing will have higher values.

A.4.6 Lack of Cohesion of Methods

Lack of Cohesion of Methods (LCOM) is a count of the number of method pairs in a class that are dissimilar minus the number of pairs that are similar. Two methods are considered to be similar if they share one or more attributes, dissimilar otherwise. A class is cohesive if its methods are similar in terms of being interrelated, all working closely within well bounded behaviour to support the class objective. Similarity of methods for this metric is measured in terms of attributes that they access that they have in common. The metric may be thought of as a measure of the disparateness in nature of the methods in the class. More formally, LCOM is given by:

Let P = set of method pairs in which no attributes are shared

Let Q = set of method pairs in which one or more attributes are shared

$$LCOM = \begin{cases} |P| - |Q| & if\ |P| > |Q| \\ 0 & otherwise \end{cases}$$

LCOM is highest when there are no attributes common to any pair of methods. Eg. if each method access a different attribute. LCOM decreases and drops to zero (that is, there is more cohesion) when the number of similar pairs, methods with attributes in common, increases to the point of being equal to the number of non-similar pairs. At this point of 'balance' between similar and dissimilar pairs, LCOM is 0. As similarity increases further, with similar pairs outnumbering dissimilar ones, the value would become negative. However, since it is a measure of lack of cohesion only, LCOM does not fall below 0. It has been suggested that the metric may be better without this constraint.

Appendix B – Metrics Tools

An example of a typical metrics tool is SDMetrics (SDMetrics). A screenshot of the metrics table in this tool is shown in Figure 104. It does not use the threshold based method for identifying high risk modules, but instead shades the specified percentile of values for each metric. This is perhaps not quite as tailored a method at identifying problem modules, not for each metric, but it is effective to the degree that the largest values tend to be associated with modules that contain faults. In the way of graphical presentation of the data, it like other tools uses histograms and Kiviat diagrams. Somewhat particular to it is that it only produces metrics on software designs specified in UML and as such only produces design metrics, not code ones.

Other tools that can be mentioned are Krakatau Professional (Power Software) for the large number of different metrics that it calculates, across the different categories of code metrics, traditional, procedural and object oriented. Another tool, CyVis (CyVis, Software Complexity Visualiser), displays the metrics it calculates in a custom plot in which classes are displayed as a series of vertical bars, modules within each as segments whose length reflects module size (either relative or absolute) and which are colour coded by risk according to whether the cyclomatic complexity of the module exceeds some specified threshold. This would seem to reflect a view by the author of the tool in which size as lines and code and the cyclomatic

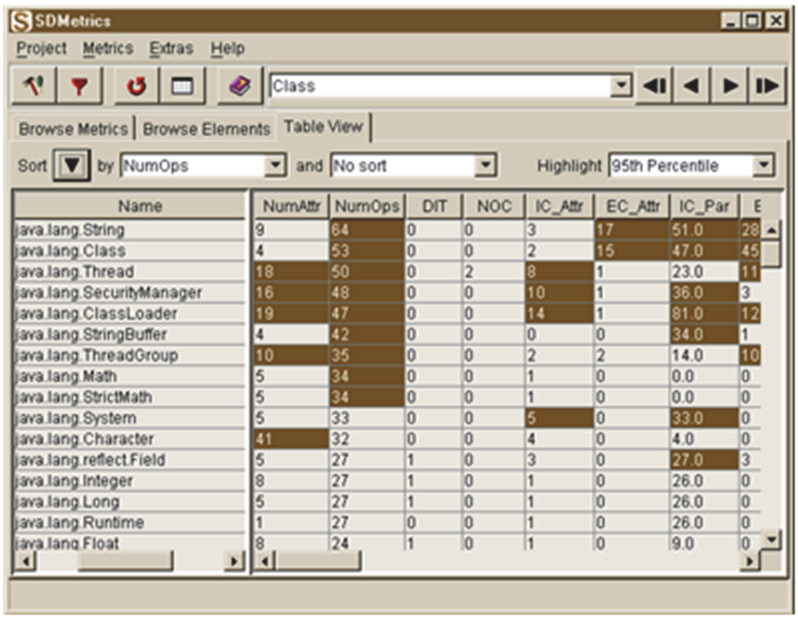


Figure 104 SDMetrics metrics collection tool and display of metrics by element in table format.

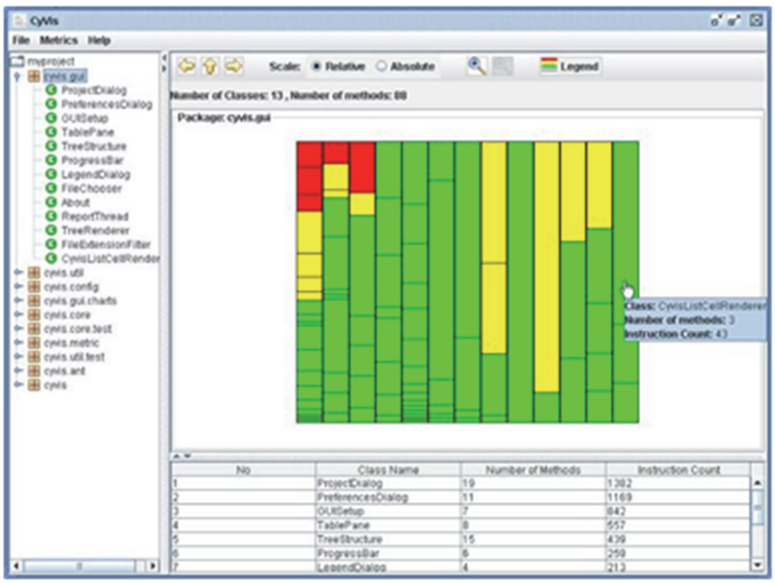


Figure 105 The CyVis metrics tool for visualising complexity metrics.

complexity metric are seen as important determinants of module quality. A screenshot of this is shown in Figure 105.

