FACTORS AFFECTING THE PERFORMANCE OF TRAINABLE MODELS FOR SOFTWARE DEFECT PREDICTION

by

DAVID HUTCHINSON BOWES

Submitted to the University of Hertfordshire in partial fulfilment
of the requirements of the degree of

DOCTOR OF PHILOSOPHY

School of Computer Sciences
University of Hertfordshire

June, 2013

Acknowledgements

My first and foremost acknowledgement goes to Tracy Hall. It is while working with Tracy that I have learnt the dogma: the need for RIGOUR and the concept of FOCUS; how can I do better. Rigour means that every word has to be justified and consistent[1]. Focus means that this dissertation is not the original 500 pages. How can I do better? means that although this dissertation has ended, the work has not.

My second acknowledgement goes to my other supervisors, Neil Davey and Bruce Christianson. It has bean a pleasure working with them and learning the different ways of doing and thinking about research.
My third acknowledgement goes to my parents who provided a stimulating and thought provoking environment in my formative years which I constantly reflect on while bringing up my own son.

My fourth acknowledgement goes to Kath Walley, my A'level physics teacher who introduced me to the problem of measurement error.

My fifth acknowledgement goes to my examiners (Prof. Barbara Kitchenham, Prof. Mark Harman and Dr. Nathan Baddoo) who have rigorously checked every equation and detail which makes this final version better than the last.

My penultimate acknowledgement goes to Jenny Jeffery for her proof reading of this dissertation and for the professionalism that she taught me while sharing an office and managing a Sixth Form in a large Secondary School.

My last acknowledgement is to the other colleagues who I have worked with while teaching and researching. Their contribution has been inspirational and usually good fun.

Finally, when my Dad died he left me his collection of books. Included in his collection was a handwritten book which contained a dissertation of all the answers to the questions I had asked him, but he had never answered. I had always thought it strange that he would not answer some questions even though he was more than capable of answering them. I have come to the conclusion that he thought it better for me to work out the answers myself rather than being fed someone else's answer. It is probably his dogma, that has made me continue asking why. At the end of his dissertation he included a glossary of all of his favourite sayings which as a child I had heard him repeat many times. However, at the start of the list was one saying I had never heard him say :

> "If you have time to waste, don't waste it on those who are busy!" D.C.Bowes.

[1] Any lack of consistency in this dissertation is down to me!

ABSTRACT

Context. Reports suggest that defects in code cost the US in excess of $50billion per year to put right. Defect Prediction is an important part of Software Engineering. It allows developers to prioritise the code that needs to be inspected when trying to reduce the number of defects in code. A small change in the number of defects found will have a significant impact on the cost of producing software.

Aims. The aim of this dissertation is to investigate the factors which affect the performance of defect prediction models. Identifying the causes of variation in the way that variables are computed should help to improve the precision of defect prediction models and hence improve the cost effectiveness of defect prediction.

Methods. This dissertation is by published work. The first three papers examine variation in the independent variables (code metrics) and the dependent variable (number/location of defects). The fourth and fifth papers investigate the effect that different learners and datasets have on the predictive performance of defect prediction models. The final paper investigates the reported use of different machine learning approaches in studies published between 2000 and 2010.

Results. The first and second papers show that independent variables are sensitive to the measurement protocol used, this suggests that the way data is collected affects the performance of defect prediction. The third paper shows that dependent variable data may be untrustworthy as there is no reliable method for labelling a unit of code as defective or not. The fourth and fifth papers show that the dataset and learner used when producing defect prediction models have an effect on the performance of the models. The final paper shows that the approaches used by researchers to build defect prediction models is variable, with good practices being ignored in many papers.

Conclusions. The measurement protocols for independent and dependent variables used for defect prediction need to be clearly described so that results can be compared like with like. It is possible that the predictive results of one research group have a higher performance value than another research group because of the way that they calculated the metrics rather than the method of building the model used to predict the defect prone modules. The machine learning approaches used by researchers need to be clearly reported in order to be able to improve the quality of defect prediction studies and allow a larger corpus of reliable results to be gathered.

Contents

1 Introduction 1
1.1 Aim 1
1.2 Introduction 2
1.3 Thesis and Research Questions 3
1.4 Contributions to Knowledge 3
1.5 Structure of this Dissertation 5

2 A Summary of Defect Prediction 7
2.1 Introduction 7
2.2 What is a Defect? 7
2.2.1 What is the Purpose of Defect Prediction? 7
2.2.2 What is Defect Prediction? 8
2.2.3 Code Defects 8
2.3 Variables Used in Defect Prediction Studies 10
2.3.1 Independent Variables 11
2.3.2 Dependent Variables 11
2.4 Modelling Techniques 11
2.4.1 Continuous Techniques 12
2.4.2 Categorical Techniques 14
2.5 Data Quality and Data Cleaning 20
2.6 Model Building Approaches 20
2.6.1 Building Generalisable Models 20
2.6.2 Testing the Generalisability of Models 21
2.7 Issues of Model Building Specific to Machine Learning 21
2.7.1 Dealing with Datasets which are Unbalanced 22
2.7.2 Machine Learning Model Optimisation 22
2.7.3 Model Tuning 23
2.7.4 Reordering the Training Data 23
2.8 Measuring the Performance of a Model 25
2.8.1 Measuring the Performance of Categorical Prediction Models 25
2.9 Summary of Techniques 28
2.10 Conclusion 28

3 Contribution of Papers 31
3.1 Introduction 31
3.2 RQ1: Does the measurement protocol for the independent variables affect the metric values produced? 31
3.3 RQ2: Is there an effective method for deriving the dependent variables for defect prediction? 33
3.4 RQ3: Which factors (dataset, learner) affect the performance of defect prediction studies? 34
3.5 RQ4: Are the results of machine learning studies reliable/trustworthy? 35
3.6 Summary of my Contribution to each Main Paper 36

4 Papers **39**
4.1 Paper 1: Calibrating program slicing metrics for practical use. 41
4.2 Paper 2: The Inconsistent Measurement of Message Chains. 45
4.3 Paper 3: Evaluating Three Approaches to Extracting Fault Data from Software Change Repositories. 55
4.4 Paper 4: A Systematic Literature Review on Fault Prediction Performance in Software Engineering. 67
4.4.1 Corrigendum 67
4.5 Paper 5: Comparing the performance of fault prediction models which report multiple performance measures: recomputing the confusion matrix. 99
4.5.1 Corrigenda 99
4.6 Paper 6: The State of Machine Learning Methodology in Software Fault Prediction. . . . 111

5 Conclusion **119**
5.1 Reflection on the Research Questions 119
5.2 Main Findings 121
5.2.1 Analysis 122
5.3 Future Work 123
5.3.1 Statistical Analysis of the Impact of Different Measurement Protocols on being able to Predict Defects 124
5.3.2 Measurement Protocols 124
5.3.3 Reporting Protocols 124
5.3.4 The Need for Replication Studies 124
5.3.5 Ensuring Consistency in Machine Learning Approaches 125
5.3.6 Building Repositories of Comparable Data 125
5.3.7 Why not What! 125
5.4 Final Remarks 126

References **127**

Appendices **134**

A Reviewers' Comments for *[Bowes et al. 2012b]* **135**

B Additional Papers **137**
B.1 Cohesion metrics: the empirical contradiction. 137
B.2 Using program slicing data to predict code faults. 145
B.3 Program slicing-based cohesion measurement: the challenges of replicating studies using metrics. 163
B.4 Developing fault-prediction models: What the research can show industry. 171
B.5 SLuRp: a tool to help large complex systematic literature reviews deliver valid and rigorous results. 177
B.6 DConfusion: A technique to allow cross study performance evaluation of fault prediction studies. 183

List of Published Papers

Main Papers

This thesis is by publication. The following 6 papers form the main contents of my submission:

Paper 1: Bowes D, Counsell S, Hall T (2008) Calibrating program slicing metrics for practical use. Proceedings of TAIC PART, Windsor, UK

Paper 2: Bowes D, Randall D, Hall T (2013) The inconsistent measurement of message chains. In: Proceeding of the 4th International Workshop on Emerging Trends in Software Metrics, ACM (accepted paper)

Paper 3: Hall T, Bowes D, Liebchen G, Wernick P (2010a) Evaluating three approaches to extracting fault data from software change repositories. In: International Conference on Product Focused Software Development and Process Improvement (PROFES), Springer, pp 107–115

Paper 4: Hall T, Beecham S, Bowes D, Gray D, Counsell S (2012) A systematic literature review on fault prediction performance in software engineering. Software Engineering, IEEE Transactions on 38(6):1276 –1304

Paper 5: Bowes D, Hall T, Gray D (2012b) Comparing the performance of fault prediction models which report multiple performance measures: reconstructing the confusion matrix. In: Proceedings of the 8th International Conference on Predictive Models in Software Engineering. ****Best Paper in Conference Award.****

Paper 6: Hall T, Bowes D (2012) The state of machine learning methodology in software fault prediction. In: Machine Learning and Applications (ICMLA), 2012 11th International Conference on, vol 2, pp 308 –313

Additional Papers in Appendix B

The following papers are also published works and can be found in Appendix B. They are included in the dissertation because they extend the work in this thesis by either using the data from the main papers or they describe extra work as a result of the main papers.

Counsell S, Bowes D, Hall T (2009) Cohesion metrics: the empirical contradiction. In: The Psychology of Programming Interest Group, Open University

Bowes D, Hall T (2010) Using program slicing data to predict code faults. In: The 3rd CREST Open Workshop, KCL

Bowes D, Hall T, Kerr A (2011) Program slicing-based cohesion measurement: the challenges of replicating studies using metrics. In: Proceeding of the 2nd International Workshop on Emerging Trends in Software Metrics, ACM, pp 75–80

Hall T, Beecham S, Bowes D, Gray D, Counsell S (2011a) Developing fault-prediction models: What the research can show industry. Software, IEEE 28(6):96 –99

Bowes D, Hall T, Beecham S (2012a) SLuRp: a tool to help large complex systematic literature reviews deliver valid and rigorous results. In: Proceedings of the 2nd International Workshop on Evidential Assessment of Software Technologies, ACM, pp 33–36

Bowes D, Hall T, Gray D (2013) DConfusion: A technique to allow cross study performance evaluation of fault prediction studies. Automated Software Engineering Journal (In Review)

List of Figures

2.1 A Diagram Showing How Dependent and Independent Variables are used to Build and Evaluate Models . . . 9

2.2 J48 Tree for the Simple Defect Data Found in Table 2.4. The values in brackets e.g. (9.0/1.0) shows the number of items in the group and the number which are incorrectly classified. . . . 19

2.3 Layout of Perceptrons in a Fully Connected Feed Forward Artificial Neural Network. . . 19

5.1 The Improvement in Image Quality by Changing the Measurement Protocol of the Hubble Space Observatory. `http://upload.wikimedia.org/wikipedia/commons/1/12/Improvement_in_Hubble_images_after_SMM1.jpg` . . . 119

5.2 The KDD Cycle. From Fayyad et al. [1996] . . . 123

List of Tables

1.1 Research Questions 4

2.1 Demonstration Data Where Each Line is for a Method Labelled as Defective or Not with Independent Variables: Method Return Type and Method Parameter Count. 12
2.2 Conditional Standard Deviation using Return Type 14
2.3 Conditional Standard Deviation using Parameter Count 14
2.4 Table of Methods Labeled as Defective or Not Based on the Return Type and Parameter Count. 16
2.5 Summary Table of Methods Labeled as Defective or Not Based on the Return Type and Parameter Count. 16
2.6 Hyper-Parameters for Some Learners. 23
2.7 Summary Statistics for NASA Datasets before Cleaning 24
2.8 A Binary Confusion Matrix 25
2.9 Compound Performance Measures from a Binary Confusion Matrix 27
2.10 Summary of Defect Prediction Modelling Techniques. 29

Chapter 1

Introduction

1.1 Aim

Defects in code cost in excess of $50billion per year to put right in the US [Levinson 2001, Runeson and Andrews 2003]. Defect prediction is an important part of software engineering. It allows developers to prioritise the code that needs to be tested and inspected when trying to reduce the number of defects in code [Li et al. 2006]. Weyuker and Ostrand [2008] demonstrate that a small change in the number of defects found will have a significant impact on the cost of producing software.

Defect prediction techniques typically use training data to train a model which then predicts the defect proneness of a unit of code (module). Initially regression techniques were used to predict defects in code using static code metrics as the independent variables [Munson and Khoshgoftaar 1990]. More recently, Lessmann et al. [2008] demonstrate machine learning techniques being applied to predicting defects. With each new technique, the number of studies increases resulting in more than two hundred academic studies into defect prediction over the last ten years **[Paper 4]**[1].

The aim of this dissertation is to demonstrate that the results of individual defect prediction studies risk not being comparable to other studies if not enough consideration has been given to how the dependent and independent variables have been computed. This dissertation will show that even when using the same source code to provide the independent variables and defect tracking data to produce the dependent variables, the way the defect data is extracted and processed may be different, which in some cases could be affecting the prediction results. Further, this dissertation aims to demonstrate that the performance measures used to justify the significance of the results can be confusing. Some reported performance values appear to be ‘good’, even though they could have been achieved by randomly predicting modules as defective [Shepperd 2013]. Finally, this dissertation will demonstrate that machine learning good practices are not always being applied which has a significant effect on the predictive performance of the models being built. The conclusion of this dissertation is that although the performance of defect prediction seems to have currently reached the limits of being able to predict defects [Menzies et al. 2008], many retrospective updates to previous experiments are needed in order to improve the reliability of the corpus of knowledge that already exists. In this dissertation I propose new techniques which allow researchers to analyse more easily the relative merit of the predictive performance of defect prediction studies.

[1] **[Paper *n*]** refers to the *n*’th paper which is part of the main argument of this thesis. A list of the 6 main papers can be found in Chapter 4

1.2 Introduction

Defects[2] in software are a side effect of creating a piece of code. Releasing code which contains defects can be costly and may threaten the confidence that users have in a system. This is exemplified by the recent problems which the Royal Bank of Scotland suffered with a system upgrade [Scott 2012].

The cost of fixing defects varies depending on the defect itself. It has been estimated that the cost of fixing defects in the US is between 50 and 78 billion dollars per year [Levinson 2001, Runeson and Andrews 2003]. Some defects are easier to fix than others, for example typos are relatively easy to find and easy to correct. Defects involving infrequently used pieces of code which require a high level of expertise to write may be more difficult to find than typos. The system domain also affects how easy defects are to fix. Defects in embedded systems tend to be more expensive to fix than defects in desktop applications [Turhan et al. 2009]. The size of the application is also likely to affect the ease with which defects can be found [Oram and Wilson 2010]. Part of the cost in fixing a defect is in the initial step of finding the defect. Historically, defects have been found during many stages of software development processes, including development and testing. Once a piece of software has been released, users may report defects. Locating where defects are will then involve many techniques, including automated methods and manual inspection which is costly. In summary, defects will occur, may need fixing and require varying amounts of time (and hence money) to fix.

Predicting where defects occur in software is an important area for research. Studies using automated techniques to determine where defects are have been published widely **[Paper 4]**. The aim of defect prediction is to reduce the amount of code that a software engineer needs to review in order to find a defect. Early techniques relied on simple regression techniques to associate code characteristics such as the number of Lines of Code (LOC) with the number of defects found. These studies have shown that relationships exist between code features and the presence of defects and have resulted in a continuous search for better ways to predict where defects are. Recently, machine learning has become a popular way of predicting which modules of code are defect prone. There have been more than 200 defect prediction studies carried out in the last decade **[Paper 4]**. These recent (2000 to 2010) studies are dominated by machine learning techniques[3].

Menzies et al. [2008; 2010] suggest that current defect prediction techniques have reached a limit to the percentage of defective code units that can be reliably predicted. This indicates that the field of defect prediction has matured and cannot be improved any further. This dissertation argues that although the results published by previous studies may have reached a ceiling, the results themselves may not be a true reflection of the ability of learners to identify where defects are with any degree of certainty. I will confirm the conjecture made by Kitchenham et al. [1990] that there are many factors which can cause variation in the values of variables used in defect prediction studies. As a result of the variation, it may be necessary to re-work the published corpus of knowledge in defect prediction to gain a more precise understanding of the ability to reliably predict where defects are using different code characteristics.

This dissertation will demonstrate that the measurement problems identified by Fenton and Neil [1999], Kitchenham et al. [1990], Rosenberg [1997] are still current with regards to defect prediction studies:

> “manual collection of data is itself error-prone...

[2] The term ’defect’ is used interchangeably in this dissertation with the terms ’fault’ or ’bug’ to mean a static fault in software code. It does not denote a ’failure’ (i.e. the possible result of a defect occurrence while the system is executing).

[3] I will describe the difference/ similarity between regression techniques and machine learning techniques in Chapter 2.

> it is unreasonable to place much significance on relatively small effects, ... ”
> Kitchenham et al. [1990]

> “Proper measurement based studies are the key to objective methods evaluation,”
> Fenton and Neil [1999]

1.3 Thesis and Research Questions

The thesis of this dissertation is that there are many factors which may affect the predictive performance of trainable models for software defect prediction. This thesis will be demonstrated by considering the following factors:

1. The independent variables. These are measures derived from the code and may include the number of lines of code (LOC) or the number of branch points in the module.
2. The dependent variables. These are usually either the number of defects in a module or a boolean value indicating if a module is defective or not defective.
3. The datasets and machine learners. The datasets are the combination of independent and dependent variables for a piece of software. The machine learner is the technique for building a model which can predict the dependent variables based on the independent variables.
4. The accepted machine learning approaches. These include the methods of calibrating the machine learners and measuring the performance of the model.

The main arguments of this thesis can be addressed by considering the research questions in Table 1.1.

1.4 Contributions to Knowledge

This dissertation makes the following contributions to knowledge:

Theoretical contributions
This dissertation makes two major theoretical contributions. The first contribution is the demonstration that the way that some independent variables (for example program slicing metrics) are computed affects the metric value (see **[Paper 1]**, *Bowes et al. [2011]*[4] and **[Paper 2]**). This contribution means that it is necessary for studies to report how the metrics have been calculated if they are to be used in cross study comparisons. It also means that systematic literature reviews based on current work may need to exclude studies which have not reported the technique for gathering metric data. The second theoretical contribution is imputing the observation that some machine learning practices (for example feature reduction techniques or data cleaning) do not appear to be being used systematically in recent defect prediction studies **[Paper 6]**. This may be due to poor reporting and means that it is important to report the detailed protocol used when performing defect prediction experiments using machine learning.

[4] Citations in italics refer to relevant papers which I have co-authored which are found in Appendix B.

Table 1.1: Research Questions

Research Question		Motivation	Paper(s)
1	Does the measurement protocol for the independent variables affect the metric values produced?	Shepperd [1995] demonstrates that the measurement protocol for different metrics can affect the value of the metric. It is possible that a change in metric value will impact on the ability to predict defects in code.	**[Paper 1]** **[Paper 2]** *Counsell, Bowes, and Hall [2009]*
2	Is there an effective method for deriving the dependent variables for defect prediction?	Being able to predict where defects are likely to be requires us to have samples where we ‘know’ where the defects are in order to be able to assess the ability of defect prediction models to predict defects. This question addresses the problem that labelling code as defective or not may introduce noise which may hamper the ability of the learner to find patterns in the data.	**[Paper 3]**
3a	Do different learners have an impact on the performance of defect prediction models?	With each new machine learning technique, we are provided with evidence that technique z has an impact on aspect Y. Is there any evidence using published results which demonstrates that different approaches are improving the ability to predict defects? Additionally, are some datasets producing better predictive performance than others because of the language they are written in or their size?	**[Paper 4]** *Bowes and Hall [2011]* **[Paper 5]** *Bowes, Hall, and Gray [2013]*
3b	Do different datasets have an impact on prediction models?		
4	Are the results of machine learning studies reliable/trustworthy?	Machine learning and defect prediction practices have evolved over time. This should mean that modern studies include all of the ‘good’ practices and the results should be ‘better’. Is there any evidence that the ‘good’ practices which are known to improve predictive performance are being adopted?	**[Paper 6]**

Key: Papers in *italics* are included in the main body of this dissertation. The rest of the papers are included in the Appendix because they assist the argument of my thesis and I had made a contribution towards them.

Methodological contribution
This dissertation makes one major methodological contribution for generating a common set of prediction performance measures for binary classification predictions based on a wide range of measures reported in different studies (see **[Paper 5]**). This means that the performance results of current defect prediction studies can be compared even though they have not reported the same set of performance measures. It enables researchers to decide which performance measures to report in their studies if they want their results to be incorporated into a larger corpus of knowledge.

Practical contributions
This dissertation makes two minor practical contributions by providing two tools: (DConfusion see *[Bowes et al. 2013]* in Appendix B) for recomputing the confusion matrix and (SLuRp see *Bowes et al. [2012a]* in Appendix B) for carrying out a systematic literature reviews. The first tool allows other researchers to check the consistency of their own results. The first tool also allows reviewers of defect prediction research papers to validate the reported results. The second tool is a web based tool which helps to manage the SLR process. SLuRp allows the reliable recording and moderation of researchers' assessment of papers. SLuRp also integrates the tool for generating a common set of prediction performance measures extracted from the papers.

Data contribution
The secondary data collected from published defect prediction studies is reported in **[Paper 4]**. The dataset in **[Paper 4]** will allow researchers to study other factors such as the impact that research group has on the performance on defect prediction studies.

1.5 Structure of this Dissertation

The thesis of this dissertation is derived from a set of published papers which have been peer reviewed. Each paper makes a contribution to this dissertation by demonstrating either how different measurement protocols lead to variation in the metrics they produce or how variation in the machine learning approaches used impact on the predictive performance of defect prediction studies. The initial chapter gives a background to defect prediction. Later chapters describe the contributions of the published work to this dissertation.

The remaining part of this dissertation is organised as follows:

Chapter 2 gives an introduction to defect prediction and a brief summary of the current state of research in defect prediction.

Chapter 3 describes how the different published papers contribute to answering the research questions for this thesis. Chapter 3 ends with a description of my contribution to each paper. The contribution of the co-authors of each paper is also included.

Chapter 4 provides the peer reviewed published papers. Chapter 4 is composed of six sections, one for each of the main published papers.

Chapter 5 discusses the overall conclusion of the work and provides suggestions for future work.

Chapter 2

A Summary of Defect Prediction

2.1 Introduction

This chapter is a review of defect prediction. It describes what a defect is, and why we should be interested in predicting where defects are. This chapter then details the data needed to build prediction models and the process of building models and measuring model performance.

2.2 What is a Defect?

The definition of a defect is best described by using the standard IEEE definitions of $error$, $defect$ and $failure$ [IEEE 1990]. An $error$[1] is an action by a developer that results in a $defect$. A $defect$ is the manifestation of an $error$ in the code whereas a $failure$ is the incorrect behaviour of the system during execution.

In summary: $error_{developer} \rightarrow \textbf{defect} \rightarrow failure_{runtime}$

2.2.1 What is the Purpose of Defect Prediction?

The practical purpose of defect prediction is to help identify modules of code which are defective by highlighting those modules which are defect prone in order to remove defects and hopefully reduce failures[2]. When a failure is reported, programmers and/or testers may carry out manual inspection of the source code to find the defect(s). Code review is an expensive process on large systems. Weyuker and Ostrand estimate that on the large commercial systems they have studied, only 20% of a system can be effectively reviewed at a time [Oram and Wilson 2010]. The amount of code which has to be manually inspected can be reduced by using a variety of techniques. These techniques include program execution traces, source code visualisation (including program slicing) and defect prediction. Execution traces can help to identify the code running when the failure occurs and hence the code which is likely to be defective. Program slicing can reduce the code comprehension problem further by allowing programmers to see the impact of code which leads up to a variable, and the impact a variable will have on future code execution [Weiser 1981; 1979]. Defect prediction uses code features such as code complexity measures to predict areas of code which are more likely to be defect prone. The purpose of defect prediction is therefore to focus the programmer on potential hot spots in the code which are likely to contain defects.

[1] A developer *error* can also be defined as a *mistake* [IEEE 1990].

[2] The removal of a defect may not remove any failure, because the end user may not use the functionality provided by the piece of code that was 'fixed'

Correctly identifying hotspots should improve the efficiency of programmers fixing defects and therefore improve the quality of the code.

2.2.2 What is Defect Prediction?

Defect prediction is the method of predicting where defects MAY be in a piece of code. When we talk about a defect predictor we are usually talking about a set of formulas or rules which allow us to assess how prone a unit of code is to having a defect in it.

There are three main components of defect prediction: the independent variables which describe the code and what has happened to it (also known as the metrics), the dependent variable which describes the defectiveness of the code (sometimes called the label) and the model which is used to associate the independent variable to the dependent variable. The model is the part of the predictor which contains the equations and/or rules for predicting the dependent variable from the independent variables. The model is trained using historic data and an algorithm which derives the coefficients and decision rules. A function which determines how well the model fits the provided data is also required in order to be able to provide feedback to the trainer on how well the model fits the training data (see Fig 2.1). A prediction function is used to predict dependent variables when presented with a set of independent variables from a test set with known labels for the defectiveness of each item. A second error function may be used to evaluate the performance of the model which can be used to compare one model against another. This second function does not have to be the same as the first error function used during training. In summary, the learner trains a model using training data, and the performance of the model is tested by predicting the defectiveness of test data which has not been used in the training set. Keeping the training and test data separate allows us to asses the ability of the model to predict future items with which the model has not previously been presented with.

2.2.3 Code Defects

Not all defects are the same. The set of defects which result from errors during coding are a subset of all possible defects which may cause a piece of software to fail. Defects can be characterised by where they come from, for example: code, configuration settings, hardware settings etc. Basili and Selby [1987] show that some defects are harder to find than others during testing. This suggests that some defects never leave the production line and are caught before the product is released. The post-release defects should have passed many of the tests created to trap defects before release and may therefore have different characteristics to pre-release defects [Radjenović et al. 2013].

In this dissertation, I scope the definition of a defect to be: “in the source code”, because other defects are related to factors over which I had no control and were not discussed while carrying out the original studies reported in the main papers of this dissertation.

$$error_{programmer} \rightarrow \mathbf{defect'_{code}} \rightarrow failure_{runtime}$$

The scope of this dissertation is limited to defects in code which the developer can modify. I intend ignoring defects which are due to errors involving configuration files or any other settings for the software including the absence of: supporting libraries, an appropriate operating system or hardware. It may not however be possible to limit the scope of this dissertation to defects in code because much research uses data derived from software systems which is not under the control of the researcher. Catal and Diri [2009] report that 60% of defect prediction datasets are based on commercial software. Companies are

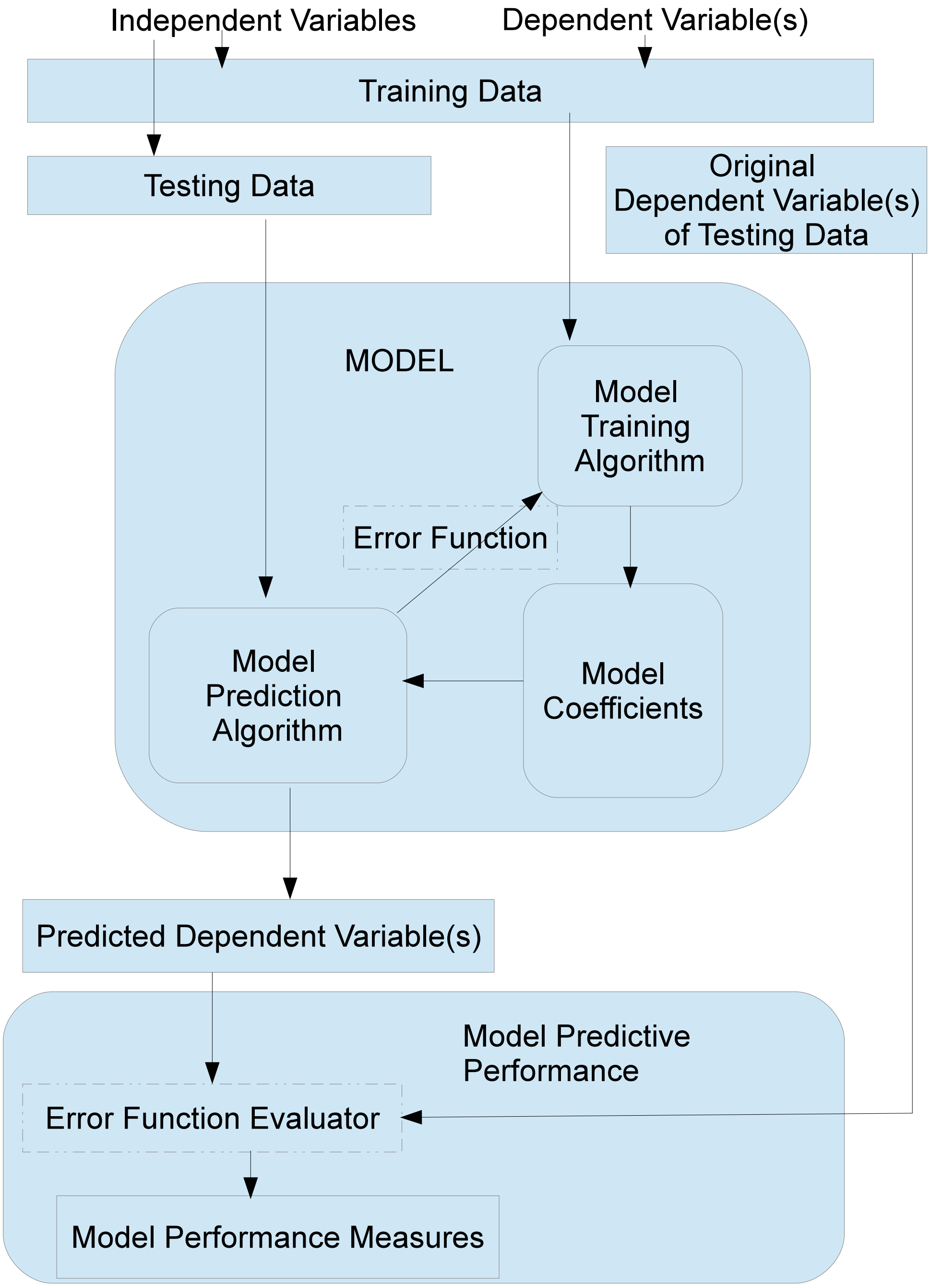

Independent Variables
Dependent Variable(s)
Training Data
Testing Data
Original Dependent Variable(s) of Testing Data
MODEL
Model Training Algorithm
Error Function
Model Prediction Algorithm
Model Coefficients
Predicted Dependent Variable(s)
Model Predictive Performance
Error Function Evaluator
Model Performance Measures

Figure 2.1: A Diagram Showing How Dependent and Independent Variables are used to Build and Evaluate Models

reluctant to release source code for reasons of confidentiality and commercial advantage. Commercial companies are however occasionally happy to provide the software metrics which have been computed from the source code on the companies premises. For example, the NASA datasets[3], currently located

[3] NASA datasets were originally available from http://mdp.ivv.nasa.gov.

in the PROMISE repository[4] , contain code metrics and the number of defects reported for closed source systems. Without the original source code or a detailed description of the measurement protocol of the closed source systems, it is not clear if the reported defects are code metrics or configuration defects. Hence it is not always possible to restrict the definition of defects to $\mathbf{defect'_{code}}$[5].

2.3 Variables Used in Defect Prediction Studies

The aim of defect prediction is to predict if a module of code is defect prone. As described earlier, the variables used to make the prediction (Lines of Code etc.) are called the independent variables and the predicted variable (defective or not, or the number of defects) is called the dependent variable. Both sets of variables need to be discussed further in order to be able to understand how they may impact on the models built and the performance of the models.

For the purpose of brevity, this dissertation will be limited to discussing variables as being either discrete or continuous. This is an oversimplification of variable types and the theory of scales of measurement which was first proposed by Stevens [1946] and applied to software metrics by Fenton and Pfleeger [1997], Fenton [1991], Kitchenham et al. [1995], Morasca and Briand [1997], Shepperd [1988] and many others. Stevens [1946] proposed four types of measurement: nominal, ordinal, interval and ratio. Nominal measurement identifies items as belonging to a group or category. Ordinal measurement identifies items as belonging to a group, each of which can be placed in an order. Interval measurement identifies items as belonging to a group, each group can be ranked and the intervals between groups are the same. Ratio measurement is similar to nominal measurement with the added restriction that there is a zero value. Nominal, ordinal and interval measures together form the discrete measures and ratio measures form the continuous measures.

The measurement of software code was intensively studied during the 1970's and the 1990's by Baker et al. [1990], Fenton and Pfleeger [1997], Fenton [1991], Halstead [1977], Kitchenham et al. [1995], McCabe [1976], Morasca and Briand [1997], Shepperd [1988] and others. Baker et al. [1990] summarises the debate about software measurement and proposed a distinction between software measures and software metrics: metrics map the computed number to an ordering of the units, whereas measures result in values which are understandable and interpretable and consistent across different measurement protocols for the same measure. Baker et al. [1990] therefore allow length of a program to be considered as a measure because we have an intuitive understanding of what it means, and the length of different programs written in the same language can be compared and ordered. In this dissertation, I use the term metric rather than measure because the things being computed do not have a well defined common meaning. For example, in **[Paper 2]**, we study how 'code bad smells' are identified and show that there is not a common understanding of what is meant by some of the common code bad smells.

[4] PROMISE is a well know repository of data used for defect prediction studies. The PROMISE repository [Menzies et al. 2012] `http://code.google.com/p/promisedata/` contains defect data for 40 projectsAs of 23/08/2012. Previous versions of the repository can be found using Boetticher et al. [2007], Shirabad and Menzies [2005].

[5] In this dissertation, I have been able to use the restricted definition ($\mathbf{defect'_{code}}$) in **[Paper 3]** because the project being studied was open source. **[Paper 4]** and **[Paper 6]** use the less restrictive definition because they are secondary studies.

2.3.1 Independent Variables

Independent variables are the set of variables used as the input for a model to make a prediction. They can be as simple as the number of Lines of Code, or as complicated as program slicing metrics such as tightness[6] [Weiser 1979]. The independent variables are not confined to those which can be computed directly from the source code (sometimes referred to as static code metrics). Nagappan and Ball [2005] use a feature of software development called 'churn' which describes the amount of change in the code over time. Pinzger et al. [2008] describe how metrics based on the network of developers can be used to successfully predict defects. Many different independent variables have been used in defect prediction studies which are summarised in **[Paper 4]** and explained in detail in [Fenton and Pfleeger 1997].

2.3.2 Dependent Variables

The dependent variable, i.e. 'defectiveness' of the code being analysed, should be the output of defect prediction models. The variable can either be continuous (the 'number of defects' is frequently used by Weyuker et al. [2010]) or discrete (not defect prone or defect prone[7]). It is possible to convert a continuous variable into a discrete variable by splitting the data into ranges. For example, Menzies et al. [2007] binarises the NASA datasets into two ranges by labelling a module as not defect prone if the number of defects reported is zero otherwise the module is labelled as defect prone.

The initial labelling of the defectiveness of a module of code is perhaps the most difficult part of acquiring defect prediction data. There are many reasons which lead to defects being discovered in code including: manual inspection of code, failure of a test, failure report in a defect tracking system. Code inspection and test failure tend to happen pre-release and are dependent on developer effort. Identification of a defect due to a failure report happens post-release by users. The number of post-release defects will depend on how long the product has been released[8] and the number and diversity of users using the software. Latent defects will remain in the source code as long as either the failure is not reported or the code is not executed.

2.4 Modelling Techniques

The modelling technique is the method of building a model which associates the independent variable with the dependent variable. This allows a prediction of the unknown dependent variable of a new instance to be made. There are many different techniques available. Some techniques can deal with continuous data such as regression techniques and others can deal with categorical data such as Naïve Bayes . This next section describes a subset of techniques used in defect prediction studies. The subset of techniques presented here have been selected because they were used in the 36 studies identified in **[Paper 4]**.

[6] Program Slicing metrics are defined later in **[Paper 1]**.

[7] Other discretisation systems have been employed, for example Khoshgoftaar et al. [2005] uses a three way classification of {not defect prone | low defect prone | high defect prone}

[8] For consistency, Weyuker counts the number of defects found due to failures reported in the first two months.

2.4.1 Continuous Techniques

There are a number of techniques which can predict the number of defects. Predicting the number of defects allows the developer to rank the modules by those which have the potential to have the most defects. Weyuker reports the Pareto effect [Newman 2005] is common with 20% of files containing 80% of defects [Oram and Wilson 2010, Chap. 9]. Weyuker also reports that inspecting 20% of the code is achievable in large systems. Therefore, as a software engineering exercise, ranking the modules by the number of defects and inspecting the top 20% is both practically achievable and likely to remove the greatest number of defects.

For the rest of the discussion on modelling techniques I will use a dummy data set to explain some of the techniques. The data is not real and has been constructed for demonstration purposes only. The data is based on a hypothetical analysis of some JAVA code where we are trying to predict the defect proneness of code by looking at the return type of a method and the number of parameters in the method signature. The data is presented in Table 2.1.

Table 2.1: Demonstration Data Where Each Line is for a Method Labelled as Defective or Not with Independent Variables: Method Return Type and Method Parameter Count.

RowId	Return Type	Parameter Count	Defect Count
1	void	1	0
2	void	2	0
3	void	1	0
4	void	2	0
5	void	2	0
6	void	2	0
7	void	4	0
8	void	2	0
9	int	0	0
10	boolean	1	0
11	void	1	2
12	int	4	4
13	ArrayList	8	8
14	JFrame	0	100

2.4.1.1 Regression

Regression techniques are based on a single mathematical equation. The equation has the dependent variable on one side and the independent variables on the other. A simple example of a regression model would be:

$$D = c_0 + c_1 \times PC \tag{2.1}$$

Where D is the predicted number of defects, PC is the number of parameters passed to the method. c_0 and c_1 are constants and can be estimated using:

$$c_0 = \frac{(\sum D)(\sum PC^2) - (\sum PC)(\sum D \times PC)}{n(\sum PC^2) - (\sum PC)^2} \tag{2.2}$$

$$c_1 = \frac{n(\sum D \times PC) - (\sum PC)(\sum D)}{n(\sum PC^2) - (\sum PC)^2} \tag{2.3}$$

Equations 2.2 and 2.3 produce values of c_0 and c_1 which minimise the following error function

$$SumsOfSquares = \sum (PC - D)^2 \tag{2.4}$$

Multiple regression involves variables being added to the right of Equation 2.5 e.g.:

$$D = c_0 + c_1 \times PC + c_2 \times r \tag{2.5}$$

Where r is 1 if the return type is void and 0 if the return type is not void. The constants can be calibrated to reduce the error by reformulating the above equation for each instance i of the training data:

$$D_i = c_0 + c_1 \times PC_i + c_2 \times r_i + \varepsilon_i \tag{2.6}$$

Where ε is an error term for the particular instance. When $i > v$ (v=the number of variables), it becomes possible to determine the coefficients $r_0 ... r_v$ by solving the set of simultaneous equations.

Negative Binomial Regression (NBR) is an especially useful regression technique for studies which use the number of defects per module as the dependent variable [Oram and Wilson 2010, Weyuker et al. 2010]. NBR is based on linear regression but uses a log transformation of the dependent variable, i.e. the log of the number of defects against the linear form from above, for example Equation 2.7.

$$\log_e(D) = c_0 + c_1 \times PC + c_2 \times r \tag{2.7}$$

2.4.1.2 Regression Decision Trees

Regression decision tree techniques are based on a simple tree model. A decision tree is composed of nodes which contain decision logic. The decision may be made on a single attribute of the independent data for example $LOC > 4$ which splits the data into different ranges. For each range the node will either lead to another node or to a final decision for example 5 defects. A tree is used by presenting the independent variables to the starting root node which then makes a decision. The attributes are passed down to other nodes until a final decision is made.

Choosing the variable to split the data on is done by observing the standard deviation of the dependent variable before the split and comparing it to the combined standard deviations of the data after the split using one of the independent variables.

Equation 2.8 is the usual equation for calculating the standard deviation of the original population:

$$S(x) = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2} \tag{2.8}$$

Where

$$\mu = \frac{1}{N}\sum_{i=1}^{N} x_i \tag{2.9}$$

The combined standard deviation of the data after a potential split is calculated as follows:

$$S(T, X) = \sum_{c \in X} P(c) S(c) \tag{2.10}$$

Using the data from Table 2.1. $S = 25.5730$

If we split on the return type we get a standard deviation of 0.6898 (Table 2.2).

Table 2.2: Conditional Standard Deviation using Return Type

Return Type	Probability	Std.Dev.	Combined
void	9/14	0.6285	0.4041
int	2/14	2.0000	0.2857
boolean	1/14	0.0000	0.0000
ArrayList	1/14	0.0000	0.0000
JFrame	1/14	0.0000	0.0000
		Total	0.6898

Splitting the data on the number of Parameters>0, we get a standard deviation of 9.1783 (Table 2.3).

Table 2.3: Conditional Standard Deviation using Parameter Count

Parameter Count	Probability	Std.Dev.	Combined
0	2/14	50.0000	7.1429
> 0	12/14	2.3746	2.0354
		Total	9.1783

This shows that the standard deviation after splitting on *Returntype* is less than the standard deviation after splitting on *Parametercount*. The lower standard deviation for *Returntype* indicates that it would be better to split on *Returntype* than *Parametercount* because the items in the groups formed by splitting on *Returntype* are more alike than items found in groups by splitting on *Parametercount*. When the data can no longer be split, the average value of the dependent variable is reported.

2.4.2 Categorical Techniques

Sometimes we do not know the number of defects, we only know if the module is defective or not defective. Fortunately categorical techniques can predict variables with values on the nominal scale. In the case of defect prediction, this would be either not defect prone or defect prone. Many techniques exist for solving this problem which are detailed below.

2.4.2.1 Statistical Methods

Logistic regression is a technique which produces a probability that the output is either 0 or 1. It uses a non linear equation based on the sigmoid function (see Equation 2.11).

$$g(z) = \frac{1}{1 + e^{-z}} \tag{2.11}$$

The probability that the output is 1 is given by:

$$Pr[y = 1|x; \theta^T] = \frac{1}{1 + e^{-f(x,\theta^T)}} \tag{2.12}$$

Where $f(x, \theta^T)$ is an expression such as $\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3$ and θ^T is a vector of constants and x is a vector of variables with $x_0 = 1$ (i.e. the intercept).

θ^T is found by repeatedly updating the coefficients using Equation 2.13 until the change in error after updating the coefficients is less than a pre-determined threshold value.

$$\theta'_j := \theta_j + \alpha \sum_{i=1}^{m} (f(X^{(i)}, \theta^T) - Y^{(i)}) X_j^{(i)} \tag{2.13}$$

For clarity:
X is a vector of vectors of independent variables.
θ_j is the j'th value of θ.
α is a constant which defines the learning rate.
m is the number of X vectors.
i is a variable to indicate which of the m vectors of X is being used.
Y is the label for the i'th vector.

Naïve Bayes is a statistical technique based in the probability theories of Thomas Bayes. It uses the following probability equation:

$$Pr[H|E] = \frac{Pr[E|H]Pr[H]}{Pr[E]} \tag{2.14}$$

Where $Pr[H|E]$ means the probability of class H based on the probability of classes E.

Naïve Bayes assumes that the independent variables are independent of each other (which is rarely the case) and that the presence of each independent variable has a different contribution to the probability of each dependent category (from now on called the class[9]). A simple example will help at this point. Consider the problem of predicting if a module is defective. We will use the data in Table 2.1 which is further summarised Table 2.4 and Table 2.5. Note that Naïve Bayes can only deal with categorical variables, therefore the count of the number of parameters has been binarised with a threshold of 3 or more, and defectiveness has been binarised with a threshold of 1 or more indicating defectiveness.

If we have a new method with a return type of `void` and 5 parameters in the method call we do not know $Pr[E]$. We can however compute the probability of the method being defective using Equations 2.15 and 2.16 resulting in Equations 2.17 and 2.18:

[9] Class and label are used interchangeably in this dissertation.

Table 2.4: Table of Methods Labeled as Defective or Not Based on the Return Type and Parameter Count.

RowId	Defective	Return void	Parameter count > 2
1	N	Y	N
2	N	Y	N
3	N	Y	N
4	N	Y	N
5	N	Y	N
6	N	Y	N
7	N	Y	Y
8	N	Y	N
9	N	N	N
10	N	N	N
11	Y	Y	N
12	Y	N	Y
13	Y	N	Y
14	Y	N	N

Table 2.5: Summary Table of Methods Labeled as Defective or Not Based on the Return Type and Parameter Count.

		Return void		Parameter count > 2		
		Y	N	N	Y	
Defective	N	8	2	9	1	10
	Y	1	3	2	2	4
		9	5	11	3	14

$$likelihoodofNOTDefective = \frac{8}{10} \times \frac{1}{10} \times \frac{10}{14} = 0.03571 \tag{2.15}$$

$$likelihoodofDefective = \frac{1}{4} \times \frac{2}{4} \times \frac{4}{14} = 0.05714 \tag{2.16}$$

$$ProbabilityofNOTDefective = \frac{0.03571}{0.03571 + 0.05714} = 0.38462 \approx 38\% \tag{2.17}$$

$$ProbabilityofDefective = \frac{0.05714}{0.03571 + 0.05714} = 0.61538 \approx 62\% \tag{2.18}$$

Showing that the probability of a method with a void return and more than two parameters being defective is 62%. On the balance of probability, we will predict this method as being defective. Naïve Bayes does give us some idea of the certainty that the method will be defective which is lost in the final classification based on the balance of probability. Naïve Bayes uses categorical data for all variables which is incompatible with LOC for example. This incompatibility is overcome by discretising none categorical data.

2.4.2.2 Classification Decision Trees

Classification decision tree techniques are another example of a tree model. Classification decision trees use InfoGain [Quinlan 1993] or Gini [Breiman et al. 1984] as the measure of how much a split is better when using a particular variable to split the data on. There are many implementations of decision classification decision tree techniques such as c4.5, J48, and rpart [Breiman et al. 1984, Quinlan 1993, Witten and Frank 2002].

For the rest of this section we will consider c4.5 as our example. c4.5 is a tree technique developed by Quinlan [1993]. c4.5 uses the measure called information entropy to decide which variable should be used at a node to divide up the data. The variable which results in the highest InfoGain is used at a node to split the data. InfoGain is the difference between the entropy of the original data and the conditional entropy of the data using a particular variable. Information entropy is a measure of how uniformly distributed different classes are in a population and was first proposed by Shannon and Weaver [1948]. For example, if we have a dataset with 10 instances and 5 are defective and 5 are not defective, the entropy is 1. If all of the values are the same, entropy is 0. The formula for entropy is:

$$H(X) = -\sum_{j=1}^{n} p_j log_2(p_j) \tag{2.19}$$

Consider the example data in Table 2.4, there are 10 none defective items and 4 defective items. The entropy of this data is:

$$H(Defective) = -\frac{10}{14}log_2\left(\frac{10}{14}\right) - \frac{4}{14}log_2\left(\frac{4}{14}\right) = 0.3467 + 0.5164 = 0.8631 \tag{2.20}$$

If we decide to split the data into two groups using the Return is void variable, we get:

$$H(Defective|ReturnIsVoid) = -\frac{8}{9}log_2\left(\frac{8}{9}\right) - \frac{1}{9}log_2\left(\frac{1}{9}\right) = 0.1510 + 0.3522 = 0.5033 \tag{2.21}$$

$$H(Defective|ReturnIsNotVoid) = -\frac{2}{5}log_2\left(\frac{2}{5}\right) - \frac{3}{5}log_2\left(\frac{3}{5}\right) = 0.5288 + 0.4422 = 0.9710 \tag{2.22}$$

The conditional entropy is the combined entropy of the two groups formed by splitting the data using the Return type variable. A low conditional entropy indicates that the groups are uniform, i.e. the items in the group are similar to each other. For the Return type, the conditional entropy is given by:

$$H(Defective|ReturnType) = 0.5033\left(\frac{9}{9+5}\right) + 0.9710\left(\frac{5}{9+5}\right) = 0.6703 \tag{2.23}$$

If we consider splitting the data using the Parameter count variable we get:

$$H(Defective|ParameterCount \leq 2) = -\frac{9}{11}log_2\left(\frac{9}{11}\right) - \frac{2}{11}log_2\left(\frac{2}{11}\right) = 0.2369 + 0.4472 = 0.6840 \tag{2.24}$$

$$H(Defective|ParameterCount > 2) = -\frac{1}{3}log_2\left(\frac{1}{3}\right) - \frac{2}{3}log_2\left(\frac{2}{3}\right) = 0.5283 + 0.3900 = 0.9183 \quad (2.25)$$

The conditional entropy using Parameter count variable is given by:

$$H(Defective|ParameterCount) = 0.6840\left(\frac{11}{14}\right) + 0.9183\left(\frac{3}{14}\right) = 0.7342 \quad (2.26)$$

InfoGain is a measure of how much information has been gained by splitting the data using a variable. InfoGain is the difference between the original entropy (Equation 2.20) and the conditional entropy for the variable (for example Equation 2.26).

$$InfoGain(ReturnType) = 0.8631 - 0.6703 = 0.1928 \quad (2.27)$$

The InfoGain using Parameter count variable is given by:

$$InfoGain(ParameterCount) = 0.8631 - 0.7342 = 0.1289 \quad (2.28)$$

This suggests that the best variable to split the data on is the Return type = `void` decision. Having split the data, the above technique is applied to both sub groups until the entropy of the group is 0 or some other stopping condition is met. This repetitive splitting is called recursive partitioning.

A c4.5 learner can also try to simplify the tree produced using a technique called pruning, so that there are fewer branches or nodes with single items. c4.5 can use continuous independent variables by determining a threshold value of the variable to split on. J48 is a JAVA implementation of c4.5. J48 produces the tree seen in Fig. 2.2 for the data found in Table 2.4.

Random Forest^{TM} is an ensemble technique which uses a collection of decision trees [Breiman 2001]. The dataset for each tree is created using a technique called bagging. Bagging involves randomly selecting a subset of independent variables from the original data set and then randomly selecting instances from the dataset with replacement. This creates a variety of models which perform differently for an item of data drawn from the original dataset. When a new item of data is presented to the Random Forest model, the predictions for each tree are aggregated together to form a collective decision. Random Forest techniques have been shown to work well for defect prediction studies [Lessmann et al. 2008].

2.4.2.3 Other Techniques

Artificial Neural Networks (ANN) are a technique which use parallel decision units which take input data and produce an output which may form the input of other decision units. A typical ANN uses a unit called a perceptron as the decision unit. The input to a perceptron is multiplied by a coefficient called a weight. The output of a perceptron is a function of the sum of the weighted inputs. An ANN is trained by repeatedly presenting input data to the ANN and then adjusting the weights using a method called back propagation. This is similar to the updating of coefficients described for logistic regression. Different arrangements of networks exist, the most common is a feed forward network with three layers: the input layer, the hidden layer and the output layer (see Fig. 2.3). It should be noted that the number of perceptrons in the hidden layer can be altered. Altering the size of the hidden layer has been observed to alter the ability of the network to predict the required outputs [Hertz et al. 1991]. The learning rate, the momentum of learning and the number of times the data is presented to the network for learning (epochs) are also known to have an impact on the ability to predict new items [Salehi et al. 1998].

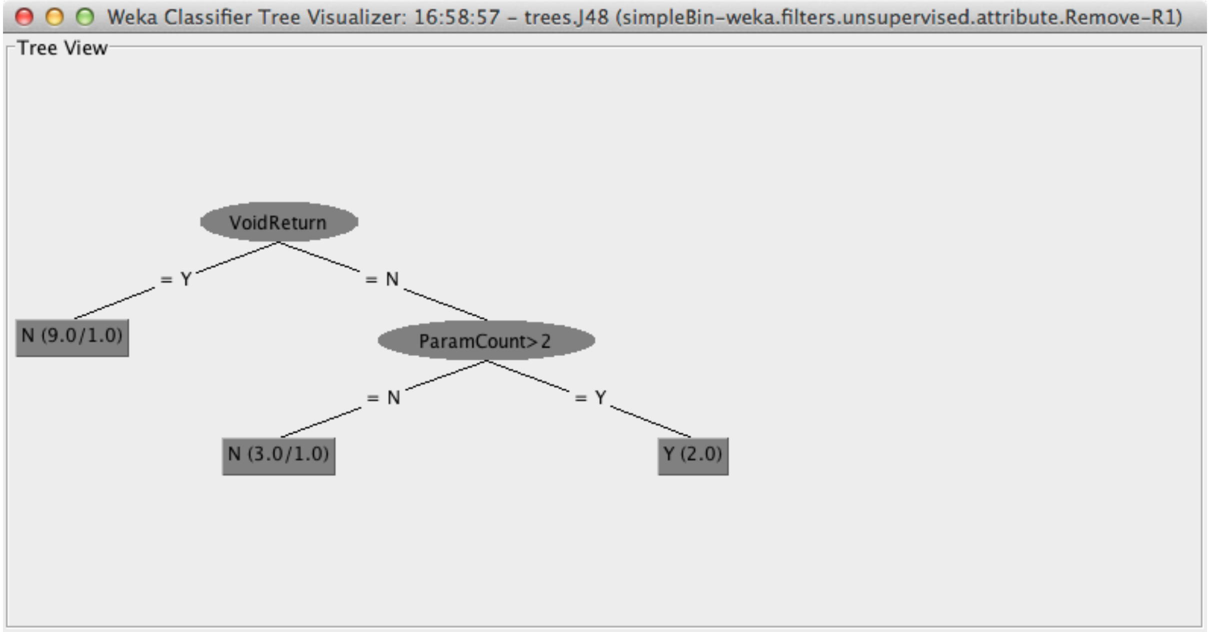

Weka Classifier Tree Visualizer: 16:58:57 - trees.J48 (simpleBin-weka.filters.unsupervised.attribute.Remove-R1)
Tree View
VoidReturn
= Y
= N
N (9.0/1.0)
ParamCount>2
= N
= Y
N (3.0/1.0)
Y (2.0)

Figure 2.2: J48 Tree for the Simple Defect Data Found in Table 2.4.
The values in brackets e.g. (9.0/1.0) shows the number of items in the group and the number which are incorrectly classified.

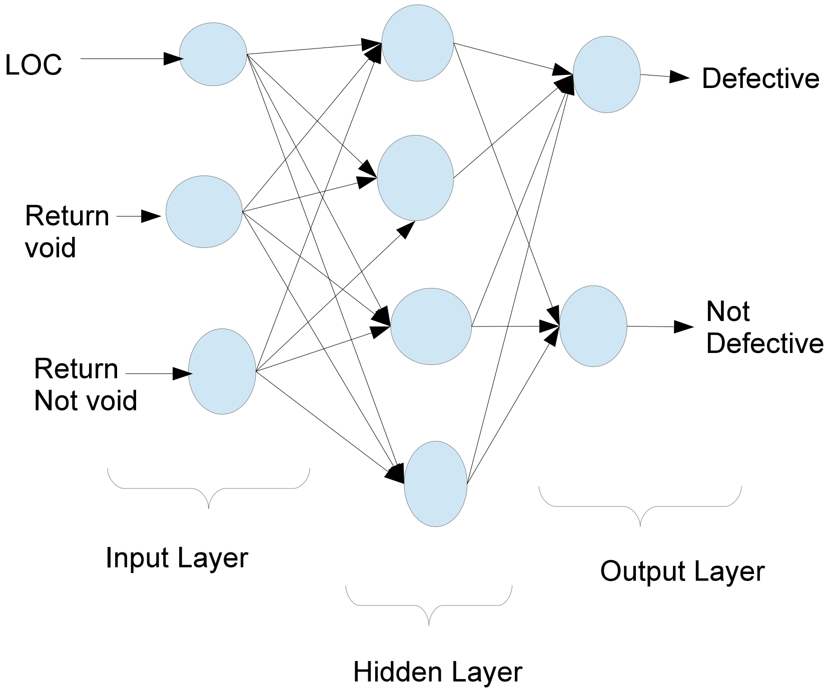


Figure 2.3: Layout of Perceptrons in a Fully Connected Feed Forward Artificial Neural Network.

Support Vector Machines (SVM) were developed by Vapnik [1963]. SVMs identify a separating plane which maximises the 'distance' between the plane and the vectors which are closest to it for linear separation problems. Boser et al. [1992] introduced the kernel trick which extended the capability of SVMs to allow solutions for non-linear problems. The kernel trick uses different functions to map the original data to a higher number of dimensions which when projected onto the original space form non linear surfaces. Such tricks usually require hype-parameters, for example the radial base function has a γ coefficient. Cortes and Vapnik [1995] extended the technique to allow the misclassification of some items by adding a cost coefficient C. High values of C result in fewer training items being misclassified by building a more complex model. SVMs work best when the coefficients for example C and γ (for radial base kernel function) are optimised. Optimisation may involve a grid search which builds models using a grid of C and γ values. Tabu searches [Glover 1989; 1990, Glover and McMillan 1986] have been used with success recently to find the optimal coefficients for SVM coefficients [Corazza et al. 2010].

K-Nearest Neighbour (kNN) is a simple technique which takes the average label for the k nearest neighbours. For continuous independent data the Euclidian distance is used to measure the distance from the test item to the possible nearest neighbours. For categorical independent variables, the Hamming distance can be used as a measure of distance. Producing a good kNN model requires finding the best value of k which can be achieved by increasing the value of k from 1 to some maximum value.

2.5 Data Quality and Data Cleaning

Witten and Frank [2005] states that machine learning should be tolerant of noise in the data. Some learners such as Naïve Bayes can make predictions in the presence of noise. It is possible to use this argument that learners can cope with noise to justify ignoring the need for data cleaning. Recently, Gray et al. [2012] identified noise in the NASA datasets which could not have been computed from the raw data (for example 1.1 LOC) and should have been removed before being used in defect prediction studies. NASA data has been used by over 30 different studies, with only one other study [Boetticher 2006] making any comment about the incorrect data items. Data cleaning and simply eye-balling the data appears to have become a forgotten part of the researchers' protocol. Gray et al. [2012] describes a sensible approach to cleaning the static code metrics of datasets used for defect prediction.

2.6 Model Building Approaches

This section describes the approaches used to build models which will work on data which has not been used to build the model. Model building approaches involve creating training and test data, as well as dealing with issues such as data quality.

2.6.1 Building Generalisable Models

Building a model which can correctly label the training set is important, however good models should have additional characteristics. Fenton and Neil [1999] describes the two aims of building prediction models as 1) building models which will correctly predict 'new' data (generalisability) and 2) building models which are understandable (comprehensibility). Some models are able to generalise the prediction problem to correctly predicting modules they have never seen before. Building models which can generalise well will depend on what data is used to train the model and what error function is used to determine how well the model performs and how the model is built [Gray 2013, Witten and Frank 2005]. Different leaners produce models of different comprehensibility. Decision trees are simple to interpret and SVMs are difficult[10].

Feature reduction is sometimes applied to the independent variables to simplify the learning process, to remove highly correlated data and to build more generalisable models. Menzies et al. [2007] demonstrated that it is possible to achieve relatively good performance on a highly reduced set of independent variables. Feature reduction can be achieved by removing attributes which have a low entropy (see Equation 2.19) and/or a high correlation with other independent variables.

2.6.2 Testing the Generalisability of Models

Testing the generalisability and hence the validity of a model involves testing it on some ‘new’ data. If the model can continue to perform well on ‘new’ data it can be considered to have been able to generalise the problem.

In software engineering, the new data could come from the next release of a software product [Weyuker et al. 2010]. Using the next release to check the model is called cross-release validation.

Some researchers also assess the performance of their models on data from different system because they want to know if a model is transferable to a wider range of environments. Testing models on different systems is called cross project validation and in defect prediction rarely works well [Turhan et al. 2009, Zimmermann et al. 2009].

The availability of data for performing cross project validation is has increased with the availability of online repositories. The PROMISE repository has made available defect data from many different systems which should allow cross project validation. Some datasets, for example, those for the Eclipse JAVA IDE development tool are available for different releases allowing cross-release validation to take place.

In the absence of data for different systems and different releases, it is possible to perform model validation by splitting the entire dataset into folds (sets of instances) and keeping one fold as the test set and the rest as the training set. This splitting of the data into folds in order to produce training and testing sets is called n-fold cross validation (where n is the number of folds). Problems can occur when forming the folds with highly imbalanced data. It is possible that some folds may not have any instances of the minority class. Having no instances of a particular class causes problems when measuring the performance of a model because the divisor of some measures is the total number of that class, hence the performance measure has a divide by zero problem (see Table 2.9 for the formula for different performance measures). To overcome this, the folds should have roughly the same distribution of each class as the original population, i.e the folds are stratified.

2.7 Issues of Model Building Specific to Machine Learning

Machine learning techniques are affected by both internal settings and the charactersistic of the data they are being trained on. The major issues are discussed below.

[10] The use of Support Vector Inductive Logic Programming combines SVMs with a technique called inductive logic programming which allows the patterns within the SVM model to be interpreted.

2.7.1 Dealing with Datasets which are Unbalanced

Most datasets used by defect prediction studies have few instances of the defective class in them (i.e. they are unbalanced). We would hope that this would be the case since defects are normally a bad thing. If we look at the NASA datasets in Table 2.7 we can see that the percentage of defective instances ranges from 0.4% in PC2, to 48.8% for KC4. The imbalance of most datasets changes after cleaning (see Table 2.7). Unfortunately the lack of instances which are labelled as defective does not help the learner. The learner c4.5 is thought to struggle with imbalanced data ([Japkowicz and Stephen 2002] and [Liu et al. 2010]) and this may explain the poor performance of models found in Arisholm et al. [2007; 2010]. For techniques which are sensitive to imbalanced datasets, there are two ways of making the training set[11] more balanced. These include under-sampling the majority class or oversampling the minority class. Under-sampling involves selecting all items from the minority class and then randomly picking the same number of items from the majority class. Over-sampling involves synthetically creating extra examples from the minority class. SMOTE is an over-sampling technique which creates new items from the minority class by taking two 'close' items a third synthetic item with independent variable values which lie between the independent variables of the two items [Chawla et al. 2002].

2.7.2 Machine Learning Model Optimisation

Model optimisation is the process of finding the set of parameters for a learner which achieves a good predictive performance on the training set (for example the cost coefficient of SVMs needs to be optimised for different datasets Gray et al. [2010]) . During model optimisation, the algorithm which trains the model has to 'know' what is good performance and what is bad performance. Model optimisation algorithms search for models which minimise an error function such as Mean Magnitude of Relative Error (Equation 2.30) for continuous dependent variables or Accuracy (Equation 2.31) for categorical dependent variables. Different error functions can be used depending on the purpose of the model. Some error functions are not always appropriate. Miyazaki et al. [1994] show that MMRE is lower for models which underestimate, and Foss et al. [2003] show that MMRE can lead to counterintuitive examples. Lokan [2005] studied different error functions and concluded that functions which minimise actual error perform better than functions which minimise relative error. Port and Korte [2008] show that MMRE (see Equation 2.30) is more sensitive to outliers than PRED (see Equation 2.32) as a measure of error. Accuracy (see Equation 2.31) should not be used as an optimisation function with imbalanced data because good values of accuracy can be achieved by simply predicting the majority class. A model optimised for Recall can be produced by making every prediction the preferred class (e.g. defective). Recent work by Shepperd [2013][12] would recommend using Mathews Correlation Coefficient (MCC see Table 2.9) as a measure of model accuracy because it is less biased to measuring the predictive capability on a particular class. Rezwan et al. [2013] also show that a binary classifier achieves better results when optimised on the same performance measure that will be used for the final performance measure.

$$MRE = \frac{|y_i - prediction(x_i)|}{y_i} \tag{2.29}$$

[11] Only the training set needs to be adjusted, because the performance of the final model should be representative of the original data.

[12] The recent work by Shepperd [2013] is based on the data we have extracted from **[Paper 4]**. Both Myself and Tracy Hall have worked with Martin to formulate the usefulness of MCC.

Technique	Coefficients to Tune
Naïve Bayes	Can be used without setting any initial coefficients.
Random Forest	Depth of Tree and min nodes per leaf
SVM	C and γ
kNN	k the number of nearest neighbours
Neural Networks	The number of nodes in each hidden layer, the number of hidden layers, the level of connectivity between layers.

Table 2.6: Hyper-Parameters for Some Learners.

$$MMRE = \frac{1}{N}\sum_{i=1}^{N}\left(\frac{|y_i - prediction(x_i)|}{y_i}\right) \tag{2.30}$$

$$Accuracy = \frac{itemscorrect}{N} \tag{2.31}$$

2.7.3 Model Tuning

In order to produce good performance, some learners need tuning. Some learners (SVMs, RandomForest) have hyper-parameters which can be altered and will affect the ability to effectively learn a particular data set. Tuning involves searching for the best hyper-parameters either in a systematic manner for example a grid search or some other way such as a Tabu search [Corazza et al. 2010]. During the search, the training data is split into folds. Each fold is then used in turn as a validation fold, with the other folds being used to train a learner with a set of hyper-parameters. The performance on each validation set is then measured and averaged for all validation folds. The hyper-parameters which produce the highest average performance (or least error) are then chosen to build a model on the entire training set. Algorithm 1 is taken from Rezwan et al. [2013] and succinctly describes a typical tuning setup for a SVM. Note that in the algorithm, data items which have the same values for the independent variables but different labels for the dependent variable[13] are removed because they hinder SVMs in their ability to produce a suitable separating hyperplane. Table 2.6 describes some of the meta-parameters which can be altered for different learners.

2.7.4 Reordering the Training Data

Some learners will produce different results with the same dataset because the order in which the items are presented to the learner will affect the model. Witten and Frank [2005] report that the clusters produced for the famous Iris (plants) dataset are highly dependent on the ordering of the items presented to the learner. It is therefore common for the training set to be reordered after creating the training and testing sets from the different folds (as seen in Algorithm 1).

[13] Due to mislabelling or the absence of information in the metrics which explains the difference in labels.

```
Algorithm

Remove all inconsistent and repeated data points;
Split the data into two-third training from the new data set, one-third test from the original data set.
Split the training data into 5 partitions;
This gives 5 different training (four-fifth) and validation (one-fifth) sets. The validation set is drawn
from the original data set;
Use sampling to produce more training sets with equal numbers of each class;
for each pair of $C/\gamma$ SVM hyper-parameters do
    for each of the 5 training sets do
        Train an SVM;
        Measure performance on the corresponding validation set, exactly as the final test will be
        measured. So use the Performance Measure, after the predictions on the validation set have
        been filtered;
    end
    Average the Performance Measure over the 5 trials;
end
Choose the hyper-parameter $C/\gamma$ pair with the best average Performance Measure;
Pre-process the complete training set and train an SVM with the best $C/\gamma$ combination Test the trained
model on the unseen test set;
Post-processing the final prediction ;
```

Algorithm 1: Finding the best hyper-parameters with modified cross-validation method taken from [Rezwan et al. 2013].

Table 2.7: Summary Statistics for NASA Datasets before Cleaning

Dataset	Language	Total KLOC	No. of Modules (pre-cleaning)	No. of Modules (post-cleaning)	%Loss Due to Cleaning	%Faulty Modules (pre-cleaning)	%Faulty Modules (post-cleaning)
CM1	C	20	505	503	0.4	9.5	9.3
KC1	C++	43	2109	1843	12.6	15.4	16.3
KC3	Java	18	458	456	0.4	9.4	9.2
KC4	Perl	25	125	116	7.2	48.8	50.0
MC1	C & C++	63	9466	9171	3.1	0.7	0.4
MC2	C	6	161	159	1.2	32.3	32.1
MW1	C	8	403	396	1.7	7.7	7.1
PC1	C	40	1107	1094	1.2	6.9	6.4
PC2	C	26	5589	5360	4.1	0.4	0.4
PC3	C	40	1563	1554	0.6	10.2	10.0
PC4	C	36	1458	1396	4.3	12.2	12.6
PC5	C++	164	17186	15278	11.1	3.0	3.1

2.8 Measuring the Performance of a Model

Having produced a model on a dataset, we would like to know how it performs on ‘new’ data. Cross-validation described earlier provides a mechanism for providing the ‘new’ test sample, but does not give us an indication of the performance. Measuring the performance of a learner on the test set is an important aspect of defect prediction because it gives us some idea of how good a model/technique is on a particular dataset. The performance measures which have commonly been used in the past to measure model performance can be categorised into two groups based on whether the dependent variable is continuous or categorical.

Continuous performance measures are usually based on the difference between the value of the prediction and the original label. MMRE described earlier (Equation 2.30), measures the relative error on values from a ratio scale. Foss et al. [2003] suggests other measures such as PRED(N) are also affected, they recommend using residual sum of squares as a less biased performance measure.

$$PRED(n) = \frac{1}{T}\sum_{i=1}^{T}\begin{cases}1 & \text{, if } MRE_i \leq \frac{n}{100};\\ 0 & \text{, otherwise;}\end{cases} \tag{2.32}$$

2.8.1 Measuring the Performance of Categorical Prediction Models

Categorical performance measures describe the level of agreement between the labelled value and the predicted value. Unlike with continuous variables, there is either a match between what was expected or not which can be best described in a confusion matrix. For binary classification, a confusion matrix is composed of a two by two grid with the following labels : TP, TN, FP and FN.

True Positives (TP) is the number of items where the actual label is the true class (defective) which is the same as the prediction.

True Negatives (TN) is the number of items where the actual label is the false class (not defective) which is the same as the prediction.

False Positives (FP) is the number of items where the actual label is the false class and the model predicted the positive class.

False Negatives (FN) is the number of items where the actual label is the true class and the model predicted the negative class.

Table 2.8 summarises the confusion matrix for binary classification.

Table 2.8: A Binary Confusion Matrix

	observed true	observed false
predicted true	TP	FP
predicted false	FN	TN

From the confusion matrix, it then becomes possible to compute a range of compound performance measures which are described in Table 2.9. Some of the frequently used compound performance measures will be briefly described in relation to defect prediction. A note of caution is required at this point because computing the confusion matrix can be achieved in different ways during cross-validation. Forman and Scholz [2010] describes how some tools compute the compound performance measure for each fold and then average the performance measures, compared to adding up all of the instances into a single confusion matrix and computing a single compound performance measure from the final confusion matrix.

Some common performance measures are described below:

- Accuracy is the ratio of the correctly predicted items compared to the total number of items.
- Recall is a measure of how many defects were correctly identified.
- Precision is a measure of how good the prediction of defective classes is.
- F-measure is the harmonic mean of precision and recall.
- Matthews Correlation Coefficient is a compound performance measure which includes all quadrants of the confusion matrix and is not biased towards an understanding of the defective class. It can have a statistical interpretation as $\chi^2 = N \times (MCC)^2$ where N is the number of instances in the data set.

Table 2.9: Compound Performance Measures from a Binary Confusion Matrix

Measures	Defined As
Accuracy (a) / Correct Classification Rate (CCR)	$\frac{TP+TN}{TP+TN+FP+FN}$
Error Rate	$\frac{FP+FN}{TP+TN+FP+FN}$
Recall (r)/ True Positive Rate / Sensitivity / Probability of Detection (pd)	$\frac{TP}{TP+FN}$
True Negative Rate / Specificity (spec)	$\frac{TN}{TN+FP}$
False Positive Rate / Type I Error Rate (t1)/ Probability of False Alarm (pf)	$\frac{FP}{TN+FP}$
False Negative Rate / Type II Error Rate (t2)	$\frac{FN}{FN+TP}$
Precision (p)	$\frac{TP}{TP+FP}$
F-Measure / F-Score	$\frac{2 \times Recall \times Precision}{Recall + Precision} = \frac{2TP}{2TP+FP+FN}$
Balance	$1 - \frac{\sqrt{(0-pf)^2+(1-pd)^2}}{\sqrt{2}}$
G-mean	$\sqrt{Recall \times Precision}$
Matthews Correlation Coefficient (MCC)	$\frac{TP \times TN - FP \times FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}}$

2.9 Summary of Techniques

To summarise, the data for defect prediction can be both categorical and/or continuous. Depending on the type of data a range of models can be built using either statistical or machine learning techniques. Table 2.10 summarises the techniques and data type for building defect prediction models.

2.10 Conclusion

There are many possible ways of performing defect prediction. Menzies [2011] describes the many defect prediction studies reported in **[Paper 4]** as permutations of solving which combination of *model* × *dataset* × *metrics* performs 'best'. To some extent, I agree that many studies are trying to find the best combination of *model* × *dataset* × *metrics*, however, there is little discussion of the measurement protocols for capturing the data in the first place; the calibration of the measurement protocols for defect prediction; the contribution of the different good practices that should take place in binary defect prediction classification using n-fold cross validation. It is evident that many of the 208 studies reported in **[Paper 4]** have not reported adequately the measurement protocols or the contextual information which is needed for other researchers to be able to interpret the reported results in the context of other results. It is therefore appropriate that this thesis explores measurement protocols and the adoption of good practices in defect prediction studies.

Table 2.10: Summary of Defect Prediction Modelling Techniques.

Technique	Statistical or Machine Learning	Independent Variable Types	Dependent Variable Types	Comment
Linear Regression	Statistical	3	1	A simple technique which has been widely used for many years and has well defined characteristics. It has the advantage that many techniques have been developed to test the statistical significance of the models produced.
Negative Binomial Regression	Statistical	3	1	A technique used when the dependent variable represents count data such as the number of defects.
Logistic Regression	Statistical	3	2	A statistical technique for dealing with data where the dependent variable can be placed in one of two categories.
Regression Decision Tree	Statistical	3	1	A technique which produces rule based decisions which in some cases may be easily interpretable.
Naïve Bayes	Statistical	2	2	A simple technique for dealing with categorical data.
Classification Decision Trees	Machine Learning	3	2	A technique based on information theory which produces rules for a tree with categorical dependent variables.
Random Forrest	Machine Learning	3	2	An ensemble of decision trees which has shown to produce good generalisable models.
Support Vector Machine	Machine Learning	3	2	A technique widely used in machine learning which produces good results when tuned.
Artifical Neural Networks	Machine Learning	3	3	A machine learning technique based on the concept of a biological neurone. Some success has been achieved with this technique but requires tuning for good results.

Key: Variable Types: 1 = categorical, 2 = continuous and 3 = both

Chapter 3

Contribution of Papers

3.1 Introduction

In this chapter I summarise the work that has been published by myself which supports my thesis that there are many factors which affect the performance of trainable models for software defect prediction. The motivation for most of the papers is derived from the work carried out as part of an EPSRC funded project (EP/E055141/1) which set out to determine if program slicing metrics can reliably predict defects in code. Some of the additional papers relating to the EPSRC funded work are included in Appendix B because they may help to answer interesting questions raised by the main published work.

The published work is described in the following sections. In Section 3.2 I describe the work which relates to determining the variation in independent variables as a result of using different measurement protocols. In Section 3.3 I describe the work which relates to the variability in the dependent variable by studying the most accurate way of labelling a module as defective or not. In Section 3.4 I outline a secondary empirical study which investigates the factors which affect the performance of defect prediction models reported by different published defect prediction studies. In Section 3.5 I describe a meta-analysis of the defect prediction approaches which have been reported by defect prediction studies which have used the NASA datasets. In Section 3.6 I describe my contribution and the contribution of my co-authors to each paper.

3.2 RQ1: Does the measurement protocol for the independent variables affect the metric values produced?

Two main papers are included in this section which relate to RQ1.

[Paper 1] Bowes D, Counsell S, Hall T (2008) Calibrating program slicing metrics for practical use. Proceedings of TAIC PART, Windsor, UK

[Paper 2] Bowes D, Randall D, Hall T (2013) The Inconsistent Measurement of Message Chains. In: Proceeding of the 4th international workshop on Emerging trends in software metrics, ACM (accepted paper)

The two papers were motivated by studies we have been carrying out related to defect prediction. The first paper was motivated by the need to compute program slicing metrics as part of the EPSRC (EP/E055141/1) funded project. The aim of the EPSRC funded project was to establish if program slicing metrics could be used to predict the defectiveness of modules. The hypothesis that program slicing metrics could be used to predict defects was first described by Black et al. [2006].

The first **[Paper 1]** is a short paper which tried to replicate the program slicing metric values of Meyers and Binkley [2007]. **[Paper 1]** addresses RQ1 by computing metrics using 15 different measurement protocols.

[Paper 1] shows that when computing program slicing metrics, the metric values vary depending on the protocol used. This is confirmed in *Bowes et al. [2011]* which shows that without knowing the protocol used, it can be difficult to replicate the work of other researchers. Reproducing the results or even the protocols of others is not a new phenomenon. Kitchenham et al. [1990] also describes difficulties in replicating the protocol for measuring Henry and Kafura information flow metrics because the description of the metrics were ambiguous and some of the count data that the metrics depend on were difficult to obtain manually. Kitchenham et al. [1990] had already described the need to report fully the measurement protocol for acquiring independent variables.

We can conclude that the measurement protocol does affect the metric values. During the 3rd CREST workshop[1] I presented defect prediction results using program slicing metrics computed using different protocols. The results (see Appendix B.2) show that the different protocols appear to impact on the predictive performance which can be achieved.

The second paper is motivated by an ongoing piece of research into the relationship between 'code bad smells' (described by Beck et al. [1999]) and defects in code as reported by Zhang et al. [2011]. The second, **[Paper 2]** studies the protocols for identifying the bad smell called 'message chains'[2]. The aim of this work was to try to define how to identify message chains in code. **[Paper 2]** demonstrates that there is little agreement between three different protocols for measuring the presence of code bad smells. The lack of agreement between techniques for identifying bad smells demonstrates that researchers need to come to some agreement about how concepts are defined if they are to be measured. The contribution of **[Paper 2]** to my thesis is that when different protocols are used for detecting code bad smells, the profile of modules which are labelled as having a bad smell will be different.

There are three supporting papers for RQ1:

Counsell S, Bowes D, Hall T (2009) Cohesion Metrics: The Empirical Contradiction. In: The Psychology of Programming Interest Group, Open University.

Bowes D, Hall T (2010) Using program slicing data to predict code faults. In: The 3rd CREST Open Workshop, KCL.

Hall T, Bowes D (2011) Issues of consistency in defining slices for slicing metrics: ensuring comparability in research findings. In: The 10th CREST Open Workshop, UCL.

Counsell et al. [2009] shows that cohesion metrics for a function based on program slicing improve over different releases. This was not what we had expected because Izurieta and Bieman [2008] describes how good design patterns acquire 'grime' during the development of a program. The addition of 'grime' is likely to have a negative impact on code cohesion.

The results of our program slicing work was also presented at two CREST workshops: *Bowes and Hall [2010]* and Hall and Bowes [2011]. The second presentation was a pre-cursor to *Bowes et al. [2011]*. During *Bowes and Hall [2010]* we repeated the findings of **[Paper 1]**. *Bowes and Hall [2010]* also

[1] http://crest.cs.ucl.ac.uk/cow/3/

[2] Fowler and Beck [1999] identified 22 patterns in code which they identified as poor 'quality'. Message Chains are usually identified by a long list of getThis() method calls.

included a report that the ability to predict defects depended on the choice of variables used to compute the program slicing metrics. The relationship between program slicing metrics and defects presented in *[Bowes and Hall 2010]* are preliminary because the open source project chosen (Barcode) was very small and provided fewer than 200 items on which to build a defect prediction model.

3.3 RQ2: Is there an effective method for deriving the dependent variables for defect prediction?

One main paper is included in this section which relates to RQ2.

[Paper 3] Hall T, Bowes D, Liebchen G, Wernick P (2010a) Evaluating three approaches to extracting fault data from software change repositories. In: International Conference on Product Focused Software Development and Process Improvement (PROFES), Springer, pp 107–115

Collecting reliable dependent variable data is essential to building defect prediction models. Collecting such data requires identifying or labelling units of code as defective or not. The problem of labelling modules as defective or not was a critical problem in our EPSRC funded project which was trying to establish if program slicing metrics could be used to predict defects. If we were to be able to use program slicing metrics to predict defects, we needed a dataset which contained a set of modules which had been correctly labelled as defective or not. In **[Paper 1]** we had already identified Barcode as an open source project that we could compute program slicing metrics from. The work in **[Paper 3]** contributes to the EPSRC funded project by trying to establish the most effective way of automatically labelling modules as defective or not. **[Paper 3]** investigates the techniques used to label a change in code as either defect fix or enhancement. Being able to determine if a change is a fix or an enhancement allows us to establish changes where the code before hand was defective and the code afterwards was not defective (or at least less defective because the code may still contain other defects which have not yet been fixed). **[Paper 3]** evaluates three different approaches which had been used previously by other researchers. **[Paper 3]** was part of a trial, with the assumption that once we had found an effective way of labelling modules as defective or not, we could then apply it to other larger open source projects.

In **[Paper 3]** we tried to identify the types of changes (i.e. defect fix or enhancement) that occurred to code by developers. If we could identify the type of change, we would be able to identify releases of code which had been defective which were now not defective. **[Paper 3]** used three different techniques for identifying the type of code change: manual inspection, analysis of commit messages and the number of changes per commit. The conclusion of **[Paper 3]** is that no two techniques analysed had a high agreement with each other for identifying the type of change. Although this was a disappointing conclusion because we again had a low confidence in the accuracy with which changes were labelled, we again realised that it is important to report the protocol for measuring the technique for labelling code change types. Finding the change type has been an ongoing problem. Śliwerski et al. [2005] first proposed an automated technique for labelling modules as defective or not based on identifying the change types of a commit. Recently, Bird et al. [2009] re-implemented the protocol in [Śliwerski et al. 2005] and were able to discover more changes that were defects. The results of [Bird et al. 2009] indicate that the defect datasets made available by Zimmerman et al.[3] may contain a relatively high proportion of items which are incorrectly labelled.

[3] http://www.st.cs.uni-saarland.de/softevo/bug-data/eclipse/

During the 3rd CREST workshop[4] I presented a second set of results which predict what type of code change[5] had occurred using program slicing metrics as the independent variable. The results (see Appendix B.2) show that the values produced by different protocols for computing the program slicing metrics do impact on the ability to predict defect fixing changes. The results in Appendix B.2 were interesting because they show that for some protocols there is an apparent large difference in the accuracy of predicting the change type. The discrepancy in being able to predict the change type may be due to an interesting relationship between the variables used to form the program slices and the types of changes identified by the different techniques for labelling the changes. Statistical analysis of the differences were not carried out because of a lack of data and because the data was based on a single project. Performing a replication study which would allow statistical analysis of the relationship between program slicing metrics and change type is the subject for future work.

The conclusion that we come to is that there is no single effective approach for identifying defective units of code using version control systems and source code and so the dependent variables used in defect prediction studies are likely to be unreliable.

3.4 RQ3: Which factors (dataset, learner) affect the performance of defect prediction studies?

Two main papers are included in this section which relate to RQ3.

[Paper 4] Hall T, Beecham S, Bowes D, Gray D, Counsell S (2012) A systematic literature review on fault prediction performance in software engineering. Software Engineering, IEEE Transactions on 38(6):1276 –1304

[Paper 5] Bowes D, Hall T, Gray D (2012b) Comparing the performance of fault prediction models which report multiple performance measures: reconstructing the confusion matrix. In: Proceedings of the 8th International Conference on Predictive Models in Software Engineering.

Best Paper in Conference Award.

The work in **[Paper 4]** was motivated again by the EPSRC funded project into establishing the ability of program slicing metrics to predict modules which are defect prone. The aim of the **[Paper 4]** was to identify from published studies, the learners and datasets which had resulted in good predictive performance results. The aim was to compute program slicing metrics for the datasets identified and then to use the same learners to see if program slicing metrics could improve on the predictive performance which had already been achieved.

[Paper 4] addresses the issue of comparing the performance of different defect prediction studies. The aim of this paper is to investigate how the context of models, the independent variables used, and the learners influence the performance of fault prediction models.

The analysis in **[Paper 4]** shows that there are many factors which are impacting on the predictive performance of defect prediction studies. Notably, the type of independent variables used has an impact on the predictive performance as does the data used. The type of classifier appears to have less of an impact

[4] http://crest.cs.ucl.ac.uk/cow/3/

[5] A code change was either classified as a defect fix or an enhancement.

on the predictive performance, with simple techniques tending to perform as well as complicated techniques. The results of Lessmann et al. [2008] are in general agreement with respect to which classifier does best. It should be noted that some classifiers which require extensive tuning (such as SVMs) are reported to perform less well even though in other disciplines, this is not the case.

[Paper 5] addresses the issue of being able to convert the performance measures reported in studies to a common comparable set of performance measures. The aim of this paper is to demonstrate the ability to re-compute the confusion matrix. This paper also reports inconsistencies in the performance measures of published papers. **[Paper 5]** supports **[Paper 4]** by showing that the results of some studies are not correctly presented (notably Elish and Elish [2008]) which can then be used to demonstrate that SVMs are not performing well when no tuning is carried out. **[Paper 5]** also demonstrates that results reported by studies need to be validated, either by checking that values reported are possible or by replication of the original studies.

There are three supporting papers for RQ3:

Hall T, Beecham S, Bowes D, Gray D, Counsell S (2011a) Developing fault-prediction models: What the research can show industry. Software, IEEE 28(6):96 –99

Bowes D, Hall T, Beecham S (2012a) SLuRp: a tool to help large complex systematic literature reviews deliver valid and rigorous results. In: Proceedings of the 2nd international workshop on Evidential assessment of software technologies, ACM, pp 33–36

Bowes D, Hall T, Gray D (2013) Dconfusion: A technique to allow cross study performance evaluation of fault prediction studies. Automated Software Engineering Journal (In Review)

[Bowes et al. 2012a] describes the tool SLuRp which was built in order to reliably collate the data for **[Paper 4]**. *[Hall et al. 2011]* is a paper published in IEEE Software which summarises the findings of *[Hall et al. 2012]* for practitioners. *[Bowes et al. 2013]* is an extended version of **[Paper 5]** which demonstrates the reliability of the recomputation technique from the results of $m \times n$ cross validation experiments.

3.5 RQ4: Are the results of machine learning studies reliable/trustworthy?

One main paper is included in this section which relates to RQ4.

[Paper 6] Hall T, Bowes D (2012) The state of machine learning methodology in software fault prediction. In: Machine Learning and Applications (ICMLA), 2012 11th International Conference on, vol 2, pp 308 –313

This last section includes one published paper **[Paper 6]**. While working on **[Paper 4]**, we realised that many different machine learning approaches were being used by researchers when carrying out machine learning experiments (for example did the researchers deal with imbalanced data or was feature reduction performed?). **[Paper 6]** analyses the machine learning approaches described in a subset of 21 papers from **[Paper 4]** which had used only the NASA datasets. We had assumed that more good approaches would be included as the year of publication increased. It is already known that some of the approaches will affect the performance of classifiers [He and Garcia 2008, Japkowicz and Stephen 2002, Kohavi et al. 1995], so the presence or absence of these approaches will affect the predictive performance of defect studies.

We found that recent studies were not systematically using good machine learning approaches.

It was clear that a few papers [He and Garcia 2008, Liebchen and Shepperd 2008, Menzies et al. 2007] were using the majority of good approaches, however, the other 18 studies that were analysed did not do as well.

The absence of key machine learning approaches (for example n-fold cross validation, feature reduction, data cleaning, dealing with imbalance) will have an effect on the predictive performance of defect prediction studies [He and Garcia 2008, Japkowicz and Stephen 2002, Kohavi et al. 1995]. I therefore conclude that the presence and absence of approaches used in defect prediction will be having an effect on predictive performance.

[Paper 6] demonstrates that the approaches themselves are factors which may affect the performance of trainable models for software defect prediction. The impact of these factors on the predictive performance of defect prediction studies is an ongoing piece of research.

3.6 Summary of my Contribution to each Main Paper

Paper 1: Calibrating program slicing metrics for practical use.

Bowes D, Counsell S, Hall T (2008) Calibrating program slicing metrics for practical use. Proceedings of TAIC PART, Windsor, UK

I formulated the methodology and collected the entire data-set. I wrote the initial draft of the paper. Tracy Hall provided direction for the work. Steve Counsell and Tracy Hall assisted in the analysis and write up of the published work.

Paper 2: The Inconsistent Measurement of Message Chains

Bowes D, Randall D, Hall T (2013) The inconsistent measurement of message chains. In: Proceeding of the 4th International Workshop on Emerging Trends in Software Metrics, ACM (accepted paper)

I modified StenchBlossom[6] so that metrics could be reported automatically. I provided the ArgoUML[7] data and processed it using StenchBlossom. I developed a website to allow manual inspection of the source code by reviewers. I supervised the MSc thesis of David Randall who performed the manual inspection of the ArgoUML source code and the capture of metrics using DECOR[8]. Tracy Hall assisted in the direction and write up of this work.

This paper will be presented at the 4th International Workshop on Emerging Trends in Software Metrics

[6] StenchBlossom is a tool for identifying code bad smells.

[7] ArgoUML is an open source application for developing UML models.

[8] DECOR is a technique for identifying code bad smells.

(WETSoM 2013) co-located with ICSE.

Paper 3: Evaluating three approaches to extracting fault data.

Hall T, Bowes D, Liebchen G, Wernick P (2010a) Evaluating three approaches to extracting fault data from software change repositories. In: International Conference on Product Focused Software Development and Process Improvement (PROFES), Springer, pp 107–115

Tracy Hall was the main architect for this study. Gernot Liebchen carried out the experiment to label modules as defective using key words and files per change-set techniques. I devised the bandwidth sliding window technique to identify change-sets. Paul Wernick, Gernot Liebchen and I performed the manual inspection of the Barcode source code. Tracy, Gernot and I devised the protocol to produce the agreed defect labels for each module. Gernot calculated the Cohen Kappa statistics. Tracy Hall and I jointly wrote the paper.

Paper 4: A systematic literature review on fault prediction performance in software engineering.

Hall T, Beecham S, Bowes D, Gray D, Counsell S (2012) A systematic literature review on fault prediction performance in software engineering. Software Engineering, IEEE Transactions on 38(6):1276 –1304

Tracy Hall, Sarah Beecham and Sue Black carried out a preliminary study to map when and where defect prediction studies were carried out during the period 2000 to 2009. Sarah Beecham carried out the initial searches and initial refinement which identified 256 papers for the period 2000 to 2010.

Myself, Tracy Hall, Sarah Beecham, David Gray and Steve Counsell devised a set of exclusion criteria based on good software engineering information and good machine learning approaches. Myself, Tracy Hall, Sarah Beecham, David Gray and Steve Counsell reviewed the 208 papers in pairs and applied a set of contextual exclusion criteria. I created a database driven website *[Bowes et al. 2012a]* called SLuRp which allowed the exclusion criteria to be refined. SLuRp allowed researchers to independently apply and moderate the exclusion criteria for each accepted paper. I extracted the performance values from the final set of 36 papers which passed the exclusion criteria. I devised the method of re-computing the confusion matrix which is the key to being able to compare the performance results of studies using binary classifiers. Tracy Hall and Steve Counsell extracted extra contextual and summary findings for the 36 papers. I produced the synthesis and analysis of the performance results with Tracy Hall. The paper was mainly authored by Tracy Hall, Sarah Beecham and myself.

Paper 5: Comparing the performance of fault prediction models which report multiple performance measures: reconstructing the confusion matrix.

Bowes D, Hall T, Gray D (2012b) Comparing the performance of fault prediction models which report multiple performance measures: reconstructing the confusion matrix. In: Proceedings of the 8th International Conference on Predictive Models in Software Engineering.

I developed the technique for recomputing the confusion matrix. I developed a JAVA library which was used in SLuRp to re-compute precision, recall and precision for a subset of papers in *Hall et al. [2012]*.

I extended the desktop tool J-Confusion (originally written by D. Gray) to allow researchers to be able to input performance measure values and re-compute a confusion matrix. I jointly wrote the paper with Tracy Hall. David Gray contributed the original J-Confusion and reviewed the paper.

Paper 6: The state of machine learning methodology in software fault prediction.

Hall T, Bowes D (2012) The state of machine learning methodology in software fault prediction. In: Machine Learning and Applications (ICMLA), 2012 11th International Conference on, vol 2, pp 308 –313

Tracy Hall provided direction for this work. Martin Shepperd and Tracy Hall assisted in the analysis of the published work. I formulated the methodology and the analysis which was included in the write up of the paper.

Chapter 4

Papers

Introduction

This chapter presents the six main papers of this thesis.

4.1 Paper 1: Calibrating program slicing metrics for practical use.

Bowes D, Counsell S, Hall T (2008) Calibrating program slicing metrics for practical use. Proceedings of TAIC PART, Windsor, UK

Calibrating program slicing metrics for practical use

David Bowes
Hertfordshire University
d.h.bowes@herts.ac.uk

Steve Counsell
Brunel University
steve.counsell@brunel.ac.uk

Tracy Hall
Brunel University
tracy.hall@brunel.ac.uk

Abstract

Program slicing metrics are an important addition to the range of static code measures available to software developers and researchers alike. However, slicing metrics still remain under-utilized due partly to the difficulty in calibrating such metrics for practical use; previous use of slicing metrics reveals a variety of calibration approaches. This paper reports on the effect of including different variables in the calibration (and collection) of slicing metrics. Findings suggest a variety of different results depending on the level of abstraction at which the metrics are collected (system or file), the inclusion or exclusion of 'printf' statements, formal 'ins', 'outs' and global variables in the metrics' calculation..

1. Introduction

Weiser [12, 13] and Ott and Thuss [9] defined a set of slice based metrics including Tightness, Coverage, Overlap, Min. Coverage and Max Coverage (see Table 1). These metrics could, potentially, be an important addition to the current range of static code metrics used by developers. Although slicing metrics have been studied in relation to code cohesion [7, 8, 9] they are rarely used by developers. Previous studies exploring the efficacy of slice-based metrics [2, 3, 4, 5, 6, 7, 8, 9, 10, 11] have tended to use different sets of variables in specifying the slices on which the metrics are based. This wide variety of approaches to data collection has made it difficult for developers (and indeed researchers) to interpret and subsequently use slice-based metrics. The aim of this paper is to provide a standard baseline upon which future use and research into slice-based metrics can be based.

2. Results

We investigated how each of the four categories of principal variables shown in Table 2 contributes empirically to the slice-based metrics of: Tightness, Overlap, Coverage, Min Coverage and Max Coverage for the Barcode Open Source Project. The results are shown in Table 3, from which a number of features emerge. First, inter-module effects are reduced significantly by slicing files individually compared to slicing on a whole project. The values on the right-hand side of Table 3 (Figure 2) are generally smaller than those on the left-hand side (Figure 1). The most notable reduction is for the Tightness metric, whose computation is heavily influenced by the cardinality of slice intersections.

Table 1. Weiser's slice-based metrics

Metric	Formula
Tightness	$= \frac{\lvert SL_{\text{int}} \rvert}{length(M)}$
Overlap	$= \frac{1}{\lvert V_o \rvert} \sum_{i=1}^{\lvert V_o \rvert} \frac{\lvert SL_{\text{int}} \rvert}{\lvert SL_i \rvert}$
Coverage	$= \frac{1}{\lvert V_o \rvert} \sum_{i=1}^{\lvert V_o \rvert} \frac{\lvert SL_i \rvert}{length(M)}$
Min Coverage	$= \frac{\min_i \lvert SL_i \rvert}{length(M)}$
Max Coverage	$= \frac{\max_i \lvert SL_i \rvert}{length(M)}$

Key	
Line	Line of code
M	Set of program lines in a module NB $length(M) \equiv \lvert M \rvert$
V_0	Set of variables used to slice a module
SL_i	Set of program lines in the slice for the *i*'th variable in V_0
SL_{int}	Intersection of all slices formed from each V_0

(Tightness measures the proportion of the module which is common to all slices; Overlap is the average proportion of common slices compared to each slice; Coverage is the average length of each slice compared to the length of the module; Min Coverage is ratio of the shortest slice compared to the length of the module and Max Coverage is the ratio of the longest slice compared to the length of the module; V_o is the set of output variables)

Table 2. Categories of principal variables and their use frequency (#) in previous studies

Categories	Description	# Studies
Formal ins	Input parameters for the function specified in the module declaration	6

Formal outs	The set of return variables	8
Global variables	The set of variables which are used or may be affected by the module	9
Printfs	Variables which appear as formal outs in the list of parameters in an output statement (e.g. printf)	7

NB: ‘Module’ is a function or method *not* a set of functions or files acting together.

Table 3. Average module metrics for different combinations of variables

Variables				Average module metrics									
				Sliced as a project					Files sliced individually				
I	O	G	pF	Over-lap	Tight-ness	Cover-age	Min C	Max C	Over-lap	Tight-ness	Cover-age	Min C	Max C
✓	✓	✓	✓	0.859	0.814	0.919	0.828	0.984	0.649	0.481	0.691	0.523	0.901
✓	✓	✓		0.861	0.820	0.926	0.833	0.984	0.643	0.482	0.705	0.524	0.901
✓	✓		✓	0.903	0.857	0.917	0.870	0.984	0.712	0.551	0.717	0.588	0.898
✓		✓	✓	0.905	0.852	0.926	0.863	0.977	0.759	0.563	0.712	0.587	0.892
	✓	✓	✓	0.898	0.837	0.918	0.842	0.966	0.745	0.519	0.671	0.543	0.845
✓	✓			0.911	0.869	0.929	0.881	0.984	0.728	0.560	0.743	0.590	0.898
		✓	✓	0.891	0.840	0.927	0.852	0.981	0.772	0.518	0.653	0.538	0.820
✓			✓	0.947	0.895	0.928	0.905	0.975	0.839	0.672	0.764	0.694	0.885
	✓	✓		0.920	0.844	0.915	0.847	0.953	0.767	0.521	0.653	0.544	0.761
✓		✓		0.911	0.869	0.929	0.881	0.984	0.728	0.560	0.743	0.590	0.898
	✓		✓	0.949	0.883	0.914	0.886	0.956	0.820	0.591	0.688	0.610	0.792
✓				0.972	0.929	0.951	0.933	0.975	0.944	0.823	0.856	0.832	0.885
	✓			1.000	0.897	0.897	0.897	0.897	1.000	0.612	0.612	0.612	0.612
		✓		0.907	0.859	0.941	0.866	0.971	0.851	0.538	0.639	0.547	0.717
			✓	0.917	0.851	0.896	0.866	0.968	0.749	0.464	0.597	0.496	0.778

I = Formal Ins, O = Formal Out, G = Globals, pF=printf; NB: Both forward and backward slices were used in all cases.

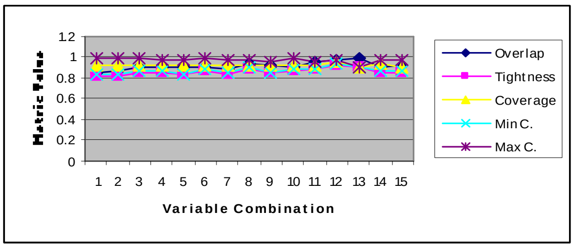


Figure 1. Sliced as a project

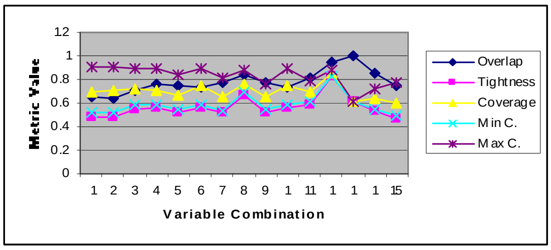


Figure 2. Files sliced individually

The least sensitive metrics appear to be those for Max Coverage, a metric influenced only by the size of the maximum slice. Second, exclusion of global variables (G) seems to cause a rise in the values of many of the metrics, both on the left *and* right-hand side of Table 3. This may be explained by the need to consider extra module inter-relationships induced by the use of global variables (which tend to push down the values of the metrics). The same effect can not be said of the inclusion or exclusion of printfs or formal ins (I) and outs (O), none of which seem to significantly influence the values of the metrics.

3. Acknowledgements

This research is kindly supported by grant from the EPSRC of the UK (EP/E055141/1).

References

[1] S. Counsell and D Bowes “A Theoretical and Empirical Analysis of Slicing Metrics for Cohesion Based on Matrices”, submitted to CSMR 2008.

[2] J. Bieman, L. Ott. “Measuring functional cohesion.” IEEE Trans. on Soft Eng., 20(8)::644–657, 1994.

[3] M. Harman, S. Danicic, B. Sivagurunathan, B. Jones, and Y. Sivagurunathan. “Cohesion metrics.” International Software Quality Week, San Francisco, CA, USA, 1995.

[4] M. Harman. “Cleaving together - program cohesion with slices.” EXE., Vol. 11 Iss. 8 pp. 35–42, 1997.

[5] A. Lakhotia. “Rule-based approach to computing module cohesion.” International Conf. on Software Engineering, Baltimore, USA, 1993. pp. 35–44.

[6] T. M. Meyers and D. Binkley. “A longitudinal and comparative study of slice-based metrics.” International Software Metrics Symposium, Chicago, USA, 2004.

[7] T. M. Meyers and D. Binkley. “Slice-based cohesion metrics and software intervention.” Working Conference on Reverse Engineering, pp. 256–265, 2004.

[8] T. M. Meyers and D. Binkley. “An empirical study of slice based cohesion and coupling metrics.” ACM Trans. on Software Engineering and Methodology, 2007.

[9] L. Ott and J. Thuss. “The relationship between slices and module cohesion.” International Conference on Software Engineering, pp. 198–204, 1989.

[10] K. Pan, S. Kim, and E. J. Whitehead Jr. “Bug classification using program slicing metrics.” IEEE Workshop on Source Code Anal and Manip., 31–42, 2006.

[11] M. Weiser. Program Slices: Formal, psychological, and practical investigations of an automatic program abstraction method. PhD Thes., Univ. Mich., MI, 1979.

[12] M. Weiser. “Program Slicing” Intl. Conference on Software Eng., San Diego, USA, 1981. pp. 439–449.

[13] Grammatech Inc. The CodeSurfer slicing system, 2000.

4.2 Paper 2: The Inconsistent Measurement of Message Chains.

Bowes D, Randall D, Hall T (2013) The Inconsistent Measurement of Message Chains. In: Proceeding of the 4th international workshop on Emerging trends in software metrics, ACM (accepted paper)

The Inconsistent Measurement of Message Chains

David Bowes
School of Computer Science
University of Hertfordshire
Hatfield, UK
+44(0)1707 286431
d.h.bowes@herts.ac.uk

David Randall
School of Computer Science
University of Hertfordshire
Hatfield, UK
+44(0)1707 280000
david@ranfam.org

Tracy Hall
Dept. of Information Systems & Computing
Brunel University
Uxbridge, UK
tracy.hall@brunel.ac.uk

***Abstract*—Fowler and Beck defined 22 Code Bad Smells. These smells are useful indicators of code that may need to be refactored. A range of tools have been developed that measure smells in Java code. We aim to compare the results of using two smell measurement tools (DECOR which is embedded in the Ptidej tool and Stench Blossom) on the same Java code (ArgoUML). This comparison identifies the code each tool identifies as containing Message Chains. We evaluate the results from these two tools using human judgment on the smells that the code contains. We look in detail at how and why the results differ. Our results show that each tool identified very different code as containing Message Chains. Stench Blossom identified very many more code instances of Message Chains than DECOR. We found three reasons why these discrepancies occurred. First there are significant differences in the definitions of Message Chains used by each tool. Second, the tools use very different measurement strategies. Third, the thresholds embedded in the tools vary. This measurement inconsistency is a problem to practitioners as they may be applying refactoring ineffectively. This inconsistency is also a problem for researchers as it undermines the reliability of making cross study comparisons and prevents mature knowledge the impact of smells being developed.**

***Index Terms*—Code Bad Smells, Message Chains, measurement tools, performance, measurement.**

I. Introduction

Code smells were first proposed by Beck and Fowler in 1999 [2]. Code can be described as 'smelly' when it contains one of the 22 poor coding structures identified by Fowler et al. [6]. These structures are low level localized instances of bad coding practices which may also be symptomatic of higher level design problems. Some of these poor code structures are very simple and are relatively easy to identify (e.g. Switch Statements), others are more complex and more difficult to identify (e.g. Speculative Generality). Smelly code is said by Fowler et al. [6] to have detrimental affects on software. Many of these affects are associated with smelly code being difficult to maintain with a higher likelihood of introducing faults during maintenance. Fowler et al. [6] provide refactoring strategies for the 22 code smells.

The aim of this paper is to investigate how consistently two smell measurement tools (DECOR and Stench Blossom) measure Message Chain bad smells in code. Our previous experience investigating smells [18, 19] suggests that smell tools are not always reliable or consistent in the smells they measure. Different tools seem to identify different code as containing a particular smell. Fontana et al. [5] have reported similar experiences of tool inconsistencies, though they did not investigate Message Chains.

In the investigation reported here we used DECOR and Stench Blossom on a large open source project (ArgoUML). We compared measurement consistency across the two tools. We analyse in detail where, how and why the code identified as containing Message Chains differs between the two tools. We evaluate these results against our manual measurement of Message Chains.

The effective measurement of code smells is important to practitioners. A variety of tools are used by practitioners to measure bad smells in order to apply appropriate refactoring. It is important that these tools perform effectively in measuring code bad smells. In particular these tools should not generate a large number of false positives (i.e. label code as containing a bad smell which turns out not to contain that smell) nor identify a large number of false negatives (i.e. label code as not containing a bad smell which turns out to contain that smell). Ineffective smell measurement tools reduce the cost effectiveness to practitioners of refactoring as: 1. time is wasted on code that is not smelly; 2. smelly code is not identified and not refactored.

Effectively measuring code bad smells is also important to researchers. Many researchers have studied the impact of code smells on software systems. The impact of smells on the maintenance and fault-proneness of code has been investigated. For example Li and Shatnawri [8] studied the relationship between six smells and fault-proneness; Olbrich et al. [12, 13] studied the impact of three smells on software evolution. To build a mature knowledge of the impact that particular smells have it is important that: 1. results reported by studies are valid, i.e. the smell data used must actually contain smelly code; 2. cross study comparisons must be possible across studies. Underpinning both of these pre-cursers to mature knowledge is that tools used by studies reliability and consistently measure smelly code. We investigate whether Stench Blossom and DECOR reliably and consistently measure Message Chains in ArgoUML.

The rest of the paper is structured as follows: In the next section we discuss Code Bad Smells, in particular Message Chains. In Section III we describe our research methods. In Section IV we present our results. In Section V we discuss our results and in Section VI we draw some conclusions.

II. Background

A. Code Smells

Fowler et al. [6] identified 22 code smells. In this paper we particularly focus on the measurement of Message Chains as this smell is measured by both tools we use.

The Message Chains smell is found in code where a client asks one object for another object, which the client then has to access through another object, which in turn it then has to access through yet another object, and so on. Message Chains are usually identified by a long list of `getThis()` methods (as below), or as a sequence of temporary objects.

Example Message Chain:

```
a.getB().getC().getD().getE().getTheData();
```

Message Chains are considered a poor coding practice as they entail the coupling of the methods in the navigation chain and any change in the relationships may cause the client to have to change. As the example above shows, the problem is that although *A* only needs to access the data from Class *E* it becomes unnecessarily coupled to classes *B, C, D* in order to achieve this. We have previously reported on the relationship between Message Chains and higher numbers of faults in code [19].

In some circumstances such delegation can be justified, and consequently Message Chains with only a couple of links are often considered harmless. The exact number of links such a chain can reasonably have is often related to other factors and is dependent upon context. Message Chains occur because objects must cooperate with others in order to perform any non-trivial operation. Problems can occur when this coupling becomes unnecessarily involved and complex.

B. Code Smell Measurement

Beck and Fowler [6] gave brief descriptions of each of their 22 bad smells alongside some refactoring techniques for their mitigation. However these explanations are fairly informal - more generally descriptive than rigorous and definitive - and do not give clear, unambiguous rules for measurement. Any process for measuring smells requires some means of definitively specifying what exactly is being looked for. Fowler et al.'s original smell descriptions do not provide such specification rules. This means that different approaches have been used by different smell measurement tools. Some approaches use standard metric measurements (e.g. [10]), some approaches use specific object oriented metrics (e.g. [7]), some approaches take into account inter-class relationships (e.g. [9]), some approaches use ad-hoc algorithms (e.g. [17]), some approaches work directly on source code (e.g. [11]), and some approaches work on system models (e.g. [9]). Our previous systematic literature review shows that considerable research effort has been dedicated to smell definition and measurement [20].

A recent approach to smell measurement is the DECOR approach [9]. Given the high number of citations for [9] (76 up to Dec 2012 according to Google Scholar), this approach is likely to be influential in smell measurement. DECOR measures well known, pre-defined smells and also provides a domain specific language allowing for the definition of other smells, based on code metrics as well as structural and lexical code properties. This extensibility potentially provides flexible and useful code smell measurement.

C. Code Smell Measurement Tools

A wide variety of smell measurement tools have been developed. Most of these are open source e.g. Stench Blossom. Such open source tools are useful as access to the tool's source code is possible. Such access means that the exact smell definitions implemented by the tool can be exposed. However some of these tools are commercial closed source tools e.g. Borland Together. The tools most commonly reported in the literature have been reported on by Fontana et al. [5]. Fontana et al. [5] suggest inconsistencies in the results of using these tools in terms of very different numbers of smells identified by different tools. Although Fontana et al. [5] provide a very useful preliminary study, they did not investigate the reasons for measurement differences nor compare in detail which smell instances were identified by which tool.

III. Methods

A. ArgoUML

This study is based on measuring smells in the open source project ArgoUML. This system is easily accessible (http://argouml.tigris.org/) and is written in Java; this is essential as the tools we use analyse only Java code. It is a fairly large project (1,963 class files). The size of the system means that it is reasonable to assume that plenty of code bad smells exist in the system. The project is also small enough to make it practical for us to use all 1,963 classes. ArgoUML has also been used in a number of previous studies of smells (e.g. [1, 9, 14, 16]) and so comparisons with previous results are possible.

B. The Smell Measurement Tools Used

In this study we used the Ptidej tool and the Stench Blossom tool. We selected Ptidej because this tool suite was developed by the Ptidej Team at the University of Montreal and is the system used both to specify the DECOR smell definition rule cards and run the associated smell measurement analysis on the target system. The DECOR approach is well recognised and we thought it important to include this tool in our evaluation. We selected Stench Blossom as it is a relatively easy to access and install open source tool and its use is well documented [11].

Ptidej is a Java application run within the Eclipse IDE. The Ptidej tool suite is an amalgam of various Java packages integrated to provide a comprehensive analysis tool for object-oriented systems. Ptidej uses a model of the software system derived by analysis of the compiled class files rather than direct examination of the source code. The first step in any Ptidej analysis process is the creation of the system model. Once this is accomplished all further investigative procedures are run against this model. Ptidej is set up to measure the following Fowler smells: Data Class, Large Class, Lazy Class, Long Method, Long Parameter List, Message Chains, Refused Bequest, Speculative Generality. We generally refer to Ptidej as DECOR throughout this paper.

Stench Blossom is an Eclipse plug-in which produces a graphical indication of the presence of certain code smells and their respective strength. The smells Stench Blossom measures [5]-Data Clumps, Feature Envy, Large Class, Large Method, Message Chain, Switch Statement. Stench Blossom enables a graphical representation of the smells present in code to be presented 'on the fly' in the edit window of Eclipse. This is in the form of a semi-circular area in the middle, right-hand side of the screen. This semi-circle, referred to as a flower, is divided in to 8 segments each representing a smell. As smells are measured in the source code being edited their corresponding petal is coloured in with the proportion of the area filled being an indication of its strength. If the cursor is placed over a petal the name of the smell represented is displayed and if clicked on a full explanation of the smell and the value of the relevant metrics defining it is revealed.

The smells are computed using metrics derived from the source code itself and not from class file models, as is the case with DECOR. The methods and metric calculations used are fairly simple by the creator's own admission [11] in order to allow for real time calculation without impinging on the performance of the IDE.

Both Ptidej and Stench Blossom identify both Message Chains and Large Class smells. We focused on Message Chains because Large Class is easily defined by a size metric. Message Chains are more complicated and therefore more difficult to define and measure. This means that the study of Message Chains should allow us to investigate how different measurement interpretations have been implemented. We also focused on Message Chains as Fontana et al. [5] do not investigate Message Chains.

C. *Message Chain Thresholds*

Bad smell tools usually use thresholds to measure whether or not code contains a smell. Thresholds are based on a variety of mechanisms and are often set differently depending on the tool and its underlying measurement strategy. These thresholds are of crucial importance to measuring smells and to understanding the basis on which different tools consider code to contain a smell or not. Tool users can vary these thresholds depending on their particular requirements (though this is not usually possible using closed source software like Borland Together).

Despite the importance of these thresholds they are often poorly documented making it difficult to know the default thresholds implemented by specific tools. Thresholds are also often very difficult to vary as doing so requires understanding and changing the code on which the tool is based. It is difficult to understand the reasons for any inconsistencies in measurement results without understanding the thresholds implemented by tools.

DECOR measures Message Chains by comparing a code metric, NOTI or number of transitive invocations (i.e how many successive method calls through different objects) against a set threshold. This approach has the advantage of differentiating between successive method calls to different objects and successive method calls on the same object (not Message Chains) as is common with string manipulations such as: `string.toLowerCase().trim()`. NOTI is computed by analysis of the reverse engineered byte-code model and is compared against a threshold level set on a rule card. The default threshold for Message Chains in DECOR is 4. This threshold can be changed by altering the rule card (which we did in this study). A lower NOTI lowers the threshold at which the number of transitive invocations are required for the code to be recognised as a Message Chain. We lowered NOTI to 3 for some runs of DECOR presented in the Results Section.

Stench Blossom uses a simpler method of parsing the source code for successive method calls and when it finds them reports back a ‘strength’ value ranging in the case of Message Chains between 0.12 and 1.1 for ArgoUML. When it does not identify a Message Chain, it reports a strength of 0.0. We have used the value of 0.12 as the lower (or default) threshold for the presence of Message Chains. We have used a strength of >0.3 for the ‘higher’ threshold (at which a greater number of successive method calls are needed to identify a Message Chain).

Fowler specifies no trigger value for the number of method object transitions which constitutes a Message Chain and so the ‘right’ threshold is open to interpretation.

D. *The Manual Approach*

To further evaluate the measurement performances of Stench Blossom and Ptidej we also manually identified Message Chains. This manual measurement was done on a large sample of ArgoUML class files (454 class files were manually evaluated). Manually measuring Message Chains in all 1,963 class files would have been too time consuming and was not practical.

We based our measurement of Message Chains on our own interpretation of Fowler et al.’s [6] description. We interpreted Fowler et al.’s [6] description of Message Chains very widely so that we were unlikely to miss any Message Chains. We counted any call which appeared to access data through method calls in an object not directly contacted by itself (i.e. completely through at least one intermediary object) as a Message Chain. One researcher with significant experience of Java development identified classes that contained Message chains. Measurement of Message Chains was done at the class level as DECOR only measures Message Chains at the class level.

Cohen's Kappa analysis was then used to calculate agreement rates between the three approaches (i.e. the two tools and the manual approach). Cohen’s Kappa is a statistical measure of the level of agreement between two ratings of the same set of data. It is considered to be an improvement upon just using the percentage agreement rates, as it takes into account the level of agreement which could be expected purely by chance. Kappa values range from 0 (no agreement) to 1.00 (total agreement), with larger values indicating better reliability. A Kappa value of greater than 0.70 is considered satisfactory.

E. *Limitations to Our Study*

Use of ArgoUML. We measured Message Chains only in the ArgoUML system. This system is not likely to be representative of all systems and so our results may not generalize across systems. However no one system is likely to be generally representative. A large sample of systems is needed to generate representative results. Our future work involves extending this study to additional systems.

Use of DECOR and Stench Blossom. We evaluated results from only two smell measurement tools. This was because setting up, running and performing a detailed analysis of results is extremely time consuming for each tool. It may be that there is more agreement in the results from other tools than we found in these two tools. Our future work involves extending this study to additional tools.

Use of Message Chains. We measured only Message Chains and no other smells. We focused on Message Chains as it was the one smell that both of the tools we selected measured. No existing tool measures every smell. Each tool measures a different (and usually small) sub-set of the 22 Fowler et al. [6] smells. It is therefore not easy to identify a range of smells that can be evaluated across several tools. There is little overlap in the smells the tools measure. It may be that there is less inconsistency in the definition and measurement of smells other than the Message Chains smell. Our future work involves extending this study to additional smells.

IV. Results

We now present the results obtained from running the DECOR and Stench Blossom tools on all 1,963 classes in the ArgoUML Open Source project. We identify and compare the classes each tool measures as containing Message Chains. We also compare the classes identified as containing Message Chains by each tool to those classes we manually identify.

TABLE I. Message Chains identified by both tools in ArgoUML

Tool	Result	Classes identified (n=1,963)	Percentage of classes identified
DECOR	Smell found	182	9%
	Smell not found	1781	91%
Stench Blossom	Smell found	1065	54%
	Smell not found	898	46%

Table I gives an overview of the classes identified as containing Message Chains by each of the tools. Each tool was run using the default threshold settings. Table I shows that Stench Blossom measures many more classes as containing Message Chains than does DECOR. Table I shows that 54% of all 1,963 ArgoUML classes are identified by Stench Blossom as containing Message Chains. This is compared to only 9% of classes similarly identified by DECOR. We then investigated the overlap in the classes identified by each tool.

Table II shows the agreement between the tools in measuring classes as containing (or not containing) Message Chains. Table II shows a low level of agreement on the classes identified. The table also shows that the relatively small number of classes identified by DECOR, are not a sub-set of the large set of classes identified by Stench Blossom. Although 145 classes out of the 182 identified by DECOR were also identified by Stench Blossom as containing Message Chains, 37 classes identified by DECOR were not identified by Stench Blossom (20% of the total identified by DECOR).

TABLE II. Measurement comparison between the two tools using default thresholds

		Stench Blossom		
		MC not identified	MC identified	Total Classes
DECOR	MC not identified	861	920	1781
	MC identified	37	145	182
	Total classes	898	1065	

NB: Cohen's Kappa = 0.09

Table II also confirms that 920 classes were identified by Stench Blossom that were not identified by DECOR. This is either potentially a very high number of false positives identified by Stench Blossom or a very high number of false negatives identified by DECOR. The very low overall agreement levels shown in Table II have a correspondingly low Cohen's Kappa result (0.09). To investigate possible reasons for the different results produced by each tool we varied the thresholds used by each tool.

TABLE III. Measurement comparison between the two tools with a default Stench Blossom threshold and a reduced DECOR threshold

		Stench Blossom (default)		
		MC not identified	MC identified	Total classes
DECOR (NOTI=3)	MC not identified	815	779	1594
	MC identified	83	286	369
	Total classes	898	1065	

NB: Cohen's Kappa = 0.17

Table III compares the classes identified by each tool when using a reduced measurement threshold for Message Chains in DECOR. We implemented this reduced threshold by changing the number of methods in the navigation chain required to decide if a class contained a Message Chain. We reduced this number from the default NOTI threshold of 4 to 3. Table III shows that this threshold reduction increased the number of Message Chains identified by DECOR to 396 (from 182 shown in Table III). However the overall agreement between the tools does not significantly improve, and still has a very low Cohen's Kappa score of 0.17 (up slightly from the previous 0.09 score). Table III also shows that although a larger subset of the classes identified by Stench Blossom are now also identified by DECOR (286), 83 classes now identified by DECOR are not identified by Stench Blossom. This means that 20% of classes identified by DECOR remain unidentified by Stench Blossom. So although the number of agreements has increased with the reduced threshold, the proportion of agreements has not. To further investigate the impact of thresholds on agreement levels we then reduced the threshold used by Stench Blossom.

TABLE IV. MEASUREMENT COMPARISON BETWEEN THE TWO TOOLS WITH AN INCREASED STENCH BLOSSOM THRESHOLD

		Stench Blossom (strength=high)		
		MC not identified	MC identified	Total classes
DECOR (default: NOTI = 4)	MC not identified	1171	610	1781
	MC identified	61	121	182
	Total classes	1232	731	

NB: Cohen's Kappa = 0.14

Table IV shows the classes each tool measures using the default DECOR threshold and an increased Stench Blossom threshold. We increased the threshold of Stench Blossom by only identifying a Message Chain only when its strength was high (a strength value of > 0.3). This change should reduce the number of Message Chains identified by Stench Blossom. Table IV shows that increasing Stench Blossom's threshold reduces the number of classes identified as containing Message Chains to 731 (from 1065 using the default threshold). However the overall agreement levels remain very low with a Cohen's Kappa score of 0.14. We then compared the performance of each tool when varied thresholds were used in each tool.

Table V shows the results of using a higher threshold in Stench Blossom and a lower threshold in DECOR. This variation increases the number of Message Chains identified by DECOR and reduces the number identified by Stench Blossom. The overall agreement levels shown in Table V are the highest of all the tables presented. However the overall agreement level remains low at 0.19. Table V shows that even using these varied thresholds Stench Blossom still measures about double the number of classes as containing Message Chains than does DECOR. About 40% of classes identified as containing Message Chains by DECOR are not identified by Stench Blossom. This percentage doubled when the thresholds are varied.

TABLE V. MEASUREMENT COMPARISON BETWEEN THE TWO TOOLS WITH A REDUCED DECOR THRESHOLD AND AN INCREASED STENCH BLOSSOM THRESHOLD

		Stench Blossom (strength = high)		
		MC not identified	MC identified	Total classes
DECOR (NOTI = 3)	MC not identified	1080	514	1594
	MC identified	152	217	369
	Total classes	1232	731	

NB: Cohen's Kappa = 0.19

The results suggest that differences in what each tool measures as a Message Chain are not fully explained by differences in thresholds. Consequently we investigated the differences in the classes identified by each tool using a manual analysis of 454 ArgoUML classes.

TABLE VI. COMPARISON BETWEEN STENCH BLOSSOM USING DEFAULT THRESHOLD AND MANUAL MEASUREMENT IN 454 CLASSES

		Manual		
		MC not identified	MC identified	Total classes
Stench Blossom	MC not identified	152	4	156
	MC identified	35	263	298
	Total classes	187	267	

NB: Cohen's Kappa = 0.82

Table VI shows the results of our manual measurement of Message Chains compared to those of Stench Blossom. Default threshold values are used in Stench Blossom. Table VI shows that the overall levels of agreement between our manual approach and Stench Blossom are very much improved. A Cohen's Kappa score of 0.82 is achieved. which is the highest level of agreement in this study. Table VI shows that not only do we identify similar numbers of classes containing Message Chains as Stench Blossom in this sample of 454 classes, but only 4 classes that we identify are not identified by Stench Blossom. The results shown in Table VI suggest that the measurement strategy used by Stench Blossom is similar to that used by humans.

Table VII shows the results of our manual measurement of Message Chains compared to those of DECOR. Default threshold values are also used in DECOR.