A metric plug-in for the development environment Eclipse produces a graph showing dependencies between modules. Focusing only on dependences, another tool, JDepend (JDepend), produces metrics for afferent and efferent coupling, reflecting a view that the relationships a module has with other modules by way of dependency is an important element of module quality. The jMetra tool (jMetra) is different in that the main program only outputs the system model in XML format. A set of secondary applications are also provided that transform the model into an HTML report which includes metrics that are calculated from the model. This approach opens access to the model, and allows new metrics to be defined simply by creating new transformations on the model. There may be a drawback however in that information might be lost in the XML model output that might restrictive on the metrics that can be calculated.

Overall it is apparent that there are a number of metrics tools available that can produce a large number of different metrics, and there is graphical representation to assist with analysis of the numbers to identify modules that may be lacking in quality, and in some cases a threshold-based approach is used as well to assist in the identification of those modules.

Appendix C - NASA and Eclipse Metrics

C.1 Nasa Metrics

The following is a table that lists all metrics in the NASA data sets, along with brief descriptions of them, and their abbreviations. Abbreviations are prefixed according to the metric group to which they belong. The table is of use for reference as the abbreviation metric names are often used instead of full names. The descriptions are based on those provided on the MDP website.

Group	Metric	Abbrev.	Description
McCabe	CYCLOMATIC_COMPLEXITY	mV	#Paths through decision graph.
	PATHOLOGICAL_COMPLEXITY	mVp	Degree of use of extremely unstructured constructs.
	ESSENTIAL_COMPLEXITY	mVe	Degree of use of unstructured constructs (#paths on reduced flow graph).
	CYCLOMATIC_DENSITY	mVd	Cyclomatic complexity / #non comment SLOC. Factors out size component of complexity.
	DECISION_DENSITY	mDd	#Conditions / #decisions.
	DESIGN_DENSITY	mId	Design complexity / cyclomatic complexity.
	ESSENTIAL_DENSITY	mEd	(Essential complexity–1) / (cyclomatic complexity–1).
	GLOBAL_DATA_DENSITY	mGd	Global data complexity / cyclomatic complexity.
	GLOBAL_DATA_COMPLEXITY	mGv	Cyclomatic complexity of structure as it relates to global/param. data.
	MAINTENANCE_SEVERITY	mMS	Essential complexity / cyclomatic complexity.
	DESIGN_COMPLEXITY	mIv	Measure of structure as it relates to calls to other modules.
	NORMALIZED_CYLOMATIC_COMPLEXITY	mVn	Cyclomatic complexity / #lines.
Error	ERROR_COUNT	eC	#Errors.
	ERROR_DENSITY	eD	#Errors / 1000 lines of code
	ERROR_REPORT_IN_1_YR	e1y	#Error reports in 1 year.
	ERROR_REPORT_IN_2_YRS	e2y	#Error reports in 2 years.
	ERROR_REPORT_IN_6_MON	e6m	#Error reports in 6 months.

Halstead	HALSTEAD_LENGTH	hN	Estimated length based on vocabulary (N from n).
	HALSTEAD_VOLUME	hV	Size in bits.
	HALSTEAD_LEVEL	hL	Level of abstraction of a module.
	HALSTEAD_DIFFICULTY	hD	Difficulty as inverse of level of abstraction.
	HALSTEAD_CONTENT	hI	Complexity independent of language.
	HALSTEAD_EFFORT	hE	Mental effort required for implementation. volume * difficulty.
	HALSTEAD_ERROR_EST	hB	Estimated #errors in module.
	HALSTEAD_PROG_TIME	hT	Estimated time (s) required for implementation based on effort.
LOC	NUMBER_OF_LINES	lN	#Lines regardless of character content.
	PERCENT_COMMENTS	lCp	Percentage of comments.
	LOC_BLANK	lB	#Lines containing white space or no text content.
	LOC_CODE_AND_COMMENT	lCC	#Lines containing both code and comment.
	LOC_COMMENTS	lC	#Lines that are purely comments.
	LOC_EXECUTABLE	lE	#Lines containing only code and white space.
	LOC_TOTAL	lT	#Lines.
Linguistic	NUM_OPERANDS	lnOpd	#Operands.
	NUM_OPERATORS	lnOpt	#Operators.
	NUM_UNIQUE_OPERANDS	lnOpdU	#Unique operands.
	NUM_UNIQUE_OPERATORS	lnOptU	#Unique operators.
Graph	NODE_COUNT	gN	#Nodes in CFG (control flow graph).
	CONDITION_COUNT	gC	#Conditions in CFG.
	DECISION_COUNT	gD	#Decisions in CFG.
	EDGE_COUNT	gE	#Edges in CFG.
	MODIFIED_CONDITION_COUNT	gCmod	#Conditions shown to independently affect a decision outcome.
	MULTIPLE_CONDITION_COUNT	gCmul	#Multiple conditions.
	BRANCH_COUNT	gB	#Branches defined as edges that exit from a decision node.
Other	CALL_PAIRS	cCP	#Calls to other functions.
	PARAMETER_COUNT	dP	#Parameters.

Table 42 NASA MDP metrics.

C.2 Eclipse Metrics

There are two groups of metrics in the Eclipse data sets. One is a small set of selected metrics, all file level with method level metrics raised to file level by taking the sum, average or maximum, and the other is a large set derived directly from the AST as counts of elements. These two sets are listed in the tables beneath. Abbreviations for the first non-AST metrics, are obtained by combining the metric name with quantity descriptor, eg FOUTAvg. For the AST metrics, abbreviations are listed.

Level	Metric	Quantity	Description
Method	FOUT	Avg, max, total	Number of method calls (fan out)
	MLOC	Avg, max, total	Method lines of code
	NBD	Avg, max, total	Nested block depth
	PAR	Avg, max, total	Number of parameters
	VG	Avg, max, total	McCabe cyclomatic complexity
Class	NOF	Avg, max, total	Number of fields
	NOM	Avg, max, total	Number of methods
	NSF	Avg, max, total	Number of static fields
	NSM	Avg, max, total	Number of static methods
File	ACD		Number of anonymous type declarations
	NOI		Number of interfaces
	NOT		Number of classes
	TLOC		Total lines of code
	pre		Number of errors pre-release.
	post		Number of errors post-release.

Table 43 Eclipse non-AST metrics.