TABLE VII. COMPARISON BETWEEN DECOR USING DEFAULT THRESHOLD AND MANUAL MEASUREMENT IN 454 CLASSES

		Manual		
		MC not identified	MC identified	Total classes
DECOR	MC not identified	148	124	272
	MC identified	39	143	182
	Total classes	187	267	

NB: Cohen's Kappa = 0.31

Table VII shows that the overall levels of agreement between our manual approach and DECOR are better than when DECOR is compared to Stench Blossom with a Cohen's Kappa score of 0.31. However there remains significant disagreement between us and DECOR on which classes contain Message Chains. In particular 20% of classes that DECOR measures as containing Message Chains we do not similarly identify. This is the same percentage difference in measurement as between DECOR and Stench Blossom.

To investigate further this 20% of Message Chains identified by DECOR that Stench Blossom and our manual approach do not identify we performed further tests. We tested whether DECOR was measuring more sophisticated Message Chains than the other two methods. In particular whether DECOR was measuring Message Chain occurrences containing intermediate results which, although still Message Chains do not necessarily have the classic a.b().c().d() source code format and are harder to detect. To test this we wrote a simple test program and used DECOR to identify Message Chains in this test program. This program, instead of making a call by an obvious chain of method calls in the main test class, transferred the major part of the chain to the second Continent class level and invoked it by a method call to the Continent class as below:

In the TestMain class:

```
String tempGreeting =
getContinent(continent).getContinentGreeting();

In the Continent class:

public String getContinentGreeting() {

return
countries.get(0).getLanguage().getGreeting(); }
```

This code effectively performs exactly the same function - returning data to TestMain by successive method calls through all the objects - but the Message Chain, although still present, would not necessarily be located in TestMain by a brief inspection of the source code. DECOR still identified this as a Message Chain and, equally importantly, still gave its location as TestMain.

This suggests that although DECOR measures fewer Message Chain occurrences than other techniques, its use of a reverse-engineered byte-code model and metric evaluation, may well identify significant instances that other techniques miss.

V. Discussion

Our results show that DECOR and Stench Blossom differ significantly in the classes they measure as containing Message Chains. Stench Blossom measures very many more instances of Message Chains (54% of all classes) than does DECOR. This measurement difference suggests that either DECOR has a high rate of false negatives or that Stench Blossom has a high rate of false positives. Identifying false negatives is bad as this means that smells are not being identified that should be identified. And this means that poorly structured code is likely to remain in the system as it is not identified for refactoring. False positives are also bad as this means that code is being identified as poorly structured that is not poorly structured. Falsely identifying such code wastes developers' time and may result in unnecessary refactoring.

The results of our manual measurement of Message Chains were very similar to the results of Stench Blossom. Both approaches measured relatively high numbers of Message Chains. It is debatable (and the subject of further work) whether all Message Chains in this large set will actually create the detrimental affects on software mentioned by Fowler et al. [6]. It is likely that many of these Message Chains exist at a minimal level and are unlikely to have significantly detrimental affects on the system. The thresholds at which a coding structure is considered to be a Message Chain may need to be higher both in our manual approach and in Stench Blossom.

Our results suggest that Stench Blossom identifies the Message Chains developers are also likely to identify. Given that an implicit aim of automatic measurement is to mimic manual measurement, one conclusion of this finding could be that Stench Blossom is the 'better' tool. However our further analysis suggests that DECOR actually identifies legitimate Message Chains that both Stench Blossom and our manual approach missed. In particular we show DECOR measuring Message Chains based on method calls which use intermediate results. DECOR uses a measurement strategy based on a reverse-engineered byte-code model and metric evaluation. This measurement strategy means that semantic structures can more easily be identified by DECOR than by the direct code analysis strategy used by Stench Blossom and by our manual approach. This difference between the strategies means there is an implicit difference in the definitions of Message Chains used by Stench Blossom and DECOR.

VI. Conclusion

Our findings show that the different measurement strategies used in DECOR and Stench Blossom mean that each tool considers different code structures as a Message Chain. There is inconsistency in the definition of Message Chains in the tools. Definition inconsistencies in software metrics and the consequences of those inconsistencies (e.g. for KLOC [4]) were first reported many years ago [3]. Despite the importance of definition consistency long being known, our results suggest that progress in achieving consistency in the definition of code smells remains slow.

The differences we report in measurement results have important implications for practitioners and researchers. The accuracy of smell measurement tools determines the effectiveness with which practitioners are able to apply refactoring. Practitioners do not want to waste time considering code that does not contain bad smells and does not require refactoring. Therefore accurate tools are important for practitioners.

The variations we report in measuring smells are very important to researchers. To build a mature understanding of the impact of smells on systems it is necessary that a corpus of evidence is developed. Studies in this corpus need to be based on smell data collected using the same smell definitions. Studies presenting smell data based on different definitions of smells are not comparable and any comparisons made are not likely to be valid. The studies reported so far on the impact of smells use a wide range of measurement approaches and tools [20]. These tools are likely to be based on different measurement strategies and definitions. Consequently it is very dangerous to compare across these studies. Until thorough and commonly used smell definitions are established we are a long way from developing a mature knowledge of the impact of smells on software systems.

REFERENCES

[1] T. Arendt and G. Taentzer. Uml model smells and model refactorings in early software development phases. Universita ̈t Marburg, 2010.

[2] K. Beck, M. Fowler, and G. Beck. Bad smells in code. Refactoring: Improving the design of existing code, pages 75–88, 1999.

[3] N. Fenton and M. Neil. A critique of software defect prediction models. Software Engineering, IEEE Transactions on, 25(5):675 –689, sep/oct 1999.

[4] N. Fenton and S. Pfleeger. Software metrics, volume 1. Chapman & Hall London, 1991.

[5] F. Fontana, E. Mariani, A. Morniroli, R. Sormani, and A. Tonello. An experience report on using code smells measurement tools. In Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on, pages 450 –457, march 2011.

[6] M. Fowler and K. Beck. Refactoring: improving the design of existing code. Addison-Wesley Professional, 1999.

[7] M. Lanza and R. Marinescu. Object-oriented metrics in practice.. Springer, 2006.

[8] W. Li and R. Shatnawi. An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. Journal of Systems and Software, 80(7):1120 – 1128, 2007

[9] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A.-F. Le Meur. Decor: A method for the specification and measurement of code and design smells. Software Engineering, IEEE Transactions on, 36(1):20 –36, jan.-feb. 2010.

[10] M. Munro. Product metrics for automatic identification of "bad smell" design problems in java source-code. In Software Metrics, 2005. 11th IEEE International Symposium, page 15, sept. 2005.

[11] E. Murphy-Hill and A. P. Black. An interactive ambient visualization for code smells. In Proceedings of the 5th international symposium on Software visualization, SOFTVIS '10, pages 5–14, New York, USA, 2010. ACM.

[12] S. Olbrich, D. Cruzes, V. Basili, and N. Zazworka. The evolution and impact of code smells: A case study of two open source systems. In Empirical Software Engineering and Measurement, 2009. ESEM 2009. 3rd International Symposium on, pages 390 –400, oct. 2009.

[13] S. Olbrich, D. Cruzes, and D. Sjoberg. Are all code smells harmful? a study of god classes and brain classes in the evolution of three open source systems. In Software Maintenance (ICSM), 2010 IEEE International Conference on, pages 1 –10, sept. 2010.

[14] R. Oliveto, M. Gethers, G. Bavota, D. Poshyvanyk, and A. De Lucia. Identifying method friendships to remove the feature envy bad smell. In Software Engineering (ICSE), 2011 33rd International Conference on, pages 820–823.

[15] D. Randall. A study of techniques for the definition and measurement of design and code bad smells. Master's thesis, Computer Science, University of Hertfordshire, 2012.

[16] D. Schaffhauser. measuring Design Violations and Code Smells by Bug-Impact Analysis. PhD thesis, 2006.

[17] F. Simon, F. Steinbruckner, and C. Lewerentz. Metrics based refactoring. In Software Maintenance and Reengineering, 2001. Fifth European Conference on, pages 30 –38, 2001.

[18] M. Zhang, N. Baddoo, P. Wernick, and T. Hall. Improving the precision of fowler's definitions of bad smells. In Software Engineering Workshop, 2008. SEW '08. 32nd Annual IEEE, pages 161 –166, oct. 2008.

[19] M. Zhang, N. Baddoo, P. Wernick, and T. Hall. Prioritising refactoring using code bad smells. In Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE International Conference on, pages 458 –464, 2011.

[20] M. Zhang, T. Hall, and N. Baddoo. Code bad smells: a review of current knowledge. Journal of Software Maintenance and Evolution: Research and Practice, 23(3):179–202, 2011.

4.3 Paper 3: Evaluating Three Approaches to Extracting Fault Data from Software Change Repositories.

Hall T, Bowes D, Liebchen G, Wernick P (2010a) Evaluating three approaches to extracting fault data from software change repositories. In: International Conference on Product Focused Software Development and Process Improvement (PROFES), Springer, pp 107–115

Evaluating Three Approaches to Extracting Fault Data from Software Change Repositories

Tracy Hall[1], David Bowes[2], Gernot Liebchen[1], and Paul Wernick[2]

[1] Brunel University, Department of Information Systems & Computing, Uxbridge, Middlesex, UK
{tracy.hall,gernot.liebchen}@brunel.ac.uk
[2] University of Hertfordshire, School of Computer Science, Hatfield, Hertfordshire, UK
{d.h.bowes,p.d.wernick}@herts.ac.uk

Abstract. Software products can only be improved if we have a good understanding of the faults they typically contain. Code faults are a significant source of software product problems which we currently do not understand sufficiently. Open source change repositories are potentially a rich and valuable source of fault data for both researchers and practitioners. Such fault data can be used to better understand current product problems so that we can predict and address future product problems. However extracting fault data from change repositories is difficult. In this paper we compare the performance of three approaches to extracting fault data from the change repository of the Barcode Open Source System. Our main findings are that we have most confidence in our manual evaluation of diffs to identify fault fixing changes. We had less confidence in the ability of the two automatic approaches to separate fault fixing from non-fault fixing changes. We conclude that it is very difficult to reliably extract fault fixing data from change repositories, especially using automatic tools and that we need to be cautious when reporting or using such data.

Key words: Software, fault, data, prediction.

1 Introduction

Identifying and fixing faults in software is a major software development cost. Preventing and removing faults is reported to cost the US between $50 and $78 billion per year [1,2]. Code faults remain a significant source of problems in software with a great deal of resources dedicated to software testing and debugging [3]. Identifying where faults are before testing could lead to higher quality software and better use of resources [4,5,6].

It is important that we understand more about the nature and cause of faults in code so that we can target our search for faults more effectively both before and during testing. Research indicates that about 60-80% of software faults are found in about 20% of the code modules (eg. [7]), with around half of code modules usually fault free [8]. Clearly there are potential resource savings if fault handling efforts are focussed on the code that actually contains faults.

M. Ali Babar, M. Vierimaa, and M. Oivo (Eds.): PROFES 2010, LNCS 6156, pp. 107–115, 2010.

Many previous researchers have explored how faults in code can be targeted and identified, with a variety of code fault prediction models reported in the literature. Building reliable fault prediction models depends on the availability and dependability of historical fault data. Most models are built on the assumption that the causes of faults in the past are similar to the causes of faults in the future. This means that models built on historical fault data should enable unfound future faults to be located.

However identifying historical fault data is not straightforward in either commercial or open source projects. The first difficulty is that it is impossible to identify all faults. Residual faults always remain in the system. Consequently previous work predominately uses fault fixing data as a proxy for faults and clearly this represents only a sub-set of faults in the system. This is a problem we do not address in this study. The second difficulty is that recording and documenting fault data is an overhead that many developers avoid. Consequently fault data is rarely maintained and very few projects use bug reporting tools like Bugzilla. Where fault data is maintained it has been reported to not be terribly reliable. As a result most researchers extract fault data from change data (eg from CVS records). Although change data accurately represents changes implemented to the system, these changes include not only fault fixes but all enhancements, improvements and refactorings made to the system. Consequently it is important to identify only those changes that represent fault fixes.

The aim of the paper is to identify the most reliable approach to identifying fault fixing changes from a change repository. We compare the effectiveness of three approaches to identifying fault fixing data using the change repository of the Barcode open source system. In the first approach we manually analyse change diffs (textual changes in code between revisions) and classify each as either a fault fixing diff or not. The other two approaches we investigate automatically analyse CVS records. The first searches the change repository for fault related key words, and the second searches for small sized changes.

In the next section we outline related previous work extracting fault data from change repositories. In Section Three we explain our methodology and describe the open source system used in this study. Section Four presents the results of collecting fault fixing data using each of the three approaches compared in this study. In Section Five we conclude and summarise our findings.

2 Background

The most common automatic method for extracting fault fixing change data from project repositories is based on searching for fault related key words in comment fields. This approach has been extensively evaluated by Zimmermann et al (eg [9,10,11]). Their work is mainly in the context of open source projects and is based on the use of CVS records and Bugzilla records. Although they report promising results they also acknowledge that natural language ambiguities reduce the success rate of the approach.

Weyuker and Ostrand [12] compare the performance of two approaches to using keywords to identify fault fixes within an industrial context. First, a system's change comments are searched through automatically for keywords indicating if a change was the result of a fault fix or if it was the result of a change of functionality. Second a change is categorised as a fault fix if an additional fault identifying field was set by developers during testing or once deployed. Their study showed that the technique using the additional field performed better than the keyword approach. Again, natural language ambiguities limited the performance of the keyword search technique.

A less common approach to identifying fault fixing changes has been proposed by Ostrand et al [13]. Following a suggestion made by developers they introduced the idea that fault fixing changes are only likely to affect one or two files. They tested this approach on a sample of 50 change submissions by reading manually through change messages [13]. Their results are promising and Ostrand et al report that they perform better than the key word search. However they applied and tested this approach only within a graphical systems development environment.

Unfortunately implementing Ostrand et al's [13] number of files involved in a change approach is hard if CVS is the only tool used to manage changes. This is because CVS does not clearly identify the set of individual changes making up one change. Other change control systems such as Subversion automatically maintain records of change sets, whereas with CVS these change sets must be reconstructed. Consequently methods for recognising the constituent changes in change sets have been developed. Zimmermann et al [14] identify a change set by either adopting a fixed window or a sliding window approach. Both techniques rely on the assumption that file submissions in a change set are carried out by the same author and within a certain time frame. The fixed window approach uses a 200 second time frame in which related submissions must occur to be part of a given change set. The sliding window approach moves the frame along after each file to see if the next file should be included. This results in a variable time frame in which submissions in a change set can be made.

3 Methodology

3.1 The Barcode Open Source System

In this study we use the change repository for the Barcode open source system (http://ar.linux.it/software/#barcode). We chose this system because initially we wanted to replicate Meyers and Binkley's program slicing metrics work [15], as our overall aim was to investigate the relationship between program slicing data and fault data. To do this we needed to extract fault data from the Barcode system and compare it to our program slicing metrics data.

Barcode is a small open source system (approx 9 KLOC) written in the C language. The program slicing tool that we use (CodeSurfer) only analyses C and C++ code, so language is a key consideration for us. Barcode is a tool for the conversion of text strings to printed bars. The Barcode project started in 1999 and a change history is available as CVS data. Although several developers participate in the project, all entries into the CVS system are carried out by one person. The Barcode project does not employ an automated fault reporting system, such as Bugzilla, but a file containing additional information about CVS submits, including pointers to issues in source code (the changelog), is maintained.

3.2 Procedures for Implementing the Three Approaches

Manual classification of change diffs. We use a manual analysis of Barcode's diffs to classify changes made to the system as either fault fixing or non fault fixing changes. Our aim is to use this classification as a baseline with which to compare the two automatic approaches. Our assumption is that manual classification of changes for a small project like Barcode is likely to be more accurate than automatic methods. However this manual classification approach is highly resource intensive and is only practical for small studies.

For manual classification we first extracted 199 module level diffs from the CVS repository for Barcode. Each diff contains the code and comments logged with the checked-in change. This data was then given to 3 researchers to independently classify each of these diffs as either: a fault fix; not a fault fix; don't know.

After this independent classification of the diffs, an inter-rater reliability analysis and a Cohen's Kappa score was calculated to measure classification agreement between the three researchers (described in Section 3.3). Disagreements in the classification of diffs were then discussed by all three researchers, the basis of these disagreements were resolved and the diffs classified for a second time.

Key word searching. We identified fault fixing changes from Barcode's change repository using a key word search based on previous studies (eg [14]) We searched the change comments logged in CVS for keywords: "bug", "fix(ed)" and "update(d)". The resulting classification of fault fixing and non fault fixing changes was compared to the manual classification of diffs using an inter rater reliability measure (described in Section 3.3).

Identifying changes involving only 1 or 2 files. We applied Ostrand et al's [13] approach to identifying fault fixing changes as those involving only one or two files. To identify how many files are involved in a given change all changes in a change set must be first identified. Identifying change sets is not straightforward for Barcode which maintains its change records only under CVS control.

In addition to implementing a fixed window approach to identifying change sets we also introduce an enhanced sliding window approach. Our sliding window approach estimates the bandwidth of an upload (bytes per second) to determine the size of the sliding window rather than use the conventional constant sized sliding window. This then allows us to calculate a more accurate variable timeframe for a particular author downloading a particular change set.

The resulting classification of fault fixing and non fault fixing changes is compared to the manual classification of diffs using an inter rater reliability measure (described in Section 3.3).

3.3 Inter Rater Reliability Measurement

We compare the performance of all 3 fault fix finding approaches using inter rater reliability scores using the Cohen's Kappa statistic. We report both the κ value of this statistic together with its categorical scale, as proposed by Landis and Koch:

< 0 No agreement
0.00 — 0.20 Slight agreement
0.21 — 0.40 Fair agreement
0.41 — 0.60 Moderate agreement
0.61 — 0.80 Substantial agreement
0.81 — 1.00 Almost perfect agreement

3.4 Limitations of the Study

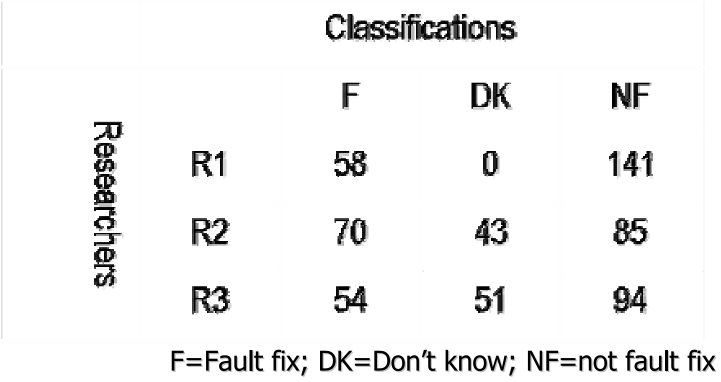
The limitations of this study are mainly related to the quality of data on which it is based. As in many change repositories CVS comment fields are not always completed and when they are not always accurately or comprehensively. However this is no different to any metrics data, as in the real world most metrics data is noisy and has missing values and our methods must be able to cope with this. In addition because each of the methods that we are evaluating tries to indirectly identify fault fixing changes, the classification will never be 100% accurate. For example there will be misclassifications in our manual diff method; ambiguity and idiosyncrasy in natural language will mean we miss keywords in our key work search; check-in variations will also mean that we will misclassify some changes involving one or two files as a fault fix when they were not. Furthermore Ostrand et al's approach may not work as well with program that do not have a graphical user interface.

4 Results

4.1 Manual Classification of Change Diffs

Table 1 shows how each researcher manually classified each of the 199 Barcode diffs.

Table 1. Overall diff classifications

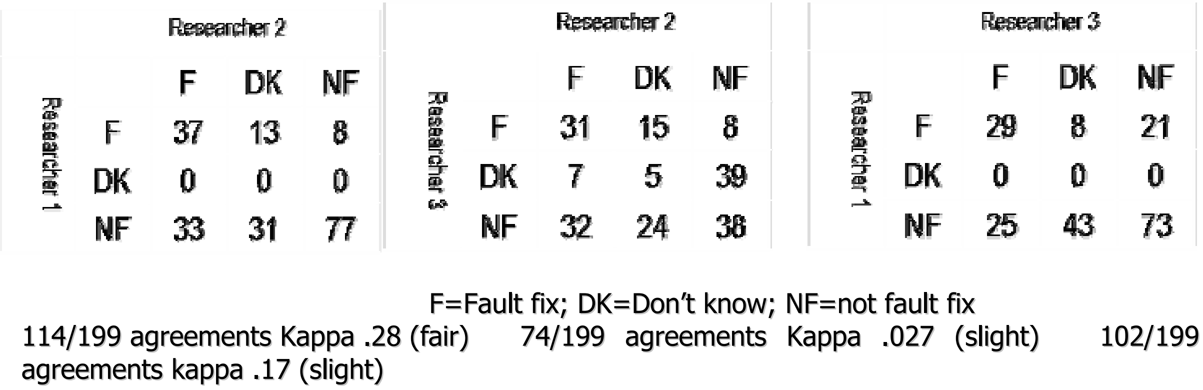


		Classifications		
		F	DK	NF
Researchers	R1	58	0	141
	R2	70	43	85
	R3	54	51	94

F=Fault fix; DK=Don't know; NF=not fault fix

Table 1 shows the different classification levels of each researcher and Table 2 shows the spread of those disagreements across the categories.

Table 2. Comparison of diff classifications between researchers



		Researcher 2		
		F	DK	NF
Researcher 1	F	37	13	8
	DK	0	0	0
	NF	33	31	77

		Researcher 2		
		F	DK	NF
Researcher 3	F	31	15	8
	DK	7	5	39
	NF	32	24	38

		Researcher 3		
		F	DK	NF
Researcher 1	F	29	8	21
	DK	0	0	0
	NF	25	43	73

F=Fault fix; DK=Don't know; NF=not fault fix

114/199 agreements Kappa .28 (fair) 74/199 agreements Kappa .027 (slight) 102/199 agreements kappa .17 (slight)

Table 2 shows substantial classification disagreement between the 3 researchers. This is likely to be the result of several factors. The first factor is the decision that Researcher 1 made to allow no Don't Know classifications and instead to assign diffs to the most likely class. The second factor is varying programming experience. Although all 3 researchers are familiar with C programming, two of them have extended experience of C programming due to working on projects in industry and academia. The other researcher's knowledge was based on being taught C during his first degree. The third factor may also be related to the difficulty of interpreting the intentions of a programmer based only on diffs.

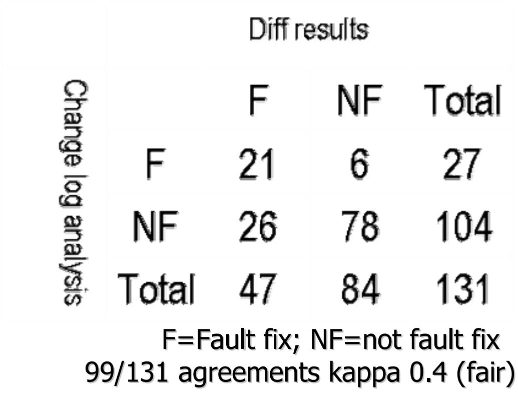
As a result of this high level of disagreement all three researchers got together and discussed the classification of each of the 199 diffs. They decided not to allow any Don't Knows as the other two approaches did not include such classifications. During this process 68 of the 199 diffs were excluded from future analysis as, on closer inspection they were based on inappropriate data (for example changes only to header files). This means that the other 2 approaches were applied to a 'cleaned' data set of 131 diffs. The following diff classification consensus was achieved: 47 fault fixing changes; 84 non fault fixing changes.

4.2 Keyword Search

We searched comments in the CVS logged changes as described in Section 3. As a result the 131 changes were classified as: 27 fault fixing changes and 104 non fault fixing changes. Table 3 compares the classification of this approach to the manual classification of diffs.

Table 3 shows that there is significant disagreement between the two approaches in the classification of fault fixing and non fault fixing changes. In particular the keywords classify far fewer changes as fault fixes than the manual diff classifications (27 as opposed to 47). This may be the result of missing comment data on which to search, as well as unexpected comments used to describe a fault fixing change. Our key words do not include for example 'patched' or 'mended' (though clearly our key word list could be extended).

Table 3. Key word compared to diff classification



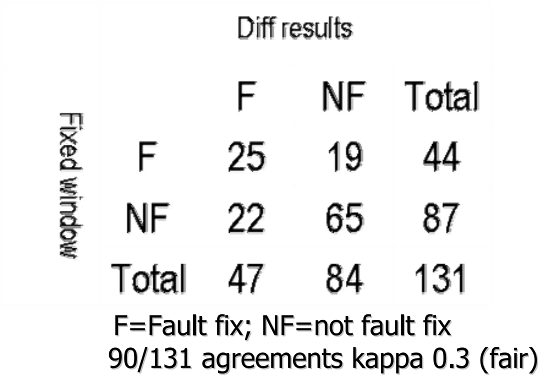
		Diff results		
Change log analysis		F	NF	Total
	F	21	6	27
	NF	26	78	104
	Total	47	84	131

F=Fault fix; NF=not fault fix
99/131 agreements kappa 0.4 (fair)

4.3 Size of Change Search

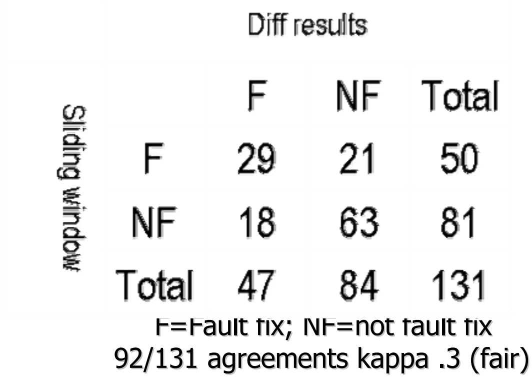
We applied both the fixed and sliding window timeframe approaches to identify those changes that involved only one or two files. Such changes are classified as fault fixing changes. Using the fixed window approach changes were classified as: 44 fault fixes; 87 non fault fixes. Using the sliding window approach changes were classified as: 50 fault fixes; 81 non fault fixes. Table 4 compares the classification of both of these timeframe approaches to our manual diff classifications.

Table 4. Size of change compared to diff classification



		Diff results		
Fixed window		F	NF	Total
	F	25	19	44
	NF	22	65	87
	Total	47	84	131

F=Fault fix; NF=not fault fix
90/131 agreements kappa 0.3 (fair)



		Diff results		
Sliding window		F	NF	Total
	F	29	21	50
	NF	18	63	81
	Total	47	84	131

F=Fault fix; NF=not fault fix
92/131 agreements kappa .3 (fair)

Table 4 shows that the approach used to calculate the timeframe in which downloaded files are assumed to be related make little difference to the classification of changes. The sliding window timeframe classifies 6 more changes as fault fixing than the fixed window timeframe. Table 4 also shows that both approaches have significant disagreement with the manual diff classification of fault fixing and non fault fixing changes.

5 Conclusions

Our results suggest that extracting fault fixing data from CVS change repositories can be unreliable. There are many factors that contribute to this. A significant factor is the completeness and quality of the data stored. Missing and unclear checked-in CVS comments make it difficult for the key word search technique to be accurate. This, together with the readability of code, also makes it difficult to manually identify fault fixing diffs. As a result it is difficult to have confidence in the precision of techniques for separating fault fixing changes from other changes. Added to which fault fixing changes represent only a sub-set of faults in the system as they are only faults that have been found, latent faults certainly remain in the system. Although our study is based on an open source system these problems are just as likely in the commercial domain where data is reported to be incomplete and have quality issues.

Our results could have important implications for researchers and practitioners. It is difficult for practitioners to have confidence in some fault prediction models as the historical fault data on which they are based could lack quality. This is likely to reduce the accuracy of predicting real faults or fault proneness. It is also difficult for researchers to build reliable fault prediction models without access to high quality data. Such data is not widely available, especially in projects which do not use fault management tools such as Bugzilla. Some projects do appear to adopt a more systematic approach to managing faults and it is these projects that are likely to generate more reliable data for analysis.

Our overall conclusions are that the quality of data used to build fault prediction models is critical to the reliability of those models and is an aspect of those models that needs to be addressed by researchers. And finally, the collection of reliable data on faults by projects is critical to improving our understanding of product quality.

Acknowledgements. This work was funded by the UK's Engineering and Physical Sciences Research Council under grant number: EP/E063039/1, Investigating code fault proneness using program slicing. We would also like to thank Sarah Beecham, Sue Black and Steve Counsell for their support during the work reported here.

References

1. Levinson, M.: Let's stop wasting $78 billion a year. CIO Magazine (2001)
2. Runeson, P., Andrews, A.: Detection or Isolation of Defects? An Experimental Comparison of Unit Testing and Code Inspection. In: ISSRE 2003, pp. 3–13 (2003)
3. Di Fatta, G., Leue, S., Stegantova, E.: Dis-criminative Pattern Mining in Software Fault Detection. In: SOQUA Workshop (2006)
4. Turhan, B., Kocak, G., Bener, A.: Data mining source code for locating software bugs: A case study in telecommunication industry. Expert Syst. Appl. 36, 6 (2009)
5. Bezerra, M.E.R., Oliveira, A.L.I., Adeodato, P.J.L., Meira, S.R.L.: Enhancing RBF-DDA Algorithm's Robustness: Neural Networks Applied to Prediction of Fault-Prone Software Modules. In: Artificial Intelligence in Theory and Practice II (2007)
6. Oral, A.D., Bener, A.: Defect prediction for embedded software. In: Proceedings of the 22nd International Symposium on Computer and Information Sciences, pp. 1–6 (2007)
7. Pai, G.J., Dugan, J.B.: Empirical Analysis of Software Fault Content and Fault Proneness Using Bayesian Methods. IEEE Trans. Software Eng. 33(10), 675–686 (2007)
8. Tomaszewski, P., Håkansson, J., Grahn, H., Lundberg, L.: Statistical models vs. expert estimation for fault prediction in modified code – An industrial case study. Journal of Systems and Software 80(8), 1227–1238 (2007)

9. Zimmermann, T., Premraj, R., Zeller, A.: Predicting defects for eclipse. In: Proceedings of the Third International Workshop on Predictor Models in Software Engineering (2007)
10. Sliwerski, J., Zimmermann, T., Zeller, A.: When do changes induce fixes? In: Proceedings of the Second International Workshop on Mining Software Repositories, pp. 24–28 (2005)
11. Schröter, A., Zimmermann, T., Premraj, R., Zeller, A.: Where do bugs come from? SIGSOFT Softw. Eng. Notes 31(6), 1–2 (2006)
12. Weyuker, E.J., Ostrand, T.J.: Comparing methods to identify defect reports in a change management database. In: DEFECTS 2008: Proceedings of the 2008 workshop on Defects in large software systems, pp. 27–31 (2008)
13. Ostrand, T.J., Weyuker, E.J., Bell, R.M.: Predicting the location and number of faults in large software systems. IEEE Trans. Software Eng. 31(4), 340–355 (2005)
14. Zimmermann, T., Weissgerber, P.: Preprocessing cvs data for fine-grained analysis. In: Proceedings of the First International Workshop on Mining Software Repositories, pp. 2–6 (2004)
15. Meyers, T.M., Binkley, D.: An empirical study of slice-based cohesion and coupling metrics. ACM Trans. Softw. Eng. Methodol. 17(1), 1–27 (2007)

4.4 Paper 4: A Systematic Literature Review on Fault Prediction Performance in Software Engineering.

Hall T, Beecham S, Bowes D, Gray D, Counsell S (2012) A systematic literature review on fault prediction performance in software engineering. Software Engineering, IEEE Transactions on 38(6):1276 –1304

4.4.1 Corrigendum

This paper requires one corrigendum:

1. On page 1291 of the published document, we would recommend that the residual sum of squares be used as the performance measure as described by Foss et al. [2003] rather than average residual as we had originally suggested.

A Systematic Literature Review on Fault Prediction Performance in Software Engineering

Tracy Hall, Sarah Beecham, David Bowes, David Gray, and Steve Counsell

Abstract—*Background*: The accurate prediction of where faults are likely to occur in code can help direct test effort, reduce costs, and improve the quality of software. *Objective*: We investigate how the context of models, the independent variables used, and the modeling techniques applied influence the performance of fault prediction models. *Method:* We used a systematic literature review to identify 208 fault prediction studies published from January 2000 to December 2010. We synthesize the quantitative and qualitative results of 36 studies which report sufficient contextual and methodological information according to the criteria we develop and apply. *Results:* The models that perform well tend to be based on simple modeling techniques such as Naive Bayes or Logistic Regression. Combinations of independent variables have been used by models that perform well. Feature selection has been applied to these combinations when models are performing particularly well. *Conclusion:* The methodology used to build models seems to be influential to predictive performance. Although there are a set of fault prediction studies in which confidence is possible, more studies are needed that use a reliable methodology and which report their context, methodology, and performance comprehensively.

Index Terms—Systematic literature review, software fault prediction

1 Introduction

This Systematic Literature Review (SLR) aims to identify and analyze the models used to predict faults in source code in 208 studies published between January 2000 and December 2010. Our analysis investigates how model performance is affected by the context in which the model was developed, the independent variables used in the model, and the technique on which the model was built. Our results enable researchers to develop prediction models based on best knowledge and practice across many previous studies. Our results also help practitioners to make effective decisions on prediction models most suited to their context.

Fault[1] prediction modeling is an important area of research and the subject of many previous studies. These studies typically produce fault prediction models which allow software engineers to focus development activities on fault-prone code, thereby improving software quality and making better use of resources. The many fault prediction models published are complex and disparate and no up-to-date comprehensive picture of the current state of fault prediction exists. Two previous reviews of the area have been performed in [1] and [2].[2] Our review differs from these reviews in the following ways:

- *Timeframes.* Our review is the most contemporary because it includes studies published from 2000-2010. Fenton and Neil conducted a critical review of software fault prediction research up to 1999 [1]. Catal and Diri's [2] review covers work published between 1990 and 2007.
- *Systematic approach.* We follow Kitchenham and Charters [3] original and rigorous procedures for conducting systematic reviews. Catal and Diri did not report on how they sourced their studies, stating that they adapted Jørgensen and Shepperd's [4] methodology. Fenton and Neil did not apply the systematic approach introduced by Kitchenham and Charters [3] as their study was published well before these guidelines were produced.
- *Comprehensiveness.* We do not rely on search engines alone and, unlike Catal and Diri, we read through relevant journals and conferences paper-by-paper. As a result, we analyzed many more papers.
- *Analysis.* We provide a more detailed analysis of each paper. Catal and Diri focused on the context of studies, including: where papers were published, year of publication, types of metrics used, datasets used, and modeling approach. In addition, we report

[1] The term "fault" is used interchangeably in this study with the terms "defect" or "bug" to mean a static fault in software code. It does not denote a "failure" (i.e., the possible result of a fault occurrence).

[2] Note that two referencing styles are used throughout this paper; [ref#] refers to papers in the main reference list while [Sref#] refers to papers in the separate systematic literature review list, located before the main reference list.

- *T. Hall and S. Counsell are with the Department of Information Systems and Computing, Brunel University, Uxbridge, Middlesex UB8 3PH, United Kingdom. E-mail: {tracy.hall, steve.counsell}@brunel.ac.uk.*
- *S. Beecham is with Lero—The Irish Software Engineering Research Centre, University of Limerick, Tierney Building, Limerick, Ireland. E-mail: sarah.beecham@lero.ie.*
- *D. Bowes and D. Gray are with the Science and Technology Research Institute, University of Hertfordshire, Hatfield, Hertfordshire AL10 9AB, United Kingdom. E-mail: {d.h.bowes, d.gray}@herts.ac.uk.*

Manuscript received 21 Oct. 2010; revised 10 July 2011; accepted 13 Sept. 2011; published online 30 Sept. 2011.
Recommended for acceptance by D. Sjøberg.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2010-10-0312.
Digital Object Identifier no. 10.1109/TSE.2011.103.

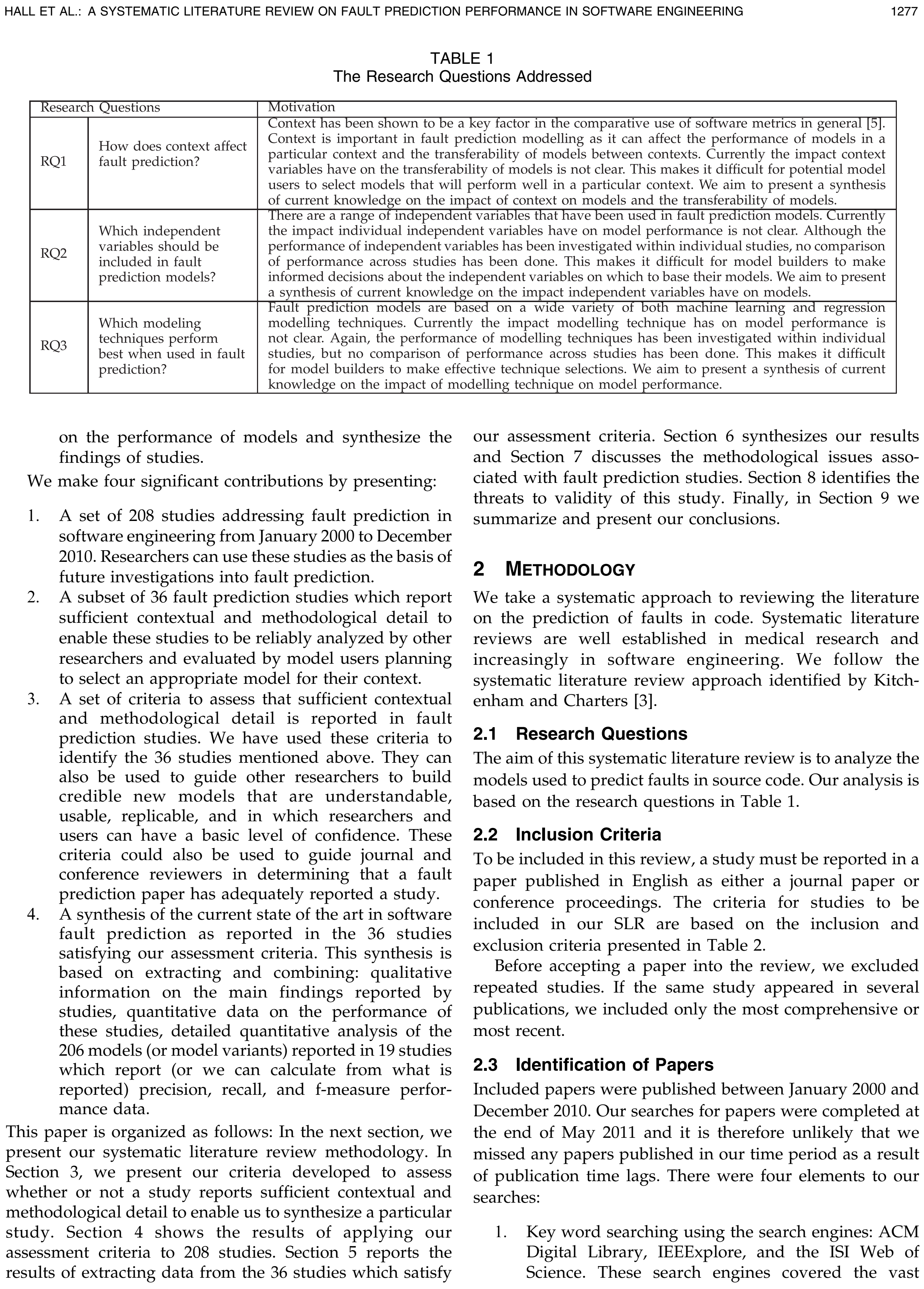


TABLE 1
The Research Questions Addressed

Research Questions		Motivation
RQ1	How does context affect fault prediction?	Context has been shown to be a key factor in the comparative use of software metrics in general [5]. Context is important in fault prediction modelling as it can affect the performance of models in a particular context and the transferability of models between contexts. Currently the impact context variables have on the transferability of models is not clear. This makes it difficult for potential model users to select models that will perform well in a particular context. We aim to present a synthesis of current knowledge on the impact of context on models and the transferability of models.
RQ2	Which independent variables should be included in fault prediction models?	There are a range of independent variables that have been used in fault prediction models. Currently the impact individual independent variables have on model performance is not clear. Although the performance of independent variables has been investigated within individual studies, no comparison of performance across studies has been done. This makes it difficult for model builders to make informed decisions about the independent variables on which to base their models. We aim to present a synthesis of current knowledge on the impact independent variables have on models.
RQ3	Which modeling techniques perform best when used in fault prediction?	Fault prediction models are based on a wide variety of both machine learning and regression modelling techniques. Currently the impact modelling technique has on model performance is not clear. Again, the performance of modelling techniques has been investigated within individual studies, but no comparison of performance across studies has been done. This makes it difficult for model builders to make effective technique selections. We aim to present a synthesis of current knowledge on the impact of modelling technique on model performance.

on the performance of models and synthesize the findings of studies.

We make four significant contributions by presenting:

1. A set of 208 studies addressing fault prediction in software engineering from January 2000 to December 2010. Researchers can use these studies as the basis of future investigations into fault prediction.
2. A subset of 36 fault prediction studies which report sufficient contextual and methodological detail to enable these studies to be reliably analyzed by other researchers and evaluated by model users planning to select an appropriate model for their context.
3. A set of criteria to assess that sufficient contextual and methodological detail is reported in fault prediction studies. We have used these criteria to identify the 36 studies mentioned above. They can also be used to guide other researchers to build credible new models that are understandable, usable, replicable, and in which researchers and users can have a basic level of confidence. These criteria could also be used to guide journal and conference reviewers in determining that a fault prediction paper has adequately reported a study.
4. A synthesis of the current state of the art in software fault prediction as reported in the 36 studies satisfying our assessment criteria. This synthesis is based on extracting and combining: qualitative information on the main findings reported by studies, quantitative data on the performance of these studies, detailed quantitative analysis of the 206 models (or model variants) reported in 19 studies which report (or we can calculate from what is reported) precision, recall, and f-measure performance data.

This paper is organized as follows: In the next section, we present our systematic literature review methodology. In Section 3, we present our criteria developed to assess whether or not a study reports sufficient contextual and methodological detail to enable us to synthesize a particular study. Section 4 shows the results of applying our assessment criteria to 208 studies. Section 5 reports the results of extracting data from the 36 studies which satisfy our assessment criteria. Section 6 synthesizes our results and Section 7 discusses the methodological issues associated with fault prediction studies. Section 8 identifies the threats to validity of this study. Finally, in Section 9 we summarize and present our conclusions.

2 METHODOLOGY

We take a systematic approach to reviewing the literature on the prediction of faults in code. Systematic literature reviews are well established in medical research and increasingly in software engineering. We follow the systematic literature review approach identified by Kitchenham and Charters [3].

2.1 Research Questions

The aim of this systematic literature review is to analyze the models used to predict faults in source code. Our analysis is based on the research questions in Table 1.

2.2 Inclusion Criteria

To be included in this review, a study must be reported in a paper published in English as either a journal paper or conference proceedings. The criteria for studies to be included in our SLR are based on the inclusion and exclusion criteria presented in Table 2.

Before accepting a paper into the review, we excluded repeated studies. If the same study appeared in several publications, we included only the most comprehensive or most recent.

2.3 Identification of Papers

Included papers were published between January 2000 and December 2010. Our searches for papers were completed at the end of May 2011 and it is therefore unlikely that we missed any papers published in our time period as a result of publication time lags. There were four elements to our searches:

1. Key word searching using the search engines: ACM Digital Library, IEEExplore, and the ISI Web of Science. These search engines covered the vast

TABLE 2
Inclusion and Exclusion Criteria

Inclusion criteria (a paper must be...)	Exclusion criteria (a paper must not be...)
- An empirical study - Focused on predicting faults in units of a software system - Faults in code is the main output (dependent variable)	- Focused on: testing, fault injection, inspections, reliability modelling, aspects, effort estimation, debugging, faults relating to memory leakage, nano-computing, fault tolerance. - About the detection or localisation of existing individual known faults.

TABLE 3
Paper Selection and Validation Process

Selection Process	# of papers	Validation
Papers extracted from databases, conferences and journals	2,073	80 random papers independently classified by 3 researchers
Sift based on title and abstract	-1,762 rejected	Fair/good inter-rater agreement on first sift (k statistic test)
Full papers considered for review	311 primary 80 secondary	Each paper is read in full and 80 secondary papers are identified from references
Rejected on full reading	-185 rejected	Papers are rejected on the basis that they do not answer our research questions
Comparison to Catal and Diri's review	2 (our searches missed)	
Papers accepted for the review	**208 papers**	

majority of software engineering publications and the search string we used is given in Appendix A.

2. An issue-by-issue manual reading of paper titles in relevant journals and conferences. The journals and conferences searched are shown in Appendix B. These were chosen as highly relevant software engineering publications found previously to be good sources of software engineering research [4].
3. A manual search for publications from key authors using DBLP.[3] These authors were selected as appearing most frequently in our list of papers: Khoshgoftaar, Menzies, Nagappan, Ostrand, and Weyuker.
4. The identification of papers using references from included studies.

Table 3 shows that our initial searches elicited 2,073 papers. The title and abstract of each was evaluated and 1,762 were rejected as not relevant to fault prediction. This process was validated using a randomly selected 80 papers from the initial set of 2,073. Three researchers separately interpreted and applied the inclusion and exclusion criteria to the 80 papers. Pairwise interrater reliability was measured across the three sets of decisions to get a fair/good agreement on the first iteration of this process. Based on the disagreements, we clarified our inclusion and exclusion criteria. A second iteration resulted in 100 percent agreement between the three researchers.

We read the remaining 311 papers in full. This resulted in a further 178 papers being rejected. An additional 80 secondary papers were identified from references and, after being read in full, accepted into the included set. We also included two extra papers from Catal and Diri's [2] review which overlapped our timeframe. Our initial searches omitted these two of Catal and Diri's papers as their search terms included the word "quality." We did not include this word in our searches as it generates a very high false positive rate. This process resulted in the 208 papers included in this review.

3 Assessing the Suitability of Papers for Synthesis

The previous section explained how we included papers which both answered our research questions and satisfied our inclusion criteria. This section describes how we identified a subset of those papers as suitable from which to extract data and synthesize an overall picture of fault prediction in software engineering. We then describe the extraction and synthesis process.

3.1 The Assessment Criteria

Our approach to identifying papers suitable for synthesis is motivated by Kitchenham and Charter's [3] notion of a quality check. Our assessment is focused specifically on identifying only papers reporting sufficient information to allow synthesis across studies in terms of answering our research questions. To allow this, a basic set of information must be reported in papers. Without this it is difficult to properly understand what has been done in a study and equally difficult to adequately contextualize the findings reported by a study. We have developed and applied a set of criteria focused on ensuring sufficient contextual and methodological information is reported in fault prediction studies. Our criteria are organized into four phases described below.

Phase 1: Establishing that the study is a prediction study.

In this SLR it is important that we consider only models which actually do some form of prediction. Some studies which seem to be reporting prediction models actually turn out to be doing very little prediction. Many of these types of studies report correlations between metrics and faults. Such studies only indicate the propensity for building a prediction model. Furthermore, a model is only doing any prediction if it is tested on unseen data (i.e., data that were not used during the training process) [S112]. To be considered a prediction model it must be trained and tested on different data [6]. Table 4 shows the criteria we apply to assess whether a study is actually a prediction study.

[3] http://www.informatik.uni-trier.de/~ley/db/.

TABLE 4
Prediction Criteria

Prediction criteria	Criteria definitions	Why the criteria is important
Is a prediction model reported?	The study must report some form of prediction. Not just report a correlation study of the relationship between faults and independent variables.	Such studies provide useful insights into observed patterns of faults but do not present prediction models as such. Nor do they validate their findings using unseen data. We do not therefore take these studies forward for synthesis.
Is the prediction model tested on unseen data?	The prediction model must be developed and tested on different data. This means that some form of hold-out or cross validation is necessary.	The performance of predictions based only on training data gives us no information on which to judge the performance of how such models generalise to new data. Such studies are therefore not taken forward for synthesis.

Table 4 shows that a study can pass this criterion as long as they have separated their training and testing data. There are many ways in which this separation can be done. Holdout is probably the simplest approach, where the original dataset is split into two groups comprising: {training set, test set}. The model is developed using the training set and its performance is then assessed on the test set. The weakness of this approach is that results can be biased because of the way the data have been split. A safer approach is often n-fold cross validation, where the data are split into n groups $\{g_1 \dots g_n\}$. Ten-fold cross validation is very common, where the data are randomly split into 10 groups, and 10 experiments carried out. For each of these experiments, one of the groups is used as the testing set, and all others combined are used as the training set. Performance is then typically reported as an average across all 10 experiments. M-N fold cross validation adds another step by generating M different N-fold cross validations, which increases the reliability of the results and reduces problems due to the order of items in the training set.

Stratified cross validation is an improvement to this process, and keeps the distribution of faulty and nonfaulty data points approximately equal to the overall class distribution in each of the n bins. Although there are stronger and weaker techniques available to separate training and testing data, we have not made a judgment on this and have accepted any form of separation in this phase of assessment.

Phase 2: Ensuring sufficient contextual information is reported.

We check that basic contextual information is presented by studies to enable appropriate interpretation of findings. A lack of contextual data limits the user's ability to: interpret a model's performance, apply the model appropriately, or repeat the study. For example, a model may have been built using legacy systems with many releases over a long time period and has been demonstrated to perform well on these systems. It may not then make sense to rely on this model for a new system where the code has only recently been developed. This is because the number and type of faults in a system are thought to change as a system evolves [S83]. If the maturity of the system on which the model was built is not reported, this severely limits a model user's ability to understand the conditions in which the model performed well and to select this model specifically for legacy systems. In this situation the model could be applied to newly developed systems with disappointing predictive performance.

The contextual criteria we applied are shown in Table 5 and are adapted from the context checklist developed by Petersen and Wohlin [7]. Our context checklist also overlaps with the 40 project characteristics proposed by Zimmermann et al. [S208] as being relevant to understanding a project sufficiently for cross project model building (it was impractical for us to implement all 40 characteristics as none of our included studies report all 40).

Context data are particularly important in this SLR as it is used to answer Research Question 1 and interpret our overall findings on model performance. We only synthesize papers that report all the required context information as listed in Table 5. Note that studies reporting several models based on different datasets can pass the criteria in this phase if sufficient contextual data are reported for one or more of these models. In this case, data will only be extracted from the paper based on the properly contextualized model.

Phase 3: Establishing that sufficient model building information is reported.

For a study to be able to help us to answer our research questions it must report its basic model building elements. Without clear information about the independent and dependent variables used as well as the modeling technique, we cannot extract sufficient data to allow synthesis. Table 6 describes the criteria we apply.

Phase 4: Checking the model building data.

Data used are fundamental to the reliability of models. Table 7 presents the criteria we apply to ensure that studies report basic information on the data they used.

In addition to the criteria we applied in Phases 1 to 4, we also developed more stringent criteria that we did not apply. These additional criteria relate to the quality of the data used and the way in which predictive performance is measured. Although we initially intended to apply these, this was not tenable because the area is not sufficiently mature. Applying these criteria would have resulted in only a handful of studies being synthesized. We include these criteria in Appendix C as they identify further important criteria that future researchers should consider when building models.

3.2 Applying the Assessment Criteria

Our criteria have been applied to our included set of 208 fault prediction studies. This identified a subset of 36 finally included studies from which we extracted data and on which our synthesis is based. The initial set of 208 included papers was divided between the five authors. Each paper was assessed by two authors independently

TABLE 5
Context Criteria

Contextual criteria	Criteria definitions	Why the criteria is important
Source of data	The source of the system data on which the study is based must be given. For example whether the system data is industrial, open source, NASA, Promise. If NASA/Promise data is used the names of the datasets used must be given. Studies using data that is in the public domain, the context of which is accessible via the public domain, need not explicitly report all these criteria to pass this phase. However the version studied must be specified to enable access to contextual data.	Different models may perform differently when applied to different data sets; for example some models may perform better on OS data than industrial data. It is therefore essential for synthesis that we can establish the source of the data.
Maturity	Some indication of the maturity of the system being studied must be reported. Readers must be able to generally determine whether the system is a mature system with many releases which has been in the field for many years, or whether it is a relatively newly developed system, or whether the system has yet to be released.	The age of a system has a significant impact on how it behaves. Especially in terms of the faults in the system. Many factors contribute to why the age of the system impacts on faults in the system, including the amount of change the system has undergone. This means that some models are likely to perform better than others on newly developed as opposed to legacy systems. It is therefore essential that we can establish the maturity of the system(s) on which the model was based, as without this it is difficult to correctly interpret study findings for synthesis.
Size in KLOC	An indication of the size of the system being studied must be given in KLOC. The overall size of the system will suffice, i.e. it is not necessary to give individual sizes of each component of the system being studied, even if only sub-sets of the system are used during prediction. Size indicated by measures other than KLOC are not acceptable (e.g. number of classes) as there are great variations in the KLOC of such other measures.	The size of systems is likely to impact on the behaviour of systems. Consequently, the faults in a system may be different in small as opposed to large systems. This means that it is also likely that different types of models will perform differently on systems of different sizes. It is therefore essential that we can establish the size of the system(s) on which the model was built as without this it is difficult to correctly interpret study findings for synthesis. Using KLOC does have limitations. The definitions of KLOC can vary and KLOC can be language dependent [8].
Application domain	A general indication of the application domain of the system being studied must be given, e.g. telecoms, customer support, etc.	Some models are likely to be domain specific. Different domains apply different development practices and result in different faults. It is therefore important that domain information is given so that this factor can be taken into account when model performance is evaluated. It is therefore essential that we establish the domain of the system(s) on which the model was built, as without this it is difficult to correctly interpret study findings for synthesis.
Programming language	The programming language(s) of the system being studied must be given.	Different languages may result in different faults. In particular it may be that OO languages perform differently from procedural languages. This makes it likely that some models will perform better for some languages. It is therefore essential that we can establish the language of the system(s) on which the model was built as without this it is difficult to correctly interpret study findings for synthesis.

TABLE 6
Model Building Criteria

Model building criteria	Criteria definitions	Why the criteria is important
Are the independent variables clearly reported?	The basis on which the model is making predictions must be clear and explicit. For example independent variables (or predictor variables) could include: static code metrics (complexity etc.), code churn metrics, previous fault metrics, etc.	In any experimental work which shows the performance of a method it is essential that the independent variables tested are explicitly identified. Without this, confidence in the work is significantly lowered and it is difficult to evaluate the impact of those variables across studies during synthesis.
Is the dependent variable clearly reported?	It must be distinguishable whether studies are predicting faults in terms of whether a module is fault prone or not fault prone (i.e. using a categorical dependent variable) or in terms of the number of faults in a code unit (i.e. using a continuous dependent variable). Some continuous studies additionally report ranked results (i.e. the identification of the faultiest 20% of files).	In any experimental work it is essential that the dependent variables are explicitly identified. Without this, confidence in the work is significantly lowered. Furthermore, the evaluation of fault prediction models is related to whether the dependent variable is categorical or continuous. It is therefore essential for synthesis that this information is clear.
Is the granularity of the dependent variable reported?	The unit of code granularity of predictions must be reported. For example fault predictions in terms of faults per module, per file, per package etc. These terms may be used differently by different authors, e.g. 'module' is often used to mean different code units by different authors. Studies must indicate the code unit being used, i.e. if the term 'module' is used, readers must be able to work out what unit of code is being defined as a module.	It is difficult to directly compare the performance of one model reporting faults per file to another model reporting faults per method. Furthermore it may be that studies reporting faults at a file level are more able to perform well than studies reporting at a method level. The granularity of the dependent variable must therefore be taken into account during synthesis and so an indication of the unit of fault granularity must be reported.
Is the modelling technique used reported?	It must be clear what modelling technique is being used, e.g. linear regression, decision trees, etc. It is not acceptable to present results from a tool based on a model that is not discussed in the paper.	Different modelling techniques are likely to perform differently in different circumstances. A study must report the modelling technique used, as we cannot examine the impact of method on performance without this information.

TABLE 7
Data Criteria

Data criteria	Criteria definitions	Why the criteria is important
Is the fault data acquisition process described?	The process by which the fault data was obtained must be described. A high level indication of this will suffice, e.g. obtained from the CVS records. However at least an overview of how the data was obtained must be provided. If the data has been obtained from a third party (e.g. NASA), some indication of how that data was obtained by that third party must be given or referenced or available in the public domain.	In order to have confidence in the data on which a model was built it is necessary to have some information on how the data was collected. Data is fundamental to the quality of the models built and the collection process has a huge impact on the quality of that data. It is not possible to have any confidence in a study where the data seems to have come from thin air. Collecting software engineering data of any sort is difficult and error-prone. We do not synthesise papers which give no indication of how the data was collected. In an ideal world studies would also indicate the development point at which fault data collection starts and stops. This is because fault data may be unreliable for several reasons: fault data may be extracted from a different version of the system to which the independent variable relates; faults are likely to emerge after fault counting ends, and data collected at the start of system development is unlikely to be stable or complete.
Is the independent variable data acquisition process described?	Some indication of how the independent variable data (e.g. static code data) was obtained should be given. For example the static analysis tools used should be reported or the process by which code churn data collected should be described. This does not need to be at a great level of detail, just an indication given. If the data has been obtained from a third party (e.g. NASA), some indication of how that data was obtained by that third party must be given, referenced or available in the public domain.	As above.
For categorical studies, has the number of faulty versus non-faulty units on which the model has been trained and tested on been reported?	The balance of faulty versus non-faulty units used for training and testing must be reported. For studies using open source systems it is not essential to report this (though preferable) as this data should be possible to identify from the public data source.	The balance of faulty versus non-faulty units used for training and testing can affect the reliability of some performance measures (see Appendix F). It is essential that class distributions are reported (i.e. number of faulty and number of non-faulty units in the data used). This makes it possible to appropriately interpret performances reported using these measures. We use this information for our synthesis of categorical studies. As where precision and recall are not reported by such studies we re-compute an approximation of it. The faulty/non-faulty balance of data is often needed in this calculation.

(with each author being paired with at least three other authors). Each author applied the assessment criteria to between 70 and 80 papers. Any disagreements on the assessment outcome of a paper were discussed between the two authors and, where possible, agreement established between them. Agreement could not be reached by the two authors in 15 cases. These papers were then given to another member of the author team for moderation. The moderator made a final decision on the assessment outcome of that paper.

We applied our four phase assessment to all 208 included studies. The phases are applied sequentially. If a study does not satisfy all of the criteria in a phase, then the evaluation is stopped and no subsequent phases are applied to the study. This is to improve the efficiency of the process as there is no point in assessing subsequent criteria if the study has already failed the assessment. This does have the limitation that we did not collect information on how a paper performed in relation to all assessment criteria. So if a paper fails Phase 1, we have no information on how that paper would have performed in Phase 4.

This assessment process was piloted four times. Each pilot involved three of the authors applying the assessment to 10 included papers. The assessment process was refined as a result of each pilot.

We developed our own MySQL database system to manage this SLR. The system recorded full reference details and references to pdfs for all papers we identified as needing to be read in full. The system maintained the status of those papers as well as providing an online process to support our assessments of 208 papers. The system collected data from all authors performing assessments. It also provided a moderation process to facilitate identifying and resolving disagreements between pairs of assessors. The system eased the administration of the assessment process and the analysis of assessment outcomes. All data that were extracted from the 36 papers which passed the assessment is also recorded on our system. An overview of the system is available from [9] and full details are available from the third author.

3.3 Extracting Data from Papers

Data addressing our three research questions was extracted from each of the 36 finally included studies which passed all assessment criteria. Our aim was to gather data that would allow us to analyze predictive performance within individual studies and across all studies. To facilitate this, three sets of data were extracted from each study:

1. **Context data.** Data showing the context of each study were extracted by one of the authors. This data give the context in terms of: the source of data studied and the maturity, size, application area, and programming language of the system(s) studied.

2. **Qualitative data.** Data related to our research questions were extracted from the findings and conclusions of each study. This was in terms of what the papers reported rather than on our own interpretation of their study. This data supplemented our quantitative data to generate a rich picture of results within individual studies.

   Two authors extracted qualitative data from all 36 studies. Each author extracted data independently and compared their findings to those of the other author. Disagreements and omissions were discussed within the pair and a final set of data agreed upon.
3. **Quantitative data.** Predictive performance data were extracted for every individual model (or model variant) reported in a study. The performance data we extracted varied according to whether the study reported their results via categorical or continuous dependent variables. Some studies reported both categorical and continuous results. We extracted only one of these sets of results, depending on the way in which the majority of results were presented by those studies. The following is an overview of how we extracted data from categorical and continuous studies.

Categorical studies. There are 23 studies reporting categorical dependent variables. Categorical studies report their results in terms of predicting whether a code unit is likely to be fault prone or not fault prone. Where possible we report the predictive performance of these studies using precision, recall, and f-measure (as many studies report both precision and recall, from which an f-measure can be calculated). F-measure is commonly defined as the harmonic mean of precision and recall, and generally gives a good overall picture of predictive performance.[4] We used these three measures to compare results across studies and, where necessary, we calculate and derive these measures from those reported (Appendix E explains how we did this conversion and shows how we calculated f-measure). Standardizing on the performance measures reported allows comparison of predictive performances across studies. Lessmann et al. [S97] recommend the use of consistent performance measures for cross-study comparison; in particular, they recommend use of Area Under the Curve (AUC). We also extract AUC where studies report this. Appendix D summarizes the measurement of predictive performance.

We present the performance of categorical models in boxplots. Box plots are useful for graphically showing the differences between populations. They are useful for our results as they make no assumptions about the distribution of the data presented. These boxplots present the precision, recall, and f-measure of studies according to a range of model factors. These factors are related to the research questions presented at the beginning of Section 2; an example is a boxplot showing model performance relative to the modeling technique used.

Continuous studies. There are 13 studies reporting continuous dependent variables. These studies report their results in terms of the number of faults predicted in a unit of code. It was not possible to convert the data presented in these studies into a common comparative measure; we report the individual measures that they use. Most measures reported by continuous studies are based on reporting an error measure (e.g., Mean Standard Error (MSE)), or measures of difference between expected and observed results (e.g., Chi Square). Some continuous studies report their results in ranking form (e.g., top 20 percent of faulty units). We extract the performance of models using whatever measure each study used.

Two authors extracted quantitative data from all 36 studies. A pair approach was taken to extracting this data since it was a complex and detailed task. This meant that the pair of authors sat together identifying and extracting data from the same paper simultaneously.

3.4 Synthesizing Data across Studies

Synthesizing findings across studies is notoriously difficult and many software engineering SLRs have been shown to present no synthesis [13]. In this paper, we have also found synthesizing across a set of disparate studies very challenging. We extracted both quantitative and qualitative data from studies. We intended to meta-analyze our quantitative data across studies by combining precision and recall performance data. However, the studies are highly disparate in terms of both context and models. Meta-analyzing this quantitative data may generate unsafe results. Such a meta-analysis would suffer from many of the limitations in SLRs published in other disciplines [14].

We combined our qualitative and quantitative data to generate a rich picture of fault prediction. We did this by organizing our data into themes based around our three research questions (i.e., context, independent variables, and modeling techniques). We then combined the data on each theme to answer our research questions. This synthesis is presented in Section 6.

4 Results of our Assessment

This section presents the results from applying our assessment criteria (detailed in Tables 4, 5, 6, and 7) to establish whether or not a paper reports sufficient contextual and methodological detail to be synthesized. The assessment outcome for each study is shown at the end of its reference in the list of included studies.

Table 8 shows that only 36 of our initially included 208 studies passed all assessment criteria.[5] Of these 36 finally included studies, three are relatively short [S116], [S110], and [S164]. This means that it is possible to report necessary contextual and methodological detail concisely without a significant overhead in paper length. Table 8 also shows that 41 papers failed at phase 1 of the assessment because they did not report prediction models as such. This includes studies that only present correlation studies or

[4] Menzies et al. [10] claim that values of precision vary greatly when used with models applied to different datasets. However, reporting precision and recall via an f-measure effectively evaluates classifier performance, even in highly imbalanced domains [11], [12].

[5] These papers are [S8], [S9], [S10], [S11], [S18], [S21], [S29], [S31], [S32], [S37], [S51], [S56], [S69], [S73], [S74], [S76], [S83], [S86], [S92], [S98], [S109], [S110], [S116], [S117], [S118], [S120], [S122], [S127], [S133], [S135], [S154], [S160], [S163], [S164], [S190], [S203].

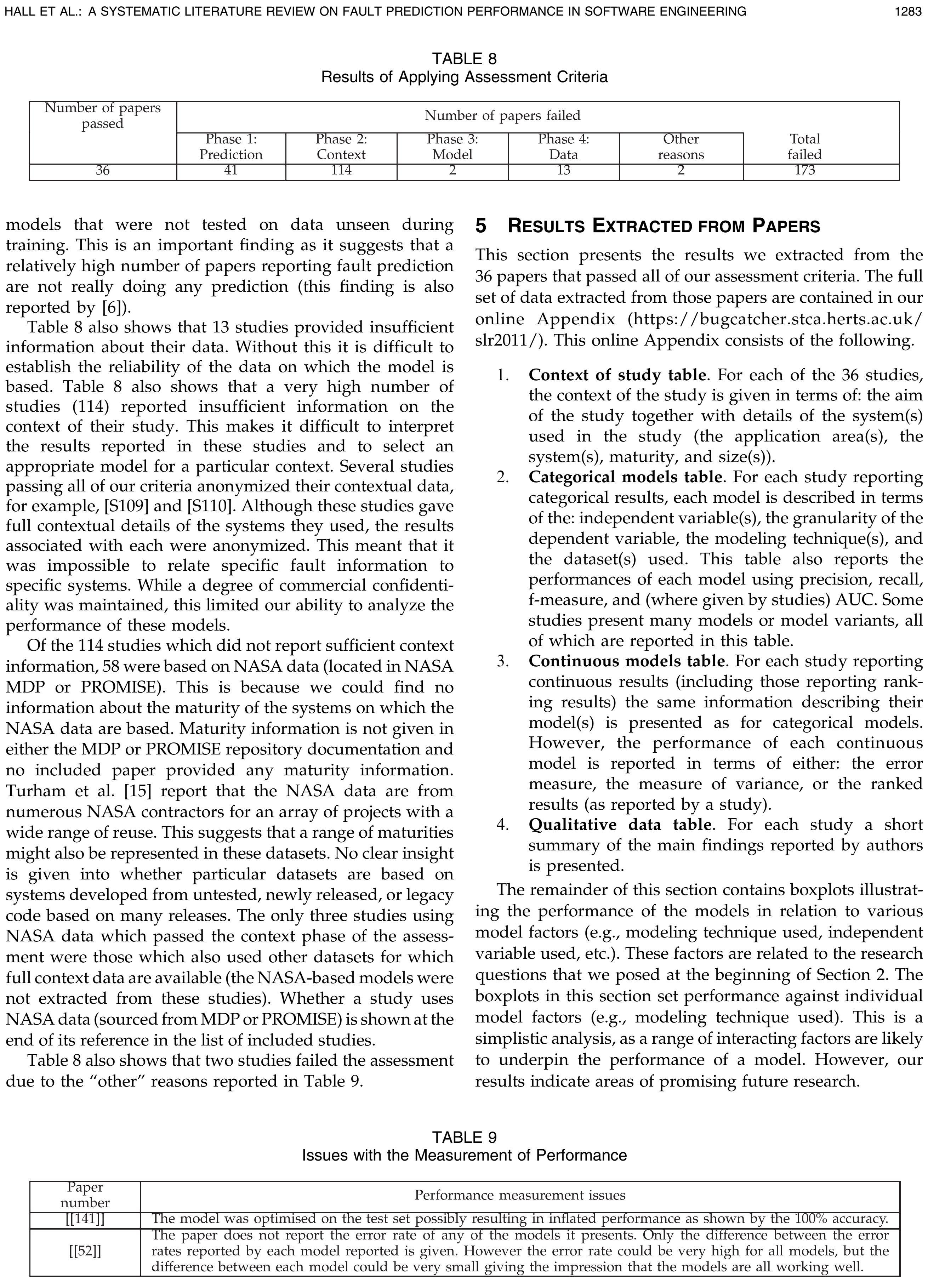


TABLE 8
Results of Applying Assessment Criteria

Number of papers passed	Number of papers failed					
	Phase 1: Prediction	Phase 2: Context	Phase 3: Model	Phase 4: Data	Other reasons	Total failed
36	41	114	2	13	2	173

models that were not tested on data unseen during training. This is an important finding as it suggests that a relatively high number of papers reporting fault prediction are not really doing any prediction (this finding is also reported by [6]).

Table 8 also shows that 13 studies provided insufficient information about their data. Without this it is difficult to establish the reliability of the data on which the model is based. Table 8 also shows that a very high number of studies (114) reported insufficient information on the context of their study. This makes it difficult to interpret the results reported in these studies and to select an appropriate model for a particular context. Several studies passing all of our criteria anonymized their contextual data, for example, [S109] and [S110]. Although these studies gave full contextual details of the systems they used, the results associated with each were anonymized. This meant that it was impossible to relate specific fault information to specific systems. While a degree of commercial confidentiality was maintained, this limited our ability to analyze the performance of these models.

Of the 114 studies which did not report sufficient context information, 58 were based on NASA data (located in NASA MDP or PROMISE). This is because we could find no information about the maturity of the systems on which the NASA data are based. Maturity information is not given in either the MDP or PROMISE repository documentation and no included paper provided any maturity information. Turham et al. [15] report that the NASA data are from numerous NASA contractors for an array of projects with a wide range of reuse. This suggests that a range of maturities might also be represented in these datasets. No clear insight is given into whether particular datasets are based on systems developed from untested, newly released, or legacy code based on many releases. The only three studies using NASA data which passed the context phase of the assessment were those which also used other datasets for which full context data are available (the NASA-based models were not extracted from these studies). Whether a study uses NASA data (sourced from MDP or PROMISE) is shown at the end of its reference in the list of included studies.

Table 8 also shows that two studies failed the assessment due to the "other" reasons reported in Table 9.

5 Results Extracted from Papers

This section presents the results we extracted from the 36 papers that passed all of our assessment criteria. The full set of data extracted from those papers are contained in our online Appendix (https://bugcatcher.stca.herts.ac.uk/slr2011/). This online Appendix consists of the following.

1. **Context of study table**. For each of the 36 studies, the context of the study is given in terms of: the aim of the study together with details of the system(s) used in the study (the application area(s), the system(s), maturity, and size(s)).
2. **Categorical models table**. For each study reporting categorical results, each model is described in terms of the: independent variable(s), the granularity of the dependent variable, the modeling technique(s), and the dataset(s) used. This table also reports the performances of each model using precision, recall, f-measure, and (where given by studies) AUC. Some studies present many models or model variants, all of which are reported in this table.
3. **Continuous models table**. For each study reporting continuous results (including those reporting ranking results) the same information describing their model(s) is presented as for categorical models. However, the performance of each continuous model is reported in terms of either: the error measure, the measure of variance, or the ranked results (as reported by a study).
4. **Qualitative data table**. For each study a short summary of the main findings reported by authors is presented.

The remainder of this section contains boxplots illustrating the performance of the models in relation to various model factors (e.g., modeling technique used, independent variable used, etc.). These factors are related to the research questions that we posed at the beginning of Section 2. The boxplots in this section set performance against individual model factors (e.g., modeling technique used). This is a simplistic analysis, as a range of interacting factors are likely to underpin the performance of a model. However, our results indicate areas of promising future research.

TABLE 9
Issues with the Measurement of Performance

Paper number	Performance measurement issues
[[141]]	The model was optimised on the test set possibly resulting in inflated performance as shown by the 100% accuracy.
[[52]]	The paper does not report the error rate of any of the models it presents. Only the difference between the error rates reported by each model reported is given. However the error rate could be very high for all models, but the difference between each model could be very small giving the impression that the models are all working well.

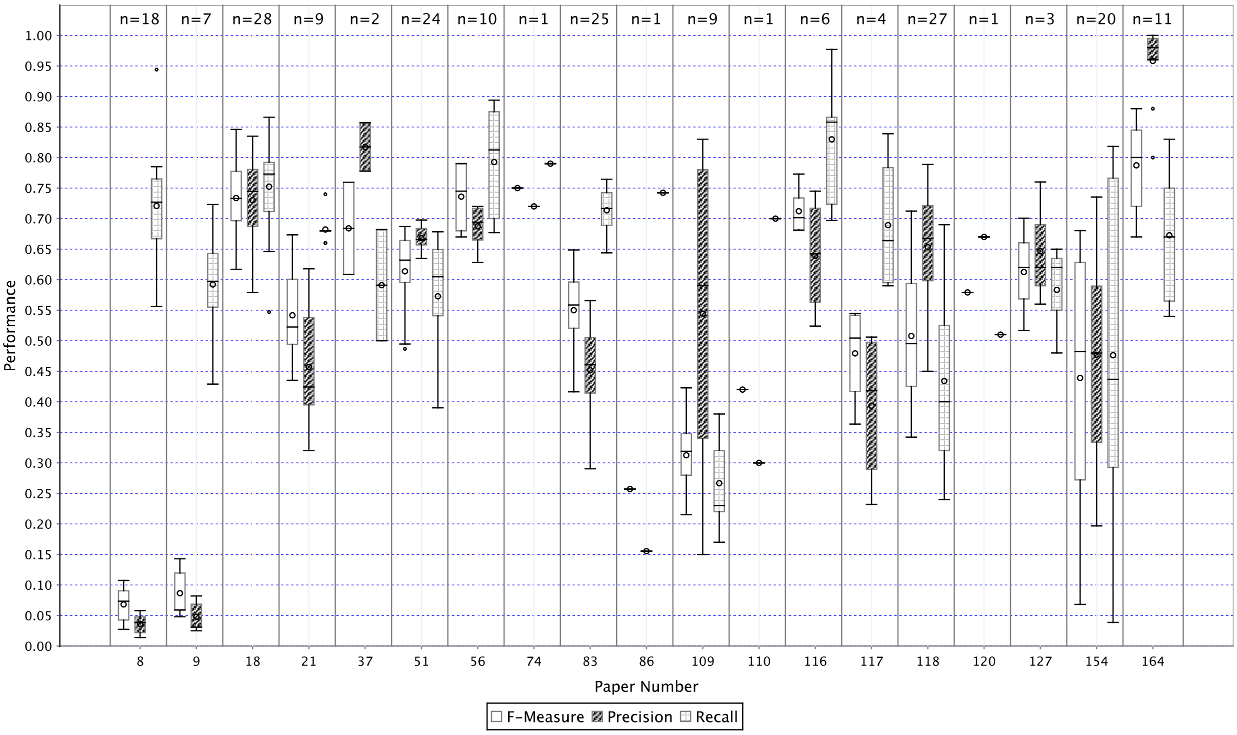


Fig. 1. Performances of the models reported in each of the categorical studies.

The boxplots represent models reporting only categorical results for which precision, recall, and f-measure were either reported or could be calculated by us. Such models are reported in 19 of the 23 categorical studies (of the remaining four, three report AUC). We are unable to present boxplots for the 13 studies using continuous data as the measures used are not comparable or convertible to comparable measures.

Each boxplot includes data only where at least three models have used a particular factor (e.g., a particular independent variable like LOC). This means that the numbers (n) at the top of the boxplots will not add up to the same number on every plot, as factors used in less than three studies will not appear; the total of ns will therefore vary from one boxplot to the next. The boxplots contain performance data based on precision, recall, and f-measure. This is for all categorical models and model variants presented by each study (206 models or model variants). Some studies present many model variants while others present only one model. We also created boxplots of only the best results from each study. These boxplots did not change the pattern of good performances but only presented limited information about poor performances. For that reason, we do not include these "best only" boxplots.

5.1 Performances of Models Reported in Individual Studies

Fig. 1 is a boxplot of the performances of all the models reported by each of the 19 categorical papers (full details of which can be found in the online Appendix). For each individual paper, f-measure, precision, and recall is reported. Fig. 1 shows that studies report on many models or variants of models, some with a wide range of performances (the details of these can be found in the Models Table in the online Appendix (https://bugcatcher.stca.herts.ac.uk/slr2011/)). For example, Schröter et al. [S154] present 20 model variants with a wide range of precision, recall, and f-measure. Many of these variants are not particularly competitive; the most competitive models that Schröter et al. [S154] report are based on training the model on only the faultiest parts of the system. This is a promising training technique and a similar technique has also been reported to be successful by Zhang et al. [S200]. Bird et al. [S18] report 28 model variants with a much smaller range of performances, all of which are fairly competitive. Fig. 1 also shows the performance tradeoffs in terms of precision and recall made by some models. For example, Bird et al. [S18] report consistent precision and recall, whereas Moser et al. [S118] and Shivaji et al. [S164] report performances where precision is much higher than recall.

Fig. 1 also shows that some models seem to be performing better than others. The models reported by Shivaji et al. [S164], based on Naive Bayes, performed extremely competitively. In general Naive Bayes performed relatively well, see Fig. 8. However, Shivaji et al. [S164] also used a good modeling process, including feature selection and appropriate measures derived during model training. In addition, their dataset contained a relatively large proportion of faulty components, making it fairly balanced. This may improve performance by providing many examples of faults from which the modeling technique can train. There are many good aspects of this study that mean it is likely to produce models which perform well.

On the other hand, the performance of Arisholm et al.'s models [S8], [S9] are low in terms of precision but

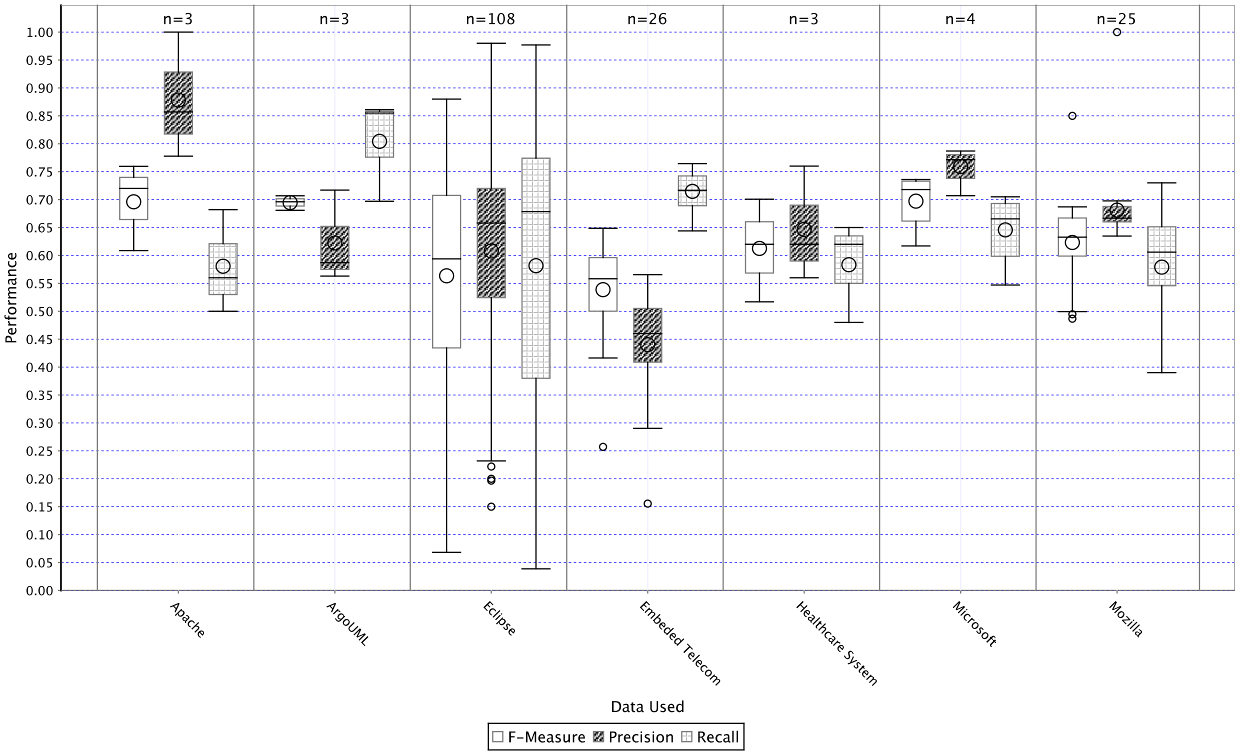


Fig. 2. Data used in models.

competitive in terms of recall. The two Arisholm et al. studies are different but use the same datasets. This low precision is reportedly because of the sampling method used to address the imbalance of the data used. Though the datasets used are also small relative to those used in other studies (148 KLOC), Arisholm et al.'s studies [S8], [S9] are interesting as they also report many good modeling practices and in some ways are exemplary studies. But they demonstrate how the data used can impact significantly on the performance of a model. It is also essential that both high and low performances be reported, as it is only by identifying these that our overall understanding of fault prediction will improve. The boxplots in the rest of this section explore in more detail aspects of models that may underpin these performance variations. Because the performances of Arisholm et al.'s models [S8], [S9] are very different from those of the other studies, we have removed them from the rest of the boxplots. We have treated them as outliers which would skew the results we report in other boxplots.

5.2 Performances in Relation to Context Factors

Fig. 2 shows the datasets used in the studies. It shows that 108 models reported in the studies are based on data from Eclipse. Eclipse is very well studied, probably because the fault data are easy to access and its utility has been well proven in previous studies. In addition, data already extracted from Eclipse are available from Saarland University (http://www.st.cs.uni-saarland.de/softevo/bug-data/eclipse/) and PROMISE (http://promisedata.org/). Fig. 2 shows that there is a wide variation in model performance using Eclipse. Fig. 2 also suggests that it may be more difficult to build models for some systems than for others. For example, the models built for embedded telecoms systems are not particularly competitive. This may be because such systems have a different profile of faults with fewer postdelivery faults relative to other systems. Developers of such systems normally prioritize reducing postdelivery faults as their embedded context makes fixing them comparatively expensive [S83].

Fig. 3 shows how models have performed relative to the size of systems on which they are based. Eclipse is the most common system used by studies. Consequently, Fig. 3 shows only the size of versions of Eclipse in relation to model performance. Fig. 3 suggests that as the size of a system increases, model performance seems to improve. This makes sense as models are likely to perform better given more data.

Fig. 4 shows the maturity of systems used by studies relative to the performance of models. The Context Table in the online Appendix shows how systems have been categorized in terms of their maturity. Fig. 4 shows that no immature systems are used by more than two models in this set of studies (i.e., where $n \geq 3$).[6] There seems to be little difference between the performance of models using mature or very mature systems. This suggests that the maturity of systems may not matter to predictive performance.[7] This finding may be linked to the finding we report on size. It may be that what was previously believed about the

[6] An exception to this is found in studies [S11], [S133], where immature systems are used with promising performances reported (see the online Appendix for full details).

[7] This may mean that it is not important to report maturity when studies describe their context (many more studies would have passed our assessment had that been the case). However, much more data on maturity is needed before firm conclusions can be drawn.

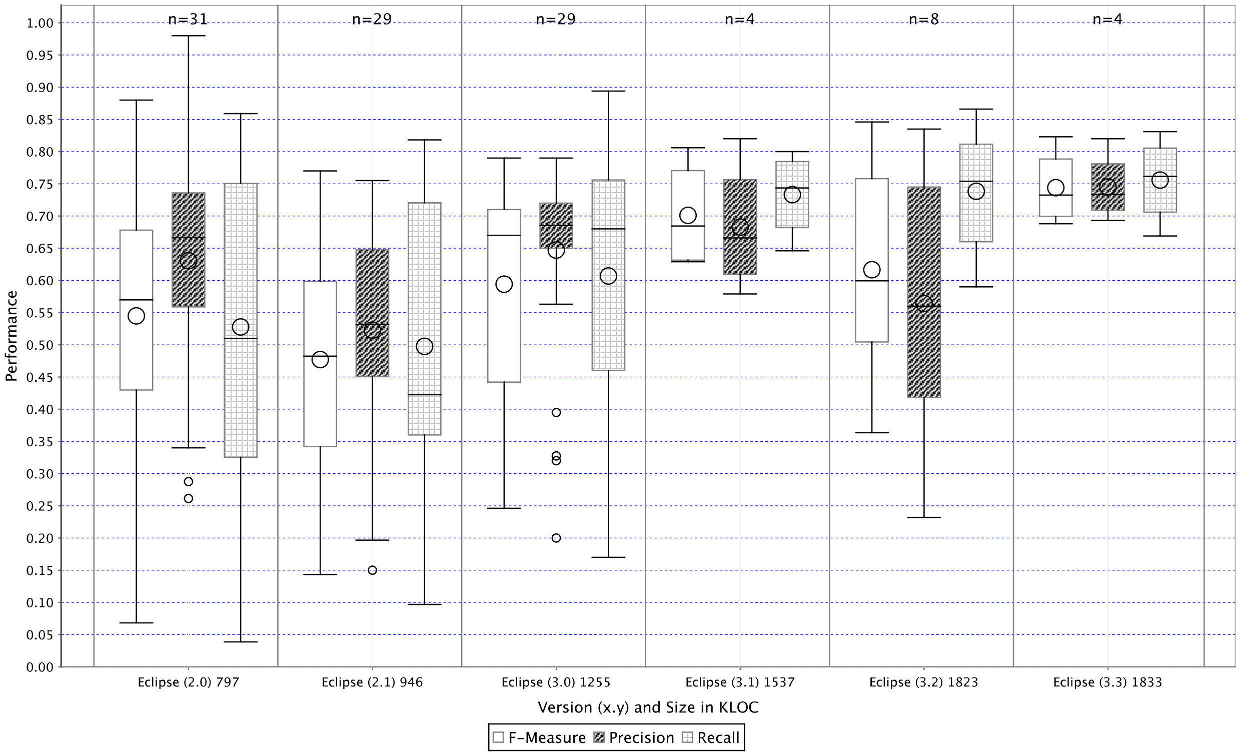


Fig. 3. The size of the datasets used for Eclipse.

importance of maturity was actually about size, i.e., maturity is a surrogate for size. Indeed, there is a significant relationship between size and maturity in the data we report here. However, we do not have enough data to draw firm conclusions as the data we analyze contain no studies using immature systems. More research is needed to test for possible association between maturity and size and whether data extracted from immature systems can be used as a basis for reliable fault prediction.

Fig. 5 shows the language used in the systems studied in relation to the performance of models. We present only studies reporting the use of either Java or C/C++. There are several single studies using other languages which we do not report. Fig. 5 suggests that model performance is not related to the language used.

Fig. 6 shows model performance relative to the granularity of dependent variables (e.g., whether fault prediction is at the class or file level). It shows no clear relationship between granularity and performance. It does not seem to be the case that higher granularity is clearly related to improved performance. Models reporting at "other" levels of granularity seem to be performing most consistently. These tend to be high levels of granularity defined specifically by individual studies (e.g., Nagappan et al. [S120]).

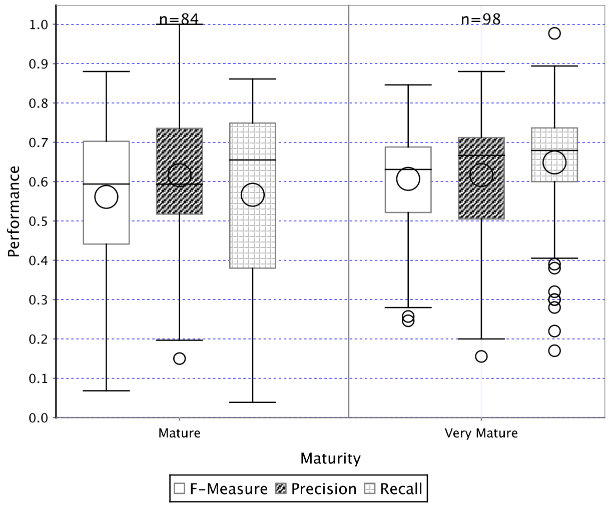


Fig. 4. The maturity of the systems used.

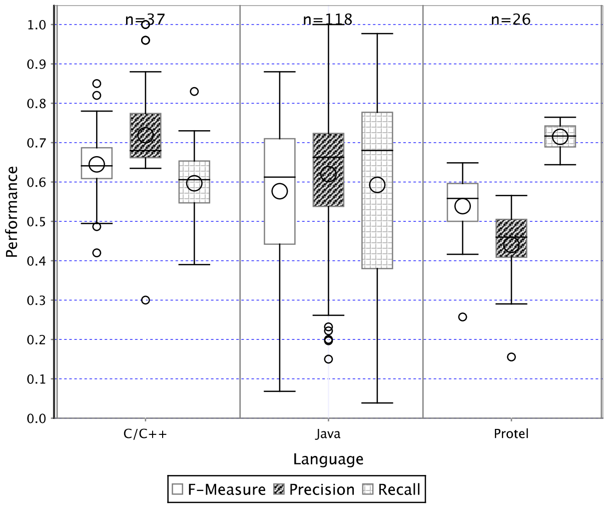
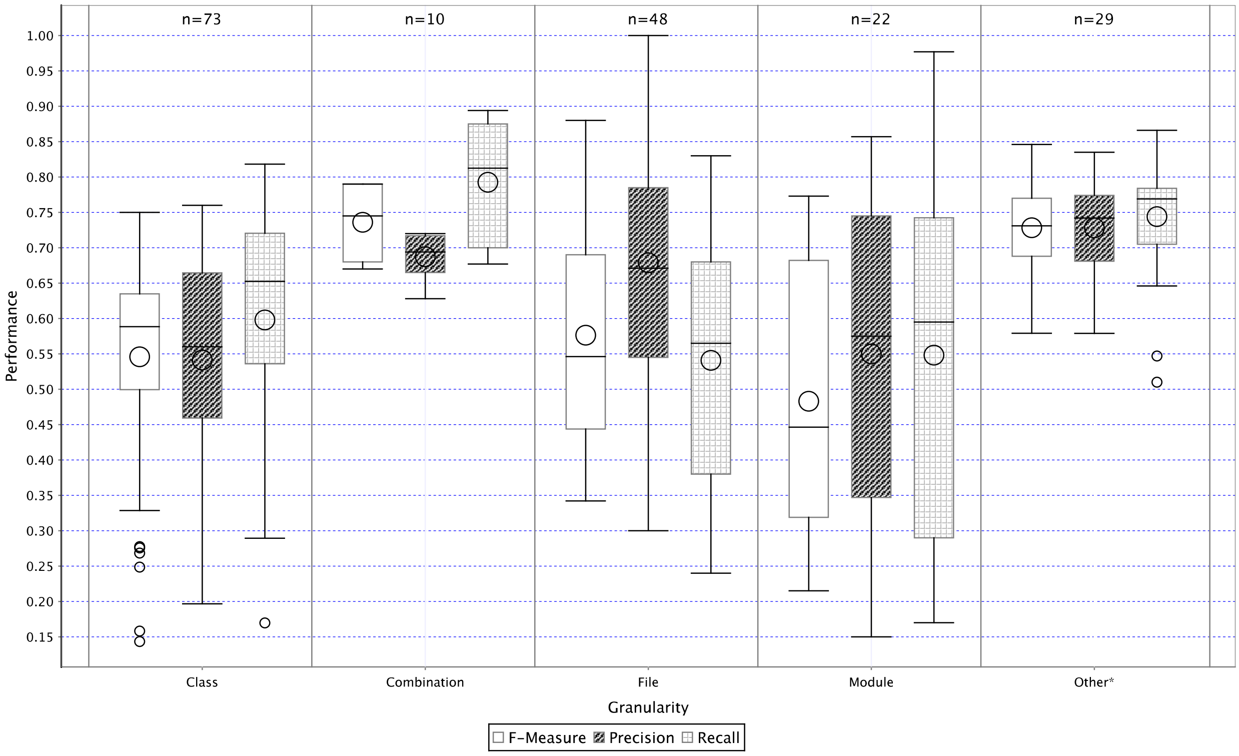


Fig. 5. The language used.



*For example plug-ins, binaries

Fig. 6. The granularity of the results.

5.3 Performance in Relation to Independent Variables

Fig. 7 shows model performance in relation to the independent variables used. The Categorical Models Table in the online Appendix shows how independent variables as expressed by individual studies have been categorized in relation to the labels used in Fig. 7. It shows that there is variation in performance between models using different independent variables. Models using a wide combination of metrics seem to be performing well. For example, models using a combination of static code metrics (scm), process metrics, and source code text seem to be performing best overall (e.g., Shivaji et al. [S164]). Similarly Bird et al.'s study [S18], which uses a wide combination of socio-technical metrics (code dependency data together with change data and developer data), also performs well (though the results from Bird et al.'s study [S18] are reported at a high level of granularity). Process metrics (i.e., metrics based on changes logged in repositories) have not performed as well as expected. OO metrics seem to have been used in studies which perform better than studies based only on other static code metrics (e.g., complexity-based metrics). Models using only LOC data seem to have performed competitively compared to models using other independent variables. Indeed, of these models using only metrics based on static features of the code (OO or SCM), LOC seems as good as any other metric to use. The use of source code text seems related to good performance. Mizuno et al.'s studies [S116], [S117] have used only source code text within a novel spam filtering approach to relatively good effect.

5.4 Performance in Relation to Modeling Technique

Fig. 8 shows model performance in relation to the modeling techniques used. Models based on Naive Bayes seem to be performing well overall. Naive Bayes is a well understood technique that is in common use. Similarly, models using Logistic Regression also seem to be performing well. Models using Linear Regression perform not so well, though this technique assumes that there is a linear relationship between the variables. Studies using Random Forests have not performed as well as might be expected (many studies using NASA data use Random Forests and report good performances [S97]). Fig. 8 also shows that SVM (Support Vector Machine) techniques do not seem to be related to models performing well. Furthermore, there is a wide range of low performances using SVMs. This may be because SVMs are difficult to tune and the default Weka settings are not optimal. The performance of models using the C4.5 technique is fairly average. However, Arisholm et al.'s models [S8], [S9] used the C4.5 technique (as previously explained, these are not shown as their relatively poor results skew the data presented). C4.5 is thought to struggle with imbalanced data [16] and [17] and this may explain the performance of Arisholm et al.'s models.

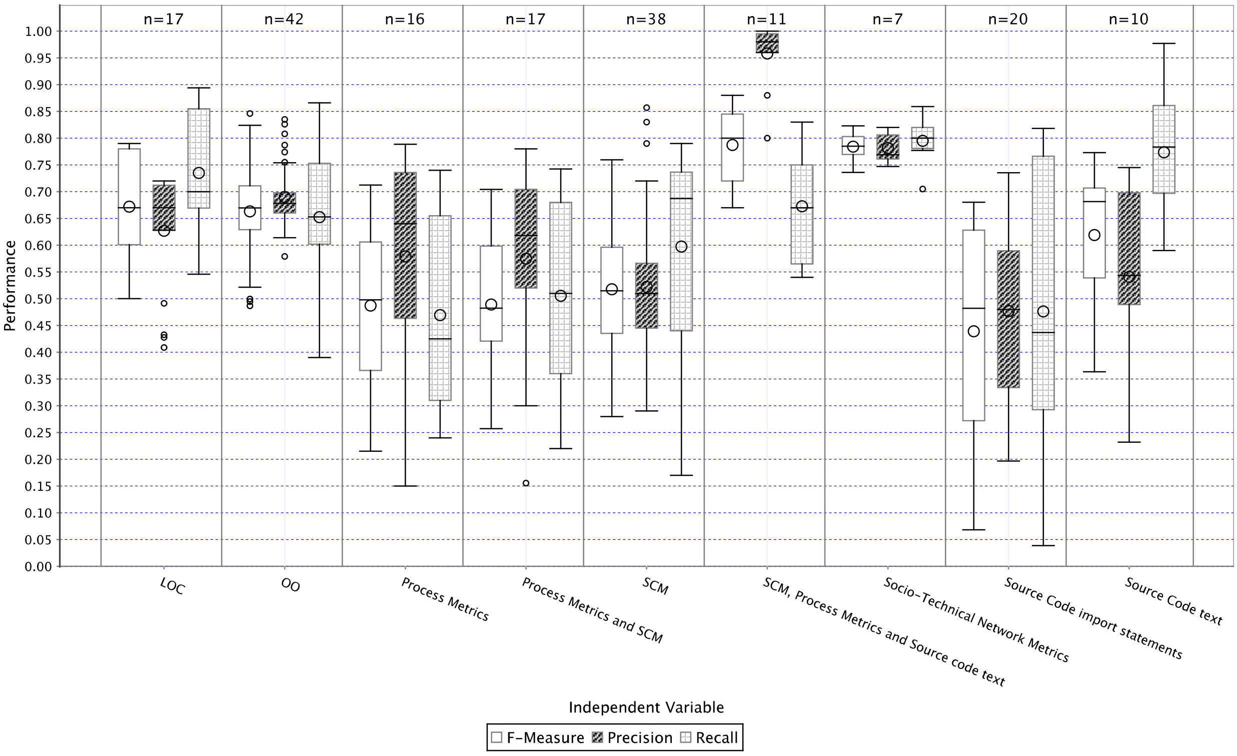


Fig. 7. Independent variables used in models.

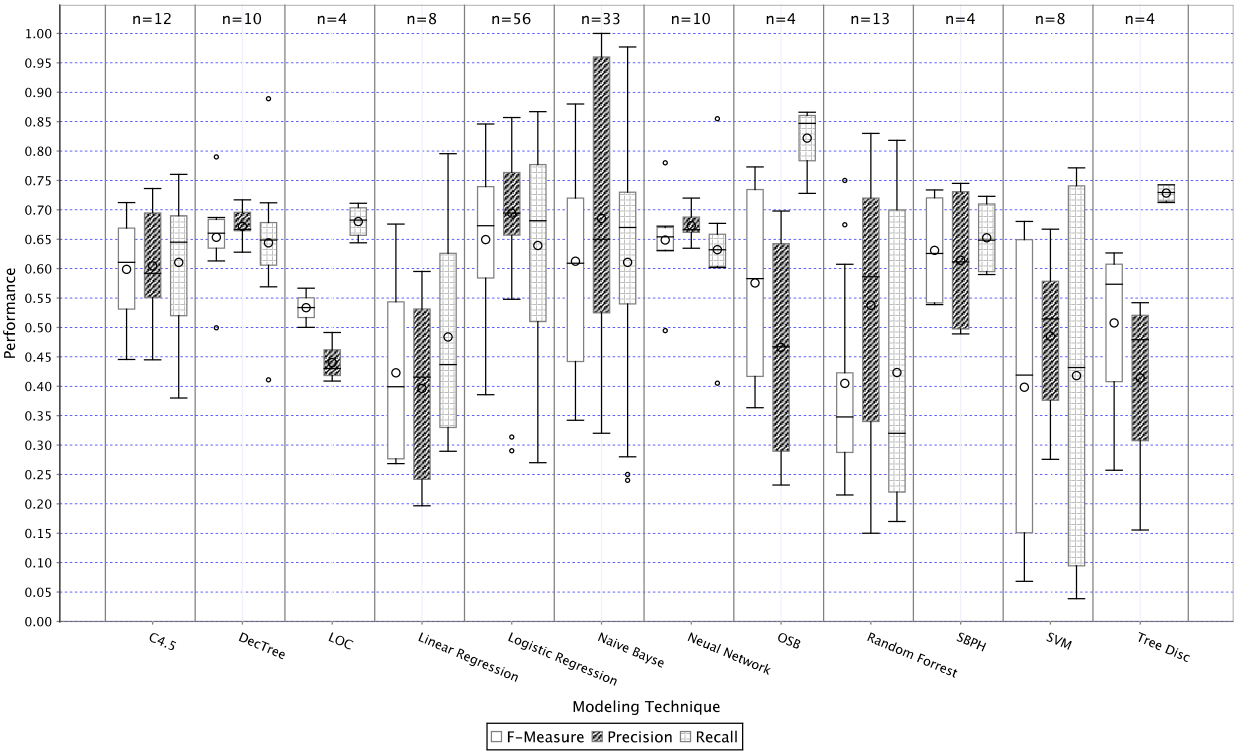


Fig. 8. Modeling technique used.

6 Synthesis of Results

This section answers our research questions by synthesizing the qualitative and quantitative data we have collected. The qualitative data consist of the main findings reported by each of the individual 36 finally included studies (presented in the Qualitative Data Table in our online Appendix). The quantitative data consist of the predictive performance of the individual models reported in the 36 studies (summarized in the Categorical and Continuous Models Tables in our online Appendix). The quantitative data also consist of the detailed predictive performance data from 19 studies (206 models or model variants) comparing performance across models (reported in Section 5). This combination of data addresses model performance across studies and within individual studies. This allows us to discuss model performance in two ways. First, we discuss performance within individual studies to identify the main influences on model performance reported within a study. Second, we compare model performances across the models reported in 19 studies. This is an important approach to discussing fault prediction models. Most studies report at least one model which performs "well." Though individual studies usually only compare performance within the set of models they present to identify their best model, we are able to then compare the performance of the models which perform well within a study across other studies. This allows us to report how well these models perform across studies.

6.1 Answering our Research Questions

RQ1: How does context affect fault prediction?

Analyzing model performance across the 19 studies in detail suggests that some context variables may influence the reliability of model prediction. Our results provide some evidence to suggest that predictive performance improves as systems get larger. This is suggested by the many models built for the Eclipse system. As Eclipse increases in size, the performance of models seems to improve. This makes some sense as models are likely to perform better with more data. We could find no evidence that this improved performance was based on the maturing of systems. It may be that size influences predictive performance more than system maturity. However, our dataset is relatively small and although we analyzed 206 models (or model variants) very few were based on immature systems. Our results also suggest that some applications may be less likely to produce reliable prediction models. For example, the many models built for embedded telecoms applications generally performed less well relative to other applications. Our results also show that many models have been built using Eclipse data. This corpus of knowledge on Eclipse provides a good opportunity for future researchers to meta-analyze across a controlled context.

The conventional wisdom is that context determines how transferrable a model is to other systems. Despite this, none of the 36 finally included studies directly investigate the impact on model performance of specific context variables such as system size, maturity, application area, or programming language. One exception is [S29], which demonstrates that transforming project data can make a model more comparable to other projects.

Many of the 36 finally included studies individually test how well their model performs when transferred to other contexts (releases, systems, application areas, data sources, or companies). Few of these studies directly investigate the contextual factors influencing the transferability of the model. Findings reported from individual studies on model transferability are varied. Most studies report that models perform poorly when transferred. In fact, Bell et al. [S11] report that models could not be applied to other systems. Denaro and Pezzè [S37] reported good predictive performance only across homogenous applications. Nagappan et al. [S122] report that different subsets of complexity metrics relate to faults in different projects and that no single set of metrics fits all projects. Nagappan et al. [S122] conclude that models are only accurate when trained on the same or similar systems. However, other studies report more promising transferability. Weyuker et al. [S190] report good performance when models are transferred between releases of systems and between other systems. However, Shatnawi and Li [S160] report that the performance of models declines when applied to later releases of a system. Shatnawi and Li [S160] conclude that different metrics should be used in models used for later releases.

The context of models has not been studied extensively in the set of studies we analyzed. Although every model has been developed and tested within particular contexts, the impact of that context on model performance is scarcely studied directly. This is a significant gap in current knowledge as it means we currently do not know what context factors influence how well a model will transfer to other systems. It is therefore imperative that studies at least report their context since, in the future, this will enable a meta-analysis of the role context plays in predictive performance.

RQ2: Which independent variables should be included in fault prediction models?

Many different independent variables have been used in the 36 finally included studies. These mainly fall into process (e.g., previous change and fault data) and product (e.g., static code data) metrics as well as metrics relating to developers. In addition, some studies have used the text of the source code itself as the independent variables (e.g., Mizuno et al. [S116], Mizuno and Kikuno [S117]).

Model performance across the 19 studies we analyzed in detail suggests that the spam filtering technique, based on source code, used by Mizuno et al. [S116], Mizuno and Kikuno [S117] performs relatively well. On the other hand, models using only static code metrics (typically complexity-based) perform relatively poorly. Model performance does not seem to be improved by combining these metrics with OO metrics. Models seem to perform better using only OO metrics rather than only source code metrics. However, models using only LOC seem to perform just as well as those using only OO metrics and better than those models only using source code metrics. Within individual studies, Zhou et al. [S203] report that LOC data performs well. Ostrand et al. [S133] report that there was some value in LOC data and Hongyu [S56] reports LOC to be a useful early general indicator of fault-proneness. Zhou et al. [S203] report that LOC performs better than all but one of the Chidamber and Kemerer metrics (Weighted Methods per Class). Within other individual studies LOC data were reported to have poor predictive power and to be outperformed by other metrics (e.g., Bell et al. [S11]). Overall, LOC seem to be generally useful in fault prediction.

Model performance across the 19 studies that we analyzed suggests that the use of process data is not particularly related to good predictive performance. However, looking at the findings from individual studies, several authors report that process data, in the form of previous history data, performs well (e.g., [S163], [S120]). D'Ambros et al. [S31] specifically report that previous bug reports are the best predictors. More sophisticated process measures have also been reported to perform well. In particular, Nagappan et al. [S120] introduce "change burst" metrics which demonstrate good predictive performance (however, these models perform only moderately when we compared them against models from other studies).

The few studies using developer information in models report conflicting results. Ostrand et al. [S135] report that the addition of developer information does not improve predictive performance much. Bird et al. [S18] report better performances when developer information is used as an element within a socio-technical network of variables. This study also performs well in our detailed comparison of performances (Bird et al. [S18] report results at a high level of granularity and so might be expected to perform better).

The models which perform best in our analysis of 19 studies seem to use a combined range of independent variables. For example, Shivaji et al. [S164] use process-based and SCM-based metrics together with source code. Bird et al. [S18] combine a range of metrics. The use of feature selection on sets of independent variables seems to improve the performance of models (e.g., [S164], [S76], [S18]). Optimized sets of metrics using, for example, feature selection, make sense.

RQ3: Which modeling techniques perform best when used in fault prediction?

While many included studies individually report the comparative performance of the modeling techniques they have used, no clear consensus on which perform best emerges when individual studies are looked at separately. Mizuno and Kikuno [S117] report that, of the techniques they studied, Orthogonal Sparse Bigrams Markov models (OSB) are best suited to fault prediction. Bibi et al. [S15] report that Regression via Classification (RvC) works well. Khoshgoftaar et al. [S86] report that modules whose fault proneness is predicted as uncertain can be effectively classified using the TreeDisc (TD) technique. Khoshgoftaar and Seliya [S83] also report that Case-Based Reasoning (CBR) does not predict well, with C4.5 also performing poorly. Arisholm et al. [S9] report that their comprehensive performance comparison revealed no predictive differences between the eight modeling techniques they investigated.

A clearer picture seems to emerge from our detailed analysis of model performance across the 19 studies. Our findings suggest that performance may actually be linked to the modeling technique used. Overall our comparative analysis suggests that studies using Support Vector Machine (SVM) techniques perform less well. These may be underperforming as they require parameter optimization (something rarely carried out in fault prediction studies) for best performance [18]. Where SVMs have been used in other prediction domains and may be better understood, they have performed well [19]. Models based on C4.5 seem to underperform if they use imbalanced data (e.g., Arisholm et al. [S8], [S9]), as the technique seems to be sensitive to this. Our comparative analysis also suggests that the models performing comparatively well are relatively simple techniques that are easy to use and well understood. Naive Bayes and Logistic regression, in particular, seem to be the techniques used in models that are performing relatively well. Models seem to have performed best where the right technique has been selected for the right set of data. And these techniques have been tuned to the model (e.g., Shivaji et al. [S164]), rather than relying on default tool parameters.

7 Methodological Issues in Fault Prediction

The methodology used to develop, train, test, and measure the performance of fault prediction models is complex. However, the efficacy of the methodology used underpins the confidence which we can have in a model. It is essential that models use and report a rigorous methodology. Without this, the maturity of fault prediction in software engineering will be low. We identify methodological problems in existing studies so that future researchers can improve on these.

Throughout this SLR, methodological issues in the published studies came to light. During our assessment of the 208 initially included studies and the extraction of data from the 36 finally included studies methodological weaknesses emerged. In this section, we discuss the most significant of these methodological weaknesses. These generally relate to the quality of data used to build models and the approach taken to measure the predictive performance of models.

7.1 Data Quality

The quality of the data used in fault prediction has significant potential to undermine the efficacy of a model. Data quality is complex and many aspects of data are important to ensure reliable predictions. Unfortunately, it is often difficult to assess the quality of data used in studies, especially as many studies report very little about the data they use. Without good quality data, clearly reported, it is difficult to have confidence in the predictive results of studies.

The results of our assessment show that data quality is an issue in many studies. In fact many studies failed our synthesis assessment on the basis that they either reported insufficient information about the context of their data or about the collection of that data. Some studies explicitly acknowledge the importance of data quality (e.g., Jiang et al. [S64]).

Collecting good quality data is very hard. This is partly reflected by the number of studies which failed our assessment by not adequately explaining how they had collected their independent or dependent data. Fault data collection has been previously shown to be particularly hard to collect, usually because fault data are either not directly recorded or recorded poorly [20]. Collecting data is made more challenging because large datasets are usually necessary for reliable fault prediction. Jiang et al. [S64] investigate the impact that the size of the training and test dataset has on the accuracy of predictions. Tosun et al. [S176] present a useful insight into the real challenges associated with every aspect of fault prediction, but particularly on the difficulties of collecting reliable metrics and fault data. Once collected, data is usually noisy and often needs to be cleaned (e.g., outliers and missing values dealt with [21]). Very few studies report any data cleaning (even in our 36 finally included studies).

The balance of data (i.e., the number of faulty as opposed to nonfaulty units) on which models are trained and tested is acknowledged by a few studies as fundamental to the reliability of models (see Appendix F for more information on class imbalance). Indeed, across the 19 studies we analyzed in detail, some of those performing best are based on data with a good proportion of faulty units (e.g., [S164], [S37], [S11], [S74]). Our analysis also suggests that data imbalance in relation to specific modeling techniques (e.g., C4.5) may be related to poor performance (e.g., [S8], [S9]). Several studies specifically investigated the impact of data balance and propose techniques to deal with it. For example, Khoshgoftaar et al. [S76] and Shivaji et al. [S164] present techniques for ensuring reliable data distributions. Schröter et al. [S154] base their training set on the faultiest parts of the system. Similarly, Seiffert et al. [S156] present data sampling and boosting techniques to address data imbalance. Data imbalance is explored further in Fioravanti and Nesi [S43] and Zhang et al. [S200]. Many studies seem to lack awareness of the need to account for data imbalance. As a consequence, the impact of imbalanced data on the real performance of models can be hidden by the performance measures selected. This is especially true where the balance of data is not even reported. Readers are then not able to account for the degree of imbalanced data in their interpretation of predictive performance.

7.2 Measuring the Predictive Performance of Models

There are many ways in which the performance of a prediction model can be measured. Indeed, many different categorical and continuous performance measures are used in our 36 studies. There is no one best way to measure the performance of a model. This depends on: the class distribution of the training data, how the model has been built, and how the model will be used. For example, the importance of measuring misclassification will vary depending on the application.

Performance comparison across studies is only possible if studies report a set of uniform measures. Furthermore, any uniform set of measures should give a full picture of correct and incorrect classification. To make models reporting categorical results most useful, we believe that the raw confusion matrix on which their performance measures are derived should be reported. This confusion matrix data would allow other researchers and potential users to calculate the majority of other measures. Pizzi et al. [22] provide a very usable format for presenting a confusion matrix. Some studies present many models and it is not practical to report the confusion matrices for all these. Menzies et al. [S114] suggest a useful way in which data from multiple confusion matrices may be effectively reported. Alternatively, Lessmann [S97] recommends that ROC curves and AUC are most useful when comparing the ability of modeling techniques to cope with different datasets (ROC curves do have some limitations [23]). Either of these approaches adopted widely would make studies more useful in the future. Comparing across studies reporting continuous results is currently even more difficult and is the reason we were unable to present comparative boxplots across these studies. To enable cross comparison we recommend that continuous studies report Average Relative Error (ARE) in addition to any preferred measures presented.

The impact of performance measurement has been picked up in many studies. Zhou et al. [S203] report that the use of some measures, in the context of a particular model, can present a misleading picture of predictive performance and undermine the reliability of predictions. Arisholm et al. [S9] discuss how model performance varies depending on how it is measured. There is an increasing focus on identifying effective ways to measure the performance of models. Cost and/or effort aware measurement is now a significant strand of interest in prediction measurement. This takes into account the cost/effort of falsely identifying modules and has been increasingly reported as useful. The concept of cost-effectiveness measurement originated with the Simula group (e.g., Arisholm et al. [S9]), but has more recently been taken up and developed by other researchers, for example, Nagappan et al. [S120] and Mende and Koschke [S109].

7.3 Fault Severity

Few studies incorporate fault severity into their measurement of predictive performance. Although some faults are more important to identify than others, few models differentiate between the faults predicted. In fact, Shatnawi and Li's [S160] was the only study in the final 36 to use fault severity in their model. They report a model which is able to predict high and medium severity faults (these levels of severity are based on those reported in Bugzilla by Eclipse developers). Lamkanfi et al. [24], Singh et al. [S167], and Zhou and Leung [S202] are other studies which have also investigated severity. This lack of studies that consider severity is probably because, although acknowledged to be important, severity is considered a difficult concept to measure. For example, Menzies et al. [S113] say that severity is too vague to reliably investigate, Nikora and Munson [S126] says that "without a widely agreed definition of severity we cannot reason about it" and Ostrand et al. [S133] state that severity levels are highly subjective and can be inaccurate and inconsistent. These problems of how to measure and collect reliable severity data may limit the usefulness of fault prediction models. Companies developing noncritical systems may want to prioritize their fault finding effort only on the most severe faults.

7.4 The Reporting of Fault Prediction Studies

Our results suggest that, overall, fault prediction studies are reported poorly. Out of the 208 studies initially included in our review, only 36 passed our assessment criteria. Many of these criteria are focused on checking that studies report basic details about the study. Without a basic level of information reported it is hard to have confidence in a study. Our results suggest that many studies are failing to report information which is considered essential when reporting empirical studies in other domains. The poor reporting of studies has consequences for both future researchers and potential users of models: It is difficult for researchers to meta-analyze across studies and it is difficult to replicate studies; it is also difficult for users to identify suitable models for implementation.

7.5 NASA Data

NASA's publicly available software metrics data have proven very popular in developing fault prediction models. We identify all 62 studies which use NASA data in the reference list of the 208 included studies. The NASA data is valuable as it enables studies using different modeling techniques and independent variables to be compared to others using the same dataset. It also allows studies to be replicated. A meta-analysis of the studies using NASA data would be valuable. However, although the repository holds many metrics and is publicly available, it does have limitations. It is not possible to explore the source code and the contextual data are not comprehensive (e.g., no data on maturity are available). It is also not always possible to identify if any changes have been made to the extraction and computation mechanisms over time. In addition, the data may suffer from important anomalies [21]. It is also questionable whether a model that works well on the NASA data will work on a different type of system; as Menzies et al. [S112] point out, NASA works in a unique niche market, developing software which is not typical of the generality of software systems. However, Turhan et al. [S181] have demonstrated that models built on NASA data are useful for predicting faults in software embedded in white goods.

8 Threats to Validity

Searches. We do not include the term "quality" in our search terms as this would have resulted in the examination of a far wider range of irrelevant papers. This term generates a high number of false positive results. We might have missed some papers that use the term "quality" as a synonym for "defect" or "fault," etc. However, we missed only two papers that Catal and Diri's [2] searches found using the term "quality." This gives us confidence that we have missed very few papers. We also omitted the term "failure" from our search string as this generated papers predominately reporting on studies of software reliability in terms of safety critical systems. Such studies of reliability usually examine the dynamic behavior of the system and seldom look at the prediction of static code faults, which is the focus of this review.

We apply our search terms to only the titles of papers. We may miss studies that do not use these terms in the title. Since we extend our searches to include papers cited in the included papers, as well as key conferences, individual journals, and key authors, we are confident that the vast majority of key papers have been included.

Studies included for synthesis. The 36 studies which passed our assessment criteria may still have limitations that make their results unreliable. In the first place, the data on which these models are built might be problematic as we did not insist that studies report data cleaning or attribute selection. Nor did we apply any performance measure-based criteria. So some studies may be reporting unsafe predictive performances. This is a particular risk in regard to how studies have accounted for using imbalanced data. This risk is mitigated in the categorical studies, where we are able to report precision, recall, and f-measure.

It is also possible that we have missed studies which should have been included in the set of 36 from which we extracted data. Some studies may have satisfied our assessment criteria but either failed to report what they did or did not report it in sufficient detail for us to be confident that they should pass the criteria. Similarly, we may have missed the reporting of a detail and a paper that should have passed a criterion did not. These risks are mitigated by two authors independently assessing every study.

The boxplots. The boxplots we present set performance against individual model factors (e.g., modeling technique used). This is a simplistic analysis, as a number of interacting factors are likely to underpin the performance of a model. For example, the technique used in combination with the dataset and the independent variables is likely to be more important than any one factor alone. Furthermore, methodological issues are also likely to impact on performance; for example, whether feature selection has been used. Our boxplots only present possible indicators of factors that should be investigated within the overall context of a model. More sophisticated analysis of a larger dataset is needed to investigate factors influencing model performance.

Our boxplots do not indicate the direction of any relationship between model performance and particular model factors. For example, we do not investigate whether a particular modeling technique performs well because it was used in a good model or whether a model performs well because it used a particular modeling technique. This is also important work for the future. In addition, some studies contribute data from many models to one boxplot, whereas other studies contribute data from only one model. This may skew the results. We do not calculate the statistical significance of any differences observed in the boxplots. This is because the data contained within them are not normally distributed and the individual points represent averages from different sizes of population.

9 Conclusions

Fault prediction is an important topic in software engineering. Fault prediction models have the potential to improve the quality of systems and reduce the costs associated with delivering those systems. As a result of this, many fault prediction studies in software engineering have been published. Our analysis of 208 of these studies shows that the vast majority are less useful than they could be. Most studies report insufficient contextual and methodological information to enable full understanding of a model. This makes it difficult for potential model users to select a model to match their context and few models have transferred into industrial practice. It also makes it difficult for other researchers to meta-analyze across models to identify the influences on predictive performance. A great deal of effort has gone into models that are of limited use to either practitioners or researchers.

The set of criteria we present identify a set of essential contextual and methodological details that fault prediction studies should report. These go some way toward addressing the need identified by Myrtveit et al. [25] for "more reliable research procedures before we can have confidence in the conclusions of comparative studies." Our criteria should be used by future fault prediction researchers. They should also be used by journal and conference reviewers. This would ensure that future studies are built reliably and reported comparably with other such reliable studies. Of the 208 studies we reviewed, only 36 satisfied our criteria and reported essential contextual and methodological details.

We analyzed these 36 studies to determine what impacts on model performance in terms of the context of models, the independent variables used by models, and the modeling techniques on which they were built. Our results suggest that models which perform well tend to be built in a context where the systems are large. We found no evidence that the maturity of systems or the language used is related to predictive performance. But we did find some evidence to suggest that some application domains (e.g., embedded systems) may be more difficult to build reliable prediction models for. The independent variables used by models performing well seem to be sets of metrics (e.g., combinations of process, product, and people-based metrics). We found evidence that where models use KLOC as their independent variable, they perform no worse than where only single sets of other static code metrics are used. In addition, models which perform well tend to use simple, easy to use modeling techniques like Naive Bayes or Logistic Regression. More complex modeling techniques, such as support vector machines, tend to be used by models which perform relatively less well.

The methodology used to build models seems to be influential to predictive performance. The models which performed well seemed to optimize three aspects of the model. First, the choice of data was optimized. In particular, successful models tend to be trained on large datasets which contain a relatively high proportion of faulty units. Second, the choice of independent variables was optimized. A large range of metrics were used on which feature selection was applied. Third, the modeling technique was optimized. The default parameters were adjusted to ensure that the technique would perform effectively on the data provided.

Overall we conclude that many good fault prediction studies have been reported in software engineering (e.g., the 36 which passed our assessment criteria). Some of these studies are of exceptional quality, for example, Shivaji et al. [S164]. However, there remain many open questions about how to build effective fault prediction models for software systems. We need more studies which are based on a reliable methodology and which consistently report the context in which models are built and the methodology used to build them. A larger set of such studies will enable reliable cross-study metaanalysis of model performance. It will also give practitioners the confidence to appropriately select and apply models to their systems. Without this increase in reliable models that are appropriately reported, fault prediction will continue to have limited impact on the quality and cost of industrial software systems.

References for the 208 Included SLR Papers [S1-S208]

References from this list are cited using the format [Sref#]). Each reference is followed by a code indicating the status of the paper in terms of whether it passed (P) or failed (F) our assessment. An indication is also given as to the assessment phase a paper failed (1, 2, 3, 4, or 5). The use of NASA data by studies is also indicated (N). A paper (n) failing an assessment criterion in phase 2 which used NASA data would be coded: (Paper=n; Status=F, Phase=2, Data=N)

Please report possible problems with our assessment of these papers to: tracy.hall@brunel.ac.uk.

[1] R. Abreu and R. Premraj, "How Developer Communication Frequency Relates to Bug Introducing Changes," *Proc. Joint Int'l and Ann. ERCIM Workshops Principles of Software Evolution and Software Evolution Workshops,* pp. 153-158, 2009. (Paper=1, Status=F, Phase=1).

[2] W. Afzal, "Using Faults-Slip-Through Metric as a Predictor of Fault-Proneness," *Proc. 17th Asia Pacific Software Eng. Conf.,* pp. 414-422, 2010. (Paper=2, Status=F, Phase=2).

[3] W. Afzal and R. Torkar, "A Comparative Evaluation of Using Genetic Programming for Predicting Fault Count Data," *Proc. Third Int'l Conf. Software Eng. Advances,* pp. 407-414, 2008. (Paper=3, Status=F, Phase=2).

[4] W. Afzal, R. Torkar, and R. Feldt, "Prediction of Fault Count Data Using Genetic Programming," *Proc. IEEE Int'l Multitopic Conf.,* pp. 349-356, 2008. (Paper=4, Status=F, Phase=2).

[5] S. Amasaki, Y. Takagi, O. Mizuno, and T. Kikuno, "A Bayesian Belief Network for Assessing the Likelihood of Fault Content," *Proc. 14th Int'l Symp. Software Reliability Eng.,* pp. 215-226, Nov. 2003. (Paper=5, Status=F, Phase=2).

[6] C. Andersson and P. Runeson, "A Replicated Quantitative Analysis of Fault Distributions in Complex Software Systems," *IEEE Trans. Software Eng.,* vol. 33, no. 5, pp. 273-286, May 2007. (Paper=6, Status=F, Phase=1).

[7] E. Arisholm and L. Briand, "Predicting Fault-Prone Components in a Java Legacy System," *Proc. ACM/IEEE Int'l Symp. Empirical Software Eng.,* pp. 8-17, 2006. (Paper=7, Status=F, Phase=4).

[8] E. Arisholm, L.C. Briand, and M. Fuglerud, "Data Mining Techniques for Building Fault-Proneness Models in Telecom Java Software," *Proc. IEEE 18th Int'l Symp. Software Reliability,* pp. 215-224, Nov. 2007. (Paper=8, Status=P).