AST Metric	**Abbrev**	**AST Metric**	**Abbrev.**
AnonymousClassDeclaration	ACD2	NORM_AnonymousClassDeclaration	n_ACD
ArrayAccess	AA	NORM_ArrayAccess	n_AA
ArrayCreation	AC	NORM_ArrayCreation	n_AC
ArrayInitializer	AI	NORM_ArrayInitializer	n_AI
ArrayType	AT	NORM_ArrayType	n_AT
AssertStatement	AS	NORM_AssertStatement	n_AS
Assignment	A	NORM_Assignment	n_A
Block	B	NORM_Block	n_B
BooleanLiteral	BL	NORM_BooleanLiteral	n_BL
BreakStatement	BS	NORM_BreakStatement	n_BS

CastExpression	CastE	NORM_CastExpression	n_CastE
CatchClause	CC	NORM_CatchClause	n_CC
CharacterLiteral	CL	NORM_CharacterLiteral	n_CL
ClassInstanceCreation	CIC	NORM_ClassInstanceCreation	n_CIC
CompilationUnit	CU	NORM_CompilationUnit	n_CU
ConditionalExpression	CE	NORM_ConditionalExpression	n_CE
ConstructorInvocation	CI	NORM_ConstructorInvocation	n_CI
ContinueStatement	CS	NORM_ContinueStatement	n_CS
DoStatement	DS	NORM_DoStatement	n_DS
EmptyStatement	ES	NORM_EmptyStatement	n_ES
ExpressionStatement	ExpS	NORM_ExpressionStatement	n_ExpS
FieldAccess	FA	NORM_FieldAccess	n_FA
FieldDeclaration	FD	NORM_FieldDeclaration	n_FD
ForStatement	FS	NORM_ForStatement	n_FS
IfStatement	LfS	NORM_IfStatement	n_LfS
ImportDeclaration	ID	NORM_ImportDeclaration	n_ID
InfixExpression	InfxE	NORM_InfixExpression	n_InfxE
Initializer	I	NORM_Initializer	n_I
Javadoc	J	NORM_Javadoc	n_J
LabeledStatement	JS	NORM_LabeledStatement	n_JS
MethodDeclaration	MD	NORM_MethodDeclaration	n_MD
MethodInvocation	MI	NORM_MethodInvocation	n_MI
NullLiteral	NL	NORM_NullLiteral	n_NL
NumberLiteral	NumL	NORM_NumberLiteral	n_NumL
PackageDeclaration	PD	NORM_PackageDeclaration	n_PD
ParenthesizedExpression	PE	NORM_ParenthesizedExpression	n_PE
PostfixExpression	PostE	NORM_PostfixExpression	n_PostE
PrefixExpression	PreE	NORM_PrefixExpression	n_PreE
PrimitiveType	PT	NORM_PrimitiveType	n_PT
QualifiedName	QN	NORM_QualifiedName	n_QN
ReturnStatement	RS	NORM_ReturnStatement	n_RS
SimpleName	SN	NORM_SimpleName	n_SN
SimpleType	ST	NORM_SimpleType	n_ST
SingleVariableDeclaration	SVD	NORM_SingleVariableDeclaration	n_SVD
StringLiteral	SL	NORM_StringLiteral	n_SL
SuperConstructorInvocation	SCI	NORM_SuperConstructorInvocation	n_SCI

SuperFieldAccess	SFA	NORM_SuperFieldAccess	n_SFA
SuperMethodInvocation	SMI	NORM_SuperMethodInvocation	n_SMI
SwitchCase	SC	NORM_SwitchCase	n_SC
SwitchStatement	SS	NORM_SwitchStatement	n_SS
SynchronizedStatement	SyncS	NORM_SynchronizedStatement	n_SyncS
ThisExpression	TE	NORM_ThisExpression	n_TE
ThrowStatement	TS	NORM_ThrowStatement	n_TS
TryStatement	TryS	NORM_TryStatement	n_TryS
TypeDeclaration	TD	NORM_TypeDeclaration	n_TD
TypeDeclarationStatement	TDS	NORM_TypeDeclarationStatement	n_TDS
TypeLiteral	TL	NORM_TypeLiteral	n_TL
VariableDeclarationExpression	VDE	NORM_VariableDeclarationExpression	n_VDE
VariableDeclarationFragment	VDF	NORM_VariableDeclarationFragment	n_VDF
VariableDeclarationStatement	VDS	NORM_VariableDeclarationStatement	n_VDS
WhileStatement	WS	NORM_WhileStatement	n_WS
InstanceofExpression	IE	NORM_InstanceofExpression	n_IE
LineComment	LC	NORM_LineComment	n_LC
BlockComment	BC	NORM_BlockComment	n_BC
TagElement	TagE	NORM_TagElement	n_TagE
TextElement	TxtE	NORM_TextElement	n_TxtE
MemberRef	MbrR	NORM_MemberRef	n_MbrR
MethodRef	MR	NORM_MethodRef	n_MR
MethodRefParameter	MRP	NORM_MethodRefParameter	n_MRP
EnhancedForStatement	EFS	NORM_EnhancedForStatement	n_EFS
EnumDeclaration	ED	NORM_EnumDeclaration	n_ED
EnumConstantDeclaration	ECD	NORM_EnumConstantDeclaration	n_ECD
TypeParameter	TP	NORM_TypeParameter	n_TP
ParameterizedType	ParT	NORM_ParameterizedType	n_ParT
QualifiedType	QT	NORM_QualifiedType	n_QT
WildcardType	WT	NORM_WildcardType	n_WT
NormalAnnotation	NA	NORM_NormalAnnotation	n_NA
MarkerAnnotation	MA	NORM_MarkerAnnotation	n_MA
SingleMemberAnnotation	SMA	NORM_SingleMemberAnnotation	n_SMA
MemberValuePair	MVP	NORM_MemberValuePair	n_MVP
AnnotationTypeDeclaration	ATD	NORM_AnnotationTypeDeclaration	n_ATD
AnnotationTypeMemberDeclaration	ATMD	NORM_AnnotationTypeMemberDeclaration	n_ATMD

Modifier	M	NORM_Modifier	n_M
SUM	S		

Table 44 Eclipse AST metrics.

Bibliography

(2009). Retrieved April 2009, from Promise Data Repository: http://promisedata.org/

Akbani, R., Kwek, S., & Japkowicz, N. (2004). Applying Support Vector Machines to Imbalanced Datasets. *Proceedings of the 15th European Conference on Machine Learning (ECML'2004)*, (pp. 39-50). Pisa, Italy.

Arisholm, E., Briand, L. C., & Fuglerud, M. (2007). Data Mining Techniques for Building Fault-proneness Models in Telecom Java Software. *Proceedings of the The 18th IEEE International Symposium on Software Reliability*, (pp. 215-224). Trollhattan, Sweden.

Basili, V., Briand, L., & Melo, W. (1996). A Validation of Object-Oriented Design Metrics as Quality Indicators. *IEEE Transactions on Software Engineering, 22*(10), 751-761.

Batista, G. E., Prati, R. C., & Monard, M. C. (2004). A study of the behavior of several methods for balancing machine learning training data. *ACM SIGKDD Explorations Newsletter, Special issue on learning from imbalanced datasets, 6*, 20-29.

Benedicenti, L., Wei Wang, V., Lee, P., & Paranjape, R. (2001). Establishing quality control in software agents. *ACM SIGAPP Applied Computing Review, 9*(3), 31 - 33.

Bernstein, A., Ekanayake, J., & Pinzger, M. (2007). Improving defect prediction using temporal features and non linear models. *Foundations of Software Engineering, Ninth international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting* (pp. 11 - 18). Dubrovnik, Croatia: ACM New York, NY, USA.

Blum, A. L., & Langley, P. (1997). Selection of relevant features and examples in machine learning. *Artificial Intelligence, 97*(1-2), 245 - 271.

Briand, L., Morasca, S., & Basili, V. (1999, September/October). Defining and Validating Measures for Object-Based High-Level Designs. *IEEE Transactions on Software Engineering, 25*(5), 722-743.

Brooks, F. A. (1995). *The Mythical Man-Month.* Reading, MA: Addison-Wesley.

Buchanan, B. G., & Wilkins, D. C. (1993). *Readings in Knowledge Acquisition and Learning: Automating the Construction and Improvement of Expert Systems.* Morgan Kaufmann.

Burges, C. J. (1998, June). A Tutorial on Support Vector Machines for Pattern Recognition. *Data Mining and Knowledge Discovery, 2*(2), 121-167.

Carbonell, J. G. (1989). Introduction: paradigms for machine learning. *Artificial Intelligence, 40*(1), 1-9.

Cartwright, M., & Shepperd, M. J. (2000). An Empirical Investigation of an Object-Oriented Software System. *IEEE Trans. Software Eng., 26*(8), 786-796.

Catal, C., & Diri, B. (2008). A Fault Prediction Model with Limited Fault Data to Improve Test Process. In *Product-Focused Software Process Improvement* (pp. 244-257). Springer.

Catal, C., Diri, B., & Ozumut, B. (2007). An Artificial Immune System Approach for Fault Prediction in Object-Oriented Software. *Proceedings of the 2nd International Conference on Dependability of Computer Systems* (pp. 238-245). Szklarska Poreba, Poland: IEEE Computer Society.

Chawla, N. V., Bowyer, K. W., & Kegelmeyer, P. W. (2002). SMOTE: Synthetic Minority Over-sampling Technique. *Journal of Artificial Intelligence Research, 16*, 321-357.

Chawla, N. V., Japkowicz, N., & Kolcz, A. (2004). Editorial: Special Issue on Learning from Imbalanced Data Sets. *SIGKDD Explorations, 6*(1), 1-6.

Chidamber, S. R., & Kemerer, C. F. (1991). Towards a metrics suite for object oriented design. *Conference proceedings on Object-oriented programming systems, languages, and applications*, (pp. 197-211). Phoenix, Arizona, United States.

Chidamber, S. R., & Kemerer, C. F. (1994). A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering, 20*(6), 476-493.

Cunningham, P. (2007). *Technical report UCD-CSI-2007-7.* University College Dublin.

CyVis, Software Complexity Visualiser. (n.d.). Retrieved April 2009, from http://cyvis.sourceforge.net/

Dash, M., Liu, H., & Motoda, H. (2000). Consistency Based Feature Selection. In *Knowledge Discovery and Data Mining. Current Issues and New Applications* (pp. 98-109). Berlin / Heidelberg: Springer.

Denaro, G., Morasca, S., & Pezzè, M. (2002). Deriving models of software fault-proneness. *SEKE, Proceedings of the 14th international conference on Software engineering and knowledge engineering* (pp. 361 - 368). Ischia, Italy: ACM New York, NY, USA.

Dick, S., Meeks, A., Last, M., Bunke, H., & Kandel, A. (2004). Data mining in software metrics databases. *Fuzzy Sets and Systems, Computational Intelligence in Software Engineering, 145*(11), 81-110.

Dietterich, T. G. (2003). *Learning and Reasoning.* School of Electrical Engineering and Computer Science, Oregon State University.

Dijkstra, E. W. (1968). Letters to the editor: go to statement considered harmful. *Communications of the ACM, 11*(3), 147 - 148.

Drown, D. J., Khoshgoftaar, T. M., & Narayanan, R. (2007). Using Evolutionary Sampling to Mine Imbalanced Data. *Proceedings of the Sixth International Conference on Machine Learning and Application, ICMLA* (pp. 363-368). Cincinnati, Ohio, USA: IEEE Computer Society.

Eclipse Bug Data. (n.d.). Retrieved April 2009, from http://www.st.cs.uni-saarland.de/softevo/bug-data/eclipse/

Elish, K. O., & Elish, M. O. (2008). Predicting defect-prone software modules using support vector machines. *JOURNAL OF SYSTEMS AND SOFTWARE, 81*(5), 649-660.

Elish, K. O., & Elishee, M. O. (2008). Predicting defect-prone software modules using support vector machines. *JOURNAL OF SYSTEMS AND SOFTWARE*.