[9] E. Arisholm, L.C. Briand, and E.B. Johannessen, "A Systematic and Comprehensive Investigation of Methods to Build and Evaluate Fault Prediction Models," *J. Systems and Software,* vol. 83, no. 1, pp. 2-17, 2010. (Paper=9, Status=P).

[10] N. Ayewah, W. Pugh, J. Morgenthaler, J. Penix, and Y. Zhou, "Evaluating Static Analysis Defect Warnings on Production Software," *Proc. Seventh ACM SIGPLAN-SIGSOFT Workshop Program Analysis for Software Tools and Eng.,* pp. 1-8, 2007. (Paper=10, Status=F, Phase=1).

[11] R. Bell, T. Ostrand, and E. Weyuker, "Looking for Bugs in All the Right Places," *Proc. Int'l Symp. Software Testing and Analysis,* pp. 61-72, 2006. (Paper=11, Status=P).

[12] P. Bellini, I. Bruno, P. Nesi, and D. Rogai, "Comparing Fault-proneness Estimation Models," *Proc. IEEE 10th Int'l Conf. Eng. Complex Computer Systems,* pp. 205-214, June 2005. (Paper=12, Status=F, Phase=2).

[13] A. Bernstein, J. Ekanayake, and M. Pinzger, "Improving Defect Prediction Using Temporal Features and Non Linear Models," *Proc. Ninth Int'l Workshop Principles of Software Evolution: In Conjunction with the Sixth ESEC/FSE Joint Meeting,* pp. 11-18, 2007. (Paper=13, Status=F, Phase=2).

[14] M. Bezerra, A. Oliveira, and S. Meira, "A Constructive RBF Neural Network for Estimating the Probability of Defects in Software Modules," *Proc. Int'l Joint Conf. Neural Networks,* pp. 2869-2874, Aug. 2007. (Paper=14, Status=F, Phase=2, Data=N).

[15] S. Bibi, G. Tsoumakas, I. Stamelos, and I. Vlahvas, "Software Defect Prediction Using Regression via Classification," *Proc. IEEE Int'l Conf. Computer Systems and Applications,* vol. 8, pp. 330-336, 2006. (Paper=15, Status=P).

[16] D. Binkley, H. Feild, D. Lawrie, and M. Pighin, "Software Fault Prediction Using Language Processing," *Proc. Testing: Academic and Industrial Conf. Practice and Research Techniques,* pp. 99-110, Sept. 2007. (Paper=16, Status=F, Phase=1).

[17] D. Binkley, H. Feild, D. Lawrie, and M. Pighin, "Increasing Diversity: Natural Language Measures for Software Fault Prediction," *J. Systems and Software,* vol. 82, no. 11, pp. 1793-1803, 2009. (Paper=17, Status=F, Phase=1).

[18] C. Bird, N. Nagappan, H. Gall, B. Murphy, and P. Devanbu, "Putting it All Together: Using Socio-Technical Networks to Predict Failures," *Proc. 20th Int'l Symp. Software Reliability Eng.*, pp. 109-119, 2009. (Paper=18, Status=P).

[19] G. Boetticher, "Improving Credibility of Machine Learner Models in Software Engineering," *Advanced Machine Learner Applications in Software Eng.*, pp. 52-72, Idea Group Publishing, 2006. (Paper=19, Status=F, Phase=2, Data=N).

[20] L. Briand, W. Melo, and J. Wust, "Assessing the Applicability of Fault-Proneness Models across Object-Oriented Software Projects," *IEEE Trans. Software Eng.*, vol. 28, no. 7, pp. 706-720, July 2002. (Paper=20, Status=F, Phase=2).

[21] B. Caglayan, A. Bener, and S. Koch, "Merits of Using Repository Metrics in Defect Prediction for Open Source Projects," *Proc. ICSE Workshop Emerging Trends in Free/Libre/Open Source Software Research and Development*, pp. 31-36, 2009. (Paper=21, Status=P).

[22] G. Calikli, A. Tosun, A. Bener, and M. Celik, "The Effect of Granularity Level on Software Defect Prediction," *Proc. 24th Int'l Symp. Computer and Information Sciences*, pp. 531-536, Sept. 2009. (Paper=22, Status=F, Phase=2).

[23] C. Catal, B. Diri, and B. Ozumut, "An Artificial Immune System Approach for Fault Prediction in Object-Oriented Software," *Proc. Second Int'l Conf. Dependability of Computer Systems*, pp. 238-245, June 2007. (Paper=23, Status=F, Phase=2, Data=N).

[24] E. Ceylan, F. Kutlubay, and A. Bener, "Software Defect Identification Using Machine Learning Techniques," *Proc. 32nd Software Eng. and Advanced Applications*, pp. 240-247, 2006. (Paper=24, Status=F, Phase=2).

[25] V.U. Challagulla, F.B. Bastani, and I.-L. Yen, "A Unified Framework for Defect Data Analysis Using the MBR Technique," *Proc. IEEE 18th Int'l Tools with Artificial Intelligence*, pp. 39-46, Nov. 2006. (Paper=25, Status=F, Phase=2, Data=N).

[26] V. Challagulla, F. Bastani, I.-L. Yen, and R. Paul, "Empirical Assessment of Machine Learning Based Software Defect Prediction Techniques," *Proc.S 10th IEEE Int'l Workshop Object-Oriented Real-Time Dependable Systems*, pp. 263-270, Feb. 2005. (Paper=26, Status=F, Phase=2, Data=N).

[27] J. Cong, D. En-Mei, and Q. Li-Na, "Software Fault Prediction Model Based on Adaptive Dynamical and Median Particle Swarm Optimization," *Proc. Second Int'l Conf. Multimedia and Information Technology*, vol. 1, pp. 44-47, 2010. (Paper=27, Status=F, Phase=2, Data=N).

[28] C. Cruz and A. Erika, "Exploratory Study of a UML Metric for Fault Prediction," *Proc. ACM/IEEE 32nd Int'l Conf. Software Eng.*, pp. 361-364, 2010. (Paper=28, Status=F, Phase=4).

[29] C. Cruz, A. Erika, and O. Koichiro, "Towards Logistic Regression Models for Predicting Fault-Prone Code across Software Projects," *Proc. Third Int'l Symp. Empirical Software Eng. and Measurement*, pp. 460-463, 2009. (Paper=29, Status=P).

[30] V. Dallmeier and T. Zimmermann, "Extraction of Bug Localization Benchmarks from History," *Proc. IEEE/ACM 22nd Int'l Conf. Automated Software Eng.*, pp. 433-436, 2007. (Paper=30, Status=F, Phase=1).

[31] M. D'Ambros, M. Lanza, and R. Robbes, "On the Relationship between Change Coupling and Software Defects," *Proc. 16th Working Conf. Reverse Eng.*, pp. 135-144, Oct. 2009. (Paper=31, Status=P).

[32] M. D'Ambros, M. Lanza, and R. Robbes, "An Extensive Comparison of Bug Prediction Approaches," *Proc. IEEE Seventh Working Conf. Mining Software Repositories*, pp. 31-41, 2010. (Paper=32, Status=P).

[33] A.B. de Carvalho, A. Pozo, S. Vergilio, and A. Lenz, "Predicting Fault Proneness of Classes through a Multiobjective Particle Swarm Optimization Algorithm," *Proc. IEEE 20th Int'l Conf. Tools with Artificial Intelligence*, vol. 2, pp. 387-394, 2008. (Paper=33, Status=F, Phase=2, Data=N).

[34] A.B. de Carvalho, A. Pozo, and S.R. Vergilio, "A Symbolic Fault-Prediction Model Based on Multiobjective Particle Swarm Optimization," *J. Systems and Software*, vol. 83, no. 5, pp. 868-882, 2010. (Paper=34, Status=F, Phase=2, Data=N).

[35] G. Denaro, "Estimating Software Fault-Proneness for Tuning Testing Activities," *Proc. Int'l Conf. Software Eng.*, pp. 704-706, 2000. (Paper=35, Status=F, Phase=2).

[36] G. Denaro, S. Morasca, and M. Pezzè, "Deriving Models of Software Fault-Proneness," *Proc. 14th Int'l Conf. Software Eng. and Knowledge Eng.*, pp. 361-368, 2002. (Paper=36, Status=F, Phase=2).

[37] G. Denaro and M. Pezzè, "An Empirical Evaluation of Fault-Proneness Models," *Proc. 24th Int'l Conf. Software Eng.*, pp. 241-251, 2002. (Paper=37, Status=P).

[38] Z. Dianqin and W. Zhongyuan, "The Application of Gray-Prediction Theory in the Software Defects Management," *Proc. Int'l Conf. Computational Intelligence and Software Eng.*, pp. 1-5, 2009. (Paper=38, Status=F, Phase=2).

[39] K. El Emam, W. Melo, and J. Machado, "The Prediction of Faulty Classes Using Object-Oriented Design Metrics," *J. Systems and Software*, vol. 56, no. 1, pp. 63-75, 2001. (Paper=39, Status=F, Phase=2).

[40] K. Elish and M. Elish, "Predicting Defect-prone Software Modules Using Support Vector Machines," *J. Systems and Software*, vol. 81, no. 5, pp. 649-660, 2008. (Paper=40, Status=F, Phase=2, Data=N).

[41] N. Fenton, M. Neil, W. Marsh, P. Hearty, L. Radlinski, and P. Krause, "Project Data Incorporating Qualitative Factors for Improved Software Defect Prediction," *Proc. Int'l Workshop Predictor Models in Software Eng.*, p. 2, May 2007. (Paper=41, Status=F, Phase=2).

[42] N. Fenton and N. Ohlsson, "Quantitative Analysis of Faults and Failures in a Complex Software System," *IEEE Trans. Software Eng.*, vol. 26, no. 8, pp. 797-814, Aug. 2000. (Paper=42, Status=F, Phase=1).

[43] F. Fioravanti and P. Nesi, "A Study on Fault-Proneness Detection of Object-Oriented Systems," *Proc. Fifth European Conf. Software Maintenance and Reeng.*, pp. 121-130, 2001. (Paper=43, Status=F, Phase=2).

[44] K. Gao and T. Khoshgoftaar, "A Comprehensive Empirical Study of Count Models for Software Fault Prediction," *IEEE Trans. Reliability*, vol. 56, no. 2, pp. 223-236, June 2007. (Paper=44, Status=F, Phase=2).

[45] N. Gayatri, S. Nickolas, A.V. Reddy, and R. Chitra, "Performance Analysis of Datamining Algorithms for Software Quality Prediction," *Proc. Int'l Conf. Advances in Recent Technologies in Comm. and Computing*, pp. 393-395, 2009. (Paper=45, Status=F, Phase=2, Data=N).

[46] I. Gondra, "Applying Machine Learning to Software Fault-Proneness Prediction," *J. Systems and Software*, vol. 81, no. 2, pp. 186-195, 2008. (Paper=46, Status=F, Phase=1).

[47] T. Graves, A. Karr, J. Marron, and H. Siy, "Predicting Fault Incidence Using Software Change History," *IEEE Trans. Software Eng.*, vol. 26, no. 7, pp. 653-661, July 2000. (Paper=47, Status=F, Phase=1).

[48] D. Gray, D. Bowes, N. Davey, S. Yi, and B. Christianson, "Software Defect Prediction Using Static Code Metrics Underestimates Defect-Proneness," *Proc. Int'l Joint Conf. Neural Networks*, pp. 1-7, 2010. (Paper=48, Status=F, Phase=1).

[49] L. Guo, B. Cukic, and H. Singh, "Predicting Fault Prone Modules by the Dempster-Shafer Belief Networks," *Proc. IEEE 18th Int'l Conf. Automated Software Eng.*, pp. 249-252, Oct. 2003. (Paper=49, Status=F, Phase=2, Data=N).

[50] L. Guo, Y. Ma, B. Cukic, and H. Singh, "Robust Prediction of Fault-Proneness by Random Forests," *Proc. 15th Int'l Symp. Software Reliability Eng.*, pp. 417-428, Nov. 2004. (Paper=50, Status=F, Phase=2, Data=N).

[51] T. Gyimothy, R. Ferenc, and I. Siket, "Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction," *IEEE Trans. Software Eng.*, vol. 31, no. 10, pp. 897-910, Oct. 2005. (Paper=51, Status=P).

[52] A.E. Hassan, "Predicting Faults Using the Complexity of Code Changes," *Proc. 31st IEEE Int'l Conf. Software Eng.*, pp. 78-88, 2009. (Paper=52, Status=F, Phase=5).

[53] A. Hassan and R. Holt, "The Top Ten List: Dynamic Fault Prediction," *Proc. 21st IEEE Int'l Conf. Software Maintenance*, pp. 263-272, Sept. 2005. (Paper=53, Status=F, Phase=1).

[54] Y. Higo, K. Murao, S. Kusumoto, and K. Inoue, "Predicting Fault-Prone Modules Based on Metrics Transitions," *Proc. Workshop Defects in Large Software Systems*, pp. 6-10, 2008. (Paper=54, Status=F, Phase=1).

[55] T. Holschuh, M. Pauser, K. Herzig, T. Zimmermann, R. Premraj, and A. Zeller, "Predicting Defects in SAP Java Code: An Experience Report," *Proc. 31st Int'l Conf. Software Eng.-Companion Volume*, pp. 172-181, 2009. (Paper=55, Status=F, Phase=2).

[56] Z. Hongyu, "An Investigation of the Relationships Between Lines of Code and Defects," *Proc. IEEE Int'l Conf. Software Maintenance*, pp. 274-283, 2009. (Paper=56, Status=P, Data=N).

[57] D. Hovemeyer and W. Pugh, "Finding Bugs Is Easy," *Proc. Companion to the 19th Ann. ACM SIGPLAN Conf. Object-Oriented Programming Systems, Languages, and Applications,* pp. 132-136, 2004. (Paper=57, Status=F, Phase=3).

[58] L. Hribar and D. Duka, "Software Component Quality Prediction Using KNN and Fuzzy Logic," *Proc. 33rd Int'l Convention MIPRO,* pp. 402-408, 2010. (Paper=58, Status=F, Phase=1).

[59] W. Huanjing, T.M. Khoshgoftaar, and A. Napolitano, "A Comparative Study of Ensemble Feature Selection Techniques for Software Defect Prediction," *Proc. Ninth Int'l Conf. Machine Learning and Applications,* pp. 135-140, 2010. (Paper=59, Status=F, Phase=4, Data=N).

[60] L. Jiang, Z. Su, and E. Chiu, "Context-Based Detection of Clone-Related Bugs," *Proc. Sixth Joint Meeting of the European Software Eng. Conf. and the ACM SIGSOFT Symp. Foundations of Software Eng.,* pp. 55-64, 2007. (Paper=60, Status=F, Phase=1).

[61] Y. Jiang, B. Cukic, and Y. Ma, "Techniques for Evaluating Fault Prediction Models," *Empirical Software Eng.,* vol. 13, no. 5, pp. 561-595, 2008. (Paper=61, Status=F, Phase=2, Data=N).

[62] Y. Jiang, B. Cukic, and T. Menzies, "Fault Prediction Using Early Lifecycle Data," *Proc. IEEE 18th Int'l Symp. Software Reliability,* pp. 237-246, Nov. 2007. (Paper=62, Status=F, Phase=2, Data=N).

[63] Y. Jiang, B. Cukic, and T. Menzies, "Cost Curve Evaluation of Fault Prediction Models," *Proc. 19th Int'l Symp. Software Reliability Eng.,* pp. 197-206, Nov. 2008. (Paper=63, Status=F, Phase=2, Data=N).

[64] Y. Jiang, J. Lin, B. Cukic, and T. Menzies, "Variance Analysis in Software Fault Prediction Models," *Proc. 20th Int'l Symp. Software Reliability Eng.,* pp. 99-108, Nov. 2009. (Paper=64, Status=F, Phase=2, Data=N).

[65] J.A. Jones and M.J. Harrold, "Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique," *Proc. IEEE/ACM 20th Int'l Conf. Automated Software Eng.,* pp. 273-282, 2005. (Paper=65, Status=F, Phase=1).

[66] J.A. Jones, M.J. Harrold, and J. Stasko, "Visualization of Test Information to Assist Fault Localization," *Proc. 24th Int'l Conf. Software Eng.,* pp. 467-477, 2002. (Paper=66, Status=F, Phase=1).

[67] H. Joshi, C. Zhang, S. Ramaswamy, and C. Bayrak, "Local and Global Recency Weighting Approach to Bug Prediction," *Proc. Fourth Int'l Workshop Mining Software Repositories,* p. 33, May 2007. (Paper=67, Status=F, Phase=2).

[68] L. Jun, X. Zheng, Q. Jianzhong, and L. Shukuan, "A Defect Prediction Model for Software Based on Service Oriented Architecture Using Expert Cocomo," *Proc. 21st Ann. Int'l Conf. Chinese Control and Decision Conf.,* pp. 2591-2594, 2009. (Paper=68, Status=F, Phase=1).

[69] Y. Kamei, S. Matsumoto, A. Monden, K.i. Matsumoto, B. Adams, and A.E. Hassan, "Revisiting Common Bug Prediction Findings Using Effort-Aware Models," *Proc. IEEE Int'l Conf. Software Maintenance,* pp. 1-10, 2010. (Paper=69, Status=P).

[70] K. Kaminsky and G. Boetticher, "Building a Genetically Engineerable Evolvable Program (GEEP) Using Breadth-based Explicit Knowledge for Predicting Software Defects," *Proc. IEEE Ann. Meeting Fuzzy Information,* vol. 1, pp. 10-15, June 2004. (Paper=70, Status=F, Phase=1).

[71] S. Kanmani, V.R. Uthariaraj, V. Sankaranarayanan, and P. Thambidurai, "Object Oriented Software Quality Prediction Using General Regression Neural Networks," *SIGSOFT Software Eng. Notes,* vol. 29, pp. 1-6, Sept. 2004. (Paper=71, Status=F, Phase=2).

[72] S. Kanmani, V. Uthariaraj, V. Sankaranarayanan, and P. Thambidurai, "Object-Oriented Software Fault Prediction Using Neural Networks," *Information and Software Technology,* vol. 49, no. 5, pp. 483-492, 2007. (Paper=72, Status=F, Phase=2).

[73] Y. Kastro and A. Bener, "A Defect Prediction Method for Software Versioning," *Software Quality J.,* vol. 16, no. 4, pp. 543-562, 2008. (Paper=73, Status=P).

[74] A. Kaur and R. Malhotra, "Application of Random Forest in Predicting Fault-Prone Classes," *Proc. Int'l Conf. Advanced Computer Theory and Eng.,* pp. 37-43, 2008. (Paper=74, Status=P).

[75] A. Kaur, P.S. Sandhu, and A.S. Bra, "Early Software Fault Prediction Using Real Time Defect Data," *Proc. Second Int'l Conf. Machine Vision,* pp. 242-245, 2009. (Paper=75, Status=F, Phase=2, Data=N).

[76] T.M. Khoshgoftaar, K. Gao, and N. Seliya, "Attribute Selection and Imbalanced Data: Problems in Software Defect Prediction," *Proc. 22nd IEEE Int'l Conf. Tools with Artificial Intelligence,* vol. 1, pp. 137-144, 2010. (Paper=76, Status=P).

[77] T. Khoshgoftaar, E. Allen, and J. Busboom, "Modeling Software Quality: The Software Measurement Analysis and Reliability Toolkit," *Proc. IEEE 12th Int'l Conf. Tools with Artificial Intelligence,* pp. 54-61, 2000. (Paper=77, Status=F, Phase=2).

[78] T. Khoshgoftaar, K. Gao, and R. Szabo, "An Application of Zero-Inflated Poisson Regression for Software Fault Prediction," *Proc. 12th Int'l Symp. Software Reliability Eng.,* pp. 66-73, Nov. 2001. (Paper=78, Status=F, Phase=2).

[79] T. Khoshgoftaar, E. Geleyn, and K. Gao, "An Empirical Study of the Impact of Count Models Predictions on Module-order Models," *Proc. Eighth IEEE Symp. Software Metrics,* pp. 161-172, 2002. (Paper=79, Status=F, Phase=2).

[80] T. Khoshgoftaar and N. Seliya, "Improving Usefulness of Software Quality Classification Models Based on Boolean Discriminant Functions," *Proc. 13th Int'l Symp. Software Reliability Eng.,* pp. 221-230, 2002. (Paper=80, Status=F, Phase=2).

[81] T. Khoshgoftaar and N. Seliya, "Software Quality Classification Modeling Using the Sprint Decision Tree Algorithm," *Proc. IEEE 14th Int'l Conf. Tools with Artificial Intelligence,* pp. 365-374, 2002. (Paper=81, Status=F, Phase=2).

[82] T. Khoshgoftaar and N. Seliya, "Tree-Based Software Quality Estimation Models for Fault Prediction," *Proc. IEEE Eighth Symp. Software Metrics,* pp. 203-214, 2002. (Paper=82, Status=F, Phase=2).

[83] T. Khoshgoftaar and N. Seliya, "Comparative Assessment of Software Quality Classification Techniques: An Empirical Case Study," *Empirical Software Eng.,* vol. 9, no. 3, pp. 229-257, 2004. (Paper=83, Status=P).

[84] T. Khoshgoftaar, N. Seliya, and K. Gao, "Assessment of a New Three-Group Software Quality Classification Technique: An Empirical Case Study," *Empirical Software Eng.,* vol. 10, no. 2, pp. 183-218, 2005. (Paper=84, Status=F, Phase=2).

[85] T. Khoshgoftaar, V. Thaker, and E. Allen, "Modeling Fault-Prone Modules of Subsystems," *Proc. 11th Int'l Symp. Software Reliability Eng.,* pp. 259-267, 2000. (Paper=85, Status=F, Phase=2).

[86] T. Khoshgoftaar, X. Yuan, E. Allen, W. Jones, and J. Hudepohl, "Uncertain Classification of Fault-Prone Software Modules," *Empirical Software Eng.,* vol. 7, no. 4, pp. 297-318, 2002. (Paper=86, Status=P).

[87] S. Kim, K. Pan, and E. Whitehead Jr., "Memories of Bug Fixes," *Proc. 14th ACM SIGSOFT Int'l Symp. Foundations of Software Eng.,* pp. 35-45, 2006. (Paper=87, Status=F, Phase=2).

[88] S. Kim, T. Zimmermann, E. Whitehead Jr., and A. Zeller, "Predicting Faults from Cached History," *Proc. 29th Int'l Conf. Software Eng.,* pp. 489-498, 2007. (Paper=88, Status=F, Phase=4).

[89] S. Kim, T. Zimmermann, K. Pan, and E. Whitehead, "Automatic Identification of Bug-Introducing Changes," *Proc. IEEE/ACM 21st Int'l Conf. Automated Software Eng.,* pp. 81-90, Sept. 2006. (Paper=89, Status=F, Phase=1).

[90] M. Kläs, F. Elberzhager, J. Münch, K. Hartjes, and O. von Graevemeyer, "Transparent Combination of Expert and Measurement Data for Defect Prediction: An Industrial Case Study," *Proc. 32nd ACM/IEEE Int'l Conf. Software Eng.,* pp. 119-128, 2010. (Paper=90, Status=F, Phase=2).

[91] M. Kläs, H. Nakao, F. Elberzhager, and J. Münch, "Predicting Defect Content and Quality Assurance Effectiveness by Combining Expert Judgment and Defect Data-A Case Study," *Proc. 19th Int'l Symp. Software Reliability Eng.,* pp. 17-26, 2008. (Paper=91, Status=F, Phase=2).

[92] P. Knab, M. Pinzger, and A. Bernstein, "Predicting Defect Densities in Source Code Files with Decision Tree Learners," *Proc. Int'l Workshop Mining Software Repositories,* pp. 119-125, 2006. (Paper=92, Status=P).

[93] A. Koru and H. Liu, "Building Effective Defect-Prediction Models in Practice," *IEEE Software,* vol. 22, no. 6, pp. 23-29, Nov./Dec. 2005. (Paper=93, Status=F, Phase=2, Data=N).

[94] A. Koru, D. Zhang, and H. Liu, "Modeling the Effect of Size on Defect Proneness for Open-Source Software," *Proc. Int'l Workshop Predictor Models in Software Eng.,* p. 10, May 2007. (Paper=94, Status=F, Phase=1).

[95] O. Kutlubay, B. Turhan, and A. Bener, "A Two-Step Model for Defect Density Estimation," *Proc. 33rd EUROMICRO Conf. Software Eng. and Advanced Applications,* pp. 322-332, Aug. 2007. (Paper=95, Status=F, Phase=2, Data=N).

[96] L. Layman, G. Kudrjavets, and N. Nagappan, "Iterative Identification of Fault-Prone Binaries Using in-Process Metrics," *Proc. ACM-IEEE Second Int'l Symp. Empirical Software Eng. and Measurement,* pp. 206-212, 2008. (Paper=96, Status=F, Phase=4).

[97] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking Classification Models for Software Defect Prediction: A Proposed Framework and Novel Findings," *IEEE Trans. Software Eng.,* vol. 34, no. 4, pp. 485-496, July/Aug. 2008. (Paper=97, Status=F, Phase=2, Data=N).

[98] P.L. Li, J. Herbsleb, M. Shaw, and B. Robinson, "Experiences and Results from Initiating Field Defect Prediction and Product Test Prioritization Efforts at Abb Inc." *Proc. 28th Int'l Conf. Software Eng.,* pp. 413-422, 2006. (Paper=98, Status=P).

[99] P.L. Li, M. Shaw, J. Herbsleb, B. Ray, and P. Santhanam, "Empirical Evaluation of Defect Projection Models for Widely-Deployed Production Software Systems," *SIGSOFT Software Eng. Notes,* vol. 29, pp. 263-272, Oct. 2004. (Paper=99, Status=F, Phase=1).

[100] P. Li, J. Herbsleb, and M. Shaw, "Finding Predictors of Field Defects for Open Source Software Systems in Commonly Available Data Sources: A Case Study of Openbsd," *Proc. IEEE 11th Int'l Symp. Software Metrics,* pp. 10-32, Sept. 2005. (Paper=100, Status=F, Phase=1).

[101] Z. Li and M. Reformat, "A Practical Method for the Software Fault-Prediction," *Proc. IEEE Int'l Conf. Information Reuse and Integration,* pp. 659-666, Aug. 2007. (Paper=101, Status=F, Phase=2, Data=N).

[102] Y. Ma, L. Guo, and B. Cukic, "A Statistical Framework for the Prediction of Fault-Proneness," *Proc. Advances in Machine Learning Applications in Software Eng.,* pp. 237-265, 2006. (Paper=102, Status=F, Phase=2, Data=N).

[103] J.T. Madhavan and E.J. Whitehead Jr., "Predicting Buggy Changes Inside an Integrated Development Environment," *Proc. OOPSLA Workshop Eclipse Technology Exchange,* pp. 36-40, 2007. (Paper=103, Status=F, Phase=4).

[104] A. Mahaweerawat, P. Sophatsathit, and C. Lursinsap, "Software Fault Prediction Using Fuzzy Clustering and Radial-Basis Function Network," *Proc. Int'l Conf. Intelligent Technologies,* pp. 304-313, 2002. (Paper=104, Status=F, Phase=2).

[105] A. Mahaweerawat, P. Sophatsathit, and C. Lursinsap, "Adaptive Self-Organizing Map Clustering for Software Fault Prediction," *Proc. Fourth Int'l Joint Conf. Computer Science and Software Eng.,* pp. 35-41, 2007. (Paper=105, Status=F, Phase=2).

[106] A. Mahaweerawat, P. Sophatsathit, C. Lursinsap, and P. Musilek, "Fault Prediction in Object-Oriented Software Using Neural Network Techniques," *Proc. InTech Conf.,* pp. 2-4, 2004. (Paper=106, Status=F, Phase=2).

[107] A. Marcus, D. Poshyvanyk, and R. Ferenc, "Using the Conceptual Cohesion of Classes for Fault Prediction in Object-Oriented Systems," *IEEE Trans. Software Eng.,* vol. 34, no. 2, pp. 287-300, Mar./Apr. 2008. (Paper=107, Status=F, Phase=1).

[108] T. Mende and R. Koschke, "Revisiting the Evaluation of Defect Prediction Models," *Proc. Fifth Int'l Conf. Predictor Models in Software Eng.,* p. 7, 2009. (Paper=108, Status=F, Phase=2, Data=N).

[109] T. Mende and R. Koschke, "Effort-Aware Defect Prediction Models," *Proc. 14th European Conf. Software Maintenance and Reeng.,* pp. 107-116, 2010. (Paper=109, Status=P, Data=N).

[110] T. Mende, R. Koschke, and M. Leszak, "Evaluating Defect Prediction Models for a Large Evolving Software System," *Proc. 13th European Conf. Software Maintenance and Reeng.,* pp. 247-250, Mar. 2009. (Paper=110, Status=P).

[111] T. Menzies and J. Di Stefano, "How Good Is Your Blind Spot Sampling Policy," *Proc. IEEE Eighth Int'l Symp. High Assurance Systems Eng.,* pp. 129-138, Mar. 2004. (Paper=111, Status=F, Phase=2, Data=N).

[112] T. Menzies, J. Greenwald, and A. Frank, "Data Mining Static Code Attributes to Learn Defect Predictors," *IEEE Trans. Software Eng.,* vol. 33, no. 1, pp. 2-13, Jan. 2007. (Paper=112, Status=F, Phase=2, Data=N).

[113] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and A.B. Bener, "Defect Prediction from Static Code Features: Current Results, Limitations, New Approaches," *Automatic Software Eng.,* vol. 17, no. 4, pp. 375-407, 2010. (Paper=113, Status=F, Phase=2, Data=N).

[114] T. Menzies, B. Turhan, A. Bener, G. Gay, B. Cukic, and Y. Jiang, "Implications of Ceiling Effects in Defect Predictors," *Proc. Fourth Int'l Workshop Predictor Models in Software Eng.,* pp. 47-54, 2008. (Paper=114, Status=F, Phase=2, Data=N).

[115] M. Mertik, M. Lenic, G. Stiglic, and P. Kokol, "Estimating Software Quality with Advanced Data Mining Techniques," *Proc. Int'l Conf. Software Eng. Advances,* p. 19, Oct. 2006. (Paper=115, Status=F, Phase=2, Data=N).

[116] O. Mizuno, S. Ikami, S. Nakaichi, and T. Kikuno, "Spam Filter Based Approach for Finding Fault-Prone Software Modules," *Proc. Fourth Int'l Workshop Mining Software Repositories,* p. 4, May 2007. (Paper=116, Status=P).

[117] O. Mizuno and T. Kikuno, "Training on Errors Experiment to Detect Fault-Prone Software Modules by Spam Filter," *Proc. Sixth Joint Meeting of the European Software Eng. Conf. and the ACM SIGSOFT Symp. Foundations of Software Eng.,* pp. 405-414, 2007. (Paper=117, Status=P).

[118] R. Moser, W. Pedrycz, and G. Succi, "A Comparative Analysis of the Efficiency of Change Metrics and Static Code Attributes for Defect Prediction," *Proc. ACM/IEEE 30th Int'l Conf. Software Eng.,* pp. 181-190, 2008. (Paper=118, Status=P).

[119] N. Nagappan and T. Ball, "Static Analysis Tools as Early Indicators of Pre-Release Defect Density," *Proc. 27th Int'l Conf. Software Eng.,* pp. 580-586, May 2005. (Paper=119, Status=F, Phase=3).

[120] N. Nagappan, A. Zeller, T. Zimmermann, K. Herzig, and B. Murphy, "Change Bursts as Defect Predictors," *Proc. IEEE 21st Int'l Symp. Software Reliability Eng.,* pp. 309-318, 2010. (Paper=120, Status=P).

[121] N. Nagappan and T. Ball, "Use of Relative Code Churn Measures to Predict System Defect Density," *Proc. 27th Int'l Conf. Software Eng.,* pp. 284-292, 2005. (Paper=121, Status=F, Phase=2).

[122] N. Nagappan, T. Ball, and A. Zeller, "Mining Metrics to Predict Component Failures," *Proc. 28th Int'l Conf. Software Eng.,* pp. 452-461, 2006. (Paper=122, Status=P).

[123] N.K. Nagwani and S. Verma, "Predictive Data Mining Model for Software Bug Estimation Using Average Weighted Similarity," *Proc. IEEE Second Int'l Advance Computing Conf.,* pp. 373-378, 2010. (Paper=123, Status=F, Phase=1).

[124] A. Neufelder, "How to Measure the Impact of Specific Development Practices on Fielded Defect Density," *Proc. 11th Int'l Symp. Software Reliability Eng.,* pp. 148-160, 2000. (Paper=124, Status=F, Phase=1).

[125] A. Nikora and J. Munson, "Developing Fault Predictors for Evolving Software Systems," *Proc. Ninth Int'l Software Metrics Symp.,* pp. 338-350, Sept. 2003. (Paper=125, Status=F, Phase=1).

[126] A. Nikora and J. Munson, "The Effects of Fault Counting Methods on Fault Model Quality," *Proc. 28th Ann. Int'l Computer Software and Applications Conf.,* vol. 1, pp. 192-201, Sept. 2004. (Paper=126, Status=F, Phase=1).

[127] A. Nugroho, M.R.V. Chaudron, and E. Arisholm, "Assessing UML Design Metrics for Predicting Fault-Prone Classes in a Java System," *Proc. IEEE Seventh Working Conf. Mining Software Repositories,* pp. 21-30, 2010. (Paper=127, Status=F, Phase=5).

[128] H. Olague, L. Etzkorn, S. Gholston, and S. Quattlebaum, "Empirical Validation of Three Software Metrics Suites to Predict Fault-Proneness of Object-Oriented Classes Developed Using Highly Iterative or Agile Software Development Processes," *IEEE Trans. Software Eng.,* vol. 33, no. 6, pp. 402-419, June 2007. (Paper=128, Status=F, Phase=2).

[129] A. Oral and A. Bener, "Defect Prediction for Embedded Software," *Proc. 22nd Int'l Symp. Computer and Information Sciences,* pp. 1-6, Nov. 2007. (Paper=129, Status=F, Phase=2, Data=N).

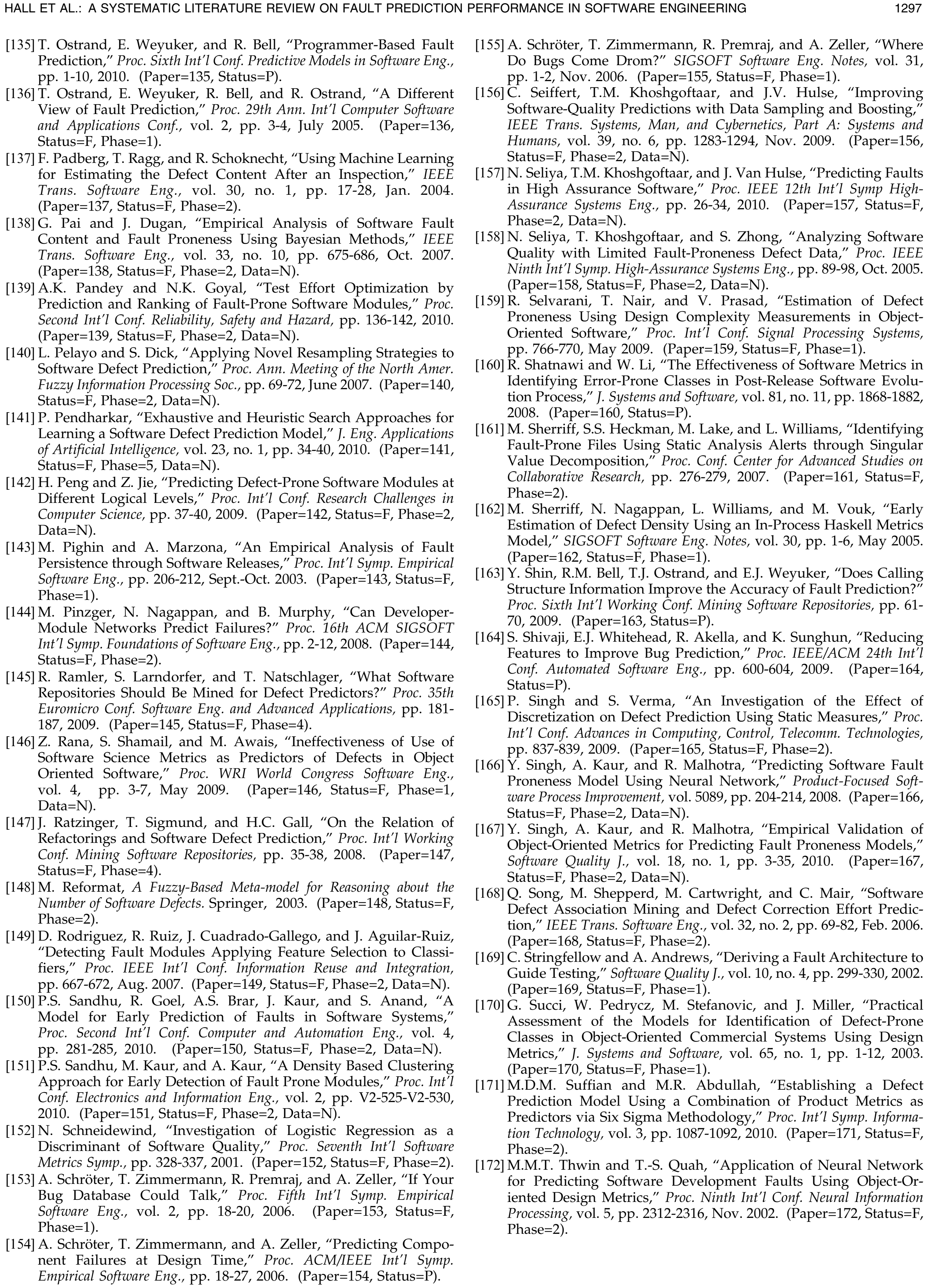
[130] T.J. Ostrand and E.J. Weyuker, "The Distribution of Faults in a Large Industrial Software System," *SIGSOFT Software Eng. Notes,* vol. 27, pp. 55-64, July 2002. (Paper=130, Status=F, Phase=1).

[131] T.J. Ostrand, E.J. Weyuker, and R.M. Bell, "Locating Where Faults Will Be," *Proc. Conf. Diversity in Computing,* pp. 48-50, 2005. (Paper=131, Status=F, Phase=2).

[132] T.J. Ostrand, E.J. Weyuker, and R.M. Bell, "Automating Algorithms for the Identification of Fault-Prone Files," *Proc. Int'l Symp. Software Testing and Analysis,* pp. 219-227, 2007. (Paper=132, Status=F, Phase=4).

[133] T.J. Ostrand, E.J. Weyuker, and R. Bell, "Where the Bugs Are," *ACM SIGSOFT Software Eng. Notes,* vol. 29, pp. 86-96, 2004. (Paper=133, Status=P).

[134] T.J. Ostrand, E.J. Weyuker, and R. Bell, "Predicting the Location and Number of Faults in Large Software Systems," *IEEE Trans. Software Eng.,* vol. 31, no. 4, pp. 340-355, Apr. 2005. (Paper=134, Status=F, Phase=4).



[135] T. Ostrand, E. Weyuker, and R. Bell, "Programmer-Based Fault Prediction," *Proc. Sixth Int'l Conf. Predictive Models in Software Eng.*, pp. 1-10, 2010. (Paper=135, Status=P).

[136] T. Ostrand, E. Weyuker, R. Bell, and R. Ostrand, "A Different View of Fault Prediction," *Proc. 29th Ann. Int'l Computer Software and Applications Conf.*, vol. 2, pp. 3-4, July 2005. (Paper=136, Status=F, Phase=1).

[137] F. Padberg, T. Ragg, and R. Schoknecht, "Using Machine Learning for Estimating the Defect Content After an Inspection," *IEEE Trans. Software Eng.*, vol. 30, no. 1, pp. 17-28, Jan. 2004. (Paper=137, Status=F, Phase=2).

[138] G. Pai and J. Dugan, "Empirical Analysis of Software Fault Content and Fault Proneness Using Bayesian Methods," *IEEE Trans. Software Eng.*, vol. 33, no. 10, pp. 675-686, Oct. 2007. (Paper=138, Status=F, Phase=2, Data=N).

[139] A.K. Pandey and N.K. Goyal, "Test Effort Optimization by Prediction and Ranking of Fault-Prone Software Modules," *Proc. Second Int'l Conf. Reliability, Safety and Hazard*, pp. 136-142, 2010. (Paper=139, Status=F, Phase=2, Data=N).

[140] L. Pelayo and S. Dick, "Applying Novel Resampling Strategies to Software Defect Prediction," *Proc. Ann. Meeting of the North Amer. Fuzzy Information Processing Soc.*, pp. 69-72, June 2007. (Paper=140, Status=F, Phase=2, Data=N).

[141] P. Pendharkar, "Exhaustive and Heuristic Search Approaches for Learning a Software Defect Prediction Model," *J. Eng. Applications of Artificial Intelligence*, vol. 23, no. 1, pp. 34-40, 2010. (Paper=141, Status=F, Phase=5, Data=N).

[142] H. Peng and Z. Jie, "Predicting Defect-Prone Software Modules at Different Logical Levels," *Proc. Int'l Conf. Research Challenges in Computer Science*, pp. 37-40, 2009. (Paper=142, Status=F, Phase=2, Data=N).

[143] M. Pighin and A. Marzona, "An Empirical Analysis of Fault Persistence through Software Releases," *Proc. Int'l Symp. Empirical Software Eng.*, pp. 206-212, Sept.-Oct. 2003. (Paper=143, Status=F, Phase=1).

[144] M. Pinzger, N. Nagappan, and B. Murphy, "Can Developer-Module Networks Predict Failures?" *Proc. 16th ACM SIGSOFT Int'l Symp. Foundations of Software Eng.*, pp. 2-12, 2008. (Paper=144, Status=F, Phase=2).

[145] R. Ramler, S. Larndorfer, and T. Natschlager, "What Software Repositories Should Be Mined for Defect Predictors?" *Proc. 35th Euromicro Conf. Software Eng. and Advanced Applications*, pp. 181-187, 2009. (Paper=145, Status=F, Phase=4).

[146] Z. Rana, S. Shamail, and M. Awais, "Ineffectiveness of Use of Software Science Metrics as Predictors of Defects in Object Oriented Software," *Proc. WRI World Congress Software Eng.*, vol. 4, pp. 3-7, May 2009. (Paper=146, Status=F, Phase=1, Data=N).

[147] J. Ratzinger, T. Sigmund, and H.C. Gall, "On the Relation of Refactorings and Software Defect Prediction," *Proc. Int'l Working Conf. Mining Software Repositories*, pp. 35-38, 2008. (Paper=147, Status=F, Phase=4).

[148] M. Reformat, *A Fuzzy-Based Meta-model for Reasoning about the Number of Software Defects.* Springer, 2003. (Paper=148, Status=F, Phase=2).

[149] D. Rodriguez, R. Ruiz, J. Cuadrado-Gallego, and J. Aguilar-Ruiz, "Detecting Fault Modules Applying Feature Selection to Classifiers," *Proc. IEEE Int'l Conf. Information Reuse and Integration*, pp. 667-672, Aug. 2007. (Paper=149, Status=F, Phase=2, Data=N).

[150] P.S. Sandhu, R. Goel, A.S. Brar, J. Kaur, and S. Anand, "A Model for Early Prediction of Faults in Software Systems," *Proc. Second Int'l Conf. Computer and Automation Eng.*, vol. 4, pp. 281-285, 2010. (Paper=150, Status=F, Phase=2, Data=N).

[151] P.S. Sandhu, M. Kaur, and A. Kaur, "A Density Based Clustering Approach for Early Detection of Fault Prone Modules," *Proc. Int'l Conf. Electronics and Information Eng.*, vol. 2, pp. V2-525-V2-530, 2010. (Paper=151, Status=F, Phase=2, Data=N).

[152] N. Schneidewind, "Investigation of Logistic Regression as a Discriminant of Software Quality," *Proc. Seventh Int'l Software Metrics Symp.*, pp. 328-337, 2001. (Paper=152, Status=F, Phase=2).

[153] A. Schröter, T. Zimmermann, R. Premraj, and A. Zeller, "If Your Bug Database Could Talk," *Proc. Fifth Int'l Symp. Empirical Software Eng.*, vol. 2, pp. 18-20, 2006. (Paper=153, Status=F, Phase=1).

[154] A. Schröter, T. Zimmermann, and A. Zeller, "Predicting Component Failures at Design Time," *Proc. ACM/IEEE Int'l Symp. Empirical Software Eng.*, pp. 18-27, 2006. (Paper=154, Status=P).

[155] A. Schröter, T. Zimmermann, R. Premraj, and A. Zeller, "Where Do Bugs Come Drom?" *SIGSOFT Software Eng. Notes*, vol. 31, pp. 1-2, Nov. 2006. (Paper=155, Status=F, Phase=1).

[156] C. Seiffert, T.M. Khoshgoftaar, and J.V. Hulse, "Improving Software-Quality Predictions with Data Sampling and Boosting," *IEEE Trans. Systems, Man, and Cybernetics, Part A: Systems and Humans*, vol. 39, no. 6, pp. 1283-1294, Nov. 2009. (Paper=156, Status=F, Phase=2, Data=N).

[157] N. Seliya, T.M. Khoshgoftaar, and J. Van Hulse, "Predicting Faults in High Assurance Software," *Proc. IEEE 12th Int'l Symp High-Assurance Systems Eng.*, pp. 26-34, 2010. (Paper=157, Status=F, Phase=2, Data=N).

[158] N. Seliya, T. Khoshgoftaar, and S. Zhong, "Analyzing Software Quality with Limited Fault-Proneness Defect Data," *Proc. IEEE Ninth Int'l Symp. High-Assurance Systems Eng.*, pp. 89-98, Oct. 2005. (Paper=158, Status=F, Phase=2, Data=N).

[159] R. Selvarani, T. Nair, and V. Prasad, "Estimation of Defect Proneness Using Design Complexity Measurements in Object-Oriented Software," *Proc. Int'l Conf. Signal Processing Systems*, pp. 766-770, May 2009. (Paper=159, Status=F, Phase=1).

[160] R. Shatnawi and W. Li, "The Effectiveness of Software Metrics in Identifying Error-Prone Classes in Post-Release Software Evolution Process," *J. Systems and Software*, vol. 81, no. 11, pp. 1868-1882, 2008. (Paper=160, Status=P).

[161] M. Sherriff, S.S. Heckman, M. Lake, and L. Williams, "Identifying Fault-Prone Files Using Static Analysis Alerts through Singular Value Decomposition," *Proc. Conf. Center for Advanced Studies on Collaborative Research*, pp. 276-279, 2007. (Paper=161, Status=F, Phase=2).

[162] M. Sherriff, N. Nagappan, L. Williams, and M. Vouk, "Early Estimation of Defect Density Using an In-Process Haskell Metrics Model," *SIGSOFT Software Eng. Notes*, vol. 30, pp. 1-6, May 2005. (Paper=162, Status=F, Phase=1).

[163] Y. Shin, R.M. Bell, T.J. Ostrand, and E.J. Weyuker, "Does Calling Structure Information Improve the Accuracy of Fault Prediction?" *Proc. Sixth Int'l Working Conf. Mining Software Repositories*, pp. 61-70, 2009. (Paper=163, Status=P).

[164] S. Shivaji, E.J. Whitehead, R. Akella, and K. Sunghun, "Reducing Features to Improve Bug Prediction," *Proc. IEEE/ACM 24th Int'l Conf. Automated Software Eng.*, pp. 600-604, 2009. (Paper=164, Status=P).

[165] P. Singh and S. Verma, "An Investigation of the Effect of Discretization on Defect Prediction Using Static Measures," *Proc. Int'l Conf. Advances in Computing, Control, Telecomm. Technologies*, pp. 837-839, 2009. (Paper=165, Status=F, Phase=2).

[166] Y. Singh, A. Kaur, and R. Malhotra, "Predicting Software Fault Proneness Model Using Neural Network," *Product-Focused Software Process Improvement*, vol. 5089, pp. 204-214, 2008. (Paper=166, Status=F, Phase=2, Data=N).

[167] Y. Singh, A. Kaur, and R. Malhotra, "Empirical Validation of Object-Oriented Metrics for Predicting Fault Proneness Models," *Software Quality J.*, vol. 18, no. 1, pp. 3-35, 2010. (Paper=167, Status=F, Phase=2, Data=N).

[168] Q. Song, M. Shepperd, M. Cartwright, and C. Mair, "Software Defect Association Mining and Defect Correction Effort Prediction," *IEEE Trans. Software Eng.*, vol. 32, no. 2, pp. 69-82, Feb. 2006. (Paper=168, Status=F, Phase=2).

[169] C. Stringfellow and A. Andrews, "Deriving a Fault Architecture to Guide Testing," *Software Quality J.*, vol. 10, no. 4, pp. 299-330, 2002. (Paper=169, Status=F, Phase=1).

[170] G. Succi, W. Pedrycz, M. Stefanovic, and J. Miller, "Practical Assessment of the Models for Identification of Defect-Prone Classes in Object-Oriented Commercial Systems Using Design Metrics," *J. Systems and Software*, vol. 65, no. 1, pp. 1-12, 2003. (Paper=170, Status=F, Phase=1).

[171] M.D.M. Suffian and M.R. Abdullah, "Establishing a Defect Prediction Model Using a Combination of Product Metrics as Predictors via Six Sigma Methodology," *Proc. Int'l Symp. Information Technology*, vol. 3, pp. 1087-1092, 2010. (Paper=171, Status=F, Phase=2).

[172] M.M.T. Thwin and T.-S. Quah, "Application of Neural Network for Predicting Software Development Faults Using Object-Oriented Design Metrics," *Proc. Ninth Int'l Conf. Neural Information Processing*, vol. 5, pp. 2312-2316, Nov. 2002. (Paper=172, Status=F, Phase=2).

[173] P. Tomaszewski, H. Grahn, and L. Lundberg, "A Method for an Accurate Early Prediction of Faults in Modified Classes," *Proc. IEEE 22nd Int'l Conf. Software Maintenance,* pp. 487-496, Sept. 2006. (Paper=173, Status=F, Phase=2).

[174] A. Tosun and A. Bener, "Reducing False Alarms in Software Defect Prediction by Decision Threshold Optimization," *Proc. Third Int'l Symp. Empirical Software Eng. and Measurement,* pp. 477-480, 2009. (Paper=174, Status=F, Phase=2, Data=N).

[175] A. Tosun, B. Turhan, and A. Bener, "Practical Considerations in Deploying AI for Defect Prediction: A Case Study within the Turkish Telecommunication Industry," *Proc. Fifth Int'l Conf. Predictor Models in Software Eng.,* p. 11, 2009. (Paper=175, Status=F, Phase=2, Data=N).

[176] A. Tosun, A.B. Bener, B. Turhan, and T. Menzies, "Practical Considerations in Deploying Statistical Methods for Defect Prediction: A Case Study within the Turkish Telecommunications Industry," *Information and Software Technology,* vol. 52, no. 11, pp. 1242-1257, 2010. (Paper=176, Status=F, Phase=2, Data=N).

[177] B. Turhan and A. Bener, "A Multivariate Analysis of Static Code Attributes for Defect Prediction," *Proc. Seventh Int'l Conf. Quality Software,* pp. 231-237, Oct. 2007. (Paper=177, Status=F, Phase=2, Data=N).

[178] B. Turhan, G. Kocak, and A. Bener, "Software Defect Prediction Using Call Graph Based Ranking (CGBR) Framework," *Proc. 34th Euromicro Conf. Software Eng. and Advanced Applications,* pp. 191-198, Sept. 2008. (Paper=178, Status=F, Phase=2).

[179] B. Turhan, G. Kocak, and A. Bener, "Data Mining Source Code for Locating Software Bugs: A Case Study in Telecommunication Industry," *Expert Systems with Applications,* vol. 36, no. 6, pp. 9986-9990, 2009. (Paper=179, Status=F, Phase=2, Data=N).

[180] B. Turhan, A.B. Bener, and T. Menzies, "Regularities in Learning Defect Predictors," *Proc. 11th Int'l Conf. Product-Focused Software Process Improvement,* pp. 116-130, 2010. (Paper=180, Status=F, Phase=4, Data=N).

[181] B. Turhan, T. Menzies, A. Bener, and J. Di Stefano, "On the Relative Value of Cross-Company and within-Company Data for Defect Prediction," *Empirical Software Eng.,* vol. 14, no. 5, pp. 540-578, 2009. (Paper=181, Status=F, Phase=2, Data=N).

[182] O. Vandecruys, D. Martens, B. Baesens, C. Mues, M. De Backer, and R. Haesen, "Mining Software Repositories for Comprehensible Software Fault Prediction Models," *J. Systems and Software,* vol. 81, no. 5, pp. 823-839, 2008. (Paper=182, Status=F, Phase=2, Data=N).

[183] R. Vivanco, Y. Kamei, A. Monden, K. Matsumoto, and D. Jin, "Using Search-Based Metric Selection and Oversampling to Predict Fault Prone Modules," *Proc. 23rd Canadian Conf. Electrical and Computer Eng.,* pp. 1-6, 2010. (Paper=183, Status=F, Phase=2, Data=N).

[184] D. Wahyudin, A. Schatten, D. Winkler, A.M. Tjoa, and S. Biffl, "Defect Prediction Using Combined Product and Project Metrics—A Case Study from the Open Source 'Apache' Myfaces Project Family," *Proc. 34th Euromicro Conf. Software Eng. and Advanced Applications,* pp. 207-215, 2008. (Paper=184, Status=F, Phase=1).

[185] T. Wang and W.-h. Li, "Naive Bayes Software Defect Prediction Model," *Proc. Int'l Conf. Computational Intelligence and Software Eng.,* pp. 1-4, 2010. (Paper=185, Status=F, Phase=2, Data=N).

[186] W. Wei, D. Xuan, L. Chunping, and W. Hui, "A Novel Evaluation Method for Defect Prediction in Software Systems," *Proc. Int'l Conf. Computational Intelligence and Software Eng.,* pp. 1-5, 2010. (Paper=186, Status=F, Phase=1).

[187] Y. Weimin and L. Longshu, "A Rough Set Model for Software Defect Prediction," *Proc. Int'l Conf. Intelligent Computation Technology and Automation,* vol. 1, pp. 747-751, 2008. (Paper=187, Status=F, Phase=2, Data=N).

[188] E. Weyuker, T. Ostrand, and R. Bell, "Using Developer Information as a Factor for Fault Prediction," *Proc. Third Int'l Workshop Predictor Models in Software Eng.,* p. 8, May 2007. (Paper=188, Status=F, Phase=2).

[189] E. Weyuker, T. Ostrand, and R. Bell, "Comparing Negative Binomial and Recursive Partitioning Models for Fault Prediction," *Proc. Fourth Int'l Workshop Predictor Models in Software Eng.,* pp. 3-10, 2008. (Paper=189, Status=F, Phase=2).

[190] E. Weyuker, T. Ostrand, and R. Bell, "Do Roo Many Cooks Spoil the Broth? Using the Number of Developers to Enhance Defect Prediction Models," *Empirical Software Eng.,* vol. 13, no. 5, pp. 539-559, 2008. (Paper=190, Status=P).

[191] E.J. Weyuker, T.J. Ostrand, and R.M. Bell, "Comparing the Effectiveness of Several Modeling Methods for Fault Prediction," *Empirical Software Eng.,* vol. 15, no. 3, pp. 277-295, 2010. (Paper=191, Status=F, Phase=2).

[192] C. Wohlin, M. Host, and M. Ohlsson, "Understanding the Sources of Software Defects: A Filtering Approach," *Proc. Eighth Int'l Workshop Program Comprehension,* pp. 9-17, 2000. (Paper=192, Status=F, Phase=1).

[193] W. Wong, J. Horgan, M. Syring, W. Zage, and D. Zage, "Applying Design Metrics to Predict Fault-Proneness: A Case Study on a Large-Scale Software System," *Software: Practice and Experience,* vol. 30, no. 14, pp. 1587-1608, 2000. (Paper=193, Status=F, Phase=2).

[194] Z. Xu, T. Khoshgoftaar, and E. Allen, "Prediction of Software Faults Using Fuzzy Nonlinear Regression Modeling," *Proc. IEEE Fifth Int'l Symp. High Assurance Systems Eng.,* pp. 281-290, 2000. (Paper=194, Status=F, Phase=2).

[195] B. Yang, L. Yao, and H.-Z. Huang, "Early Software Quality Prediction Based on a Fuzzy Neural Network Model," *Proc. Third Int'l Conf. Natural Computation,* vol. 1, pp. 760-764, Aug. 2007. (Paper=195, Status=F, Phase=2).

[196] L. Yi, T.M. Khoshgoftaar, and N. Seliya, "Evolutionary Optimization of Software Quality Modeling with Multiple Repositories," *IEEE Trans. Software Eng.,* vol. 36, no. 6, pp. 852-864, Nov. 2010. (Paper=196, Status=F, Phase=2, Data=N).

[197] H. Youngki, B. Jongmoon, K. In-Young, and C. Ho-Jin, "A Value-Added Predictive Defect Type Distribution Model Based on Project Characteristics," *Proc. IEEE/ACIS Seventh Int'l Conf. Computer and Information Science,* pp. 469-474, 2008. (Paper=197, Status=F, Phase=2).

[198] P. Yu, T. Systa, and H. Muller, "Predicting Fault-Proneness Using OO Metrics an Industrial Case Study," *Proc. Sixth European Conf. Software Maintenance and Reeng.,* pp. 99-107, 2002. (Paper=198, Status=F, Phase=1).

[199] X. Yuan, T. Khoshgoftaar, E. Allen, and K. Ganesan, "An Application of Fuzzy Clustering to Software Quality Prediction," *Proc. IEEE Third Symp. Application-Specific Systems and Software Eng. Technology,* pp. 85-90, 2000. (Paper=199, Status=F, Phase=2).

[200] H. Zhang, A. Nelson, and T. Menzies, "On the Value of Learning from Defect Dense Components for Software Defect Prediction," *Proc. Sixth Int'l Conf. Predictive Models in Software Eng.,* p. 14, Sept. 2010. (Paper=200, Status=F, Phase=2, Data=N).

[201] S. Zhong, T. Khoshgoftaar, and N. Seliya, "Unsupervised Learning for Expert-Based Software Quality Estimation," *Proc. IEEE Eighth Int'l Symp. High Assurance Systems Eng.,* pp. 149-155, Mar. 2004. (Paper=201, Status=F, Phase=2, Data=N).

[202] Y. Zhou and H. Leung, "Empirical Analysis of Object-Oriented Design Metrics for Predicting High and Low Severity Faults," *IEEE Trans. Software Eng.,* vol. 32, no. 10, pp. 771-789, Oct. 2006. (Paper=202, Status=F, Phase=2, Data=N).

[203] Y. Zhou, B. Xu, and H. Leung, "On the Ability of Complexity Metrics to Predict Fault-Prone Classes in Object-Oriented Systems," *J. Systems and Software,* vol. 83, no. 4, pp. 660-674, 2010. (Paper=203, Status=P).

[204] T. Zimmermann and N. Nagappan, "Predicting Subsystem Failures Using Dependency Graph Complexities," *Proc. IEEE 18th Int'l Symp. Software Reliability,* pp. 227-236, Nov. 2007. (Paper=204, Status=F, Phase=2).

[205] T. Zimmermann and N. Nagappan, "Predicting Defects Using Network Analysis on Dependency Graphs," *Proc. ACM/IEEE 30th Int'l Conf. Software Eng.,* pp. 531-540, 2008. (Paper=205, Status=F, Phase=4).

[206] T. Zimmermann and N. Nagappan, "Predicting Defects with Program Dependencies," *Proc. Third Int'l Symp. Empirical Software Eng. and Measurement,* pp. 435-438, 2009. (Paper=206, Status=F, Phase=2).

[207] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting Defects for Eclipse," *Proc. Int'l Workshop Predictor Models in Software Eng.,* p. 9, May 2007. (Paper=207, Status=F, Phase=2).

[208] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy, "Cross-Project Defect Prediction: A Large Scale Experiment on Data vs. Domain vs. Process," *Proc. Seventh Joint Meeting of the European Software Eng. Conf. and the ACM SIGSOFT Symp. The Foundations of Software Eng.,* pp. 91-100, 2009. (Paper=208, Status=F, Phase=4).

TABLE 10
Conferences and Journals Manually Searched

Conference manually searched	Journals manually searched
International Conference on Software Engineering (ICSE)	IEEE Transactions of Software Engineering
International Conference on Software Maintenance (ICSM)	Journal of Systems and Software
IEEE Int'l Working Conference on Source Code Analysis and Manipulation (SCAM)	Journal of Empirical Software Engineering
International Conference on Automated Software Engineering	Software Quality Journal
IEEE Int'l Symposium and Workshop on Engineering of Computer Based Systems	Information & Software Technology
International Symposium on Automated Analysis-driven Debugging	
International Symposium on Software Testing and Analysis (ISSTA)	
International Symposium on Software Reliability Engineering	
ACM SIGPLAN Conference on Programming language Design and Implementation	
Int'l Workshop on Mining Software Repositories	
Empirical Software Engineering & Measurement	
PROMISE	
Foundations of Software Engineering	

TABLE 11
Additional Data Quality Criteria

Data quality criteria	Criteria definitions	Why the criteria is important
Has any data cleaning been done?	An indication that the quality of the data has been considered is necessary and that any data cleaning needed has been addressed e.g. missing values handled, outliers and errorful data been removed.	Data sets are often noisy. They often contain outliers and missing values that can skew results (the impact of this depends on the analysis methods used). Our confidence in the predictions made by a model is impacted by the quality of the data used while building the model. Few studies have cleaned their data and so we did not apply this criterion.
Have repeated attributes been removed?	An indication that the impact of repeated attributes has been considered should be given. For example machine learning studies could mention attribute selection while other studies could consider, for example, Principal Component Analysis.	Repeated attributes and related attributes have been shown to bias the outcomes of models. Confidence is affected in the predictions of studies which have not considered the impact of repeated/related attributes. Few studies have considered repeated attributes and so we did not apply this criterion.