Emam, K. E., Benlarbi, S., Goel, N., & Rai, S. N. (2001). The Confounding Effect of Class Size on the Validity of Object-Oriented Metrics. *IEEE Trans. Software Eng., 27*(7), 630-650.

Emam, K. E., Melo, W. L., & Machado, J. C. (2001). The prediction of faulty classes using object-oriented design metrics. *Journal of Systems and Software, 56*(1), 63-75.

Fenton, N., & Pfleeger, S. (1997). *Software Metrics. A Rigorous and Practical Approach* (2nd ed.). International Thomson Computer Press.

Foreman, G., Guyon, I., & Elisseeff, A. (2003). An extensive empirical study of feature selection metrics for text classification. *The Journal of Machine Learning Research, 3*, 1289 - 1305.

Gau, P., & Lyu, M. R. (2000). Software Quality Prediction Using Mixture Models with EM Algorithm. *Proceedings of the First Asia-Pacific Conference on Quality Software (APAQS 2OOO)*, (pp. 69-78). Hong Kong.

Gibbs, W. (1994, September). Software's chronic crisis. *Scientific American, 271*(3), 86-95.

Golub, T. R., Slonim, D. K., Tamayo, P., Huard, C., Gaasenbeek, M., Mesirov, J. P., et al. (1999). Molecular classification of cancer: class discovery and class prediction by gene expression monitoring. *Science, 286*(5439), 531-537.

Gondra, L. (2008). Applying machine learning to software fault proneness prediction. *The Journal of Systems & Software, 81*(2), 186-195.

Graves, T. L., Karr, A. F., Marron, J. S., & Siy, H. (2000). Predicting fault incidence using software change history. *Software Engineering, IEEE Transactions on, 26*(7), 653 - 661.

Guo, L., Ma, Y., & Harshinder, S. (2004). Robust prediction of fault-proneness by random forests. *15th International Symposium on Software Reliability Engineering, 2004. ISSRE 2004.*, (pp. 417- 428). St-Malo, France.

Guo, X., Yin, Y., Dong, C., Yang, G., & Zhou, G. (2008). On the Class Imbalance Problem. *Proceedings of the 2008 Fourth International Conference on Natural Computation - Volume 04* (pp. 192-201). Washington, DC, USA: IEEE Computer Society.

Gupta, D., Brar, A. S., & Sandhu, P. S. (2008). Modeling Of Fault Prediction Using Machine Learning Techniques. *Proceedings of 2nd National Conference on Challenges & Opportunities in Information Technology (COIT-2008).* Mandi Gobindgarh, India.

Guyon, I. (2008). Practical Feature Selection: from Correlation to Causality. In F. Fogelman-Soulié, D. Perrotta, J. Piskorski, & R. Steinberger, *Mining Massive Data Sets for Security* (pp. 27 - 43). IOS Press.

Guyon, I., & Elisseeff, A. (2003). An introduction to variable and feature selection. *The Journal of Machine Learning Research, 3*(7), 1157 - 1182.

Hall, M. A. (1999). Correlation-based Feature Selection for Machine Learning, Thesis.

Halstead, M. (1977). *Elements of Software Science.* New York: Elsevier-North Holland.

Han, J., & Micheline, K. (2006). *Data Mining: Concepts and Techniques 2nd Edition.* San Francisco: Morgan Kaufmann Publishers.

Hartmann, D. (2009, August 25). *Standish CHAOS Report Methods Questioned*. Retrieved October 2009, from InfoQ: http://www.infoq.com/news/Standish-Chaos-Report-Questioned

Hughes, R. (2000). *Practical Software Measurement.* Mcgraw-Hill (Tx).

Itzfeldt, W. D. (1990). Quality metrics for software management and engineering. In R. J. Mitchell, *Managing Complexity in software engineering, IEE Computing Series 17* (pp. 127-151). London: Peregrinus.

Japkowicz, N. (2001). Concept-Learning in the Presence of Between-Class and Within-Class Imbalances. *Proceedings of the 14th Biennial Conference of the Canadian Society on Computational Studies of Intelligence: Advances in Artificial Intelligence*, (pp. 67-77).

Japkowicz, N., & Stephen, S. (2002). The class imbalance problem: A systematic study. *Intelligent Data Analysis, 6*(5), 429 - 449.

JDepend. (n.d.). Retrieved April 2009, from Clarkware Consulting: http://clarkware.com/software/JDepend.html

Jiang, Y., Cukic, B., & Menzies, T. (2007). Fault Prediction using Early Lifecycle Data. *Proceedings of the The 18th IEEE International Symposium on Software Reliability* (pp. 237-246). Trollhättan, Sweden: IEEE Computer Society.

Jiang, Y., Cukic, B., & Menzies, T. (2008). Can data transformation help in the detection of fault-prone modules? *International Symposium on Software Testing and Analysis, Proceedings of the 2008 workshop on Defects in large software systems*, (pp. 16-20). Seattle, Washington.

jMetra. (n.d.). Retrieved April 2009, from hyperCision Inc.: http://hc.techstylesusa.com/jMetra.html

John, G. H., Kohavi, R., & Pfleger, K. (1994). Irrelevant Features and the Subset Selection Problem. *Proceedings of the 11th International Conference on Machine Learning*, (pp. 121-129). San Francisco, CA.

Kastro, Y., & Bener, B. A. (2008). A defect prediction method for software versioning. *Software Quality Control, 16*(4), 543 - 562.

Khoshgoftaar, T. M., & Seliya, N. (2002). Improving usefulness of software quality classification models based on Boolean discriminant functions. *13th International Symposium on Software Reliability Engineering, 2002. ISSRE 2002. Proceedings.*, (pp. 221- 230). Annapolis, MD, USA.

Khoshgoftaar, T. M., & Seliya, N. (2004). The necessity of assuring quality in software measurement data. *10th International Symposium on Software Metrics, 2004. Proceedings.*, (pp. 119- 130). Chicago, Illinois, USA.

Khoshgoftaar, T. M., Geleyn, E., & Nguyen, L. (2003). Empirical case studies of combining software quality classification models. *Third International Conference on Quality Software, 2003. Proceedings.*, (pp. 40 - 49). Dallas, Texas, USA.

Khoshgoftaar, T. M., Rebours, P., & Seliya, N. (2009). Software quality analysis by combining multiple projects and learners. *Software Quality Control, 17*(1), 25 - 49.

Khoshgoftaar, T. M., Seliya, N., & Sundaresh, N. (2006). An Empirical Study of Predicting Software Faults with Case-Based Reasoning. *Software Quality Journal, 14*(2), 85-110.

Khoshgoftaar, T. M., Zhong, S., & Joshi, V. (2005). Enhancing software quality estimation using ensemble-classifier based noise filtering. *Intelligent Data Analysis, 9*(1), 3-27.

Kohavi, R., & John, G. H. (1997). Wrappers for feature subset selection. *Artificial Intelligence, 97*(1-2), 273 - 324.

Kolence, K., & Kiviat, P. (1973). Software Unit Profiles and Kiviat Figures. *ACM Performance Evaluation Review, 2*(3), 2–12.

Koller, D., & Sahami, M. (1996). Toward Optimal Feature Selection. *Proceedings of the Thirteenth International Conference on Machine Learning (ICML)* (pp. 284--292). Bary, Italy: Morgan Kaufmann Publishers.

Koru, A. G., & Liu, H. (2005). An investigation of the effect of module size on defect prediction using static measures. *International Conference on Software Engineering archive, Proceedings of the 2005 workshop on Predictor models in software engineering* (pp. 1 - 5). St. Louis, Missouri: ACM.

Kotsiantis, S. B., Zaharakis, I. D., & Pintelas, P. E. (2006). Machine learning: a review of classification and combining techniques. *ARTIFICIAL INTELLIGENCE REVIEW, 26*(3), 159-190.

Langley, P. (1996). *Elements of Machine Learning.* San Francisco: Morgan Kaufman Publishers, Inc.

Lecocke, M. L., & Hess, K. (2004). *An Empirical Study of Optimism and Selection Bias in Binary Classification with Microarray Data.* UT MD Anderson Cancer Center Department of Biostatistics Working Paper Series. Working Paper 3.

Lessmann, S., Baesens, B., Mues, C., & Pietsch, S. (2008). Benchmarking Classification Models for Software Defect Prediction: A Proposed Framework and Novel Findings. *IEEE Transactions on Software Engineering, 34*(4), 485 - 496.

Levinson, M. (2001, October). Let's Stop Wasting $78 Billion a Year. *CIO Magazine*.

Li, L. P., Herbsleb, J., & Shaw, M. (2005). Finding Predictors of Field Defects for Open Source Software Systems in Commonly Available Data Sources: A Case Study of OpenBSD. *Proceedings of the 11th IEEE International Software Metrics Symposium* (p. 32). Como, Italy: IEEE Computer Society.

Lipton, R. (1991). New Directions in Testing. In *Distributed Computing and Cryptography, DIMACS Series on Discrete Mathematics and Theoretical Computer Science 2* (pp. 191-202). American Mathematical Society.

Liu, H., & Wong, L. (2003). Data Mining Tools for Biological Sequences. *Journal of Bioinformatics and Computational Biology, 1*(1), 139-168.

Lorenz, M., & Kidd, J. (1994). *Object-Oriented Software Metrics: A Practical Approach.* Prentice-Hall.

Ma, Y., & Cukic, B. (2007). Adequate and Precise Evaluation of Quality Models in Software Engineering Studies. *International Conference on Software Engineering, Proceedings of the Third International Workshop on Predictor Models in Software Engineering* (p. 1). Washington, DC, USA: IEEE Computer Society.

Macro, L. (1997, April). Measuring software complexity. *Enterprise Systems Journal*.

McCabe, T. J. (1976). A Complexity Measure. *IEEE Transactions on Software Engineering, 2*(4), 308-320.

McCall, J. A., Richards, P. K., & Walters, G. F. (1978). *Factors in Software Quality.* RADC Reports.

McLachlan, G. J., Chevelu, J., & Zhu, J. (2008). Correcting for Selection Bias via Cross-Validation in the Classification of Microarray Data. In N. Balakrishnan, E. A. Pena, & M. J. Silvapulle, *Beyond Parametrics in Interdisciplinary Research: Festschrift in Honor of Professor Pranab K. Sen* (pp. 364-376). United States: Institute of Mathematical Statistics.

Menzies, T. (2006). *From Static Code Measures to Defect Predictors.* Retrieved November 2010, from IV&V Facility Research Program Results and SARP Results: http://sarpresults.ivv.nasa.gov/DownloadFile/107/12/See More, Tell More, Learn More Papers.pdf

Menzies, T., Ammar, K., Nikora, A., & Di Stefano, J. (2003). How Simple is Software Defect Detection? *International Symposium on Software Reliability Engineering.* Denver, CO, USA: Kluwer Academic Publishers.

Menzies, T., Greenwald, J., & Frank, A. (2007). Data Mining Static Code Attributes to Learn Defect Predictors. *IEEE Transactions on Software Engineering, 33*(1), 2-13.

Menzies, T., Raffo, D., Setamanit, S., DiStefano, J., & Chapman, R. M. (2004). *Why Mine Software Repositories?* Retrieved November 2010, from Tim Menzies: http://menzies.us/pdf/04whymine.pdf

Menzies, T., Stefano, J. D., Ammar, K., McGill, K., Callis, P., Davis, J., et al. (2003). When can we test less? *Ninth International Software Metrics Symposium, 2003. Proceedings.*, (pp. 98 - 110). Sydney, Australia.

Menzies, T., Turhan, B., Bener, A., Gay, G., Cukic, B., & Jiang, Y. (2008). Implications of ceiling effects in defect predictors. *International Conference on Software Engineering archive, Proceedings of the 4th international workshop on Predictor models in software engineering* (pp. 47-54). ACM.

Metrics Data Program. (n.d.). Retrieved April 2009, from NASA IV&V Facility: http://mdp.ivv.nasa.gov/

Michalski, R. (1986). Understanding The Nature Of Learning: Issues and Research. In R. Michalski, J. Carbonnel, & T. Mitchell, *Machine Learning: A Multistrategy Approach, Volume II.* Los Altos: Kaufmann Publishers.