APPENDIX A

SEARCH STRING

The following search string was used in our searches:

(Fault* OR bug* OR defect* OR errors OR corrections OR corrective OR fix*) *in title only*

AND (Software) *anywhere in study*

APPENDIX B

CONFERENCES AND JOURNALS MANUALLY SEARCHED

See Table 10

APPENDIX C

ADDITIONAL ASSESSMENT CRITERIA

Data quality criteria. The efficacy of the predictions made by a model is determined by the quality of the data on which the model was built. Leibchen and Shepperd [26] report that many studies do not seem to consider the quality of the data they use. Many fault prediction models are based on machine learning, where it has been shown that a lack of data cleaning may compromise the predictions obtained [21]. The criteria shown in Table 11 are based on [21], [S168], [S192], [S194], and [S19].

Predictive performance criteria. Measuring the predictive performance of a model is an essential part of demonstrating the usefulness of that model. Measuring model performance is complex and there are many ways in which the performance of a model may be measured. Furthermore, the value of measures varies according to context. For example, safety critical system developers may want models that identify as many faults as possible, accepting the cost of false alarms, whereas business system developers may want models which do not generate many false alarms as testing effort is short to ensure the timely release of a product at the cost of missing some faults. Appendix D reports the principles of predictive performance measurement and provides the basis of our performance measurement criteria. Table 12 shows our predictive performance measurement criteria.

APPENDIX D

THE PRINCIPLES OF PREDICTIVE PERFORMANCE MEASUREMENT

This overview of measuring predictive performance is based on [30], [S61], and [S97]. The measurement of predictive performance is often based on the analysis of data in a confusion matrix (shown in Table 13 and explained further in Table 14). This matrix reports how the model classified the different fault categories compared to their actual classification (predicted versus observed). Many performance measures are related to components of the confusion matrix shown in Table 14. Confusion matrix-based measures are most relevant to fault prediction models producing categorical outputs, though continuous outputs can be converted to categorical outputs and analyzed in terms of a confusion matrix.

TABLE 12
Additional Predictive Performance Measurement Criteria

Measurement criteria	Criteria definitions	Why the criteria is important
Have imbalanced data sets been accounted for, sufficient to enable confidence in predictive performance?	No one approach adequately accounts for imbalanced data in every circumstance. See Appendix F for an overview of the approaches available. Each study should be assessed on a case-by-case basis according to the specific model reported.	See Appendix F for an overview of the importance of dealing with imbalanced data. There is no one best way of accounting for data imbalance and it was thus difficult to identify a precise enough criterion to apply consistently across studies. In addition there remains significant debate on data imbalance (see [27], [[179]], [28], [29]).
Has predictive performance been reported appropriately?	For each model reporting categorical results a study should report either: - A confusion matrix - Area Under the Curve (AUC) For each model reporting continuous results a study choosing to report measures of error should not report only Mean Squared Error. Average Relative Error should also be reported. Or else results based on Chi Square should be reported.	The particular set of performance measures reported by studies can make it difficult to understand how a model performs overall in terms of correct and incorrect predictions. Confusion matrix constructs form the basis of most other ways of reporting predictive performance. Reporting the confusion matrix (possibly in addition to other measures reported by studies) would allow subsequent analysis of performance in ways other than those preferred by the model developer. Menzies et al. [[114]] suggests a useful way in which data from multiple confusion matrices may be effectively reported. AUC performance data is reported to be an effective way in which to compare the ability of a modelling technique to cope with different datasets (Lessmann et al. [[97]]). Reporting AUC means that models using different datasets can then be meta-analysed. This would enable a much richer understanding of the abilities of particular modelling techniques. Ideally data for the whole ROC curve would be given by each study. It is impractical to report this amount of data and would require the use of on-line data stores. However AUC has limitations for imbalanced data sets as reported by [23] [12]. Mean Squared Error (MSE) generates results that can only be interpreted within the data set from which they originated as the measurement scales used dictate the size of the error. MSE limits the comparability of results across other data sets. Chi Square or Average Relative Error should be used instead. Only a small number of models currently report confusion matrix and AUC data. Consequently, applying this criterion was untenable as so few studies would have passed. Similarly MSE is a poorly understood measure and its limitations are not widely reported and so the current application of this requirement is not tenable.

Composite performance measures can be calculated by combining values from the confusion matrix (see Table 15). "Recall" (otherwise known as the true positive rate, probability of detection (pd), or sensitivity) describes the proportion of faulty code units (usually files, modules, or packages) correctly predicted as such, while "precision" describes how reliable a prediction is in terms of what proportion of code predicted as faulty actually was faulty. Both are important when test sets are imbalanced, but there is a tradeoff between these two measures [S61]. An additional composite measure is the false positive rate (pf), which describes the proportion of erroneous defective predictions. Thus, the optimal classifier would achieve a pd of 1, precision of 1, and a pf of 0. The performance measure balance combines pd and pf. A high-balance value (near 1) is achieved with a high pd and low pf. Balance can also be adjusted to a factor in the cost of false alarms which typically do not result in fault fixes. When the combinations of pd and pf are plotted, they produce a Receiver Operator Curve (ROC). This gives a range of balance figures, and it is usual to report the area under the curve as varying between 0 and 1, with 1 being the ideal value. Table 16 shows other ways in which the performance of a model can be measured. Such measures are usually used in models that produce continuous or ranking results.

TABLE 13
Confusion Matrix

	Predicted defective	Predicted defect free
Observed defective	True Positive (TP)	False Negative (FN)
Observed defect free	False Positive (FP)	True Negative (TN)

Appendix E

Calculating Precision, Recall, and F-Measure for Categorical Studies (Reported in [31])

Many studies report precision and recall, but others report pd and pf. If we are to compare the results we need to convert the results of one paper into the performance

TABLE 14
Confusion Matrix-Based Performance Indicator

Construct	Also known as	Description
False Positive	FP, and Type I Error	Classifies non faulty unit as faulty
False Negative	FN, and Type II Error	Classifies faulty unit as not faulty
True Positive	TP	Correctly classified as faulty
True Negative	TN	Correctly classified as non-faulty

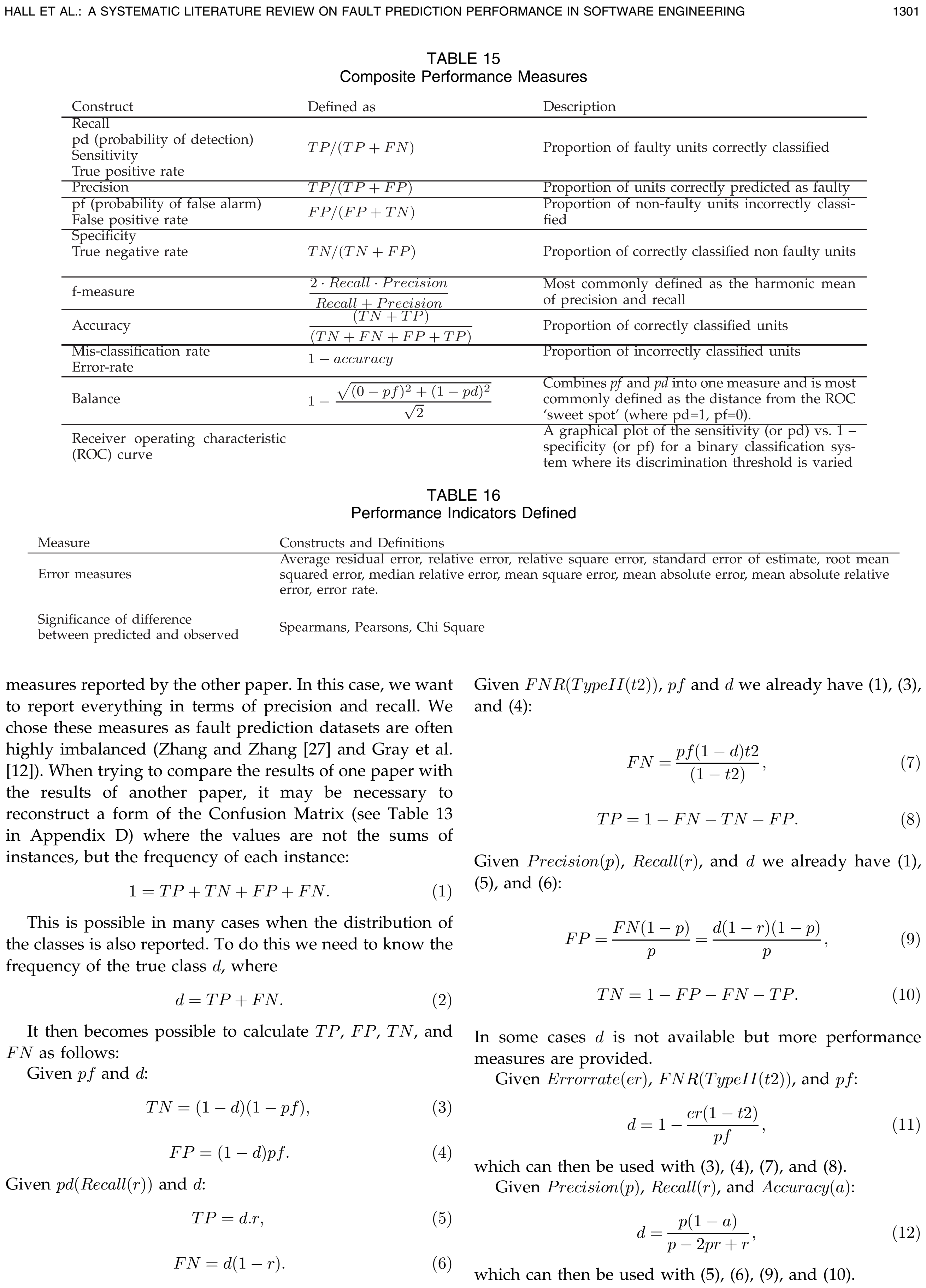


TABLE 15
Composite Performance Measures

Construct	Defined as	Description
Recall pd (probability of detection) Sensitivity True positive rate	$TP/(TP+FN)$	Proportion of faulty units correctly classified
Precision	$TP/(TP+FP)$	Proportion of units correctly predicted as faulty
pf (probability of false alarm) False positive rate	$FP/(FP+TN)$	Proportion of non-faulty units incorrectly classified
Specificity True negative rate	$TN/(TN+FP)$	Proportion of correctly classified non faulty units
f-measure	$\frac{2 \cdot Recall \cdot Precision}{Recall + Precision}$	Most commonly defined as the harmonic mean of precision and recall
Accuracy	$\frac{(TN+TP)}{(TN+FN+FP+TP)}$	Proportion of correctly classified units
Mis-classification rate Error-rate	$1 - accuracy$	Proportion of incorrectly classified units
Balance	$1 - \frac{\sqrt{(0-pf)^2 + (1-pd)^2}}{\sqrt{2}}$	Combines *pf* and *pd* into one measure and is most commonly defined as the distance from the ROC 'sweet spot' (where pd=1, pf=0).
Receiver operating characteristic (ROC) curve		A graphical plot of the sensitivity (or pd) vs. 1 – specificity (or pf) for a binary classification system where its discrimination threshold is varied

TABLE 16
Performance Indicators Defined

Measure	Constructs and Definitions
Error measures	Average residual error, relative error, relative square error, standard error of estimate, root mean squared error, median relative error, mean square error, mean absolute error, mean absolute relative error, error rate.
Significance of difference between predicted and observed	Spearmans, Pearsons, Chi Square

measures reported by the other paper. In this case, we want to report everything in terms of precision and recall. We chose these measures as fault prediction datasets are often highly imbalanced (Zhang and Zhang [27] and Gray et al. [12]). When trying to compare the results of one paper with the results of another paper, it may be necessary to reconstruct a form of the Confusion Matrix (see Table 13 in Appendix D) where the values are not the sums of instances, but the frequency of each instance:

$$1 = TP + TN + FP + FN. \tag{1}$$

This is possible in many cases when the distribution of the classes is also reported. To do this we need to know the frequency of the true class d, where

$$d = TP + FN. \tag{2}$$

It then becomes possible to calculate TP, FP, TN, and FN as follows:

Given pf and d:

$$TN = (1-d)(1-pf), \tag{3}$$

$$FP = (1-d)pf. \tag{4}$$

Given $pd(Recall(r))$ and d:

$$TP = d.r, \tag{5}$$

$$FN = d(1-r). \tag{6}$$

Given $FNR(TypeII(t2))$, pf and d we already have (1), (3), and (4):

$$FN = \frac{pf(1-d)t2}{(1-t2)}, \tag{7}$$

$$TP = 1 - FN - TN - FP. \tag{8}$$

Given $Precision(p)$, $Recall(r)$, and d we already have (1), (5), and (6):

$$FP = \frac{FN(1-p)}{p} = \frac{d(1-r)(1-p)}{p}, \tag{9}$$

$$TN = 1 - FP - FN - TP. \tag{10}$$

In some cases d is not available but more performance measures are provided.

Given $Errorrate(er)$, $FNR(TypeII(t2))$, and pf:

$$d = 1 - \frac{er(1-t2)}{pf}, \tag{11}$$

which can then be used with (3), (4), (7), and (8).

Given $Precision(p)$, $Recall(r)$, and $Accuracy(a)$:

$$d = \frac{p(1-a)}{p - 2pr + r}, \tag{12}$$

which can then be used with (5), (6), (9), and (10).

Given $Accuracy(a)$, pf, and $FNR(TypeII(t2))$:

$$FP = \frac{(1 - t2 - a)pf}{pf - t2}, \qquad (13)$$

$$TN = \frac{(1 - t2 - a)(1 - pf)}{pf - t2}, \qquad (14)$$

$$TP = \frac{(1 - t2)(pf - 1 + 1)}{pf - t2}, \qquad (15)$$

$$FN = 1 - TP - TN - FP. \qquad (16)$$

Given FP, FN, and d:

$$TP = d - FP, \qquad (17)$$

which can then be used with (10).

The following values were extracted from [S83]:

$$er = 0.3127, pf = 0.3134, t2 = 0.2826.$$

We compute

$$d = 0.2842.$$

Giving,

$$FN = 0.0884, TN = 0.4915, FP = 0.2243, TP = 0.1958.$$

Finally,

$$Precision = 0.4661, Recall = 0.6891,$$
$$F\text{-}measure = 0.5561.$$

Appendix F

The Class Imbalance Problem

Substantially imbalanced datasets are commonly used in binary fault prediction studies (i.e., there are usually many more nonfaulty units than faulty units) [32], [27]. An extreme example of this is seen in NASA dataset PC2, which has only 0.4 percent of data points belonging to the faulty class (23 out of 5,589 data points). This distribution of faulty and nonfaulty units—known as the class distribution —should be taken into account during any binary fault prediction task. This is because imbalanced data can strongly influence both the training of a classification model and the suitability of classifier performance metrics.

When training a classifier using imbalanced data, an algorithm can struggle to learn from the minority class. This is typically due to an insufficient quantity of minority class data. The most common symptom when this occurs is for a classifier to predict all data points as belonging to the majority class, which is of little practical worth. To avoid this happening, various approaches can be used and are typically based around training-set sampling and/or learning algorithm optimization. Note that these techniques are entirely optional, and may not be necessary. This is because learning techniques vary in their sensitivity to imbalanced data. For example, C4.5 decision trees have been reported to struggle with imbalanced data [16] and [17], whereas fuzzy-based classifiers have been reported to perform robustly regardless of class distribution [33].

Sampling methods involve the manipulation of training data in order to reduce the level of imbalance and therefore alleviate the problems associated with learning from imbalanced data. Undersampling methods involve reducing the size of the majority class, whereas oversampling methods involve increasing the size of the minority class. Such techniques have been reported to be useful [11]; however, they do suffer from drawbacks. With undersampling methods, the main problem is deciding which majority class data points should be removed. With oversampling methods, there is a risk of the learning algorithm overfitting the oversampled data. This will probably result in good training data performance, but low performance when the classifier is presented with unseen data (data independent from that used during training) [11].

Many learning algorithms can have their various parameters adjusted in order to boost performance on imbalanced data. This can be very effective, as many algorithms by default assume an equal class distribution during training. By increasing the misclassification cost of the minority class, it is possible to construct models that are better suited to imbalanced domains. Such methods can be difficult and/or time consuming to approximate appropriate misclassification costs.

Additional problems caused by imbalanced data are that selecting appropriate classifier performance measures is more difficult. This is because measures which favor the majority class (such as accuracy and error rate) are no longer sufficient [11]. More appropriate measures in imbalanced domains include: precision, recall, f-measure (see Appendix D), and g-mean [11].

In contrast to the training data, the balance of test data should be representative of that which will be encountered in the real world.

There remains significant debate on data imbalance in fault prediction (see [12], [27], [S179], [28], [29]).

Acknowledgments

The authors are grateful to the United Kingdom's Engineering and Physical Science Research Council who supported this research at Brunel University under grant EPSRC EP/E063039/1 and to Science Foundation Ireland grant 3/CE2/I303_1 who partially supported this work at Lero. They are also grateful to Dr. Sue Black and Dr. Paul Wernick who provided input to the early stages of the work reported in this paper and to Professor Martin Shepperd for his suggestions throughout the work. They are also grateful for the detailed and insightful comments from the reviewers that enabled them to significantly improve the quality of this paper.

References

[1] N. Fenton and M. Neil, "A Critique of Software Defect Prediction Models," *IEEE Trans. Software Eng.,* vol. 25, no. 5, pp. 675-689, Sept. 1999.

[2] C. Catal and B. Diri, "A Systematic Review of Software Fault Prediction Studies," *Expert Systems with Application,* vol. 36, no. 4, pp. 7346-7354, 2009.

[3] B. Kitchenham and S. Charters, "Guidelines for Performing Systematic Literature Reviews in Software Engineering (Version 2.3)," Technical Report EBSE-2007-01, Keele Univ., EBSE, 2007.

[4] M. Jorgensen and M. Shepperd, "A Systematic Review of Software Development Cost Estimation Studies," *IEEE Trans. Software Eng.*, vol. 33, no. 1, pp. 33-53, Jan. 2007.

[5] B. Kitchenham, "What's Up with Software Metrics?—A Preliminary Mapping Study," *J. Systems and Software*, vol. 83, no. 1, pp. 37-51, 2010.

[6] Q. Song, Z. Jia, M. Shepperd, S. Ying, and J. Liu, "A General Software Defect-Proneness Prediction Framework," *IEEE Trans. Software Eng.*, vol. 37, no. 3, pp. 356-370, May 2011.

[7] K. Petersen and C. Wohlin, "Context in Industrial Software Engineering Research," *Proc. Third Int'l Symp. Empirical Software Eng. and Measurement*, pp. 401-404, 2009.

[8] P.G. Armour, "Beware of Counting LOC," *Comm. ACM*, vol. 47, pp. 21-24, Mar. 2004.

[9] D. Bowes and T. Hall, "SLuRp: A Web Enabled Database for Effective Management of Systematic Literature Reviews," Technical Report 510, Univ. of Hertfordshire, 2011.

[10] T. Menzies, A. Dekhtyar, J. Distefano, and J. Greenwald, "Problems with Precision: A Response to "Comments on 'Data Mining Static Code Attributes to Learn Defect Predictors,'" *IEEE Trans. Software Eng.*, vol. 33, no. 9, pp. 637-640, Sept. 2007.

[11] H. He and E. Garcia, "Learning from Imbalanced Data," *IEEE Trans. Knowledge and Data Eng.*, vol. 21, no. 9, pp. 1263-1284, Sept. 2008.

[12] D. Gray, D. Bowes, N. Davey, Y. Sun, and B. Christianson, "Further Thoughts on Precision," *Proc. Evaluation and Assessment in Software Eng.*, 2011.

[13] D.S. Cruzes and T. Dybå, "Research Synthesis in Software Engineering: A Tertiary Study," *Information Software Technology*, vol. 53, pp. 440-455, May 2011.

[14] R. Rosenthal and M. DiMatteo, "Meta-Analysis: Recent Developments in Quantitative Methods for Literature Reviews," *Ann. Rev. Psychology*, vol. 52, no. 1, pp. 59-82, 2001.

[15] B. Turhan, T. Menzies, A. Bener, and J. Di Stefano, "On the Relative Value of Cross-Company and Within-Company Data for Defect Prediction," *Empirical Software Eng.*, vol. 14, no. 5, pp. 540-578, 2009.

[16] N. Japkowicz and S. Stephen, "The Class Imbalance Problem: A Systematic Study," *Intelligent Data Analysis*, vol. 6, no. 5, pp. 429-449, 2002.

[17] W. Liu, S. Chawla, D.A. Cieslak, and N.V. Chawla, "A Robust Decision Tree Algorithm for Imbalanced Data Sets," *Proc. 10th SIAM Int'l Conf. Data Mining*, pp. 766-777, 2010.

[18] C. Hsu, C. Chang, and C. Lin, "A Practical Guide to Support Vector Classification," technical report, Dept. of Computer Science and Information Eng., Nat'l Taiwan Univ., 2003.

[19] N. Cristianini and J. Shawe-Taylor, *An Introduction to Support Vector Machines: And Other Kernel-Based Learning Methods.* Cambridge Univ. Press, 2006.

[20] T. Hall, D. Bowes, G. Liebchen, and P. Wernick, "Evaluating Three Approaches to Extracting Fault Data from Software Change Repositories," *Proc. 11th Int'l Conf. Product-Focused Software Process Improvement*, pp. 107-115, 2010.

[21] D. Gray, D. Bowes, N. Davey, Y. Sun, and B. Christianson, "The Misuse of the NASA Metrics Data Program Data Sets for Automated Software Defect Prediction," *Proc. Evaluation and Assessment in Software Eng.*, 2011.

[22] N. Pizzi, A. Summers, and W. Pedrycz, "Software Quality Prediction Using Median-Adjusted Class Labels," *Proc. Int'l Joint Conf. Neural Networks*, vol. 3, pp. 2405-2409, 2002.

[23] J. Davis and M. Goadrich, "The Relationship between Precision-Recall and ROC Curves," *Proc. 23rd Int'l Conf. Machine Learning*, pp. 233-240, 2006.

[24] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals, "Predicting the Severity of a Reported Bug," *Proc. IEEE Working Conf. Seventh Mining Software Repositories*, pp. 1-10, 2010.

[25] I. Myrtveit, E. Stensrud, and M. Shepperd, "Reliability and Validity in Comparative Studies of Software Prediction Models," *IEEE Trans. Software Eng.*, vol. 31, no. 5, pp. 380-391, May 2005.

[26] G. Liebchen and M. Shepperd, "Data Sets and Data Quality in Software Engineering," *Proc. Fourth Int'l Workshop Predictor Models in Software Eng.*, pp. 39-44, 2008.

[27] H. Zhang and X. Zhang, "Comments on 'Data Mining Static Code Attributes to Learn Defect Predictors,'" *IEEE Trans. Software Eng.*, vol. 33, no. 9, pp. 635 -637, Sept. 2007.

[28] G. Batista, R. Prati, and M. Monard, "A Study of the Behavior of Several Methods for Balancing Machine Learning Training Data," *ACM SIGKDD Explorations Newsletter*, vol. 6, no. 1, pp. 20-29, 2004.

[29] Y. Kamei, A. Monden, S. Matsumoto, T. Kakimoto, and K. Matsumoto, "The Effects of Over and Under Sampling on Fault-Prone Module Detection," *Proc. First Int'l Symp. Empirical Software Eng. and Measurement*, pp. 196-204, Sept. 2007.

[30] T. Ostrand and E. Weyuker, "How to Measure Success of Fault Prediction Models," *Proc. Fourth Int'l Workshop Software Quality Assurance: In Conjunction with the Sixth ESEC/FSE Joint Meeting*, pp. 25-30, 2007.

[31] D. Bowes and D. Gray, "Recomputing the Confusion Matrix for Prediction Studies Reporting Categorical Output," Technical Report 509, Univ. of Hertfordshire, 2011.

[32] N.V. Chawla, N. Japkowicz, and A. Kotcz, "Editorial: Special Issue on Learning from Imbalanced Data Sets," *SIGKDD Explorations*, vol. 6, no. 1, pp. 1-6, 2004.

[33] S. Visa and A. Ralescu, "Fuzzy Classifiers for Imbalanced, Complex Classes of Varying Size," *Proc. Information Processing and Management of Uncertainty in Knowledge-Based Systems*, pp. 393-400, 2004.

Tracy Hall received the PhD degree in software metrics from City University in 1998. Currently she is working as a reader in software engineering at Brunel University. Previously she was the head of the Systems & Software Research Group at the University of Hertfordshire. Over the last 15 years she has conducted many empirical software engineering studies with a variety of industrial collaborators. Her research interests have become increasingly centered on fault prediction.

Sarah Beecham is a research fellow working at Lero, The Irish Software Engineering Research Centre. She is currently collaborating closely with industry to develop a process model of global software development recommended practices. Her research interests include software engineer motivation, requirements engineering, fault prediction, effort estimation, and open source software development.

David Bowes is currently working toward the PhD degree, focusing on defect prediction studies. He has been a senior lecturer at the University of Hertfordshire since 2005. He has published mainly in the area of program slicing metrics, defect prediction, and epigenetic robotics.

David Gray received the BSc and MSc degrees in computer science from the University of Hertfordshire. He is currently working toward the PhD degree at the Science and Technology Research Institute of the same university. His research interests include software defect prediction and, in particular, issues with methodology and data quality. His main areas of academic interests include software engineering, machine learning, data quality, and data cleansing.

Steve Counsell received the BSc degree in computing in 1987, the MSc degree in systems analysis in 1988, and the PhD degree from the University of London in 2002. From 1998-2004, he was a lecturer in computer science at Birkbeck, London. He is currently a reader in the School of Information Systems, Computing and Mathematics at Brunel University. His research interests include software metrics, refactoring, software testing, and, more generally, empirical studies of software engineering artifacts.

4.5 Paper 5: Comparing the performance of fault prediction models which report multiple performance measures: recomputing the confusion matrix.

Bowes D, Hall T, Gray D (2012b) Comparing the performance of fault prediction models which report multiple performance measures: reconstructing the confusion matrix. In: Proceedings of the 8th International Conference on Predictive Models in Software Engineering.
Best Paper in Conference Award.

4.5.1 Corrigenda

This paper requires two corrigendum:

1. Section 3.2 paragraph 1. The equation is incorrect and should read: $d = \frac{TP + FN}{TP + TN + FP + FN}$
2. Equation 12 is incorrect and should be replaced by Equation 15 (which can then be removed).

Comparing the performance of fault prediction models which report multiple performance measures: recomputing the confusion matrix

David Bowes
Science and Technology
Research Institute
University of Hertfordshire
College Lane
Hatfield, AL10 9AB
United Kingdom
d.h.bowes@herts.ac.uk

Tracy Hall
Department of Information
Systems and Computing
Brunel University
Uxbridge
Middlesex, UB8 3PH
United Kingdom
tracy.hall@brunel.ac.uk

David Gray
Science and Technology
Research Institute
University of Hertfordshire
College Lane
Hatfield, AL10 9AB
United Kingdom
d.gray@herts.ac.uk

ABSTRACT

There are many hundreds of fault prediction models published in the literature. The predictive performance of these models is often reported using a variety of different measures. Most performance measures are not directly comparable. This lack of comparability means that it is often difficult to evaluate the performance of one model against another. Our aim is to present an approach that allows other researchers and practitioners to transform many performance measures of categorical studies back into a confusion matrix. Once performance is expressed in a confusion matrix alternative preferred performance measures can then be derived. Our approach has enabled us to compare the performance of 600 models published in 42 studies. We demonstrate the application of our approach on several case studies, and discuss the advantages and implications of doing this.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Measurement, Performance, Reliability, Theory

Keywords

fault, confusion matrix, machine learning

1. INTRODUCTION

Imagine the following simplified scenario:

You are a practitioner thinking about starting to use fault prediction models. You hope that such models will help you to identify the most fault prone parts of your system. You then plan to target your test effort on those parts of the system. You think that doing this may reduce the faults delivered to your users and reduce the cost of your system. You are not an expert in fault prediction models yourself, but you have seen many such models published in the literature. You identify several published models that have been developed in a similar software development context to your own. You decide to evaluate the performance of these models with a view to trying out the top three models in your project. However when you look at the model performance figures they are reported using a variety of different performance measures. Several studies report Precision[1] *and Recall. Some report Error Rate. Some report pd and pf. Others report Popt. A few report Area Under the Curve of the Receiver Operator Curve. One provides a confusion matrix. It is beyond your expertise to identify a comparative point of reference amongst these different measures. You struggle to understand how the overall performance of a model compares to the others. And so you decide that fault prediction models are too complicated to use and abandon the idea.*

This type of scenario may partially explain why the uptake of fault prediction models is low in industry. This low uptake is important as finding and fixing faults in code costs the software industry many millions of dollars every year. Predicting effectively where faults are in code occupies many researchers and practitioners. Our previous work [14] showed that 208 software fault prediction studies were published between January 2000 and December 2010. These 208 studies contained many hundreds of fault prediction models.

Despite this significant research effort it remains difficult or inconvenient to compare the performance of these models. The difficulty in comparing predictive performance means that identifying which fault prediction models perform best in a given context is complex. This complexity in comparing the performance of models is not only a likely barrier to practitioners using fault prediction models, but also makes it difficult for researchers to meta-analyse fault prediction studies [8]. This lack of opportunity to meta-analyse limits the ability of the fault prediction community to mature, as

PROMISE '12, September 21–22, 2012, Lund, Sweden
Copyright 2012 ACM 978-1-4503-1241-7/12/09 ...$15.00.

[1] Definitions of particular measures are given in Section Two.

we are not building an evidence base that is as useful as it should be.

One of the difficulties when comparing the performance of fault prediction models stems from the many performance measurement schemes devised, used and reported by studies. Many of the schemes used by studies highlight different aspects of predictive performance. For example, Menzies et al. [25] use pd and pf to highlight standard predictive performance, while Mende and Koschke [23] use Popt to assess effort-awareness. The different performance measurement schemes used mean that directly comparing the performance reported by individual studies is difficult and potentially misleading. Such comparisons cannot compare like with like as there is no adequate point of comparison.

It is perfectly legitimate for studies to report different performance measures. Studies may be interested in reporting prediction models with particular qualities. Some studies may be interested in reporting models which reduce the amount of effort wasted on code predicted as faulty which turns out not to be faulty. In these cases, measures based on the number of false positives will be of most interest. Other studies may be developing models focused on identifying the maximum number of faults in the system. In which case measures related to the number of true positives are likely to be the performance focus. The qualities needed in a fault prediction model depend on, for example, application domain. Models used in the safety critical domain are likely to need different predictive qualities to those in other domains. However developers and potential users of models may want to compare performance in terms of a particular predictive quality. This requires a conversion of performance figures from those reported to those reflecting the predictive quality of interest. The ability to convert predictive measures in this way allows the predictive performance of a wide range of models to be compared.

We previously found [14] that Precision and Recall were the most commonly reported predictive performance measures used with binary[2] fault prediction models (e.g. [1, 6, 11, 20]). However, many studies provide only limited predictive performance data, often only reporting performance using their preferred performance measures. This preferred data often represents the performance of specific models in the most positive light. An issue also highlighted by Zeller et al. [34]. This preferred measurement data may be unusual and rarely reported in other studies. For example, only a few studies report the use of Error Rate [19, 29, 33] or Popt [23]. Without additional performance data that is more commonly reported by studies, it is difficult to satisfactorily compare the predictive performance of such models. A common point of comparison is needed.

The confusion matrix is usually at the centre of measuring the predictive performance of binary classification models (the confusion matrix is discussed in detail in Section Two). Most other predictive performance measures are calculated from the confusion matrix. The confusion matrix is a powerful point of comparative reference. All models reporting binary results can have their predictive performance expressed via a confusion matrix [26]. This means that it is a relatively universal comparative basis. It is also a simple and understandable way to show predictive performance. More sophisticated measures of predictive performance can be calculated from a confusion matrix. The confusion matrix provides measurement flexibility as specific measures may be derived from the confusion matrix which evaluate particular model qualities. The importance of the confusion matrix is discussed in detail by Pizzi et al. [28].

In this paper we present a process by which we transform a variety of reported predictive performance measures back to a confusion matrix [5]. These measures cover most of those reported by the 208 fault prediction studies we previously reviewed [14]. We illustrate this process by constructing the confusion matrix for a number of published models. From these confusion matrices we compute a range of alternative performance measures. We finally evaluate the use of our transformation process.

In Section Two we describe the measurement of predictive performance by discussing in detail the basis of the confusion matrix and related compound measures of performance. In Section Three we present our method of transforming a variety of performance measures to the confusion matrix and explain how alternative measures can then be derived from this matrix. Section Four provides the results of worked examples from the literature in which we transform the reported performance measures back to the confusion matrix. Section Five identifies the threats to the validity of the study. Section Six discusses the implications of transforming performance measures. We conclude and summarise in Section Seven.

2. MEASURING PREDICTIVE PERFORMANCE

This section is based on several previous studies which provide an excellent overview of measuring the predictive performance of fault models (e.g. [26], [16] and [22]).

2.1 The Confusion Matrix

The measurement of predictive performance is often based on the analysis of data in a confusion matrix (see [26]). Pizzi et al. [28] discuss the confusion matrix in more detail. This matrix reports how the model classified the different fault categories compared to their actual classification (i.e. predicted versus observed). This is represented by four pieces of data:

- True Positive (TP): An item is predicted as faulty and it is faulty
- False Positive (FP): An item is predicted as faulty and it is not faulty
- True Negative (TN): An item is predicted as not faulty and it is not faulty
- False Negative (FN): An item is predicted as not faulty and it is faulty

[2] Binary models are those predicting that code units (e.g. modules or classes) are either fault prone (fp) or not fault prone (nfp). Binary models do not predict the number of faults in code units. In this paper we restrict ourselves to considering only binary models that are based on machine learning techniques.

Table 1 shows the structure of a confusion matrix.

Table 1: Confusion matrix

	observed true	observed false
predicted true	TP	FP
predicted false	FN	TN

Table 2: Confusion matrix with example summed instances

	observed true	observed false
predicted true	33	2
predicted false	17	98

In a confusion matrix, it is normal for the sum of the instances of each possibility to be reported, see Table 2.

Few studies report complete confusion matrices for their experiments, these include [27], [36] and [18]. Most studies prefer to report a sub-set of the compound performance measures shown in Table 3.

2.2 Compound Measures

Many performance measures are related to components of the confusion matrix. Table 3 shows how some commonly used performance measures are calculated relative to the confusion matrix.

Table 3 shows that Accuracy is the proportion of units correctly classified. Table 3 also shows that Recall (otherwise known as the true positive rate, probability of detection (pd) or Sensitivity) describes the proportion of faulty code units (usually files, modules or packages) correctly predicted as such. Precision describes how reliable a prediction is in terms of what proportion of code predicted as faulty actually was faulty. Both Recall and Precision are important when test sets are imbalanced (see the following sub-section), but there is a trade-off between these two measures (see [16] for a more detailed analysis of this trade-off). An additional composite measure is the false positive rate (pf) which describes the proportion of erroneously predicted faulty units. The optimal classifier would achieve a pd of 1, Precision of 1, a pf of 0 and an f-measure of 1. The performance measure balance combines pd and pf. A high Balance value (near 1) is achieved with a high pd and low pf. Balance can also be adjusted to factor in the cost of false alarms which typically do not result in fault fixes. Matthews Correlation Coefficient (MCC) is a measure rarely used in software fault prediction [3]. MCC is more commonly used in medical research and bioinformatics e.g. [3, 30]. It is a Chi Square based performance measure on which all four quadrants of the confusion matrix are included in the calculation. MCC results are the equivalent of reporting R^2 in regression modelling and results range from -1 to 1 (with 0 indicating random results). Popt defined by Mende and Koschke [23] is a an effort aware performance measure which ranges between 0 and 1 with 1 being desirable.

The Receiver Operator Curve (ROC) is an important measure of predictive performance for comparing different models. When the combinations of Recall and pf for a series of experiments are plotted they produce a ROC. It is usual to report the area under the curve (AUC) as varying between 0 and 1, with 1 being the ideal value. Because the AUC is a result of a series of experiments where the meta-parameters are varied, it is not possible to compute the confusion matrix from AUC and visa versa[3].

Previous studies have critiqued the use of these various measures of performance. For example, Zhang and Zhang [35], Menzies et al. [24] and Gray et al. [13] discuss the use of precision. However such a critique is beyond the scope of the work reported here.

2.3 Imbalanced Data

Substantially imbalanced data sets are commonly used in fault prediction studies (i.e. there are usually many more non-faulty units than faulty units in the data sets used in fault prediction) [7], [35]. An extreme example of this is seen in the NASA data set PC2, which has only 0.4% of data points belonging to the faulty class (23 out of 5589 data points). This distribution of faulty and non-faulty units has important implications in fault prediction. Imbalanced data can strongly influence the suitability of predictive performance measures. Measures which favour the majority class (such as Accuracy and Error Rate) are not sufficient by themselves [15]. More appropriate measures for imbalanced data sets include: Precision, Recall, f-measure, MCC and G-mean [15]. Consequently data imbalance is an important consideration in our method of recomputing the confusion matrix[4].

3. OUR METHOD OF RECOMPUTING THE CONFUSION MATRIX

To compare the results of one study with the results of another we recompute the confusion matrix for each study and then calculate the preferred compound measures from this. Zhang and Zhang [35] did something similar to this by recomputing Precision for Menzies et al.'s [25] study which originally reported pd and pf. Our approach is motivated by Zhang and Zhang's [35] work. We now describe the process by which transformation from a variety of compound measures to the confusion matrix can be achieved.

3.1 Creating a Frequency-Based Confusion Matrix

The precise method needed to recompute the confusion matrix varies slightly depending upon the original measures reported. In most cases the first thing that needs to be done is to produce a frequency-based confusion matrix. These confusion matrices are different from instance based confusion matrices (an example of which was shown in Table 2).

[3] Although AUC is a valuable measure of performance, it is beyond the scope of our work as it is not possible to construct a confusion matrix from AUC data.

[4] Data imbalance also has serious implications for the training of prediction models. Discussion of this is beyond the scope of this work (instead see [13], [35], [31], [4] and [17]).

Table 3: Compound Performance Measures

Measures	Defined As
Accuracy / Correct Classification Rate (CCR)	$\frac{TP+TN}{TP+TN+FP+FN}$
Error Rate	$\frac{FP+FN}{TP+TN+FP+FN}$
Recall / True Positive Rate / Sensitivity / Probability of Detection (pd)	$\frac{TP}{TP+FN}$
True Negative Rate / Specificity	$\frac{TN}{TN+FP}$
False Positive Rate / Type I Error Rate / Probability of False Alarm (pf)	$\frac{FP}{TN+FP}$
False Negative Rate / Type II Error Rate	$\frac{FN}{FN+TP}$
Precision	$\frac{TP}{TP+FP}$
F-Measure / F-Score	$\frac{2 \times Recall \times Precision}{Recall + Precision}$
Balance	$1 - \frac{\sqrt{(0-pf)^2+(1-pd)^2}}{\sqrt{2}}$
G-mean	$\sqrt{Recall \times Precision}$
Matthews Correlation Coefficient (MCC)	$\frac{TP \times TN - FP \times FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}}$

Table 4 shows the frequencies (or proportions for each confusion matrix quadrant) based on the instances in Table 2. These frequencies are derived by dividing the instances in each quadrant by the total number of instances in the matrix. This shows the relative proportion each quadrant represents of the whole confusion matrix. From now on we will append $_f$ to TP, TN, FP and FN to distinguish frequency values from instance based values for example TP_f.

3.2 Calculating Faulty and Non-Faulty Data Distributions

Constructing a frequency based confusion matrix is possible when the class distribution (i.e. the proportion of faulty versus non-faulty units) is reported[5]. To do this we use d as the frequency of the faulty units, where:

$$d = \frac{TN+FN}{TN+TP+FP+FN} \tag{1}$$

Applying (1) to the example instances reported in Table 2 would result in:

$$n = 33+17+2+98, d = \frac{33+17}{n} = 0.3333$$

This shows that given the confusion matrix shown in Table 2, 33% of the units in the data set on which the model was applied, were faulty.

Table 4: Frequency Confusion Matrix

	observed true	observed false
predicted true	0.2200	0.0133
predicted false	0.1133	0.6533

$$TN_f + TP_f + FP_f + FN_f = 1$$
$$d = 0.2200 + 0.1133 = 0.3333$$

3.3 Transforming Specific Compound Measures

A wide variety of compound measures are reported by studies. Our approach is successful when a particular subset of these measures is reported by studies. Table 5 shows

[5] When this class distribution is not provided it is often possible to calculate the proportion of faulty units in a data set.

the pre-requisite combinations of performance measures that must be available.

Each of these combinations of measures requires a specific method by which to recompute the confusion matrix. Formulae for the most common measures reported are now described.

1. **Transforming** *Precision*, *Recall* **and** *pf*

We first need to know the frequency of the true class d.

$$1 = TP_f + TN_f + FP_f + FN_f \tag{2}$$

$$d = TP_f + FN_f \tag{3}$$

It then becomes possible to calculate TP_f, FP_f, TN_f and FN_f as follows:
Given pf and d

$$TN_f = (1 - d)(1 - pf) \tag{4}$$

$$FP_f = (1 - d)pf \tag{5}$$

Given $Recall(r)$ and d

$$TP_f = d \times r \tag{6}$$

$$FN_f = d(1 - r) \tag{7}$$

Given $FNR(TypeII(t2))$, pf and d we already have (2), (4) and (5)

$$FN_f = t2 \times d \tag{8}$$

$$TP_f = 1 - FN_f - TN_f - FP_f \tag{9}$$

Given $Precision(p)$, $Recall(r)$ and d we already have (2), (6) and (7)

$$FP_f = \frac{TP_f(1 - p)}{p} = \frac{d(1 - p)r}{p} \tag{10}$$

$$TN_f = 1 - FP_f - FN_f - TP_f \tag{11}$$

2. **Transforming** $ErrorRate(er)$, $TypeII(t2)$ **and** pf

$$d = \frac{er - pf + pf \times er}{t2} \tag{12}$$

which can then be used with (4),(5),(8) and (9)

3. **Transforming** $Precision(p)$, $Recall(r)$ **and** $Accuracy(a)$

$$d = \frac{p(1 - a)}{p - 2pr + r} \tag{13}$$

which can then be used with (6),(7),(10) and (11)

4. **Transforming** $Accuracy(a)$, pf **and** $FNR(TypeII(t2))$

$$er = 1 - a \tag{14}$$

$$d = \frac{er - pf}{t2 - pf} \tag{15}$$

which can be used with (8) to give FN_f and (5) to give FP_f.

$$TP_f = d(1 - t2) \tag{16}$$

which can be used with (11) to give TN_f.

We have automated these conversations by developing a tool[6]. This tool allows individual performance measurement data to be input and will automatically recompute the confusion matrix by iterating over the the equations until no extra performance measures can be derived.

4. CONSTRUCTING THE CONFUSION MATRIX FOR SOME EXAMPLE STUDIES

We have transformed the predictive performance data produced by 600 models reported in 42 published studies. A list of these studies is provided in the Appendix. Space restrictions make it is impossible to report the detail for all these transformations. Consequently in this section we present transformations for four examples. We chose these four examples to illustrate recomputing the confusion matrix from a range of different original measures.

4.1 Case Studies

Table 6 illustrates the original performance measurement data reported by our four case study papers. Table 6 shows that a wide range of different measurement data is reported by these four papers. Given this range it is difficult to evaluate how the performance of these models compares against each other.

We have recomputed the confusion matrix for these four case studies (shown in Table 7). Based on this confusion matrix data, we have computed the f-measure and MCC data for each case study (also shown in Table 7). It is now possible to comparatively evaluate the predictive performance of these case studies using this common set of data[7].

5. THREATS TO VALIDITY

There are internal and external validity issues that need to be considered when using our approach to recomputing the confusion matrix.

5.1 Internal Validity

The impact of cross-validation. Performance data reported are usually based on some form of cross-validation. This means that the numbers reported are usually average figures across a specific number of folds and / or experiments. This averaging process may introduce some minor inaccuracies into our calculations[8]. The study by Elish and Elish [12] provide a clearer understanding of the variation in performance values across a series of experiments. The results of rounding errors and variations in performance values helps to explain the negative TN_f value computed in Table 11.

[6] This tool is available at: https://bugcatcher.stca.herts.ac.uk/JConfusion/

[7] The aim of this paper is to provide an approach by which others may perform comparative analysis of fault prediction models. A comparative analysis is complex and requires many factors to be taken into account, e.g. the aims of the predictive model. It is beyond the scope of this paper to provide a full comparative analysis of studies against each other.

[8] An examination of the code for LibSVM shows that the performance measure is an average value of the performance measure for each fold.

Table 5: Pre-Requisite Combinations of Performance Measures for Re-Computing the Confusion Matrix.

Fault Frequency	Type I	Type II	Precision	Recall	Accuracy	pf	Error Rate	Specificity
			✔	✔		✔		
				✔	✔			✔
			✔	✔	✔			
✔			✔	✔				
✔				✔				✔
✔				✔	✔			
✔				✔		✔		
		✔			✔	✔		
		✔				✔	✔	
	✔	✔					✔	

All columns with checkmarks in each row correspond to "enough information" to recompute the frequency confusion matrix. NB this is not an exhaustive list. For example, it is possible to calculate d by dividing the number of defective instances by the total number of instances.

Table 6: Reported Performance Measurement Data

Study	pd	pf	Error Rate	Type I	Type II	Precision	Recall	Accuracy	Total Instances	Faulty Instances
[19]		0.3134	0.3127		0.2826					
[6]						0.682	0.621	0.641		
[21]	0.471	0.0834						0.8515		
[29]			0.1615	0.1304	0.2830				520	106

Divide by zero problems. Several of our formulas are based on divisions. Where some figures are very similar we encounter divide by zero problems. This division problem is exacerbated by rounding of very small numbers. These small numbers may be very different, but when rounded become the same number. Such numbers suffer from divide by zero issues.

Data uncertainty. Identifying the balance of faulty and non-faulty units is an important part of our recomputing method. However in a few studies there is inconsistency in the class distribution figures. For example, although an author may have cited a particular class distribution, when we calculate the distribution figure inherent within the results reported (i.e. via the calculation of d), the distribution is different to that stated by the authors. Similarly in some papers where the same data set has been used the distribution varies between experiments. This inconsistency casts some uncertainty over the results in such cases. We suspect that this distribution inconsistency is partly the result of a particular machine learner dealing with the data that it is processing differently to other learners, and partly the result of studies not reporting the data pre-processing that they have applied.

5.2 External Validity

Model tuning. Some models may have been developed to maximise a particular quality (e.g. to reduce false positives). Such models are likely to perform best when their performance is expressed using measures that are sympathetic to the qualities for which the model has been built. Interpreting the performance of such models via alternative performance measures should be treated with caution.

6. DISCUSSION

The process of translating the performance measures reported by studies to the confusion matrix reveals a variety of performance issues with studies that we now discuss.

6.1 Erroneous Results

In some cases our translation to the confusion matrix demonstrated that the original results reported by some studies could not have been possible. For example we found an error in [32]. This error was revealed as our transformations would not work correctly. As a result of this we emailed the authors to clarify the problem. The authors confirmed that a typographical error had crept into their final draft. False Alarms were reported instead of False Positives. It is easy for such errors to creep into published work, especially in an area as complex as fault prediction. Without very careful interpretation such errors can easily be missed and be misleading.

6.2 Definitions of Measures

While performing our transformations we have had difficulty in making sense of the figures reported in some studies. The reason for this was that a number of studies have used non-standard definitions for some well-known performance measures (e.g. [36] does not use a standard definition of Precision and [27] does not use the standard definitions of Sensitivity and Specificity (in both cases, the issues were confirmed by emailing the authors)). Although the definitions used were given in the paper, it is difficult for a reader to pick-up on the nuances of measurement definitions (usually provided via formulae). Consequent mis-understanding could have serious implications for subsequent model users.

Table 7: Computed Performance Measurement Data

Study	TP_f	TN_f	FP_f	FN_f	f-measure	MCC
[19]	0.0163	0.6710	0.3063	0.0064	0.0944	0.1288
[6]	0.3335	0.3075	0.1555	0.2035	0.6501	0.2845
[21]	0.0575	0.7940	0.0732	0.0646	0.4549	0.3755
[29]	0.1422	0.6963	0.1000	0.0615	0.6377	0.5381

6.3 Reporting Performance Based on Predicting Non-Faulty Units

Some papers have reported performance measures based on predicting the majority (non-faulty class) rather than the minority (faulty) class. In some of these cases it is also not made clear that the predictive performance is on the majority class. These issues can be very misleading when trying to evaluate predictive performance. For example, Elish and Elish [12] report a very influential fault prediction study using Support Vector Machines (SVM). Their study has been cited more than 60 times and is considered a pivotal paper in the use of SVMs. Table 8 shows the very good Accuracy, Precision and Recall performances reported by Elish and Elish for SVM using datasets cm1, pc1, kc1 and kc3 (taken from [12]).

Table 8: Accuracy, Precision and Recall [12]

Dataset	Accuracy	Precision	Recall
cm1	0.9069	0.9066	1.0000
pc1	0.9310	0.9353	0.9947
kc1	0.8459	0.8495	0.9940
kc3	0.9328	0.9365	0.9958

Our process to recompute the confusion matrix would not work on these figures when we assumed that the values for Precision and Recall were based on the non-faulty class. Tables 9 and 10 show our workings for this recomputation. Our workings suggest that Elish and Elish have reported the performance of their SVM models based on predicting non-faulty units rather than faulty units, this was confirmed by emailling the authors. Since the vast majority of units in data sets are non-faulty (ranging between 84.6% and 93.7% in their case), predicting the majority class is very easy and so high performance figures are to be expected. Such models are not useful. Our findings are complementary to those of several other authors who report problems reproducing the high predictive performances reported by Elish and Elish when using their SVM settings. For example [2] reports that most papers report a far lower Recall value. [9] and [10] used the same SVM settings. [9] reported Specificity and Sensitivity values and [10] reported Precision and Recall for both the faulty and non-faulty classes which are similar to our recomputed values.

Using our technique it is possible to calculate the Precision and Recall of the faulty units in Elish and Elish's study. Table 11 shows the results of this calculation. Table 11 suggests that the performance of the SVMs in the Elish and Elish study is much less positive. Table 11 shows that f-measure ranges from 0.0 to 0.12. This is compared to their original maximum f-measure of 0.96.

Table 9: Frequency of the Class Identified as "True" and the Frequency of the Faulty class

Dataset	computed d	1 - computed d	Reported fault frequency
cm1	0.9037	0.0963	0.097
pc1	0.9311	0.0689	0.069
kc1	0.8462	0.1538	0.154
kc3	0.9370	0.0630	0.063

Table 10: SVM Confusion Matrix of the Majority Class

Dataset	TP_f	FN_f	FP_f	TN_f
cm1	0.9037	0.0000	0.0931	0.0032
pc1	0.9261	0.0049	0.0641	0.0049
kc1	0.8412	0.0047	0.1490	0.0051
kc3	0.9330	0.0039	0.0633	-0.0002*

* demonstrates that rounding errors occur.

Table 11: Performance Measures for the Faulty Class using the Values from Table 10

Dataset	Accuracy	Precision	Recall	f-measure
cm1	0.9069	1.0000	0.0332	0.0643
pc1	0.9310	0.5000	0.0710	0.1244
kc1	0.8463	0.5204	0.0331	0.0622
kc3	0.9328	-0.0541	-0.0032	-0.0060

7. CONCLUSION

The predictive performance of published fault prediction models is expressed using a variety of different performance measures. This makes it difficult to compare the performance of published prediction models. We have presented an approach that enables the recomputation of the confusion matrix for studies originally reporting a variety of performance measures. From the confusion matrix a range of other performance measures can be calculated. Expressing the performance of fault prediction models using a consistent set of measures allows comparative analysis. Our approach has several advantages, including that it:

- allows comparative analysis of a set of fault prediction models in terms of a preferred predictive quality.

- makes meta-analysis possible across the many fault prediction studies published.
- enables the validation of the performance figures reported in published studies.

The advantages of our approach have benefits for fault prediction researchers, practitioners and reviewers. Researchers can use our approach to evaluate predictive performance across sets of models and perform meta-analysis of these models. An evidence base of fault prediction can be built by researchers that will enable more informed future model building research. Practitioners can express model performance to reflect the qualities that they are interested in, for example practitioners wanting a model that values finding as many faults as possible might might predominately focus on Recall. Practitioners are then in a more informed position to select a model that is appropriate for their development context. Reviewers of fault prediction studies can use our process as a relatively easy way to check that no errors have crept into fault prediction studies. Without our 'ready reckoner' checking performance figures in studies submitted for review is difficult. Model builders could themselves use our process as a 'ready reckoner' to check their own figures are correct. Model builders and reviewers doing this checking could improve the quality of the fault prediction work that is published.

Overall the approach that we present could significantly improve the quality of fault prediction studies and enable meta-analysis across studies. Achieving this is very important as it will help this research area to mature and grow. Such maturation could ultimately expand the industrial uptake of fault prediction modelling.

8. REFERENCES

[1] E. Arisholm, L. C. Briand, and M. Fuglerud. Data mining techniques for building fault-proneness models in telecom java software. In *Software Reliability, 2007. ISSRE '07. The 18th IEEE International Symposium on*, pages 215 –224, nov. 2007.

[2] E. Arisholm, L. C. Briand, and E. B. Johannessen. A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. *Journal of Systems and Software*, 83(1):2–17, 2010.

[3] P. Baldi, S. Brunak, Y. Chauvin, C. Andersen, and H. Nielsen. Assessing the accuracy of prediction algorithms for classification: an overview. *Bioinformatics*, 16(5):412–424, 2000.

[4] G. Batista, R. Prati, and M. Monard. A study of the behavior of several methods for balancing machine learning training data. *ACM SIGKDD Explorations Newsletter*, 6(1):20–29, 2004.

[5] D. Bowes and D. Gray. Recomputing the confusion matrix for prediction studies reporting categorical output. Technical Report 509, University of Hertfordshire, 2011.

[6] C. Catal, B. Diri, and B. Ozumut. An artificial immune system approach for fault prediction in object-oriented software. In *Dependability of Computer Systems, 2007. DepCoS-RELCOMEX '07. 2nd International Conference on*, pages 238 –245, june 2007.

[7] N. V. Chawla, N. Japkowicz, and A. Kotcz. Editorial: special issue on learning from imbalanced data sets. *SIGKDD Explorations*, 6(1):1–6, 2004.

[8] D. S. Cruzes and T. Dybå. Research synthesis in software engineering: A tertiary study. *Inf. Softw. Technol.*, 53:440–455, May 2011.

[9] A. B. de Carvalho, A. Pozo, S. Vergilio, and A. Lenz. Predicting fault proneness of classes trough a multiobjective particle swarm optimization algorithm. In *Tools with Artificial Intelligence, 2008. ICTAI '08. 20th IEEE International Conference on*, volume 2, pages 387–394, 2008.

[10] A. B. de Carvalho, A. Pozo, and S. R. Vergilio. A symbolic fault-prediction model based on multiobjective particle swarm optimization. *Journal of Systems and Software*, 83(5):868–882, 2010.

[11] G. Denaro and M. Pezzè. An empirical evaluation of fault-proneness models. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 241–251, New York, NY, USA, 2002. ACM.

[12] K. O. Elish and M. O. Elish. Predicting defect-prone software modules using support vector machines. *Journal of Systems and Software*, 81(5):649 – 660, 2008.

[13] D. Gray, D. Bowes, N. Davey, Y. Sun, and B. Christianson. Further thoughts on precision. In *Evaluation and Assessment in Software Engineering (EASE)*, 2011.

[14] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell. A systematic review of fault prediction performance in software engineering. *Software Engineering, IEEE Transactions on*, PP(99):1, 2011.

[15] H. He and E. Garcia. Learning from imbalanced data. *IEEE Transactions on Knowledge and Data Engineering*, pages 1263–1284, 2008.

[16] Y. Jiang, B. Cukic, and Y. Ma. Techniques for evaluating fault prediction models. *Empirical Software Engineering*, 13(5):561–595, 2008.

[17] Y. Kamei, A. Monden, S. Matsumoto, T. Kakimoto, and K. Matsumoto. The effects of over and under sampling on fault-prone module detection. In *Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on*, pages 196 –204, sept. 2007.

[18] A. Kaur, P. S. Sandhu, and A. S. Bra. Early software fault prediction using real time defect data. In *Machine Vision, 2009. ICMV '09. Second International Conference on*, pages 242–245. accept, 2009.

[19] T. Khoshgoftaar and N. Seliya. Comparative assessment of software quality classification techniques: An empirical case study. *Empirical Software Engineering*, 9(3):229–257, 2004.

[20] A. Koru and H. Liu. Building effective defect-prediction models in practice. *Software, IEEE*, 22(6):23 – 29, nov.-dec. 2005.

[21] O. Kutlubay, B. Turhan, and A. Bener. A two-step model for defect density estimation. In *Software Engineering and Advanced Applications, 2007. 33rd EUROMICRO Conference on*, pages 322 –332, aug. 2007.

[22] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *Software Engineering, IEEE Transactions on*, 34(4):485 –496, july-aug. 2008.

[23] T. Mende and R. Koschke. Effort-aware defect prediction models. In *Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on*, pages 107–116, 2010.

[24] T. Menzies, A. Dekhtyar, J. Distefano, and J. Greenwald. Problems with precision: A response to "comments on 'data mining static code attributes to learn defect predictors'". *Software Engineering, IEEE Transactions on*, 33(9):637 –640, sept. 2007.

[25] T. Menzies, J. Greenwald, and A. Frank. Data mining static code attributes to learn defect predictors. *Software Engineering, IEEE Transactions on*, 33(1):2 –13, jan. 2007.

[26] T. Ostrand and E. Weyuker. How to measure success of fault prediction models. In *Fourth international workshop on Software quality assurance: in conjunction with the 6th ESEC/FSE joint meeting*, pages 25–30. ACM, 2007.

[27] G. Pai and J. Dugan. Empirical analysis of software fault content and fault proneness using bayesian methods. *Software Engineering, IEEE Transactions on*, 33(10):675 –686, oct. 2007.

[28] N. Pizzi, A. Summers, and W. Pedrycz. Software quality prediction using median-adjusted class labels. In *Neural Networks, 2002. IJCNN '02. Proceedings of the 2002 International Joint Conference on*, volume 3, pages 2405 –2409, 2002.

[29] N. Seliya, T. Khoshgoftaar, and S. Zhong. Analyzing software quality with limited fault-proneness defect data. In *High-Assurance Systems Engineering, 2005. HASE 2005. Ninth IEEE International Symposium on*, pages 89 –98, oct. 2005.

[30] Y. Sun, C. Castellano, M. Robinson, R. Adams, A. Rust, and N. Davey. Using pre & post-processing methods to improve binding site predictions. *Pattern Recognition*, 42(9):1949–1958, 2009.

[31] B. Turhan, G. Kocak, and A. Bener. Data mining source code for locating software bugs: A case study in telecommunication industry. *Expert Systems with Applications*, 36(6):9986–9990, 2009.

[32] T. Wang and W.-h. Li. Naive bayes software defect prediction model. In *Computational Intelligence and Software Engineering (CiSE), 2010 International Conference on*, pages 1–4. accept, 2010.

[33] L. Yi, T. M. Khoshgoftaar, and N. Seliya. Evolutionary optimization of software quality modeling with multiple repositories. *Software Engineering, IEEE Transactions on*, 36(6):852–864, 2010.

[34] A. Zeller, T. Zimmermann, and C. Bird. Failure is a four-letter word: a parody in empirical research. In *Proceedings of the 7th International Conference on Predictive Models in Software Engineering*, Promise '11, pages 5:1–5:7, New York, NY, USA, 2011. ACM.

[35] H. Zhang and X. Zhang. Comments on "data mining static code attributes to learn defect predictors". *Software Engineering, IEEE Transactions on*, 33(9):635 –637, sept. 2007.

[36] Y. Zhou and H. Leung. Empirical analysis of object-oriented design metrics for predicting high and low severity faults. *Software Engineering, IEEE Transactions on*, 32(10):771 –789, oct. 2006.

APPENDIX

List of Papers from which we have Recomputed Confusion Matrices.

E. Arisholm, L. C. Briand, and M. Fuglerud. Data mining techniques for building fault-proneness models in telecom java software. In *ISSRE '07. The 18th IEEE Intern Symp on*, pages 215 –224, 2007.

E. Arisholm, L. C. Briand, and E. B. Johannessen. A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. *Journal of Systems and Software*, 83(1):2–17, 2010.

C. Bird, N. Nagappan, H. Gall, B. Murphy, and P. Devanbu. Putting it all together: Using socio-technical networks to predict failures. In *20th International Symposium on Software Reliability Engineering*, pages 109–119. IEEE, 2009.

L. Briand, W. Melo, and J. Wust. Assessing the applicability of fault-proneness models across object-oriented software projects. *Software Engineering, IEEE Transactions on*, 28(7):706 – 720, 2002.