Michalski, R. (1994). Inferential Theory of Learning: Developing Foundations for Multistrategy Learning. In R. S. Michalski, & G. Tecuci, *Machine Learning: A Multistrategy Approach, Volume IV*. San Mateo: Morgan Kaufmann.

Michalski, R. S., Carbonell, J. G., & Mitchell, T. M. (1983). *Machine Learning: An Artificial Intelligence Approach.* Palo Alto, California.

Mitchell, T. (1997). *Machine Learning.* McGraw Hill.

Mizuno, O., Ikami, S., Nakaichi, S., & Kikuno, T. (2007). Fault-Prone Filtering: Detection of Fault-Prone Modules Using Spam Filtering Technique. *First International Symposium on Empirical Software Engineering and Measurement, 2007. ESEM 2007.* (pp. 374-383). Madrid, Spain: IEEE Computer Society.

Nagappan, N., & Ball, T. (2005). Static analysis tools as early indicators of pre-release defect density. *International Conference on Software Engineering, Proceedings of the 27th international conference on Software engineering* (pp. 580 - 586). St. Louis, MO, USA: ACM New York, NY, USA.

Ostrand, T. J., Weyuker, E. J., & Bell, R. M. (2004). Where the bugs are. *International Symposium on Software Testing and Analysis*, (pp. 86-96). Boston, Massachusetts.

Pelayo, L., & Dick, S. (2007). Applying Novel Resampling Strategies To Software Defect Prediction. *Annual Meeting of the North American Fuzzy Information Processing Society, 2007. NAFIPS apos;07.* , (pp. 24-27).

Pizzi, N. J. (2007). Software quality prediction using fuzzy integration: a case study. *Soft Computing - A Fusion of Foundations, Methodologies and Applications, 12*(1), 67 - 76.

Power Software. (n.d.). Retrieved April 2009, from http://www.powersoftware.com/

Prati, R. C., Batista, G. E., & Monard, M. C. (2004). Class Imbalances versus Class Overlapping: An Analysis of a Learning System Behavior. In *MICAI 2004: Advances in Artificial Intelligence: Third Mexican International Conference on Artificial Intelligence* (pp. 312-321). Mexico City, Mexico: Springer.

Quinlan, J. R. (1986). Induction of Decision Trees. *Machine Learning, 1*(1), 81 - 106.

Ramanna, S., Bhatt, R., & Biernot, P. (2007). Software Defect Classification: A Comparative Study with Rough Hybrid Approaches. *Proceedings of the international conference on Rough Sets and Intelligent Systems Paradigms* (pp. 630 - 638). Warsaw, Poland: Springer-Verlag.

Raskutti, B., & Kowalczyk, A. (2004). Extreme re-balancing for SVMs: a case study. *ACM SIGKDD Explorations Newsletter, Special issue on learning from imbalanced datasets*, 60 - 69.

Riguzzi, F. (1996). *A survey of software metrics.* Technical Report DEIS-LIA-96-010, LIA Series n.17, DEIS, University of Bologna.

Robnik-Sikonja, M., & Kononenko, I. (1997). An adaptation of Relief for attribute estimation in regression. *Proceedings of the Fourteenth International Conference on Machine Learning* (pp. 296 - 304). Nashville, Tennessee, USA: Morgan Kaufmann Publishers Inc.

Rodriguez, D., Ruiz, R., Cuadrado-Gallego, J., & Aguilar-Ruiz, J. (2007). Detecting Fault Modules Applying Feature Selection to Classifiers. *IEEE International Conference on Information Reuse and Integration, 2007. IRI 2007.*, (pp. 667 - 672). Las Vegas, Nevada, USA.

Russell, S. J., & Norvig, P. (1995). *Artificial Intelligence: A Modern Approach* (2 ed.). NJ: Prentice Hall.

Schroeder, M. (1999). A practical guide to object-oriented metrics. *IT Professional, 1*(6), 30-36.

Schröter, A., Zimmermann , T., & Zeller, A. (2006). Predicting component failures at design time. *International Symposium on Empirical Software Engineering, Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering* (pp. 18 - 27). Rio de Janeiro, Brazil: ACM New York, NY, USA.

SDMetrics. (n.d.). Retrieved April 2009, from http://www.sdmetrics.com/

Selby, R. W., & Porter, A. A. (1988). Learning from Examples: Generation and Evaluation of Decision Trees for Software Resource Analysis. *IEEE Trans. Software Eng, 14*(12), 1743--1757.

Seliya, N., & Khoshgoftaar, T. M. (2007). Software quality estimation with limited fault data: a semi-supervised learning perspective. *Software Quality Control, 15*(3), 327 - 344.

Seliya, N., & Khoshgoftaar, T. M. (2007). Software Quality Modeling With Limited Apriori Defect Data. In X. Zhu, & I. Davidson, *Knowledge Discovery and Data Mining: Challenges and Realities with Real World Data* (pp. 1-15). Idea Group Publishing.

Sharma, P. (2004). *Software Engineering.* Aph Publishing Corporation.

Shin, M., Goel, A. L., Ratanothayanon, S., & Paul, R. A. (2007). Parsimonious classifiers for software quality assessment. *High Assurance Systems Engineering Symposium* (pp. 411 - 412). Dallas, Texas, USA: IEEE Computer Society 2007.

Singh, Y., Kaur, A., & Malhotra, R. (2008). Predicting Software Fault Proneness Model Using Neural Network. In *Agile Processes in Software Engineering and Extreme Programming* (Vol. 9, pp. 215-217). Springer.

Singhi, S. K., & Liu, H. (2006). Feature subset selection bias for classification learning. *ACM International Conference Proceeding Series, Proceedings of the 23rd international*

conference on Machine learning. 148, pp. 849 - 856. Pittsburgh, Pennsylvania: ACM New York, NY, USA.

Srinivasa, K. G., Venugopal, K. R., & Patnaik, L. M. (2006). Feature Extraction using Fuzzy C - Means Clustering for Data Mining Systems. *International Journal of Computer Science and Network Security, 6*(3A), 230-236.

Standish Group International. (n.d.). *The Chaos Report*. Retrieved June 28, 2004, from www.standishgroup.com/sample_research/PDFpages/Chaos2004.pdf

Subramanyam, R., & Krishnan, M. S. (2003). Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects. *Software Engineering, IEEE Transactions on, 29*(4), 297- 310.

Tang, M.-H., Kao, M.-H., & Chen, M.-H. (1999). An Empirical Study on Object-Oriented Metrics. *6th IEEE International Software Metrics Symposium (METRICS 1999)* (pp. 242-249). Boca Raton, FL, USA: IEEE Computer Society.

Tang, W., & Khoshgoftaar, T. M. (2004). Noise identification with the k-means algorithm. *16th IEEE International Conference on Tools with Artificial Intelligence, 2004. ICTAI 2004.*, (pp. 373- 378). Boca Raton, FL, USA.

The Spider. (n.d.). Retrieved August 2009, from http://www.kyb.mpg.de/bs/people/spider/main.html

Thornton, C. J. (1992). *Techniques in Computational Learning: An Introduction.* London ; New York: Chapman & Hall Computing.

Tosun, A., Turhan, B., & Bener, A. (2008). Ensemble of software defect predictors: a case study. *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, (pp. 318-320). Kaiserslautern, Germany.

Turhan, B., & Bener, A. (2007). Software Defect Prediction: Heuristics for Weighted Naive Bayes. *Proceedings of the 2nd International Conference on Software and Data Technologies (ICSOFT 2007)*, (pp. 244-249). Barcelona, Spain.

van Rijsbergen, C. J. (1979). *Information Retrieval* (2nd Edition ed.). Butterworth.

Vandecruys, O., Martens, D., Baesens, B., Mues, C., De Backe, M., & Haesen, R. (2008). Mining software repositories for comprehensible software fault prediction models. *Journal of Systems and Software, 81*(5), 823-839.

Vapnik, V. N. (1995). *The nature of statistical learning theory.* Springer-Verlag New York, Inc.

Wagner, S., & Jurjens, J. (2005). Model-Based Identification of Fault-Prone Components. In *Dependable Computing - EDCC 2005* (Vol. 3463/2005, pp. 435-452). SpringerLink.

Weiss, G. M. (2004). Mining with rarity: a unifying framework. *ACM SIGKDD Explorations Newsletter, Special issue on learning from imbalanced datasets, 6*(1), 7 - 19.

Weiss, S. M., & Indurkhya, N. (1988). *Predictive data mining: a practical guide.* San Francisco, CA: Morgan Kaufmann Publishers Inc.

Witten, I. H., & Frank, E. (2000). *DataMining: practical machine learning tools and techniques with Java implementations.* Morgan Kaufmann Publishers.

Xing, F., Guo, P., & Lyu, M. R. (2005). A novel method for early software quality prediction based on support vector machine. *16th IEEE International Symposium on Software Reliability Engineering, 2005. ISSRE 2005.*, (pp. 213-222). Chicago, IL, USA.

Yang, B., Yao, L., & Huang, H.-Z. (2007). Early Software Quality Prediction Based on a Fuzzy Neural Network Model. *Proceedings of the Third International Conference on Natural Computation - Volume 01* (pp. 760-764). Haikou, China: IEEE Computer Society.

Yang, Y., Xia, Y., Chi, Y., & Muntz, R. R. (2003). *Learning Naive Bayes Classifier from Noisy Data.* Technical Report CSD-TR No. 030056 1, UCLA Computer Science Department.

Yu, L., & Liu, H. (2003). Feature Selection for High-Dimensional Data: A Fast Correlation-Based Filter Solution. *In Proceedings of The Twentieth International Conference on Machine Leaning (ICML-03)*, (pp. 856–863). Washington, D.C.

Yu, L., & Liu, H. (2004). Efficient Feature Selection via Analysis of Relevance and Redundancy. *The Journal of Machine Learning Research, 5*, 1205 - 1224.

Yu, T., Debenham, J., Jan, T., & Simoff, S. (2006). Combine Vector Quantization and Support Vector Machine for Imbalanced Datasets. In *Artificial Intelligence in Theory and Practice* (Vol. 217, pp. 81-88). Springer Boston.

Zhang, D., & Tsai, J. J. (2003). Machine Learning and Software Engineering. *Software Quality Journal, 11*(2), 87-119.

Zhang, H., & Zhang, X. (2007, June). Comments on "Data Mining Static Code Attributes to Learn Defect Predictors". *IEEE Transactions on Software Engineering, 33*(9), 635-637.

Zheng, J., Williams, L., Nagappan, N., Snipes, W., Hudepohl, J., Vouk, M., et al. (2005). *On the Effectiveness of Static Analysis Tools for Fault-Detection.* North Carolina State University.

Zheng, J., Williams, L., Naggapan, N., Snipes, W., Hudepohl, J. P., & Vouk, M. A. (2006, April). On the value of static analysis for fault detection in software. *IEEE Transactions on Software Engineering, 32*(4), 240-253.

Zheng, Z., Wu, X., & Srihari, R. (2004). Feature selection for text categorization on imbalanced data. *ACM SIGKDD Explorations Newsletter, 6*(1), 80 - 89.

Zhong, S., Khoshgoftaar, T. M., & Seliya, N. (2004). Analyzing Software Measurement Data with Clustering Techniques. *IEEE Intelligent Systems, 19*(2), 20-27.

Zhong, S., Khoshgoftaar, T. M., & Seliya, N. (2004). Unsupervised learning for expert-based software quality estimation. *Proceedings on the Eighth IEEE International Symposium on High Assurance Systems Engineering*, (pp. 149-155). Tampa, Florida, USA.

Zimmerman, T., Nagappan, N., & Zeller, A. (2007). Predicting Bugs from History. *PROMISE '07: Proceedings of the Third International Workshop on Predictor Models in Software Engineering, 2007* (p. 9). Springer.

Zimmermann, T., Premraj, R., & Zeller, A. (2007). Predicting Defects for Eclipse. *Proceedings of the Third International Workshop on Predictor Models in Software Engineering* (p. 9). Minneapolis, MN, USA: IEEE Computer Society.