B. Caglayan, A. Bener, and S. Koch. Merits of using repository metrics in defect prediction for open source projects. In *FLOSS '09. ICSE Workshop on*, pages 31–36, 2009.

G. Calikli, A. Tosun, A. Bener, and M. Celik. The effect of granularity level on software defect prediction. In *Computer and Information Sciences, 2009. ISCIS 2009. 24th International Symposium on*, pages 531 –536, 2009.

C. Catal, B. Diri, and B. Ozumut. An artificial immune system approach for fault prediction in object-oriented software. In *Dependability of Computer Systems, 2007. DepCoS-RELCOMEX '07. 2nd International Conference on*, pages 238 –245, 2007.

C. Cruz and A. Erika. Exploratory study of a uml metric for fault prediction. In *Proceedings of the 32nd ACM/IEEE Intern Conf on Software Engineering*, pages 361–364. 2010.

A. B. de Carvalho, A. Pozo, S. Vergilio, and A. Lenz. Predicting fault proneness of classes trough a multiobjective particle swarm optimization algorithm. In *Tools with Artificial Intelligence, 2008. ICTAI '08. 20th IEEE International Conference on*, volume 2, pages 387–394, 2008.

A. B. de Carvalho, A. Pozo, and S. R. Vergilio. A symbolic fault-prediction model based on multiobjective particle swarm optimization. *J of Sys & Soft*, 83(5):868–882, 2010.

G. Denaro and M. Pezzè. An empirical evaluation of fault-proneness models. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 241–251, NY, USA, 2002. ACM.

L. Guo, Y. Ma, B. Cukic, and H. Singh. Robust prediction of fault-proneness by random forests. In *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on*, pages 417 – 428, 2004.

T. Gyimothy, R. Ferenc, and I. Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *Software Engineering, IEEE Transactions on*, 31(10):897 – 910, 2005.

Z. Hongyu. An investigation of the relationships between

lines of code and defects. In *Software Maintenance, 2009. IEEE Intern Conf on*, pages 274–283, 2009.

Y. Jiang, B. Cukic, and Y. Ma. Techniques for evaluating fault prediction models. *Empirical Software Engineering*, 13(5):561–595, 2008.

S. Kanmani, V. Uthariaraj, V. Sankaranarayanan, and P. Thambidurai. Object-oriented software fault prediction using neural networks. *Information and Software Technology*, 49(5):483–492, 2007.

A. Kaur and R. Malhotra. Application of random forest in predicting fault-prone classes. In *Advanced Computer Theory and Engineering, 2008, Internl Conf on*, 37–43, 2008.

A. Kaur, P. S. Sandhu, and A. S. Bra. Early software fault prediction using real time defect data. In *Machine Vision, 2009. Intern Conf on*, pages 242–245.

T. Khoshgoftaar and N. Seliya. Comparative assessment of software quality classification techniques: An empirical case study. *Empirical Software Engineering*, 9(3):229–257, 2004.

T. Khoshgoftaar, X. Yuan, E. Allen, W. Jones, and J. Hudepohl. Uncertain classification of fault-prone software modules. *Empirical Software Engineering*, 7(4):297–318, 2002.

A. Koru and H. Liu. Building effective defect-prediction models in practice. *Software, IEEE*, 22(6):23 – 29, 2005.

O. Kutlubay, B. Turhan, and A. Bener. A two-step model for defect density estimation. In *Software Engineering and Advanced Applications, 2007. 33rd EUROMICRO Conference on*, pages 322 –332, 2007.

Y. Ma, L. Guo, and B. Cukic. *Advances in Machine Learning Applications in Software Engineering*, chapter A statistical framework for the prediction of fault-proneness, pages 237–265. IGI Global, 2006.

T. Mende and R. Koschke. Effort-aware defect prediction models. In *Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on*, pages 107–116, 2010.

T. Menzies, J. Greenwald, and A. Frank. Data mining static code attributes to learn defect predictors. *Software Engineering, IEEE Transactions on*, 33(1):2 –13, 2007.

O. Mizuno, S. Ikami, S. Nakaichi, and T. Kikuno. Spam filter based approach for finding fault-prone software modules. In *Mining Software Repositories, 2007. ICSE '07. International Workshop on*, page 4, 2007.

O. Mizuno and T. Kikuno. Training on errors experiment to detect fault-prone software modules by spam filter. In *Procs European Software Engineering Conf and the ACM SIGSOFT symp on The foundations of software engineering*, ESEC-FSE '07, pages 405–414, 2007. ACM.

R. Moser, W. Pedrycz, and G. Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Software Engineering, 2008. ICSE '08. ACM/IEEE 30th Intern Conf on*, pages 181–190.

N. Nagappan, A. Zeller, T. Zimmermann, K. Herzig, and B. Murphy. Change bursts as defect predictors. In *Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on*, pages 309–318.

A. Nugroho, M. R. V. Chaudron, and E. Arisholm. Assessing uml design metrics for predicting fault-prone classes in a java system. In *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, pages 21–30.

G. Pai and J. Dugan. Empirical analysis of software fault content and fault proneness using bayesian methods. *Software Engineering, IEEE Trans on*, 33(10):675–686, 2007.

A. Schröter, T. Zimmermann, and A. Zeller. Predicting component failures at design time. In *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, pages 18–27. ACM, 2006.

N. Seliya, T. Khoshgoftaar, and S. Zhong. Analyzing software quality with limited fault-proneness defect data. In *High-Assurance Systems Engineering, 2005. IEEE Internl Symp on*, pages 89 –98, 2005.

S. Shivaji, E. J. Whitehead, R. Akella, and K. Sunghun. Reducing features to improve bug prediction. In *Automated Software Engineering, 2009. ASE '09. 24th IEEE/ACM International Conference on*, pages 600–604.

Y. Singh, A. Kaur, and R. Malhotra. Predicting software fault proneness model using neural network. *Product-Focused Software Process Improvement*, 5089:204–214, 2008.

A. Tosun and A. Bener. Reducing false alarms in software defect prediction by decision threshold optimization. In *Empirical Software Engineering and Measurement, ESEM 2009. International Symposium on*, pages 477–480.

B. Turhan and A. Bener. A multivariate analysis of static code attributes for defect prediction. In *Quality Software, 2007. Intern Conf on*, pages 231 –237, 2007.

O. Vandecruys, D. Martens, B. Baesens, C. Mues, M. De Backer, and R. Haesen. Mining software repositories for comprehensible software fault prediction models. *Journal of Systems and Software*, 81(5):823–839, 2008.

R. Vivanco, Y. Kamei, A. Monden, K. Matsumoto, and D. Jin. Using search-based metric selection and oversampling to predict fault prone modules. In *Electrical and Computer Engineering, 2010, Canadian Conf on*, pages 1–6.

L. Yi, T. M. Khoshgoftaar, and N. Seliya. Evolutionary optimization of software quality modeling with multiple repositories. *Soft Engin, IEEE Trans on*, 36(6):852–864, 2010.

Y. Zhou and H. Leung. Empirical analysis of object-oriented design metrics for predicting high and low severity faults. *Software Engineering, IEEE Trans on*, 32(10):771–789, 2006.

T. Zimmermann, R. Premraj, and A. Zeller. Predicting defects for eclipse. In *Predictor Models in Software Engineering, 2007. PROMISE'07*, page 9, 2007.

4.6 Paper 6: The State of Machine Learning Methodology in Software Fault Prediction.

Hall T, Bowes D (2012) The state of machine learning methodology in software fault prediction. In: Machine Learning and Applications (ICMLA), 2012 11th International Conference on, vol 2, pp 308 –313

The State of Machine Learning Methodology in Software Fault Prediction

Tracy Hall
Department of Information Systems and Computing
Brunel University
Uxbridge, UK
Email: tracy.hall@brunel.ac.uk

David Bowes
Science and Technology Research Institute
University of Hertfordshire
Hatfield, UK
d.h.bowes@herts.ac.uk

***Abstract*—The aim of this paper is to investigate the quality of methodology in software fault prediction studies using machine learning. Over two hundred studies of fault prediction have been published in the last 10 years. There is evidence to suggest that the quality of methodology used in some of these studies does not allow us to have confidence in the predictions reported by them. We evaluate the machine learning methodology used in 21 fault prediction studies. All of these studies use NASA data sets. We score each study from 1 to 10 in terms of the quality of their machine learning methodology (e.g. whether or not studies report randomising their cross validation folds). Only 10 out of the 21 studies scored 5 or more out of 10. Furthermore 1 study scored only 1 out of 10. When we plot these scores over time there is no evidence that the quality of machine learning methodology is better in recent studies. Our results suggest that there remains much to be done by both researchers and reviewers to improve the quality of machine learning methodology used in software fault prediction. We conclude that the results reported in some studies need to be treated with caution.**

***Index Terms*—machine learning; experimental techniques; methodology; fault prediction; software engineering.**

I. Introduction

In this paper we investigate the quality of methodology used in 21 studies using machine learning for software fault prediction. We focus on four methodological aspects of these studies. First, we look at the data used and how it is cleaned and pre-processed. Second, we look at the cross validation schemes implemented. Third, we look at how machine learning techniques have been applied and tuned. Fourth, we look at how studies have drawn conclusions on predictive performance. Our aim is to establish whether the quality of methodology used allows us to have confidence in the results reported in machine learning studies for software fault prediction.

The quality of methodology underpins the validity of reported results in all disciplines. Methodological weakness and error are regularly brought to light in published studies in many disciplines [1]. Such methodological weaknesses have subsequently invalidated many reported results. Important examples include the spurious link reported between the MMR (Measles, Mumps, and Rubella) vaccine and autism[1]. The repercussions of this are still reflected in sub-optimal take-up rates of this vaccination[2]. More recently methodological problems led to false reports of neutrino particles travelling faster than the speed of light[3].

Methodological weaknesses can also be found in the software fault prediction literature. Song et al [2] report that the quality of data used to build software fault prediction models can be low and can affect the efficacy of predictions. Gray et al [3] report that the way predictive performance of software fault prediction models is measured can be flawed. Gray et al also show that the poor data used by some models can introduce predictive bias [4]. Zeller et al [5] suggest that methodological rigour might be compromised in response to the pressure researchers feel under to publish positive fault prediction results.

In our previous systematic literature review [6] only 36 out of 208 fault prediction studies passed our fairly basic methodological quality check[4]. Furthermore we have also shown that methodological problems seriously compromise our confidence in the results of several previous studies. For example we show [7] that Elish and Elish's [8] highly cited work using Support Vector Machine (SVM) models in software fault prediction reports unexpected results.

It is very easy for researchers to inadvertently design weaknesses into their methodology. It is also very easy to make mistakes in the deployment of that methodology. Indeed we have made many methodological mistakes in our own studies. Methodological weakness and error is especially easy when using machine learning in software fault prediction. This is because researchers are likely to be software engineering experts rather than machine learning experts. This expertise problem is exacerbated by the large range of machine learners to choose from and the complexity involved in effectively deploying each machine learner, as well as the availability of seemingly easy to use tools (e.g. Weka). Song et al [9] observe that the complexity of predictive model building means that

[1] http://www.bmj.com/content/342/bmj.c7452

[2] http://www.hpa.org.uk/ProductsServices/LocalServices/NorthWest/ NorthWestPressReleases/nwest110629measlesvaccine15yearhigh/

[3] http://www.bbc.co.uk/news/science-environment-17139635

[4] Appendix B shows the basic empirical methods check we applied in our previous study. We have not previously analysed the machine learning methodology (e.g. cross validation scheme used etc. used by studies of fault prediction).

it is difficult to know how best to construct a model from all the options available.

In this paper we look systematically at the quality of machine learning methodology used in software fault prediction studies. We analyse the methodology of studies published 2000-2010. We analyse all 21 of these studies which use NASA data and machine learning classifiers. We develop a simple scoring scheme based on the literature and apply this scoring scheme to each of these 21 studies. Appendix A lists these 21 papers which are referenced in the text using [[x]].

The rest of this paper is structured as follows. The next section provides an overview of the background to methodological aspects of machine learning. Section Three describes our methods. Section Four outlines the threats to validity. Section Five presents our results. Section Six concludes by discussing the implications of our results.

II. Background

Using machine learning to build software fault prediction models has become very popular. Our previous systematic literature review [6] identified 208 such studies published between the beginning of 2000 and the end of 2010. Building such models is complex and the efficacy of the methods used underpin the validity of the results produced. There are many important and complex methodological elements that must be effectively addressed by studies. In this section we give a very brief overview of these elements and provide pointers to more detailed information.

A. Data

It is essential that there is integrity in the data used. This section discusses several important aspects of data integrity.

The origin of the data: It is important that data is obtained from a trusted source. For example, the NASA data (used by all the studies we analyse here) is available from two sites - the original NASA sponsored MDP site[5] and the PROMISE data site[6]. The origin of the NASA data used in the studies is important as the quality of the data hosted on PROMISE has been reported to be compromised [4].

Data cleaning: The quality of the data used in fault prediction models determines the reliability of predictions. It is common for there to be quality problems with fault data [4]. Data sets are often noisy and contain outliers and missing values. Such data problems can skew results and invalidate findings. Leibchen and Shepperd [10] report that many studies in software engineering do not seem to consider the quality of the data they use. It is very important that studies do explicitly consider the quality of the data on which models are built.

Data pre-processing: There are many ways in which data can be pre-processed. Repeated attributes are one important aspect of pre-processing data for machine learning. Results have been shown to be skewed if repeated or related attributes are allowed to dominate predictions [4]. Various techniques can be used to guard against this, including Principal Component Analysis and attribute selection. Menzies et al [11] suggest that applying feature reduction techniques is important in terms of reducing computational effort.

Data balance: Imbalanced data can severely compromise the reliability of some machine learners, especially when reporting predictive performance using some measures that are particularly sensitive to imbalanced data (e.g. accuracy). It is important that imbalanced data is acknowledged and dealt with where necessary. It is also important that the balance of data used in studies is reported. Reporting this data allows subsequent researchers to replicate and investigate reported results further. Substantially imbalanced data sets are commonly used in fault prediction studies (i.e. there are usually many more non-faulty units than faulty units) ([12], [13]). For example NASA data set PC2 has only 0.4% of data points belonging to the faulty class (23 out of 5589 data points). This class distribution should be taken into account during fault prediction because imbalanced data can strongly influence both: the training of a model, and the way predictive performance should be measured. Various approaches can be used including over/under sampling [14], and/or learning algorithm optimisation [15]. However learning techniques vary in their sensitivity to imbalanced data and a decision about the appropriateness of the data needs to be made for each model.

B. Cross validation

The separation of training and test data is an important aspect of prediction. There are many ways in which this separation can be done. Holdout is probably the simplest approach, where the original data set is split into two groups comprising: training set, test set. The model is developed using the training set and its performance is then assessed on the test set. The weakness of this approach is that results can be biased because of the way the data has been split. A safer approach is often n-fold cross validation, where the data is split into n groups $\{g_1..g_n\}$. Ten-fold cross validation is very common, where the data is randomly split into ten groups, and ten experiments carried out. For each of these experiments, one of the groups is used as the testing set, and all others combined together and preferably randomised to form a training set. Performance is then typically reported as an average across all ten experiments. M-N fold cross validation adds another step by generating M different N-fold cross validations which increases the reliability of the results and reduces problems due to the order of items in the training set. Stratified cross validation is an improvement to this process, and keeps the distribution of faulty and non-faulty data points approximately equal to the overall class distribution in each of the n bins.

C. Machine learning techniques

There are many different machine learning techniques being used to build software fault prediction models. These range from relatively simple statistical classifiers (e.g. Naive Bayes) to more sophisticated Support Vector Machine (SVM) classifiers. Witten [15] describes these classifiers in detail and Lessmann et al [16] provide a comparative analysis of their

[5] Hosted at: http://filesanywhere.com/fs/v.aspx?v=896a648c5e5e6f799b

[6] http://promisedata.org/

performance in software fault prediction. Most classification techniques are complex to understand and to deploy correctly. Many require expertise to tune them appropriately to the particular circumstances of the experiment. For example SVMs have been shown to require intensive tuning to function effectively [17]. It is also important that the right technique is selected for the right experimental circumstances. Some techniques perform at their best when deployed with data evenly balanced between binary classes, for example C4.5 decision trees ([18] and [19]). Other techniques perform well with imbalanced data, for example fuzzy based classifiers [20]. Many learning algorithms can have their various parameters adjusted in order to boost performance on imbalanced data. To preserve the integrity of predictive results it is essential that the right technique is correctly used in the right circumstances.

A variety of tools exist to automate the production of fault prediction models using machine learning. Weka[7] is a very popular such tool. Again, it is important that tools are used correctly and that techniques are tuned. The default settings of tools make producing a predictive model deceptively simple. But without tuning, tools may not produce an effective model given the particular technique and the particular data.

D. Predictive performance measurement

There are many ways in which the predictive performance of models can be measured (see [21], [3]). Many measures are based on elements of the confusion matrix or are derivatives of the confusion matrix. This matrix reports how the model classified the different fault categories compared to their actual classification (predicted versus observed). The efficacy of the performance measurement scheme used depends on the overall construction of the model and the data being used. Designing the right measurement scheme for the right model is not necessarily straightforward. However the appropriateness of that scheme underpins the confidence that we can have in the predictions reported.

Prediction studies usually aim to compare the performance of one model against another (e.g. [11], [16]). The performance of a range of machine learning techniques is often the focus of such comparative studies. When reporting the results of such comparisons the use of statistical inferencing methods are important. Such methods can formally test the existence or not of any apparent superior performance. Manually observing the performance results of a comparative study and informally reporting what the results seem to imply, without testing the statistical basis of these results, makes it difficult to have confidence in the conclusions draw.

E. Methodological quality criteria

Based on the methodological aspects of machine learning reported in the literature (and summarised in this section), we propose a set of basic criteria that a study using machine learning for fault prediction should adhere to. These criteria are based on practices recommended in the literature as previously described in this section. Table I shows those criteria. The criteria in the table are at a high level of granularity and are those methodological issues that most studies should apply and all studies should be cognisant of. The few studies which need not apply a particular criteria should still report the reasons for this. Studies should not only address the methodological criteria shown in Table I, but they should also explicitly report how they have addressed these criteria.

Criteria
1. Data cleaning should be used with PROMISE data otherwise MDP data should be used
2. Data cleaning should be done
3. Imbalanced data should be dealt with
4. Feature reduction techniques should be applied
5. At least 10 cross validation folds should be used
6. Cross validation folds should be randomised
7. At least 10 experiment repeats should be performed
8. Fold stratification should be used
9. Classifiers should be tuned
10. Comparative results should be based on statistical analysis

TABLE I: Machine learning methodological criteria

III. Methods

We systematically identified all studies predicting faults in code published from the beginning of 2000 to the end of 2010. We identified these studies using using the following search terms in the ACM Digital Library, IEEExplore and the ISI Web of Science[8]:

(Fault* OR bug* OR defect* OR errors OR corrections OR corrective OR fix*) *in title only* AND (Software) *anywhere in study*

Our searches identified 2,073 papers. After ensuring that each paper was an empirical study focused on predicting faults in software published in English we were left with 208 papers.

We then identified all those 62 studies based on using NASA data. We selected this sub-set of studies for detailed analysis as 1) we did not analyse these studies in our first study [6] and 2) we wanted to reduce the variability in the studies and so controlled the data on which the studies are based to only NASA data.

We applied a quality check to each paper to ensure that basic information about the study was reported. The details of this quality check are reported in [6] and summarised in Appendix B. Of the 62 NASA studies 21 which used machine learning techniques passed the quality check[9].

For the study reported here we then scored each of these 21 studies according to the 10 criteria shown in Table I. This results in each of the 21 studies receiving a score between

[7] http://www.cs.waikato.ac.nz/ml/weka/

[8] Full details of our search strategy are reported in [6]

[9] One quality check item was not applied to the NASA studies. This item required that studies reported the maturity of systems on which data was based. None of the NASA studies were able to report system maturity.

0 and 10. This score is a very rough and ready guide to the general quality of machine learning methodology reported in the studies. The scoring system works on the basis that if a paper reports any indication that a criterion has been met, then 1 will be added to the cumulative score for the paper. A study gains a score if the criteria has clearly been addressed by the study or the criteria is explicitly mentioned (even if only to explain why this criteria was not necessary).

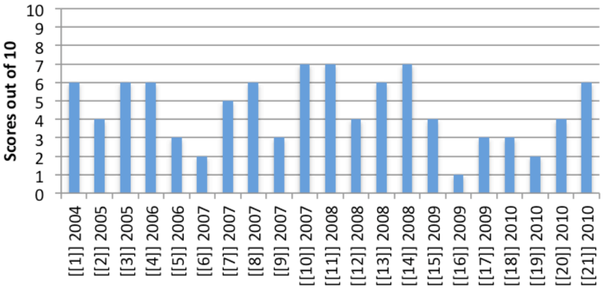


Fig. 1. Scores of each study in publication date order

Criteria	Papers passed (n)
1. Data cleaning should be used with PROMISE data otherwise MDP data should be used	14
2. Data cleaning should be done	10
3. Imbalanced data should be dealt with	5
4. Feature reduction techniques should be applied	6
5. At least 10 cross validation folds should be used	16
6. Cross validation folds should be randomised	7
7. At least 10 experiment repeats should be performed	11
8. Fold stratification should be used	4
9. Classifiers should be tuned	9
10. Comparative results should be based on statistical analysis	11

TABLE II: Results of applying the methodological criteria

IV. Threats to Validity

1. Our criteria are a very rough indication of quality as some studies may have done an extensive job of, for example, data cleaning whereas another study may have done a minimal job. Both such studies would receive the same score of 1. Consequently the scores we have awarded are likely to be somewhat inflated as any implementation of the criteria has been credited even when that implementation was relatively weak.

2. No detailed analysis of the interplay between the study factors has been done. For example we did not analyse the detailed methodological requirements of a particular study implementing a particular technique with a particular NASA data set, consequently our scoring system may miss the fact that a criteria was not actually necessary for a particular study. However the need to miss any criteria is rare. The vast majority of these criteria will apply to all studies. Consequently any score inaccuracies will be minimal.

3. All criteria are scored equally. It could be argued that some criteria impact more on model efficacy than other criteria. However there is little consensus on the relative importance on each criteria. This lack of consensus makes it difficult to implement a finer grained scoring system.

4. All the studies analysed in this paper use NASA data. It is not clear if the methodological approaches used with this subset of studies is representative of all fault prediction models. However there is nothing to lead us to believe that studies using these particular data sets are any different to studies using other data sets.

V. Results

Figure 1 presents the scores that individual studies received when we applied the criteria shown in Table I. Figure 1 shows that no study scored more than 7 out of 10. Only 10 of the 21 studies scored 5 or more. Indeed 1 of the studies scored only 1. Overall this suggests that the methodology used by most studies could be significantly improved. Figure 1 is ordered by publication date. This ordering shows that the scores of studies have not improved over time (this finding is confirmed by an autocorrelation test). Figure 1 provides no evidence that researchers are improving their machine learning techniques as more information on methodology comes to light over time in the literature.

Table II shows how many papers passed each of our 10 methodological criteria. Table II shows that 14 from the 21 studies passed our criteria ensuring that the origin of the data is safe (criterion 1). All 14 of these studies used the NASA data as hosted by MDP. None of the studies using PROMISE cleaned the data to overcome the known quality problems with that data.

Table II shows that only 10 from the 21 studies passed the criterion ensuring that data cleaning has been considered by studies (criterion 2). This means that less than half of the studies reported any form of data cleaning. The quality of the data cleaning done varies hugely.

Table II shows that 5 of the 21 studies passed the criterion ensuring that studies have considered the impact of the balance of data on their model (criterion 3). Two studies used oversampling while three studies used other data balancing techniques. A very low proportion of studies mention the impact of the balance of data they are using at all. In some cases balance is likely to be highly problematic given the severely imbalanced nature of some NASA data sets.

A few studies consider the balance of data where data preprocessing or cleaning has been applied. It is important that the impact of changing the data has on the balance of data is considered. Although the majority of studies do not need to re-consider the balance of data (as they did not apply data cleaning or pre-processing), of the 9 studies which did change the data, 5 studies did re-consider data balance. We consider this a sign of high quality studies.

Table II shows that 6 of the 21 studies passed the criteria ensuring that feature selection has been done (criterion 4). Of these 6 studies, 3 use feature sub-set selection.

Robust cross validation strategies form an important basis for confidence in predictive performance. There are a variety of elements involved in a good quality cross validation strategy. The number of folds used is one element. Table II shows that 16 out of the 21 studies passed the criteria ensuring that cross validation strategies were based on at least 10 data folds (criterion 5). Table II shows that only 7 studies report that those folds are randomised (criterion 6). Such randomisation adds rigour to the cross validation. Table II also shows that only 4 studies report using stratification (criteria 8). Table II shows that 11 out of the 21 studies report using at least 10 experiment repeats (criterion 7). Overall only 1 study [[10]] reports using the most sophisticated cross validation scheme (i.e. cross validation using at least 10 stratified randomised folds where at least 10 experiments are repeated).

Table II shows that 9 of the 21 studies report tuning their technique (criterion 9). Of those that do not, 4 explicitly report using the default parameters. Reporting this is a useful practice for subsequent replicability.

The vast majority of studies aim to draw some conclusions on the most effective approach to building a predictive model using the NASA data. In drawing these conclusions it is important that reliable inferences are made. Table II shows that 11 of the 21 studies draw their comparative conclusions based on statistical inference (criterion 10). This means that many studies use only informal argumentation and discussion with which to draw conclusions. This is a weak form of inference.

VI. Discussion and conclusions

It is clear that approaches to building models and checking the data on which they are built vary between studies. Some studies use data cleaning, report the balance of new datasets, balance training sets and present a well described and executed study (e.g. [[10]],[[11]] and [[14]]). More worrying is the number of studies which do not report key machine learning aspects of their methodology, e.g. tuning techniques, randomisation of folds etc.

Our results suggest that relatively novice approaches are being used to build some models. Basic principles of machine learning (as outlined in for example [15]) do not always seem to be followed. In particular very little technique tuning is reported by studies. Tuning has been shown to be particularly important when using some techniques (e.g. SVMs). Some studies report an unsophisticated approach to cross validation. It is not clear that studies are using cross validation schemes shown to be most effective for the fault prediction domain. Furthermore feature selection seems to be performed infrequently even though [11] demonstrates that feature selection can improve the production of predictive models. Our results also suggest that fault data imbalance does not really seem to be addressed adequately. Dealing with data imbalance is especially important with some of the highly imbalanced NASA data sets that are used by the studies we analysed. This omission could seriously compromise the results of studies. Very little data cleaning is being reported. This suggests that some researchers are not getting to know the data they are using (e.g. identifying impossible erroneous data points etc.). Such a failure to clean data can result in serious bias ([9] and [16]). Overall a significant minority of papers seem to demonstrate a serious lack of expertise in using machine learning.

A high proportion of studies seem to draw their comparative conclusions using informal argument rather than statistical testing. This means that conclusions on the ’best’ performing technique/cross validation scheme/data balancing scheme/etc. may not be sound. In addition many studies do not benchmark their results against those of others. This lack of benchmarking means that no corpus of understanding is being developed and every study may as well be the first. Such an approach does not improve the maturity of the machine learning use in software fault prediction.

Acknowledgment

The authors would like to thank Professor Martin Shepperd for his help in extracting the data on which this paper is based. And also for his helpful and insightful comments on drafts of this paper.

References

[1] K. Dickersin, “The existence of publication bias and risk factors for its occurrence,” *JAMA: the journal of the American Medical Association*, vol. 263, no. 10, pp. 1385–1389, 1990.

[2] Q. Song, M. Shepperd, M. Cartwright, and C. Mair, “Software defect association mining and defect correction effort prediction,” *Software Engineering, IEEE Transactions on*, vol. 32, no. 2, pp. 69 – 82, feb. 2006.

[3] D. Gray, D. Bowes, N. Davey, Y. Sun, and B. Christianson, “Further thoughts on precision,” in *Evaluation and Assessment in Software Engineering (EASE)*, 2011.

[4] ——, “The misuse of the nasa metrics data program data sets for automated software defect prediction,” in *Evaluation and Assessment in Software Engineering (EASE)*, 2011.

[5] A. Zeller, T. Zimmermann, and C. Bird, “Failure is a four-letter word: a parody in empirical research,” in *Proceedings of the 7th International Conference on Predictive Models in Software Engineering*, ser. Promise ’11. New York, NY, USA: ACM, 2011, pp. 5:1–5:7.

[6] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, “A systematic review of fault prediction performance in software engineering,” *Software Engineering, IEEE Transactions on*, vol. PP, no. 99, p. 1, 2011.

[7] D. Bowes, T. Hall, and D. Gray, “Comparing the performance of fault prediction models which report multiple performance measures: reconstructing the confusion matrix,” in *Proceedings of the 8th International Conference on Predictive Models in Software Engineering*, 2012.

[8] K. Elish and M. Elish, “Predicting defect-prone software modules using support vector machines,” *Journal of Systems and Software*, vol. 81, no. 5, pp. 649–660, 2008.

[9] Q. Song, Z. Jia, M. Shepperd, S. Ying, and J. Liu, “A general software defect-proneness prediction framework,” *Software Engineering, IEEE Transactions on*, vol. 37, no. 3, pp. 356 –370, may-june 2011.

[10] G. Liebchen and M. Shepperd, “Data sets and data quality in software engineering,” in *Proceedings of the 4th international workshop on Predictor models in software engineering*. ACM, 2008, pp. 39–44.

[11] T. Menzies, J. Greenwald, and A. Frank, “Data mining static code attributes to learn defect predictors,” *Software Engineering, IEEE Transactions on*, vol. 33, no. 1, pp. 2 –13, jan. 2007.

[12] N. V. Chawla, N. Japkowicz, and A. Kotcz, “Editorial: special issue on learning from imbalanced data sets,” *SIGKDD Explorations*, vol. 6, no. 1, pp. 1–6, 2004.

[13] H. Zhang and X. Zhang, “Comments on ”data mining static code attributes to learn defect predictors”,” *Software Engineering, IEEE Transactions on*, vol. 33, no. 9, pp. 635 –637, sept. 2007.

[14] H. He and E. Garcia, “Learning from imbalanced data,” *IEEE Transactions on Knowledge and Data Engineering*, pp. 1263–1284, 2008.

[15] I. Witten, E. Frank, and M. Hall, *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2011.

[16] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, “Benchmarking classification models for software defect prediction: A proposed framework and novel findings,” *Software Engineering, IEEE Transactions on*, vol. 34, no. 4, pp. 485 –496, july-aug. 2008.

[17] T. Segaran, *Programming collective intelligence: building smart web 2.0 applications*. O’Reilly Media, 2007.

[18] N. Japkowicz and S. Stephen, “The class imbalance problem: A systematic study,” *Intell. Data Anal*, vol. 6, no. 5, pp. 429–449, 2002.

[19] W. Liu, S. Chawla, D. A. Cieslak, and N. V. Chawla, “A robust decision tree algorithm for imbalanced data sets,” in *SDM*. SIAM, 2010, pp. 766–777.

[20] S. Visa and A. Ralescu, “Fuzzy classifiers for imbalanced, complex classes of varying size,” in *Information Processing and Management of Uncertainty in Knowledge-Based Systems*, 2004, pp. 393–400.

[21] T. Ostrand and E. Weyuker, “How to measure success of fault prediction models,” in *Fourth international workshop on Software quality assurance: in conjunction with the 6th ESEC/FSE joint meeting*. ACM, 2007, pp. 25–30.

Appendix A
Papers from which data was extracted

[[1]] L. Guo, Y. Ma, B. Cukic, and H. Singh, “Robust prediction of fault-proneness by random forests,” in *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on*, nov. 2004, pp. 417 – 428.

[[2]] A. Koru and H. Liu, “Building effective defect-prediction models in practice,” *Software, IEEE*, vol. 22, no. 6, pp. 23 – 29, nov.-dec. 2005.

[[3]] N. Seliya, T. Khoshgoftaar, and S. Zhong, “Analyzing software quality with limited fault-proneness defect data,” in *High-Assurance Systems Engineering, 2005. HASE 2005. Ninth IEEE International Symposium on*, oct. 2005, pp. 89 –98.

[[4]] Y. Ma, L. Guo, and B. Cukic, *Advances in Machine Learning Applications in Software Engineering*. IGI Global, 2006, ch. A statistical framework for the prediction of fault-proneness, pp. 237–265.

[[5]] Y. Zhou and H. Leung, “Empirical analysis of object-oriented design metrics for predicting high and low severity faults,” *Software Engineering, IEEE Transactions on*, vol. 32, no. 10, pp. 771 –789, oct. 2006.

[[6]] C. Catal, B. Diri, and B. Ozumut, “An artificial immune system approach for fault prediction in object-oriented software,” in *Dependability of Computer Systems, 2007. DepCoS-RELCOMEX ’07. 2nd International Conference on*, june 2007, pp. 238 –245.

[[7]] O. Kutlubay, B. Turhan, and A. Bener, “A two-step model for defect density estimation,” in *Software Engineering and Advanced Applications, 2007. 33rd EUROMICRO Conference on*, aug. 2007, pp. 322 –332.

[[8]] T. Menzies, J. Greenwald, and A. Frank, “Data mining static code attributes to learn defect predictors,” *Software Engineering, IEEE Transactions on*, vol. 33, no. 1, pp. 2 –13, jan. 2007.

[[9]] G. Pai and J. Dugan, “Empirical analysis of software fault content and fault proneness using bayesian methods,” *Software Engineering, IEEE Transactions on*, vol. 33, no. 10, pp. 675 –686, oct. 2007.

[[10]] B. Turhan and A. Bener, “A multivariate analysis of static code attributes for defect prediction,” in *Quality Software, 2007. QSIC ’07. Seventh International Conference on*, oct. 2007, pp. 231 –237.

[[11]] O. Vandecruys, D. Martens, B. Baesens, C. Mues, M. De Backer, and R. Haesen, “Mining software repositories for comprehensible software fault prediction models,” *Journal of Systems and Software*, vol. 81, no. 5, pp. 823–839, 2008.

[[12]] Y. Singh, A. Kaur, and R. Malhotra, “Predicting software fault proneness model using neural network,” *Product-Focused Software Process Improvement*, vol. 5089, pp. 204–214, 2008.

[[13]] Y. Jiang, B. Cukic, and Y. Ma, “Techniques for evaluating fault prediction models,” *Empirical Software Engineering*, vol. 13, no. 5, pp. 561–595, 2008.

[[14]] A. B. de Carvalho, A. Pozo, S. Vergilio, and A. Lenz, “Predicting fault proneness of classes trough a multiobjective particle swarm optimization algorithm,” in *Tools with Artificial Intelligence, 2008. ICTAI ’08. 20th IEEE International Conference on*, vol. 2, 2008, pp. 387–394.

[[15]] Z. Hongyu, “An investigation of the relationships between lines of code and defects,” in *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, 2009, pp. 274–283.

[[16]] A. Kaur, P. S. Sandhu, and A. S. Bra, “Early software fault prediction using real time defect data,” in *Machine Vision, 2009. ICMV ’09. Second International Conference on*. 2009, pp. 242–245.

[[17]] A. Tosun and A. Bener, “Reducing false alarms in software defect prediction by decision threshold optimization,” in *Empirical Software Engineering and Measurement, 2009. ESEM 2009. 3rd International Symposium on*. 2009, pp. 477–480.

[[18]] T. Mende and R. Koschke, “Effort-aware defect prediction models,” in *Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on*, 2010, pp. 107–116.

[[19]] R. Vivanco, Y. Kamei, A. Monden, K. Matsumoto, and D. Jin, “Using search-based metric selection and oversampling to predict fault prone modules,” in *Electrical and Computer Engineering (CCECE), 2010 23rd Canadian Conference on*. 2010, pp. 1–6.

[[20]] L. Yi, T. M. Khoshgoftaar, and N. Seliya, “Evolutionary optimization of software quality modeling with multiple repositories,” *Software Engineering, IEEE Transactions on*, vol. 36, no. 6, pp. 852–864, 2010.

[[21]] A. B. de Carvalho, A. Pozo, and S. R. Vergilio, “A symbolic fault-prediction model based on multiobjective particle swarm optimization,” *Journal of Systems and Software*, vol. 83, no. 5, pp. 868–882, 2010.

Appendix B
Summary of Methodology Check Applied [6]

Methodological criteria
1. Is a prediction model reported?
2. Is the prediction model tested on unseen data?
3. Is the source of data reported?
4. Is the maturity of data reported?
5. Is the size of data reported?
6. Is the application domain of data reported?
7. Is the programming language of the data reported?
8. Are the independent and dependent variables clearly reported?
9. Is the granularity of the dependent variables reported?
10. Are the modelling techniques used reported?
11. Is the fault data acquisition process described?
12. Is the independent variables data acquisition process described?
13. Is the faulty and non-faulty balance of data reported?

Chapter 5

Conclusion

Understanding the measurement protocols enables us to see the patterns better.

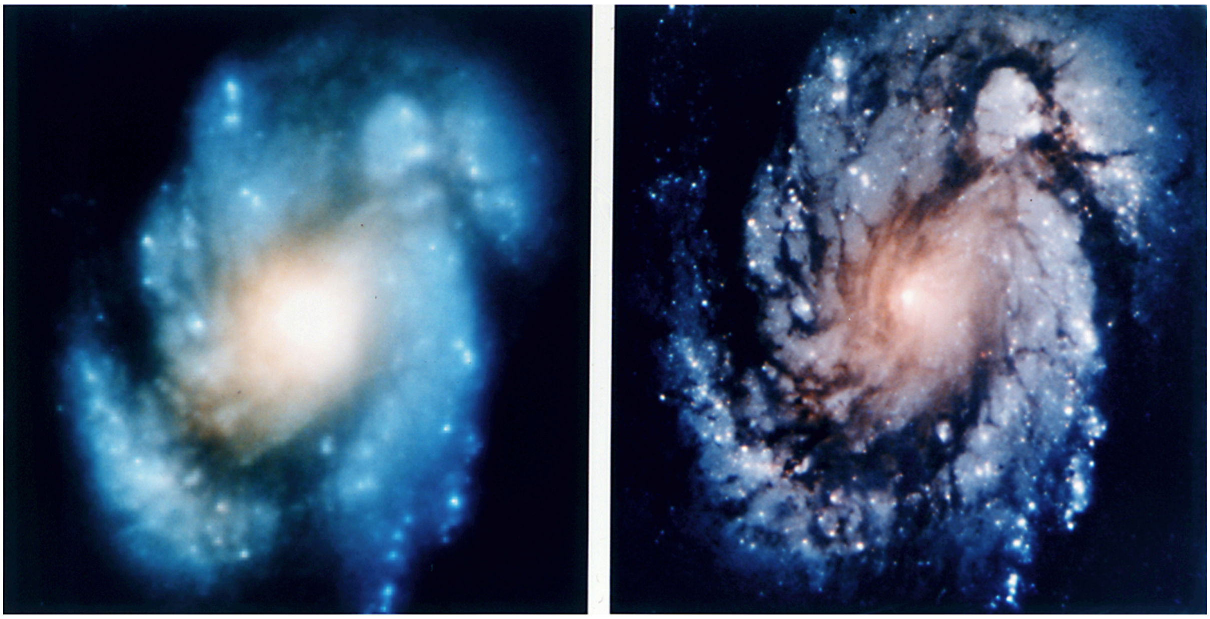


Figure 5.1: The Improvement in Image Quality by Changing the Measurement Protocol of the Hubble Space Observatory.
http://upload.wikimedia.org/wikipedia/commons/1/12/Improvement_in_Hubble_images_after_SMM1.jpg

5.1 Reflection on the Research Questions

To start the conclusion of this dissertation I will reflect on the initial research questions which were posed in the Introduction and answered in Chapter 3.

RQ1: Does the measurement protocol for the independent variables affect the metric values produced?

The results and conclusions from **[Paper 1]** and **[Paper 2]** demonstrate that the measurement protocol does affect the value of the metrics being computed. Having answered the question, we now want to know if this variation in values determined by the measurement protocol has an impact on defect prediction performance. The presentation in Appendix B.2 suggests that the measurement protocol does impact on the ability to predict defects. The suggestion that measurement protocol affects the ability to predict defects opens up the need for more research into how the measurement protocols impact on the ability of learners to predict defect prone modules. The results of Appendix B.2 also suggest that different measurement protocols may find different defect types. Being able to build a diverse set of models which identify different defect types is important when building ensembles of learners [Kuncheva and Whitaker 2003], therefore we may be able to exploit the different measurement protocols to build better ensembles.

RQ2: Is there an effective method for deriving the dependent variables for defect prediction?

In **[Paper 3]**, we conclude that the approaches for identifying where defects are in code do not agree strongly. This means that the additional noise caused by the presence of modules labelled as not defective when they are actually defective will reduce the opportunity to learn the characteristics of defects. Bird et al. [2009] reported that they had found more defects than Zimmermann et al. [2007] for the eclipse dataset. **[Paper 3]** in conjunction with [Zimmermann et al. 2007] and Bird et al. [2009] suggests that the impact of the noise generated by not being able to correctly label modules as defective or not should now be studied further. Kim et al. [2011] suggests that the performance of learners is affected when the percentage of false positives or false negatives is more than 20% of all instances. Our findings suggest that this value may be exceeded (24%) based on the low level of agreement that we discovered between the techniques used to label defects in **[Paper 3]**.

RQ3 a) Do different model building techniques have an impact on prediction models?

The results and conclusion of **[Paper 4]** show that there is little impact of using different modelling techniques on the predictive performance in defect prediction. This is in agreement with other studies [Lessmann et al. 2008] which also found that no single technique dominates when applied to the NASA datasets. **[Paper 4]** concludes that simple modelling techniques perform slightly better than more complicated modelling techniques.

RQ3 b) Do different datasets have an impact on prediction models?

The results of **[Paper 4]** show that the dataset does have an impact on the predictive performance of defect prediction studies. **[Paper 4]** also studied which metrics produced higher predictive performance measure values. Radjenović et al. [2013] have already extended our work by carrying out a more detailed analysis of the characteristics of the metrics which allow researchers to build better defect prediction models.

[Paper 4] has already been cited in 40 studies[1] because of its contribution to knowledge relating to the state of the art in defect prediction performance and the corpus of data which allows for other analysis such as the joint work with Martin Shepperd [Shepperd 2011; 2012; 2013][2].

[Paper 5] is an important contribution to the work in analysing the performance of defect prediction models across different studies. It has already been cited by others as a point of reference when reporting subsets of performance measures which can be used to re-compute the confusion matrix [Turhan et al. 2012]. The report of the first reviewer is included in Appendix A. In summary, the first reviewer highlighted the contribution of the technique for re-computing the the confusion matrix as follows: The technique produces a common performance measure (the confusion matrix) which can be used by others to compute their own preferred predictive performance measure. The technique does not constrain researchers to report a standard single predictive performance measure which may not in itself be a clear indicator of the effect that a researcher is trying to demonstrate. A final contribution is the explicit use of MCC as a performance measure for comparing predictive performance. The use of MCC in **[Paper 5]** makes the comparison of the performance of different studies both transparent. The use of MCC also highlights studies which are not doing better than making random predictions.

RQ4: Are the results of machine learning studies reliable/trustworthy?

[Paper 6] answers RQ4 by showing that machine learning approaches which are known to have an impact on the performance of machine learners, are not being applied systematically to defect prediction studies. This strongly suggests that the machine learning approach is an important factor which needs to be evaluated when assessing the predictive performance of a defect prediction study.

5.2 Main Findings

As was stated in the Introduction of this dissertation, defect prediction is an important technique which can help practitioners to improve the quality of software code. Helping to make the predictions as accurate as possible requires us to understand what factors may impact on the defect prediction models that we build. This dissertation has demonstrated that there are many factors which may affect the predictive performance of trainable models for software defect prediction. I have demonstrated the need for having well defined measurement protocols for both independent and dependent variables. I have also helped to synthesise the factors which affect defect prediction performance by studying the

[1] Citation count from Google scholar (5/06/2013).

[2] Martin Shepperd, Tracy Hall and I have completed a meta analysis of the results from **[Paper 4]** which shows that the single major factor which determines the predictive performance of defect prediction studies is which group of researchers worked together.

predictive performance of results published in a range of defect prediction studies. I have developed the idea that comparing the results of different studies is not simple, requiring a deeper understanding of the context of each result.

The papers that I have presented for this dissertation make a clear contribution to the thesis that there are many factors which may affect the performance of trainable models for software defect prediction. **[Paper 1]** and **[Paper 2]** demonstrate that there is a clear need to describe the measurement protocol for different independent variables. **[Paper 3]** demonstrates that there is a need to improve the measurement protocol for determining the dependent variables of defect prediction studies. **[Paper 4]** provides us with a clearer picture of the factors which have affected the predictive performance of models reported in published studies. **[Paper 5]** provides a technique for re-computing the confusion matrix which then allows us to compare the predictive performance of different studies. **[Paper 5]** also demonstrates that the re-computation of the confusion matrix is an important part of checking the validity of results published by researchers when carrying out defect prediction using binary classifiers. Finally, **[Paper 6]** shows that the good machine learning practices which should be used by researchers when carrying out defect prediction studies are not being reported in recent studies. The poor reporting of good machine learning practices is a concern and reduces the trust that we have in the results that are produced.

5.2.1 Analysis

It is clear that every area of defect prediction has factors which will result in some form of variability in the results. The measurement protocols for calculating the code metrics can vary significantly. No two techniques for labelling modules as defective or not have a good level of agreement with each other. The approaches used to perform machine learning experiments vary across different studies. Put together, all of these factors may affect the predictive performance reported by the many studies analysed.

Menzies and Shepperd [2012] and Shepperd and MacDonell [2012] illuminate the idea that the conclusions for different studies are contradictory or 'unstable' when applying machine learning to the problems of defect prediction and effort estimation. Menzies and Shepperd use the diagram from [Fayyad et al. 1996] (see Figure 5.2) to motivate some of the factors which may affect the conclusion. The conclusion of [Menzies and Shepperd 2012] is that we need better reporting of prediction studies and that learning local patterns may be more productive. It would be interesting to ask the question: Is the conclusion of Menzies and Shepperd [2012] reliable, or could it be affected by some other factor? On reflection we find that they have not included the measurement protocol factors which will impact on their result. I would suggest that conclusions are usually stable as long as good practice has been followed. They are stable because they work for the context in which they were placed. Asking the question: "Do bears live in woods?" will produce unstable conclusions, but it is obvious that the context is key: it depends on the continent. The conclusions may be achieved when internal threats to validity are low, but external threats to validity are high. I therefore suggest that the problem is context instability which is not reported rather than conclusion instability. To be able to join up the the results of different studies to form a global conclusion, requires us to be able to understand the context of the results. The argument that we need to investigate the locality of learning is probably correct. Learning from subsets of the data may overcome one of the pitfalls of comparing the performance of models from different systems. Cross project validation has a hidden assumption that the metric values are of similar ranges. Most statisticians know not to extrapolate a prediction beyond the end of the graph, yet cross project validation asks models to predict outcomes for metrics which are significantly outside of the metric space that the model is trained on. The problem with using subsets of data is that the different measurement protocols may mean that even when using the same source, the measurement protocol may project the values to different spaces. Chapter 4 clearly shows that the measurement protocol for program slicing metrics affects the values of the metrics computed. If we use the models built on small parts of the dataset and apply them to someone else's data using a different measurement protocol (on the same source), the models have no reason to work because the shift in metrics values between the two datasets caused by the different measurement protocols will make it appear that we are dealing with a different locality.

Aggregating data from many different sources may not be meaningful unless sufficient contextual information is provided. If we know the context and have sufficient data, it may be possible to re-compute a common dataset from which a global conclusion can be formed. DConfusion *[Bowes et al. 2013]* has allowed us to compare the predictive performance for defect prediction studies. We need to be aware that we have not been able to assess the measurement protocols for the different studies when applying DConfusion which may result in the findings of conclusion instability described in Shepperd [2012]. Having openly available datasets such as PROMISE allows us to remove some of the factor instability by removing the measurement protocol factors but context is still key. The process of being able to re-compute a comparable measurement allows researchers to report results in the way that they want to for the purpose of their paper while retaining the ability to use the data in cross study analysis.

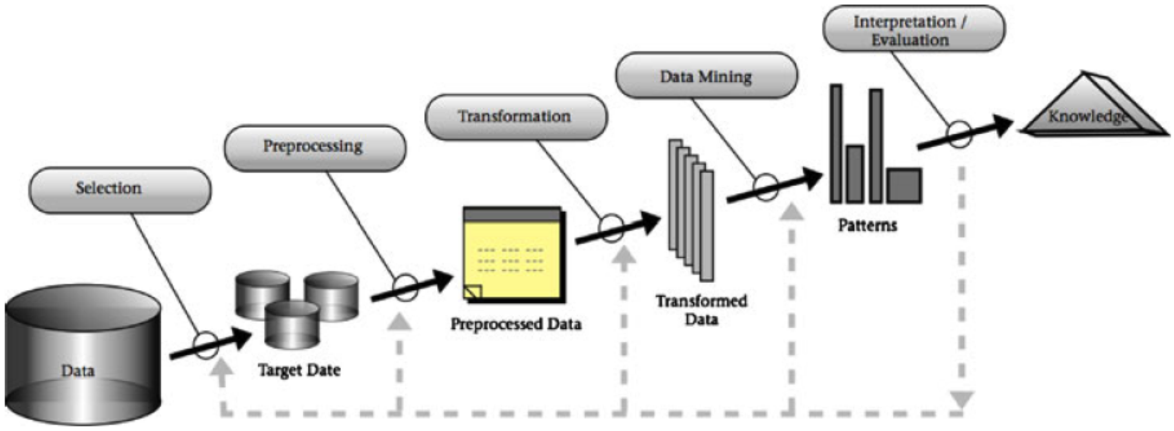


Figure 5.2: The KDD Cycle. From Fayyad et al. [1996]

5.3 Future Work

This dissertation has studied both empirically and experimentally some of the factors which may affect defect prediction studies which use machine learning. There are many questions which this dissertation raises which are areas for future work; some of which are described below.

5.3.1 Statistical Analysis of the Impact of Different Measurement Protocols on being able to Predict Defects

This dissertation has suggested that different measurement protocols will impact on the ability to predict defects. As was mentioned in the introduction, not all defects are the same. It is therefore completely possible that program slicing metrics based on print statements are limited to a few defects which cause output failures. Statistically proving the relationship is an ongoing study which the data from the papers included in this dissertation should help to answer. Using the approach of varying the measurement protocol and seeing if this gives a more reliable defect prediction model will hopefully give a better understanding of defects and the cause of defects.

5.3.2 Measurement Protocols

It is clear that changing the measurement protocol for program slicing metrics results in different results. What is not clear, is how other metrics can be and are affected by different measurement protocols. Shepperd [1995] goes some way to discussing the issues with computing LOC which when put to a first year University student appear to be simple. Measuring cyclomatic complexity and other metrics (such as the presence or absence of Fowlers' bad smells [Fowler and Beck 1999]) is also open to interpretation and this may affect experimental results which use these metrics.

5.3.3 Reporting Protocols

The method of reporting results has historically been by publications in archival papers. These papers are usually limited in space and may not be able to report the full contextual information needed to allow us to compare the results of one paper against another. There are at least two possible ways forward, the first is to establish a reporting protocol which succinctly describes the experimental methods carried out during the work and the second is to build open repositories which contain both the data and the tools used to produce the results.

5.3.4 The Need for Replication Studies

The replication of studies is an important part of scientific practice [Jasny et al. 2011]. In physics, it was important that Lonchampt et al. [1996] replicated the work of Fleischmann et al. [1989] in order to show that the measurement protocol could in itself could lead to the conclusion that cold fusion was possible. In software engineering, Sjøberg et al. [2003] suggest that the aim of academics is to have their work transfer into an industrial setting. Sjøberg et al. [2003] discuss how industry needs to have trust in the experiments that software engineers perform and recommend that replicating studies along with full reporting of the context of work is an important factor in building trust.

Bowes et al. [2011] clearly demonstrates that more studies need to be replicated in order to test the applicability of published results. The inability to replicate the results of Meyers and Binkley [2007] caused us to significantly delay publication of our results because we initially doubted our own results. By replicating **[Paper 1]** using an independent programmer, we were able to improve confidence in our own results.

It was while reading [Mende and Koschke 2009] that I fully realised the impact that poor reporting has on the trust that could be placed on the findings of a paper. Mende and Koschke [2009] show that while replicating two defect prediction studies, it was possible to only replicate the results of one paper. The second paper reported in [Mende and Koschke 2009] could not be replicated because of poor protocol reporting. Replication studies allow us to find the holes in papers by making us work through the problem again. Bruce Christianson extols the virtue of making PhD students at the beginning of their programme of work replicate the studies of others. He suggests that it stops students from just reading the paper and makes them think critically about the methodology and results. David Gray while carrying out his PhD [Gray 2013] repeated many of the defect prediction studies reported in **[Paper 6]** from which he could synthesise the methodological issues in defect prediction reported in his thesis. Removing the noise of poor papers will improve the chance that the next generation of researchers will have a good starting point on which to base their ideas.

5.3.5 Ensuring Consistency in Machine Learning Approaches

[Gray 2013] goes some way towards formalising the methodology for defect prediction studies. The results in **[Paper 6]** show that good practice has either not been carried out or not reported. The challenge now is to make sure that the good practices are continued and reported.

Using re-computation techniques will help to spot errors in papers, but so far, the technique has only been developed for binary classification problems. Extending the technique to other problems is possible but requires more data to be stored in repositories. If we record the prediction for every instance in every fold, we can truly compare models against each other. We can identify instances that are always predicted as defective and those which are occasionally predicted as defective. Knowing which instances are reliably predicted the same will allow us to better focus our attention and understanding on the less predictable instances.

5.3.6 Building Repositories of Comparable Data

Repositories of data need to be constructed which extend the PROMISE concept by including not only the initial data, but also the results and code. Being open and transparent will improve the trust we have in results and will give us a better understanding of the features of code which increase the probability that it is defective.

5.3.7 Why not What!

Throughout this dissertation I have constantly answered the question ‘what?’. The secret purpose has always been to put us in a better position to address the question ‘why?’. The purpose of ‘what?’ is irrelevant naval gazing if it does not allow us the better understand ‘why?’. If we do not attempt to find root cause and effect we will be limited to mere observers rather than participants in changing the reasons why defects are occurring in code. Creating models which are clearer and more understandable should be the goal of defect prediction. Although this dissertation has added to the problems by including yet another set of factors which need to be considered when building a model, I hope that the contribution will eventually enable us to build simpler, clearer models which will help others in their future work ask the question ‘why do these metrics predict defects?’

5.4 Final Remarks

The conclusion for this dissertation is short and stark:

- How independent variables are computed from the source code can have a statistically significant impact on the values of the metrics collected.
- There is no gold standard for collecting the dependent variables for defect prediction.
- There are many defect prediction studies which have used a large number of different datasets, independent variables and learners. The performance reported in these studies varies significantly depending on the dataset and to some extent the learner.
- The machine learning approaches developed over the last decade have not been adopted by many recent papers.

We therefore seem to be no further on than the studies (which motivated this dissertation) carried out in the 1980's, 1990's and now the 2000's. Papers are still ignoring sound practices in terms of measurement protocol validation and machine learning approaches. It is no wonder we have 'conclusion instability'. Having taken an empirical approach using real data rather than the synthetic data of Shepperd and Kadoda [2001], we still come to the same conclusion: how you do the experiment will affect how the model performs.

This dissertation helps to understand the problem of conclusion instability by showing that we need to be critical of the way that factors are computed and used in defect prediction and beyond.

This dissertation has made significant contributions to knowledge (see Section 1.4) and posses as many questions as it answers (see Section 5.3).

References

Arisholm E, Briand LC, Fuglerud M (2007) Data mining techniques for building fault-proneness models in telecom java software. In: Software Reliability, 2007. ISSRE '07. The 18th IEEE International Symposium on, pp 215 –224 (Cited on page 22.)

Arisholm E, Briand LC, Johannessen EB (2010) A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. Journal of Systems and Software 83(1):2–17 (Cited on page 22.)

Baker AL, Bieman JM, Fenton N, Gustafson DA, Melton A, Whitty R (1990) A philosophy for software measurement. Journal of Systems and Software 12(3):277 – 281, oregon Workshop on Software Metrics (Cited on page 10.)

Basili VR, Selby RW (1987) Comparing the effectiveness of software testing strategies. IEEE Trans Softw Eng 13(12):1278–1296 (Cited on page 8.)

Beck K, Fowler M, Beck G (1999) Bad smells in code. Refactoring: Improving the design of existing code pp 75–88 (Cited on page 32.)

Bird C, Bachmann A, Aune E, Duffy J, Bernstein A, Filkov V, Devanbu P (2009) Fair and balanced?: bias in bug-fix datasets. In: Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, ACM, New York, NY, USA, ESEC/FSE '09, pp 121–130 (Cited on pages 33 and 120.)

Black S, Counsell S, Hall T, Wernick P (2006) Using program slicing to identify faults in software. In: Binkley DW, Harman M, Krinke J (eds) Beyond Program Slicing, Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, Dagstuhl, Germany, no. 05451 in Dagstuhl Seminar Proceedings (Cited on page 31.)

Boetticher G (2006) Advanced machine learner applications in software engineering, Idea Group Publishing, Hershey, PA, USA, chap Improving credibility of machine learner models in software engineering, pp 52 – 72 (Cited on page 20.)

Boetticher G, Menzies T, Ostrand T (2007) Promise repository of empirical software engineering data. West Virginia University, Department of Computer Science (Cited on page 10.)

Boser BE, Guyon IM, Vapnik VN (1992) A training algorithm for optimal margin classifiers. In: Proceedings of the fifth annual workshop on Computational learning theory, ACM, New York, NY, USA, COLT '92, pp 144–152 (Cited on page 20.)

Bowes D, Hall T (2010) Using program slicing data to predict code faults. In: The 3rd CREST Open Workshop (Cited on pages 32 and 33.)

Bowes D, Hall T (2011) SLuRp: A web enabled database for effective management of systematic literature reviews. Tech. Rep. 510, University of Hertfordshire (Cited on page 4.)

Bowes D, Hall T, Kerr A (2011) Program slicing-based cohesion measurement: the challenges of replicating studies using metrics. In: Proceeding of the 2nd international workshop on Emerging trends in software metrics, ACM, pp 75–80 (Cited on pages 3, 32 and 124.)

Bowes D, Hall T, Beecham S (2012a) SLuRp: a tool to help large complex systematic literature reviews deliver valid and rigorous results. In: Proceedings of the 2nd international workshop on Evidential assessment of software technologies, ACM, pp 33–36 (Cited on pages 5, 35 and 37.)

Bowes D, Hall T, Gray D (2012b) Comparing the performance of fault prediction models which report multiple performance measures: reconstructing the confusion matrix. In: Proceedings of the 8th International Conference on Predictive Models in Software Engineering (Cited on pages vi, 135 and 136.)

Bowes D, Hall T, Gray D (2013) DConfusion: A technique to allow cross study performance evaluation of fault prediction studies. Automated Software Engineering Journal (In Review) (Cited on pages 4, 5, 35 and 123.)

Breiman L (2001) Random forests. Machine Learning 45:5–32 (Cited on page 18.)

Breiman L, Friedman J, Stone CJ, Olshen RA (1984) Classification and regression trees. Chapman & Hall/CRC (Cited on page 17.)

Catal C, Diri B (2009) A systematic review of software fault prediction studies. Expert Systems with Applications 36(4):7346–7354 (Cited on page 8.)

Chawla N, Bowyer K, Hall L, Kegelmeyer W (2002) Smote: synthetic minority over-sampling technique. Journal of Artificial Intelligence Research 16(1):321–357 (Cited on page 22.)

Corazza A, Di Martino S, Ferrucci F, Gravino C, Sarro F, Mendes E (2010) How effective is tabu search to configure support vector regression for effort estimation? In: Proceedings of the 6th International Conference on Predictive Models in Software Engineering, ACM, New York, NY, USA, PROMISE '10, pp 4:1–4:10 (Cited on pages 20 and 23.)

Cortes C, Vapnik V (1995) Support-vector networks. Mach Learn 20(3):273–297 (Cited on page 20.)

Counsell S, Bowes D, Hall T (2009) Cohesion Metrics: The Empirical Contradiction. In: The Psychology of Programming Interest Group, Open University (Cited on pages 4 and 32.)

Elish K, Elish M (2008) Predicting defect-prone software modules using support vector machines. Journal of Systems and Software 81(5):649–660 (Cited on page 35.)

Fayyad U, Piatetsky-Shapiro G, Smyth P (1996) From data mining to knowledge discovery in databases. AI magazine 17(3):37 (Cited on pages ix, 122 and 123.)

Fenton N, Neil M (1999) A critique of software defect prediction models. Software Engineering, IEEE Transactions on 25(5):675 –689 (Cited on pages 2, 3 and 20.)

Fenton N, Pfleeger SL (1997) Software metrics (2nd ed.): a rigorous and practical approach. PWS Publishing Co., Boston, MA, USA (Cited on pages 10 and 11.)

Fenton NE (1991) Software Metrics: A Rigorous Approach. Chapman & Hall, Ltd., London, UK, UK (Cited on page 10.)

Fleischmann M, Pons S, Hawkins M (1989) Electrochemically induced nuclear fusion of deuterium. Journal of Electroanalytical Chemistry 261(2):301–308 (Cited on page 124.)

Forman G, Scholz M (2010) Apples-to-apples in cross-validation studies: pitfalls in classifier performance measurement. ACM SIGKDD Explorations Newsletter 12(1):49–57 (Cited on page 26.)

Foss T, Stensrud E, Kitchenham B, Myrtveit I (2003) A simulation study of the model evaluation criterion mmre. Software Engineering, IEEE Transactions on 29(11):985 – 995 (Cited on pages 22, 25 and 67.)

Fowler M, Beck K (1999) Refactoring: Improving the Design of Existing Code. Addison-Wesley, Boston, MA, USA (Cited on pages 32 and 124.)

Glover F (1989) Tabu search—part i. ORSA Journal on computing 1(3):190–206 (Cited on page 20.)

Glover F (1990) Tabu search—part ii. ORSA Journal on computing 2(1):4–32 (Cited on page 20.)

Glover F, McMillan C (1986) The general employee scheduling problem: an integration of ms and ai. Comput Oper Res 13(5):563–573 (Cited on page 20.)

Gray D (2013) Software defect prediction using static code metrics : Formulating a methodology. PhD thesis, Computer Science, University of Hertfordshire (Cited on pages 21 and 125.)

Gray D, Bowes D, Davey N, Yi S, Christianson B (2010) Software defect prediction using static code metrics underestimates defect-proneness. In: Neural Networks (IJCNN), The 2010 International Joint Conference on, pp 1–7 (Cited on page 22.)

Gray D, Bowes D, Davey N, Sun Y, Christianson B (2012) Reflections on the nasa mdp data sets. Software, IET 6(6):549 –558 (Cited on page 20.)

Hall T, Bowes D (2011) Issues of consistency in defining slices for slicing metrics: ensuring comparability in research findings. In: The 10th CREST Open Workshop (Cited on page 32.)

Hall T, Beecham S, Bowes D, Gray D, Counsell S (2011) Developing fault-prediction models: What the research can show industry. Software, IEEE 28(6):96 –99 (Cited on page 35.)

Hall T, Beecham S, Bowes D, Gray D, Counsell S (2012) A systematic literature review on fault prediction performance in software engineering. Software Engineering, IEEE Transactions on 38(6):1276 –1304 (Cited on pages 35 and 37.)

Halstead MH (1977) Elements of Software Science (Operating and programming systems series). Elsevier Science Inc., New York, NY, USA (Cited on page 10.)

He H, Garcia E (2008) Learning from imbalanced data. IEEE Transactions on Knowledge and Data Engineering pp 1263–1284 (Cited on page 36.)

Hertz JA, Krogh AS, Palmer RG (1991) Introduction to the theory of neural computation, vol 1. Westview press (Cited on page 18.)

IEEE (1990) Ieee standard glossary of software engineering terminology. IEEE Std 61012-1990 p 1 (Cited on page 7.)

Izurieta C, Bieman J (2008) Testing consequences of grime buildup in object oriented design patterns. In: Software Testing, Verification, and Validation, 2008 1st International Conference on, pp 171 –179 (Cited on page 32.)

Japkowicz N, Stephen S (2002) The class imbalance problem: A systematic study. Intell Data Anal 6(5):429–449 (Cited on pages 22 and 36.)

Jasny BR, Chin G, Chong L, Vignieri S (2011) Again, and again, and again .... Science 334(6060):1225 (Cited on page 124.)

Khoshgoftaar T, Seliya N, Gao K (2005) Assessment of a new three-group software quality classification technique: An empirical case study. Empirical Software Engineering 10(2):183–218 (Cited on page 11.)

Kim S, Zhang H, Wu R, Gong L (2011) Dealing with noise in defect prediction. In: Proceedings of the 33rd International Conference on Software Engineering, ACM, New York, NY, USA, ICSE '11, pp 481–490 (Cited on page 120.)

Kitchenham B, Pickard L, Linkman S (1990) An evaluation of some design metrics. Software Engineering Journal 5(1):50 –58 (Cited on pages 2, 3 and 32.)

Kitchenham B, Pfleeger S, Fenton N (1995) Towards a framework for software measurement validation. Software Engineering, IEEE Transactions on 21(12):929 –944 (Cited on page 10.)

Kohavi R, et al. (1995) A study of cross-validation and bootstrap for accuracy estimation and model selection. In: International joint Conference on artificial intelligence, Lawrence Erlbaum Associates Ltd, vol 14, pp 1137–1145 (Cited on page 36.)

Kuncheva L, Whitaker C (2003) Measures of diversity in classifier ensembles and their relationship with the ensemble accuracy. Machine Learning 51:181–207 (Cited on page 120.)

Lessmann S, Baesens B, Mues C, Pietsch S (2008) Benchmarking classification models for software defect prediction: A proposed framework and novel findings. Software Engineering, IEEE Transactions on 34(4):485 –496 (Cited on pages 1, 18, 35 and 120.)

Levinson M (2001) Let's stop wasting $78 billion a year'. CIO, 15th October pp 78–83 (Cited on pages 1 and 2.)

Li PL, Herbsleb J, Shaw M, Robinson B (2006) Experiences and results from initiating field defect prediction and product test prioritization efforts at abb inc. In: Proceedings of the 28th international conference on Software engineering, ACM, New York, NY, USA, ICSE '06, pp 413–422 (Cited on page 1.)

Liebchen G, Shepperd M (2008) Data sets and data quality in software engineering. In: Proceedings of the 4th international workshop on Predictor models in software engineering, ACM, pp 39–44 (Cited on page 36.)

Liu W, Chawla S, Cieslak DA, Chawla NV (2010) A robust decision tree algorithm for imbalanced data sets. In: SDM, SIAM, pp 766–777 (Cited on page 22.)

Lokan C (2005) What should you optimize when building an estimation model? In: Software Metrics, 2005. 11th IEEE International Symposium, pp 10 pp. –34 (Cited on page 22.)

Lonchampt G, Bonnetain L, Hieter P (1996) Reproduction of fleischmann and pons experiments. In: Sixth International Conference on Cold Fusion, Progress in New Hydrogen Energy, p 113 (Cited on page 124.)

McCabe T (1976) A complexity measure. Software Engineering, IEEE Transactions on SE-2(4):308 – 320 (Cited on page 10.)

Mende T, Koschke R (2009) Revisiting the evaluation of defect prediction models. In: Proceedings of the 5th International Conference on Predictor Models in Software Engineering, ACM, p 7 (Cited on page 125.)

Menzies T (2011) S.p.a.c.e. exploration for software engineering. In: The 15th CREST Open Workshop (Cited on page 28.)

Menzies T, Shepperd M (2012) Special issue on repeatable results in software engineering prediction. Empirical Software Engineering pp 1–17 (Cited on page 122.)

Menzies T, Greenwald J, Frank A (2007) Data mining static code attributes to learn defect predictors. Software Engineering, IEEE Transactions on 33(1):2 –13 (Cited on pages 11, 21 and 36.)

Menzies T, Turhan B, Bener A, Gay G, Cukic B, Jiang Y (2008) Implications of ceiling effects in defect predictors. In: Proceedings of the 4th international workshop on Predictor models in software engineering, ACM, New York, NY, USA, PROMISE '08, pp 47–54 (Cited on pages 1 and 2.)

Menzies T, Milton Z, Turhan B, Cukic B, Jiang Y, Bener AB (2010) Defect prediction from static code features: current results, limitations, new approaches. Autom Softw Eng 17(4):375–407 (Cited on page 2.)

Menzies T, Caglayan B, Kocaguneli E, Krall J, Peters F, Turhan B (2012) The promise repository of empirical software engineering data. URL http://promisedata.googlecode.com (Cited on page 10.)

Meyers TM, Binkley D (2007) An empirical study of slice-based cohesion and coupling metrics. ACM Trans Softw Eng Methodol 17(1):2:1–2:27 (Cited on pages 32 and 124.)

Miyazaki Y, Terakado M, Ozaki K, Nozaki H (1994) Robust regression for developing software estimation models. Journal of Systems and Software 27(1):3 – 16 (Cited on page 22.)

Morasca S, Briand L (1997) Towards a theoretical framework for measuring software attributes. In: Software Metrics Symposium, 1997. Proceedings., Fourth International, pp 119 –126 (Cited on page 10.)

Munson J, Khoshgoftaar T (1990) Regression modelling of software quality: empirical investigation. Information and Software Technology 32(2):106 – 114 (Cited on page 1.)

Nagappan N, Ball T (2005) Use of relative code churn measures to predict system defect density. In: Proceedings of the 27th international conference on Software engineering, ACM, New York, NY, USA, ICSE '05, pp 284–292 (Cited on page 11.)

Newman M (2005) Power laws, pareto distributions and zipf's law. Contemporary physics 46(5):323–351 (Cited on page 12.)

Oram A, Wilson G (eds) (2010) Making Software What Really Works, and Why We Believe It. O'Reilly Media (Cited on pages 2, 7, 12 and 13.)

Pinzger M, Nagappan N, Murphy B (2008) Can developer-module networks predict failures? In: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering, ACM, pp 2–12 (Cited on page 11.)

Port D, Korte M (2008) Comparative studies of the model evaluation criterions mmre and pred in software cost estimation research. In: Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement, ACM, New York, NY, USA, ESEM '08, pp 51–60 (Cited on page 22.)

Quinlan J (1993) C4. 5: programs for machine learning, vol 1. Morgan kaufmann (Cited on page 17.)

Radjenović D, Heričko M, Torkar R, Živkovič A (2013) Software fault prediction metrics: A systematic literature review. Information and Software Technology Accepted for publication (Cited on pages 8 and 121.)

Rezwan F, Davey N, Sun Y, Adams R (2013) Effect of using varying negative examples in transcription factor binding site predictions. Applied Soft Computing (Cited on pages 22, 23 and 24.)

Rosenberg J (1997) Some misconceptions about lines of code. In: Software Metrics Symposium, 1997. Proceedings., Fourth International, pp 137 –142 (Cited on page 2.)

Runeson P, Andrews A (2003) Detection or isolation of defects? an experimental comparison of unit testing and code inspection. In: Software Reliability Engineering, 2003. ISSRE 2003. 14th International Symposium on, IEEE, pp 3–13 (Cited on pages 1 and 2.)

Salehi F, Lacroix R, Wade KM (1998) Effects of learning parameters and data presentation on the performance of backpropagation networks for milk yield prediction (Cited on page 18.)

Scott JSJSJ (2012) Ca technologies works with rbs on technical fault as outage enters fifth day. URL http://www.computerweekly.com/news/2240158589/CA-Technologies-works-with-RBS-on-technical-fault-as-outage-enters-fifth-day (Cited on page 2.)

Shannon CE, Weaver W (1948) A mathematical theory of communication (Cited on page 17.)

Shepperd M (1988) A critique of cyclomatic complexity as a software metric. Software Engineering Journal 3(2):30 –36 (Cited on page 10.)

Shepperd M (1995) Foundations of software measurement. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK (Cited on pages 4 and 124.)

Shepperd M (2011) It doesn't matter what you do but does matter who does it! In: The 15th CREST Open Workshop (Cited on page 121.)

Shepperd M (2012) The scientific basis for prediction research. In: Proceedings of the 8th International Conference on Predictive Models in Software Engineering, ACM, pp 1–2 (Cited on pages 121 and 123.)

Shepperd M (2013) Assessing the predictive performance of machine learners in software engineering. In: The 24th CREST Open Workshop (Cited on pages 1, 22 and 121.)

Shepperd M, Kadoda G (2001) Comparing software prediction techniques using simulation. Software Engineering, IEEE Transactions on 27(11):1014 –1022 (Cited on page 126.)

Shepperd M, MacDonell S (2012) Evaluating prediction systems in software project estimation. Information and Software Technology 54(8):820 – 827, special Issue: Voice of the Editorial Board (Cited on page 122.)

Shirabad J, Menzies T (2005) The promise repository of software engineering databases. School of Information Technology and Engineering, University of Ottawa, Canada 24 (Cited on page 10.)

Sjøberg D, Anda B, Arisholm E, Dybå T, Jørgensen M, Karahasanović A, Vokáč M (2003) Challenges and recommendations when increasing the realism of controlled software engineering experiments. Empirical methods and studies in software engineering pp 24–38 (Cited on page 124.)

Śliwerski J, Zimmermann T, Zeller A (2005) When do changes induce fixes? In: ACM SIGSOFT Software Engineering Notes, ACM, vol 30, pp 1–5 (Cited on page 33.)

Stevens SS (1946) On the theory of scales of measurement. Science 103(2684):677–680 (Cited on page 10.)

Turhan B, Menzies T, Bener A, Di Stefano J (2009) On the relative value of cross-company and within-company data for defect prediction. Empirical Software Engineering 14(5):540–578 (Cited on pages 2 and 21.)

Turhan B, MÄ±sÄ±rlÄ± AT, Bener A (2012) Empirical evaluation of the effects of mixed project data on learning defect predictors. Information and Software Technology (0):– (Cited on page 121.)

Vapnik V (1963) Pattern recognition using generalized portrait method. Automation and Remote Control 24:774–780 (Cited on page 20.)

Weiser M (1981) Program slicing. In: Proceedings of the 5th international conference on Software engineering, IEEE Press, Piscataway, NJ, USA, ICSE ’81, pp 439–449 (Cited on page 7.)

Weiser MD (1979) Program slices: formal, psychological, and practical investigations of an automatic program abstraction method. PhD thesis, Computer and Communication Sciences Dept., Ann Arbor, MI, USA, aAI8007856 (Cited on pages 7 and 11.)

Weyuker EJ, Ostrand TJ (2008) What can fault prediction do for you? In: Beckert B, Hähnle R (eds) Tests and Proofs, Lecture Notes in Computer Science, vol 4966, Springer Berlin / Heidelberg, pp 18–29 (Cited on page 1.)

Weyuker EJ, Ostrand TJ, Bell RM (2010) Comparing the effectiveness of several modeling methods for fault prediction. Empirical Software Engineering 15(3):277–295 (Cited on pages 11, 13 and 21.)

Witten I, Frank E (2002) Weka. Machine Learning Algorithms in Java In: Witten I, Frank E(eds) Data Mining: Practical Machine Learning Tools and Techniques with Java (Cited on page 17.)

Witten I, Frank E (2005) Data Mining: Practical machine learning tools and techniques. Morgan Kaufmann (Cited on pages 20, 21 and 23.)

Zhang M, Baddoo N, Wernick P, Hall T (2011) Prioritising refactoring using code bad smells. In: Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on, pp 458 –464 (Cited on page 32.)

Zimmermann T, Premraj R, Zeller A (2007) Predicting defects for eclipse. In: Predictor Models in Software Engineering, 2007. PROMISE’07: ICSE Workshops 2007. International Workshop on, p 9 (Cited on page 120.)

Zimmermann T, Nagappan N, Gall H, Giger E, Murphy B (2009) Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In: Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, ACM, New York, NY, USA, ESEC/FSE '09, pp 91–100 (Cited on page 21.)

APPENDIX A

Reviewers' Comments for *[Bowes et al. 2012b]*

Reviewer 1 for *Bowes et al. [2012b]* summarises the benefits of re-computing a common measure better than I can :

> “TITLE: Comparing the performance of defect prediction models which report multiple performance measures: reconstructing the confusion matrix
>
> AUTHORS: David Bowes, Tracy Hall and David Gray
>
> This paper compares 601 predictive models from 42 paper using a very nice approach to combine the different performance measures reported.
>
> I think that this is probably one of the most important steps forward in comparative meta level research that I have read for some time and I would strongly recommend that it is accepted for PROMISE. ...
>
> ... The contribution is really important in my view. That is, we know that most papers report some form of results, but find it very hard to compare them; often we feel we are comparing "apples and oranges", or so it seems.
>
> Of course we could expect a set of papers to opine the lack of standard measures and to "mandate" that authors use a prescribed approach to facilitate comparison. I have some sympathy with these "mandated standard of reporting" papers, but they always have a slight Orwellian ring to them; they demand that authors comply to facilitate comparison, sublimating their natural human creativity for the greater scientific good. It's all very laudable, but it seems to go against the creative grain of human nature.
>
> This paper takes a very different approach and one that I think has more chance of success because it goes with the grain of human nature rather than seeking to take up arms against it with a polemic against the "proliferation of measurement standards".
>
> The work builds on the foundations of the previous (very detailed and comprehensive) review of predictive models by these authors (and colleagues) in TSE. What the authors noticed in that work was the way in which precision and recall formed a natural base point and (also) the way in which reported measures could often be defined using a confusion matrix. The authors therefore set about to "reverse engineer" the confusion matrix from 42 different papers to facilitate comparison.
>
> The confusion matrix is like a kind of "atomic indivisible" reducible core of information about predictive models. It consist of the "core four": true positives, false positives, true negatives and false negatives. It is at the heart of any predictive model, yet, as the authors explain, these four numbers are not always reported. One could "mandate" that they be

reported in all future paper (nice try, big brother), but what the authors do is just so much better; they propose concrete approaches to reverse engineer these core four numbers and demonstrate that this approach is realistic: that is, they show that it is possible to extract from a large corpus of work, the confusion matrix using their "reverse engineering" approach.

The techniques used to reverse engineer the confusion matrix are synthesised from the literature and are not, in themselves, the primary novel contribution of the paper. However, the use of these techniques in this way, though simple and perhaps even (to some) obvious (once stated as an approach!) is extremely important. I would go as far as to suggest that this could well prove to be a game changing paper that really marks a point in time after which, meta analysis will clearly be changed (for the better).

The confusion matrix makes sense to me as a lingua franca. Naturally, one can compute things like precision and recall from this information (provided the predictive model always answers positive or negative in a binary classification test; no "undefined" values allowed).

This is a really nice idea. It means that we can use this approach to seek to compare many different results. It also *will* affect the bahaviour of future authors in ways that polemic and tirades against non standardness cannot. That is, I believe that authors will start to more routinely report in a way that facilities this kind of analysis (not wanting to be left out).

By showing that a large number of models (601) can be compared using their approach, the authors have done a huge service; they have created a sufficiency "corpus of comparison" to make it worth other authors' while to make their work sufficiently compliant to be included an any future studies using this approach. In this way I believe that this paper will act as a catalyst.

The paper includes a nice short explanation of the kinds of compound measure most commonly used and the approach the authors used to extract from such measures the "core four" needed for the confusion matrix.

The authors take four example papers as a case study and show the diversity of measurements reported. They then use their approach to construct the confusion matrices for these four papers, from which a better and more informative comparison becomes possible.

I think that this paper represents an important step forward in the construction of meta analysis. Such meta analyses (when done properly) are hugely important and valuable. This paper opens a path through which future meta analyses can truly be important and valuable. I have no doubt that the paper should be accepted for PROMISE 2012 and I would like to advocate for it to be considered for the award of best paper." Reviewer 1

Appendix B

Additional Papers

B.1 Cohesion metrics: the empirical contradiction.

Counsell S, Bowes D, Hall T (2009) Cohesion Metrics: The empirical contradiction. In: The Psychology of Programming Interest Group, Open University

Evolutionary Cohesion Metrics: The Empirical Contradiction

Steve Counsell
Dept. Information Systems
Brunel University
steve.counsell@brunel.ac.uk

David Bowes
Dept. Computer Science
Hertfordshire University
d.h.bowes@herts.ac.uk

Tracy Hall
Dept. Information Systems
Brunel University
tracy.hall@brunel.ac.uk

Keywords: Cohesion, Slicing, Barcode, Function.

Abstract

As a software engineering concept, the nuances of software cohesion have been notoriously difficult to interpret, comprehend or obtain any real consensus about. This has arisen because cohesion can be interpreted in many different ways, each of which is usually as valid as any other. In this paper, we describe a contradictory facet of slice-based cohesion from automatically extracted data that illustrates just this problem; the system in question (Barcode) was studied over multiple versions and two, slice-based cohesion metrics collected. Counter-intuitively, we found that function cohesion, rather than deteriorating over time, actually improved over time as code grew in size. The implication of our analysis and a conclusion that we should have perhaps drawn a long time ago is that cohesion is not commonly understood, is unlikely to be and, consequently, a metric that captures it will always elude us.

1. Introduction

Cohesion has been the subject of many previous studies [5, 6, 11]. It is also a characteristic of software that we still know very little about compared with, for example, coupling. The problem with cohesion is that unlike coupling, it can only be 'quantified' based on a single impression of what cohesion actually measures and that impression is often debatable. Herein, we explore the evolution of two cohesion metrics based on program slices [12] from a C system called Barcode. We collected the metrics from versions of the system using the CodeSurfer tool [4] and studied the evolutionary features therein. Results point to a revealing, counter-intuitive characteristic feature of the two metrics. The remainder of the paper is organised as follows. We describe the motivation for the work and related studies (Section 2). In Section 3, we describe the empirical data drawn from the Barcode system and in Section 4 present a discussion of some the issues raised by the study. Finally, we draw some conclusions in Section 5.

2. Motivation and Related Work

The motivation for the research in this paper stems from two sources. First, we know very little, empirically about cohesion metrics and their behaviour. Second, from an evolutionary perspective, and theoretically speaking and in the absence of consistent refactoring [8], we would expect cohesion of a system to deteriorate as a system decays evolves. The authors know of no study that demonstrates whether or not this is the case.

Slicing metrics have been used in various program analysis studies, the majority of which have used the procedural paradigm as a basis [1, 2, 3, 13]. In terms of slicing literature, the paper from which the slicing metrics were analyzed and is considered the seminal slicing text is that of Weiser [12]. Since then, the techniques of program slicing have been adopted and adapted by many disciplines and in a multitude of contexts. Ott and Thuss explored some of Weiser's original metrics [11] and also introduced several of their own. These metrics were then analyzed from a largely empirical viewpoint. Meyers and Binkley [10] undertook a large-scale empirical study of five slice-based metrics (largely those of Ott and Thuss) and provide baseline values for those metrics on a longitudinal basis; lowly-rated modules according to those baselines would be candidates for re-engineering.

3. Data Analysis

3.1 Preliminaries

Formally, we denote a set of variables used by a function M as $V_{F.}$ We denote V_O as the subset of V_F representing output variables. We denote a slice SL_i as that obtained for $v_i \in V_O$ and SL_{int} as the intersection of SL_i over all $v_i \in V_O$. We define the 'Tightness' and 'Overlap' metrics as originally defined by Weiser [12] (see the example in Appendix A) as:

$$\text{Tightness(F)} = \frac{|SL_{\text{int}}|}{length(F)} \quad \text{and} \quad \text{Overlap(F)} = \frac{1}{|\text{Vo}|} \sum_{i=1}^{|\text{Vo}|} \frac{|SL_{\text{int}}|}{|\text{SLi}|}$$

Tightness measures the number of statements that occur in every slice and Overlap 'how many statements in a slice are found only in that slice' [12]. Appendix A is taken from [10] and shows how these two metrics are calculated from a function to calculate the largest and smallest of an array of numbers. From the definitions of Tightness and Overlap, we obtain the following values for the function in Appendix A:

$$\text{Tightness} = \frac{11}{19} = 0.58$$

$$\text{Overlap} = \frac{1}{2}\left(\frac{11}{14} + \frac{11}{16}\right) = 0.74$$

The relatively high value of Overlap is due to high value of SL_{int} relative to the size of the two slices for 'smallest' and 'largest'. The value of Tightness reflects the fact that SL_{int} accounts for just over half the module length.

3.2 Summary Data

Table 1 shows the summary data (maximum (Max.), minimum (Min.), median and standard deviation (SD)) for the Tightness and Overlap values for the first version (v1) and the latest version (v22) of all modules in the Barcode system. Barcode is a C system for processing barcode information and the subject of previous empirical research by Meyers and Binkley [10]. The CodeSurfer tool [4] was used to extract the data. We note that in the Barcode system, a module can contain many functions (i.e. there is a 'one-to-many' relationship between the two) and it is the metrics extracted on a function basis that we report in this paper.

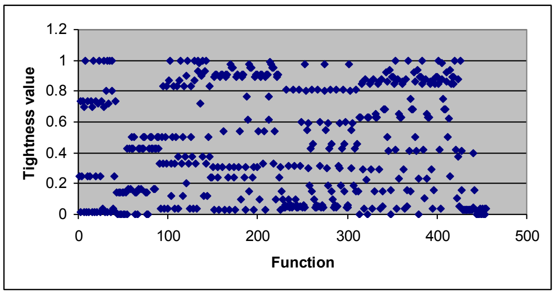
	Max.	Min.	Median	SD
Tightness v1	1	0	0.54	0.34
Tightness v22	1	0	0.61	0.33
Overlap v1	1	0	0.92	0.30
Overlap v22	1	0	0.84	0.28

Table 1. Summary Data for Tightness and Overlap

The most striking feature of Table 1 is the low values for SD suggesting that there is very little variance in the range of values for either metric. The relatively high values of the Overlap metric are reflective of the fact that the functions in Barcode had many slices – and the greater the number of slices, the higher the probability of high 'overlap'.

3.3 Data analysis

Figures 1a and 1b show the trend in the Tightness values for v1 and v22 of all modules in Barcode. Figures 2a and 2b show the equivalent Overlap values. Neither set of values show any specific characteristics. *We would expect cohesion to fall consistently during evolution as the code decays and functions grow in size*. This is based on the belief that as code is maintained it decays because of consistent corrective and requirement-change induced maintenance. As a result it becomes more fault-prone (not less fault-prone) and consequently a vicious circle of: maintenance-fault-maintenance, sets in. Function cohesion will inevitably suffer as a result. The average value for Tightness in v1 is 0.45 and for v22, higher at 0.54. The average value for Overlap in v1 is 0.71 and for v22, again higher at 0.76. In other words, counter-intuitively, evolution has *not* caused the values of the two metrics to fall – they have increased. The average LOC per function in v1 was 46.6 and for v22, 66.1, i.e., cohesion *increased* as function size increased.



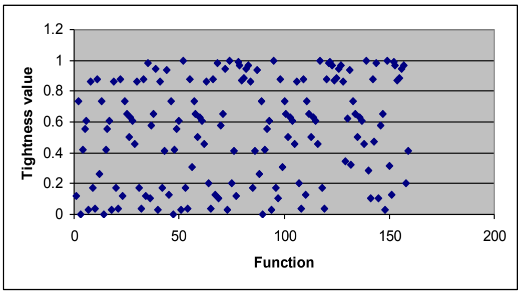
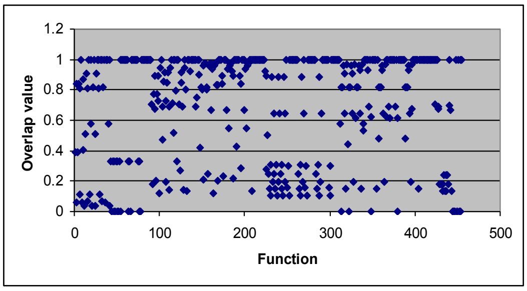


Figure 1a. Tightness values version 1

Figure 1b. Tightness values version 22



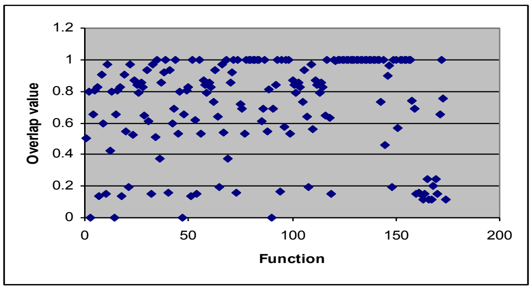


Figure 2a. Overlap values version 1

Figure 2b. Overlap values version 22

As the Barcode system has evolved, the cohesion of its functions has gone up and it has grown in size rather than what we expected.

4. Discussion

The preceding analysis raises a number of interesting issues. The stance that the authors adopt is that cohesion is a subjective concept that we should collect and analyse based on the understanding of the limitations of each metric. We suggest that it is unlikely that there will be a consensus on 'the' cohesion metric. Even though the OO community has adopted the Lack of Cohesion of the Methods of a Class (LCOM) metric of Chidamber and Kemerer [7] in a widespread way, it has not enjoyed full acceptance as the definitive OO metric; many attempts since have tried to redefine or re-invent new cohesion metrics. As a community, we can understand the notion of coupling since this can be quantified. We can say that one function has more coupling than another function with certainty; we can even normalise that coupling data to account for program size to make it more meaningful. In other words, we should collect coupling metrics and not cohesion.

One question that arises is whether the values of the two metrics fluctuated much during the course of the 22 versions over which data was extracted. Figure 3a shows the values for the Tightness metric at version 11 and Figure 3b, those for Overlap at version 11.

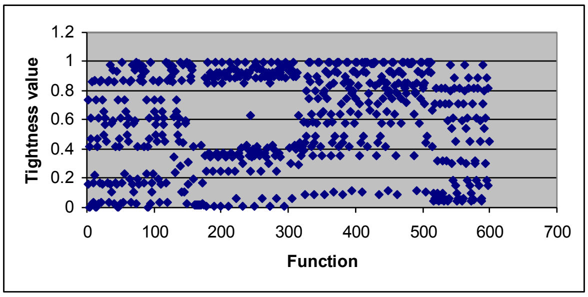


Figure 3a. Tightness values version 11

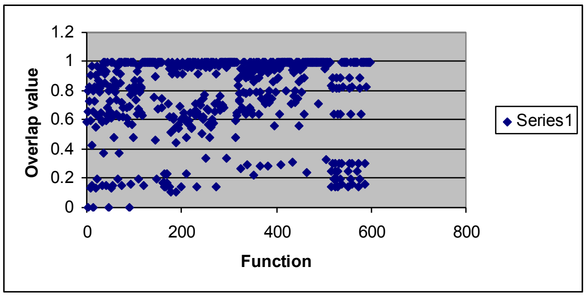


Figure 3b. Overlap values version 11

Table 2 shows the summary values for version 11 of Barcode. The values for median and SD are comparable with those of version 1 and 22 (in Table 1). The mean value of Tightness at v11 was 0.60 and in the same version, that of Overlap 0.80. In other words, there has only been a marginal decrease in cohesion from version 11 to 22, (a 0.06 difference for Tightness and a 0.04 difference for Overlap), this is not a convincing argument for using either metric.

	Max.	Min.	Median	SD
Tightness v11	1	0	0.66	0.33
Overlap v11	1	0	0.95	0.27

Table 2. Summary Data for Tightness and Overlap (v11)

Of course, we have no evidence on how much refactoring effort has been applied to the Barcode system and this represents one validity threat to the study. However, that still leaves the question as to whether the values of Tightness and Overlap actually represent high cohesion and what they measure.

5. Conclusions

In this paper, we have explored the characteristics of two slice-based cohesion metrics (Tightness and Overlap). The basis of the study was that if a metric measures cohesion, then by implication, as a system decays and cohesion deteriorates, this should be reflected in the values of any cohesion metric. Our empirical analysis of version of the Barcode system showed that the Tightness and Overlap cohesion values actually rose from version 1 to version 22. The implications of the study are therefore that, while metrics do provide a comparative indication of function features, they do fail to provide reinforcement of intuitive beliefs about the behaviour of software systems. Future work will focus on extracting faults from the same system as well as a study of the ‘entropy’ of the metrics data.

6. Acknowledgements

The research described is kindly supported by a grant from the UK Engineering and Physical Sciences Research Council (EPSRC Grant No: EP/E055141/1).

7. References

[1] Binkley, D. Gold, N. and Harman, M. An empirical study of static program slice size. ACM Trans. Software Engineering Methodology (TOSEM) 16(2):1-32, 2007.

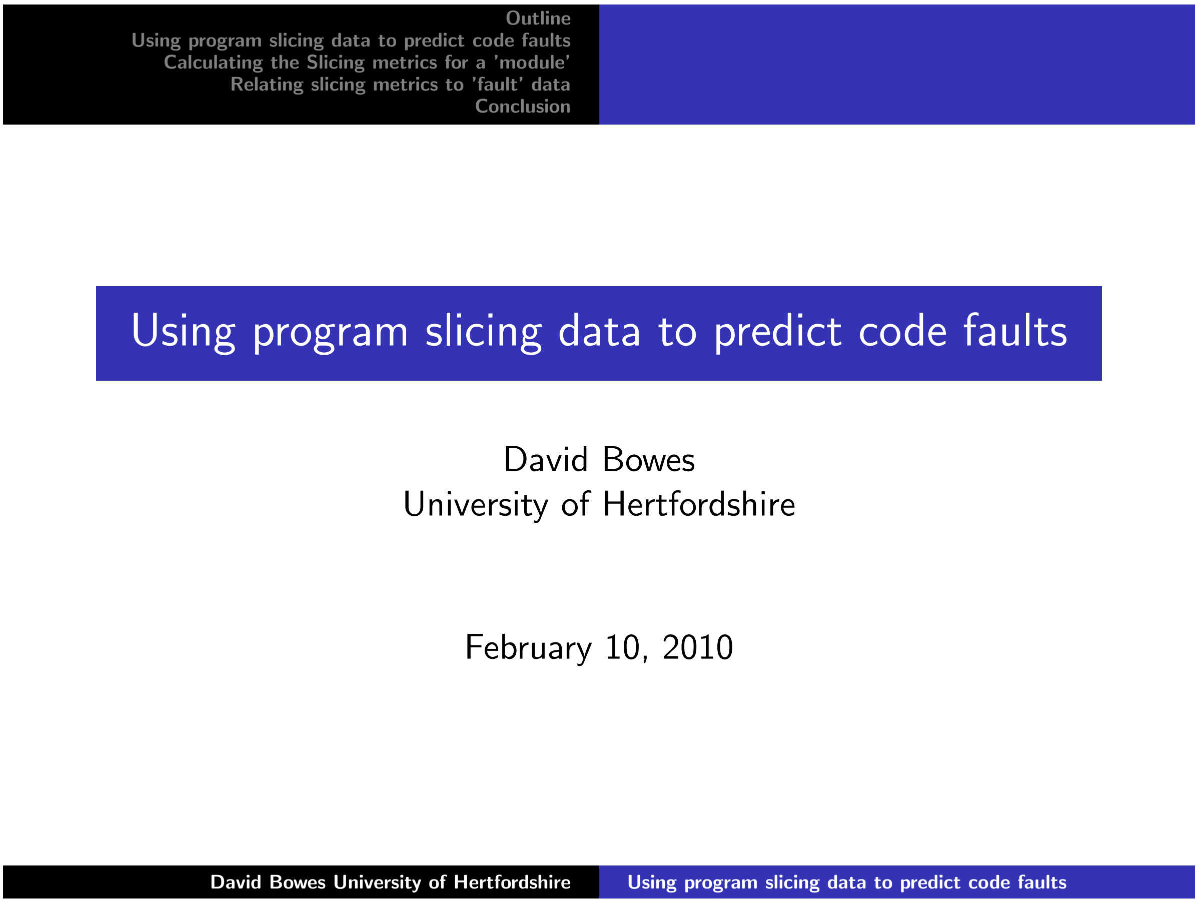
[2] Binkley, D., Harman, M., and Krinke, J., Empirical study of optimization techniques for massive slicing. ACM Trans. Program. Lang. Syst. 30(1): (2007)
[3] Binkley, D., Harman, M., Raszewski, I., and Smith, C. An empirical study of amorphous slicing as a program comprehension tool. Proc. of the Intl. Workshop on Program Comprehension, Limerick, Ireland, pp. 161-170, 2000.
[4] Codesurfer at: www.grammatech.com/products/codesurfer/
[5] Counsell, S., Swift. S. and Crampton J. The Interpretation and Utility of Three Cohesion Metrics for Object-Oriented Design. ACM Trans. on Software Eng. and Meth., 15(2):123 – 149, 2006.
[6] Counsell, S. Mendes, E., and Swift, S. Comprehension of Object-Oriented Software Cohesion: The Empirical Quagmire. Intl. Workshop on Program Comprehension, Paris, France, pp.33-42, 2002.
[7] Chidamber, S., and Kemerer, C. A metrics suite for object oriented design. IEEE Trans. on Software Engineering 20(6) (1994), 467-493.
[8] Fowler, M. Refactoring (Improving the Design of Existing Code). Addison Wesley, 1999.
[9] Gold, N., Harman, M., Binkley, D., Hierons, R., Unifying program slicing and concept assignment for higher-level executable source code extraction. Softw., Pract. Exper. 35(10): 977-1006 (2005)
[10] Meyers, T and Binkley, D. Slice-based Cohesion Metrics and Software Intervention, Proceeding s of the 11th Working Conf. on Reverse Engineering (WCRE 04), Delft, Netherlands, pages 256-265.
[11] Ott L. and Thuss, J., The relationship between slices and module cohesion. Proceedings of International Conference on Software Engineering, Pittsburgh, US, 1989, pages 198-204.
[12] Weiser, M. 1981. Program slicing. Proc. Int. Conf. on Soft Eng., San Diego, IEEE, pp.439-449.
[13] Weiser M (1982) Programmers use slices when debugging, Communications of the ACM, 25(7), pp.446-452, July 1982

APPENDIX A – Example function from [10]

Program function	$SL_{smallest}$	$SL_{largest}$	SL_{int}
Main()			
{			
int i;	|	|	|
int smallest;	|	|	
int largest;		|	
int A[10];	|	|	|
for (i=0; i <10; i++)	|	|	|
{			
int num;	|	|	|
scanf(“%d”, &num);	|	|	|
A[i] = num;	|	|	|
}			
smallest = A[0];	|	|	|
largest=smallest;		|	
i=1;	|	|	|
while (i <10)	|	|	|
{			
if (smallest > A[i])	|		
smallest = A[i];	|		
if (largest < A[i])		|	
largest = A[i];		|	
i = i +1;	|	|	|
}			
printf(“%d \n”, smallest);	|		
printf(”%d \n”, largest);		|	
}			
length =19	14	16	11

B.2 Using program slicing data to predict code faults.

Bowes D, Hall T (2010) Using program slicing data to predict code faults. In: The 3rd CREST Open Workshop, KCL



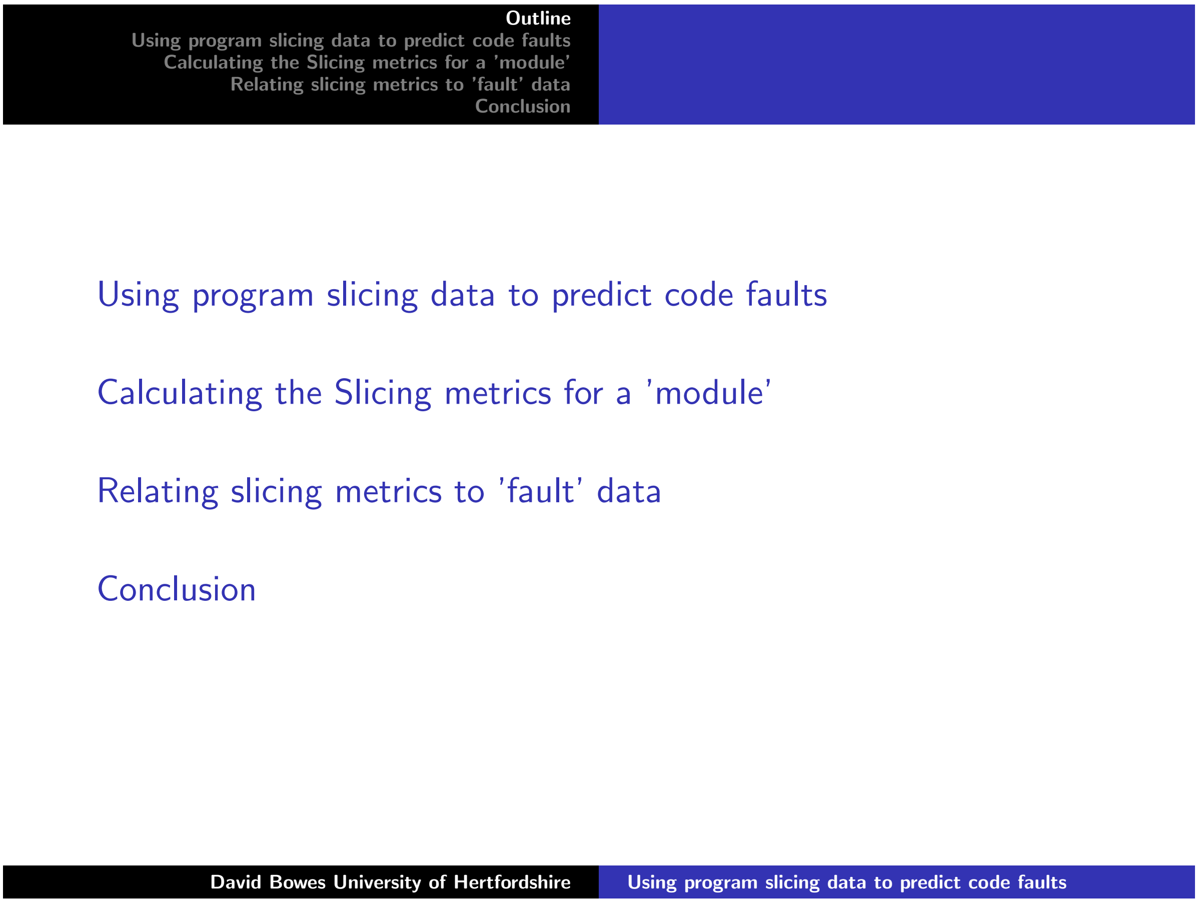
Outline
Using program slicing data to predict code faults
Calculating the Slicing metrics for a 'module'
Relating slicing metrics to 'fault' data
Conclusion

Using program slicing data to predict code faults

David Bowes
University of Hertfordshire

February 10, 2010

David Bowes University of Hertfordshire | Using program slicing data to predict code faults



Outline
Using program slicing data to predict code faults
Calculating the Slicing metrics for a 'module'
Relating slicing metrics to 'fault' data
Conclusion

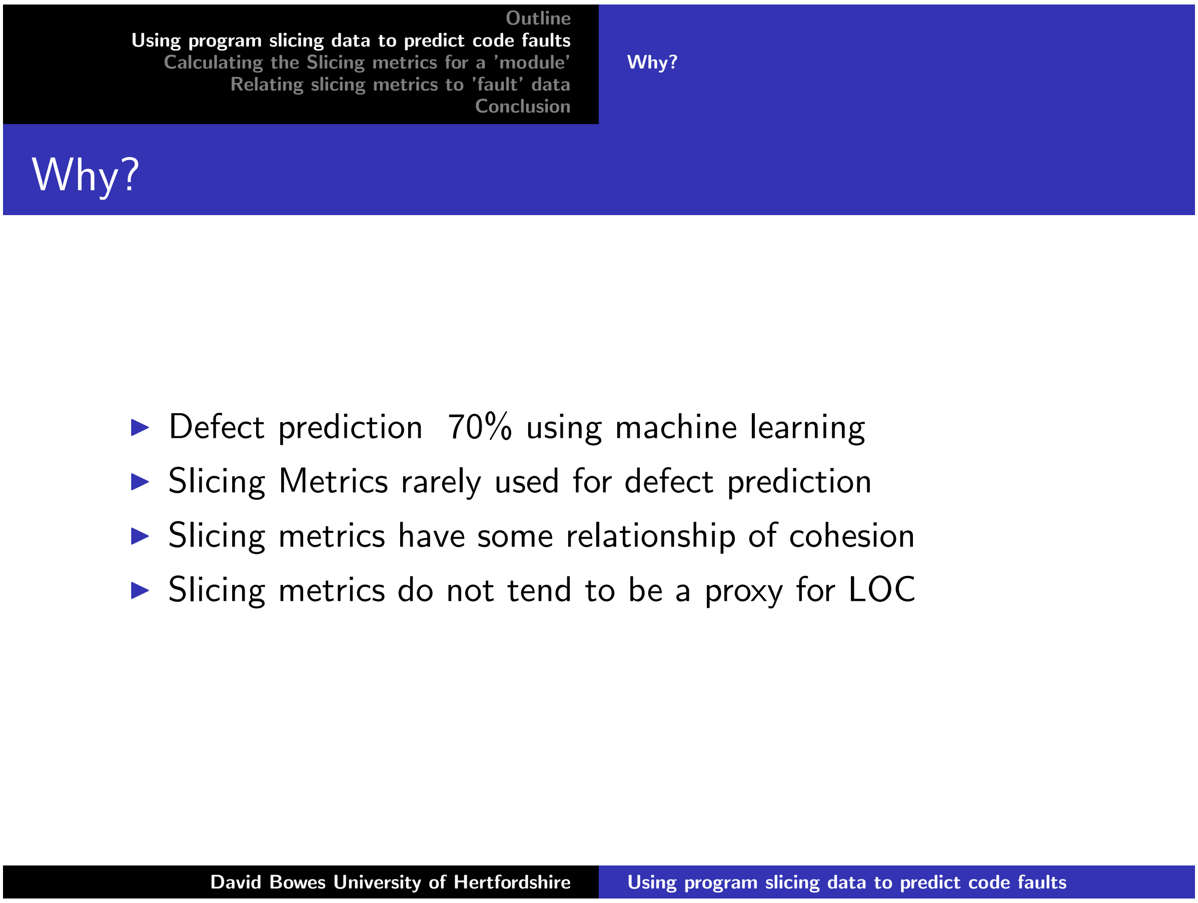
Using program slicing data to predict code faults

Calculating the Slicing metrics for a 'module'

Relating slicing metrics to 'fault' data

Conclusion

David Bowes University of Hertfordshire | Using program slicing data to predict code faults



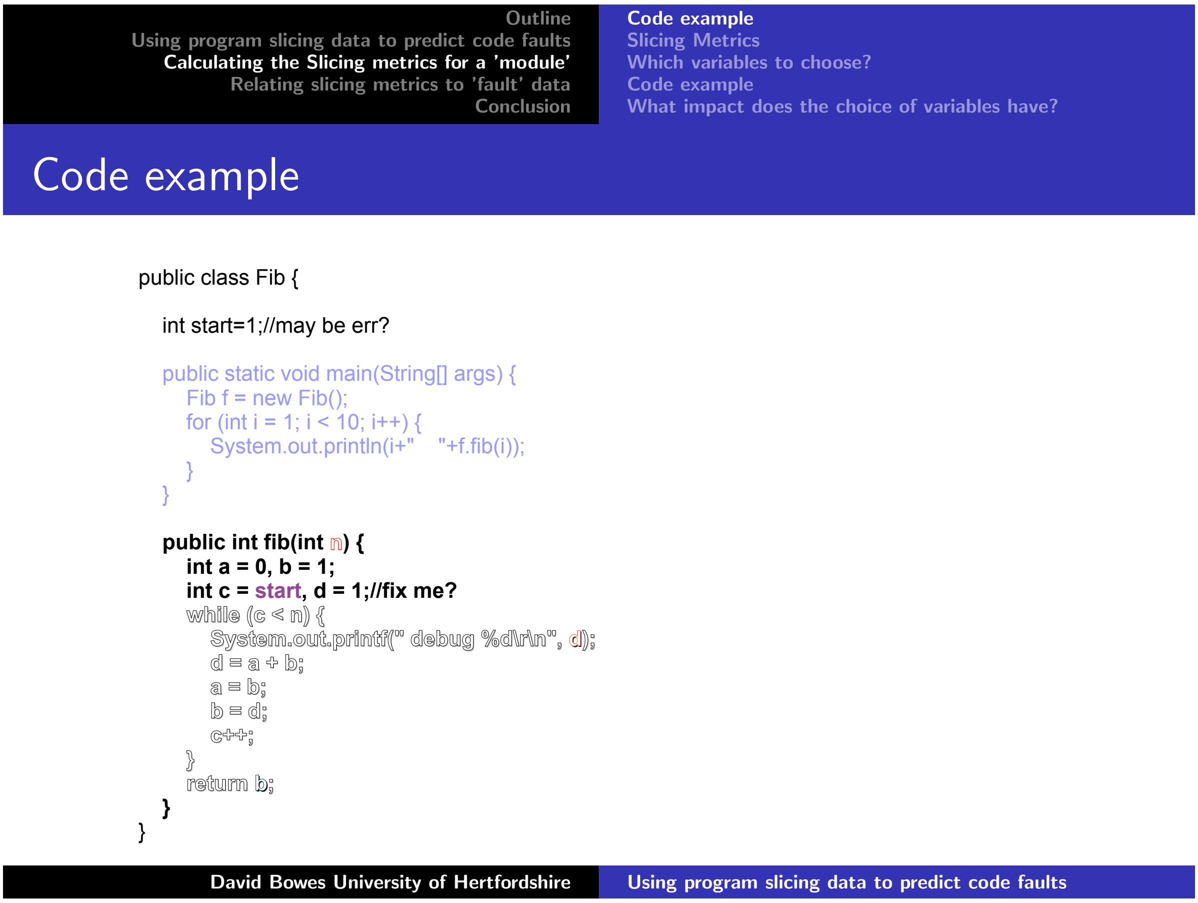
Outline
Using program slicing data to predict code faults
Calculating the Slicing metrics for a 'module'
Relating slicing metrics to 'fault' data
Conclusion

Why?

Why?

- Defect prediction 70% using machine learning
- Slicing Metrics rarely used for defect prediction
- Slicing metrics have some relationship of cohesion
- Slicing metrics do not tend to be a proxy for LOC

David Bowes University of Hertfordshire **Using program slicing data to predict code faults**



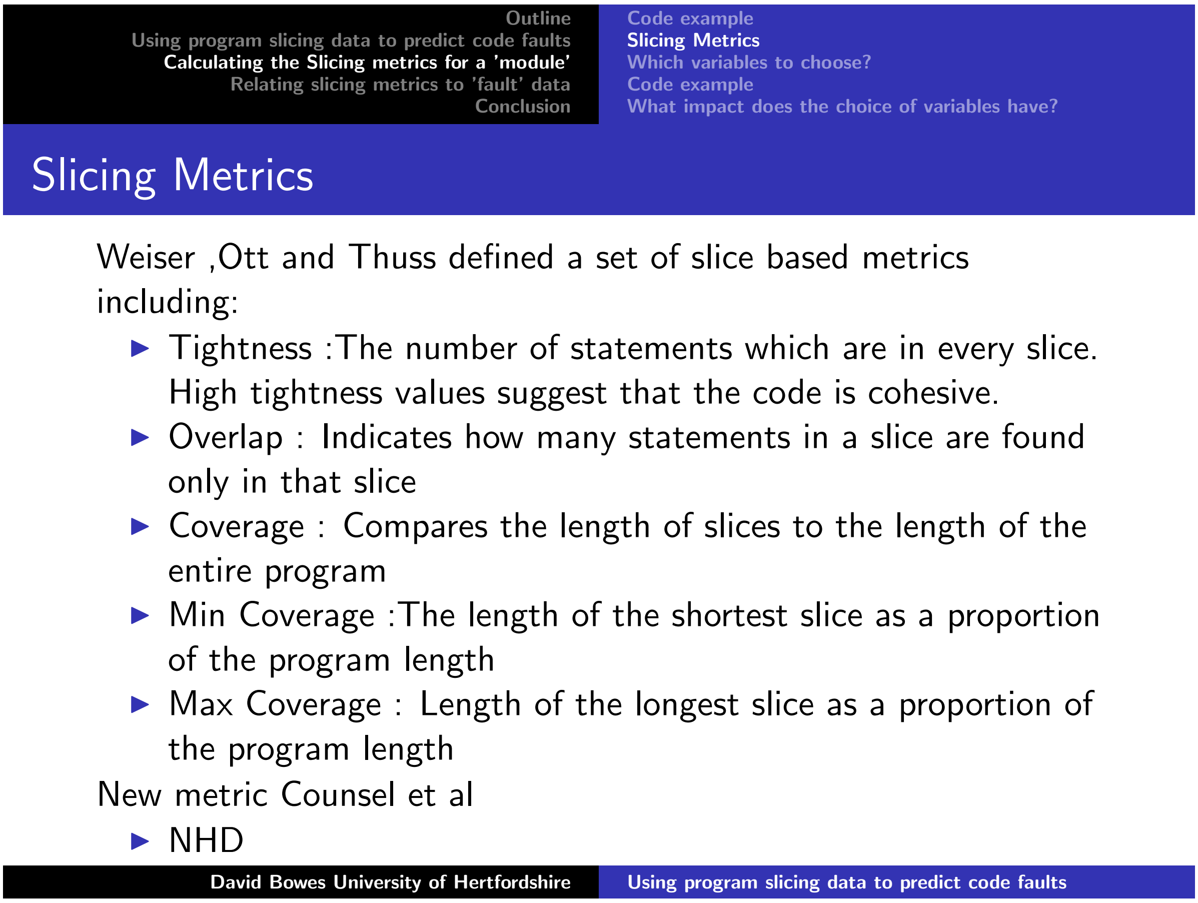
Outline
Using program slicing data to predict code faults
Calculating the Slicing metrics for a 'module'
Relating slicing metrics to 'fault' data
Conclusion

Code example
Slicing Metrics
Which variables to choose?
Code example
What impact does the choice of variables have?

Code example

```
public class Fib {

    int start=1;//may be err?

    public static void main(String[] args) {
        Fib f = new Fib();
        for (int i = 1; i < 10; i++) {
            System.out.println(i+"    "+f.fib(i));
        }
    }

    public int fib(int n) {
        int a = 0, b = 1;
        int c = start, d = 1;//fix me?
        while (c < n) {
            System.out.printf(" debug %d\r\n", d);
            d = a + b;
            a = b;
            b = d;
            c++;
        }
        return b;
    }
}
```

David Bowes University of Hertfordshire | Using program slicing data to predict code faults



Outline
Using program slicing data to predict code faults
Calculating the Slicing metrics for a 'module'
Relating slicing metrics to 'fault' data
Conclusion

Code example
Slicing Metrics
Which variables to choose?
Code example
What impact does the choice of variables have?

Slicing Metrics

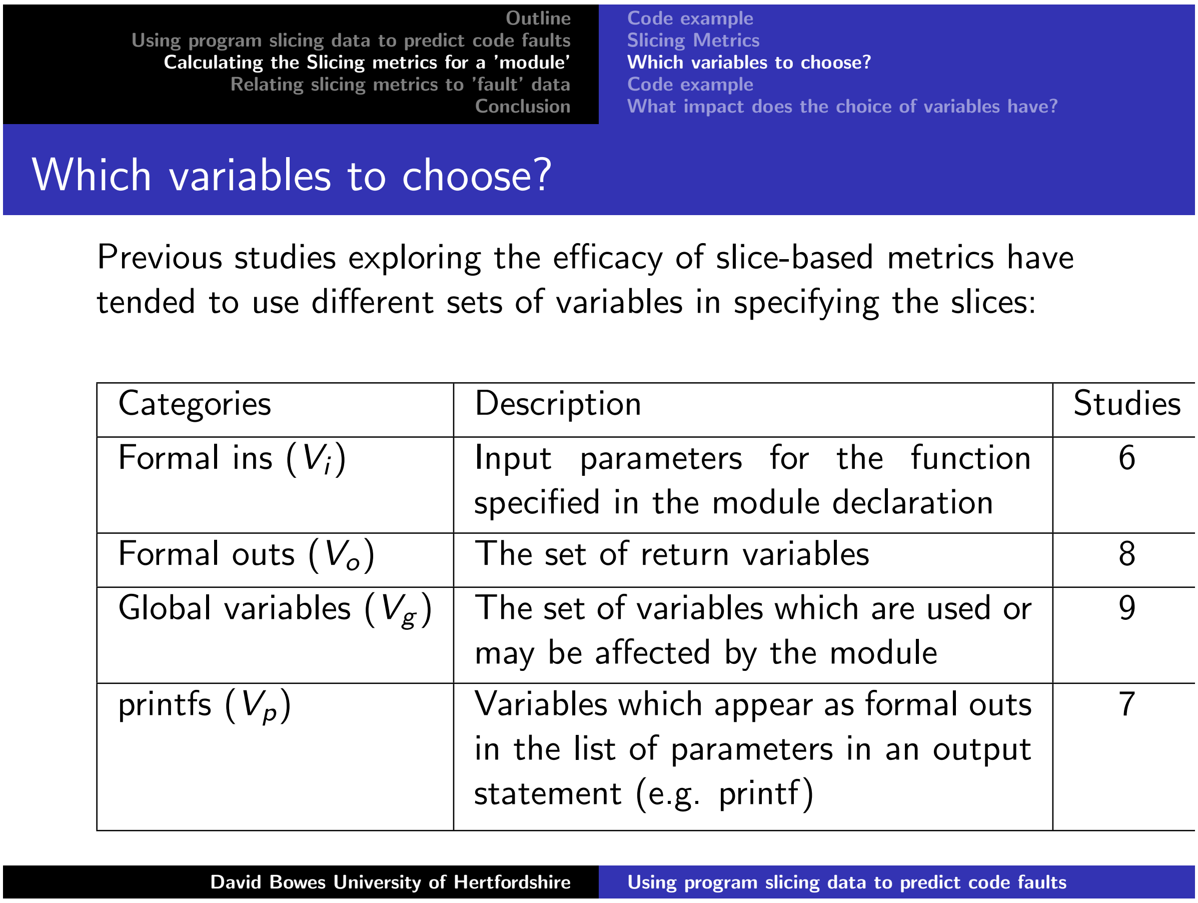
Weiser ,Ott and Thuss defined a set of slice based metrics including:

- Tightness :The number of statements which are in every slice. High tightness values suggest that the code is cohesive.
- Overlap : Indicates how many statements in a slice are found only in that slice
- Coverage : Compares the length of slices to the length of the entire program
- Min Coverage :The length of the shortest slice as a proportion of the program length
- Max Coverage : Length of the longest slice as a proportion of the program length

New metric Counsel et al

- NHD

David Bowes University of Hertfordshire | Using program slicing data to predict code faults



Outline
Using program slicing data to predict code faults
Calculating the Slicing metrics for a 'module'
Relating slicing metrics to 'fault' data
Conclusion

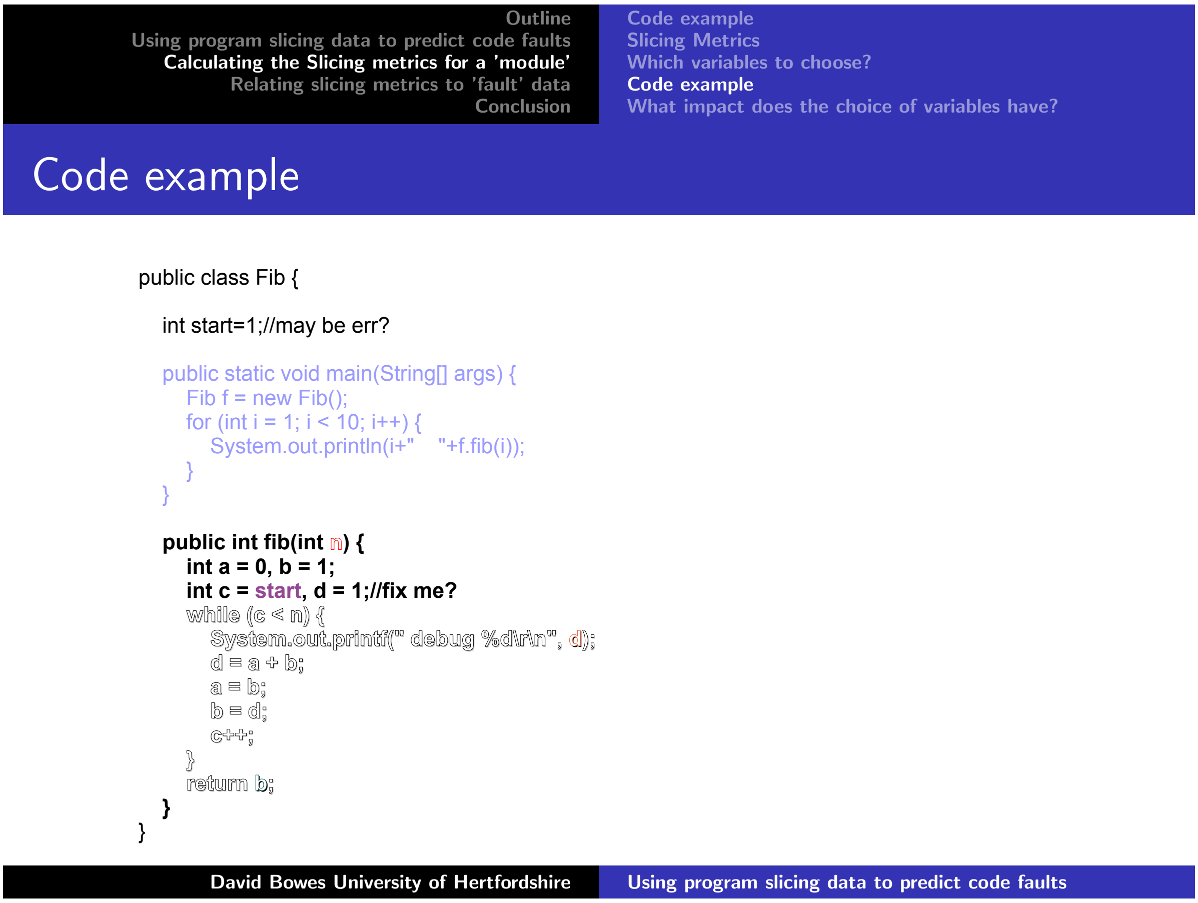
Code example
Slicing Metrics
Which variables to choose?
Code example
What impact does the choice of variables have?

Which variables to choose?

Previous studies exploring the efficacy of slice-based metrics have tended to use different sets of variables in specifying the slices:

Categories	Description	Studies
Formal ins (V_i)	Input parameters for the function specified in the module declaration	6
Formal outs (V_o)	The set of return variables	8
Global variables (V_g)	The set of variables which are used or may be affected by the module	9
printfs (V_p)	Variables which appear as formal outs in the list of parameters in an output statement (e.g. printf)	7

David Bowes University of Hertfordshire Using program slicing data to predict code faults



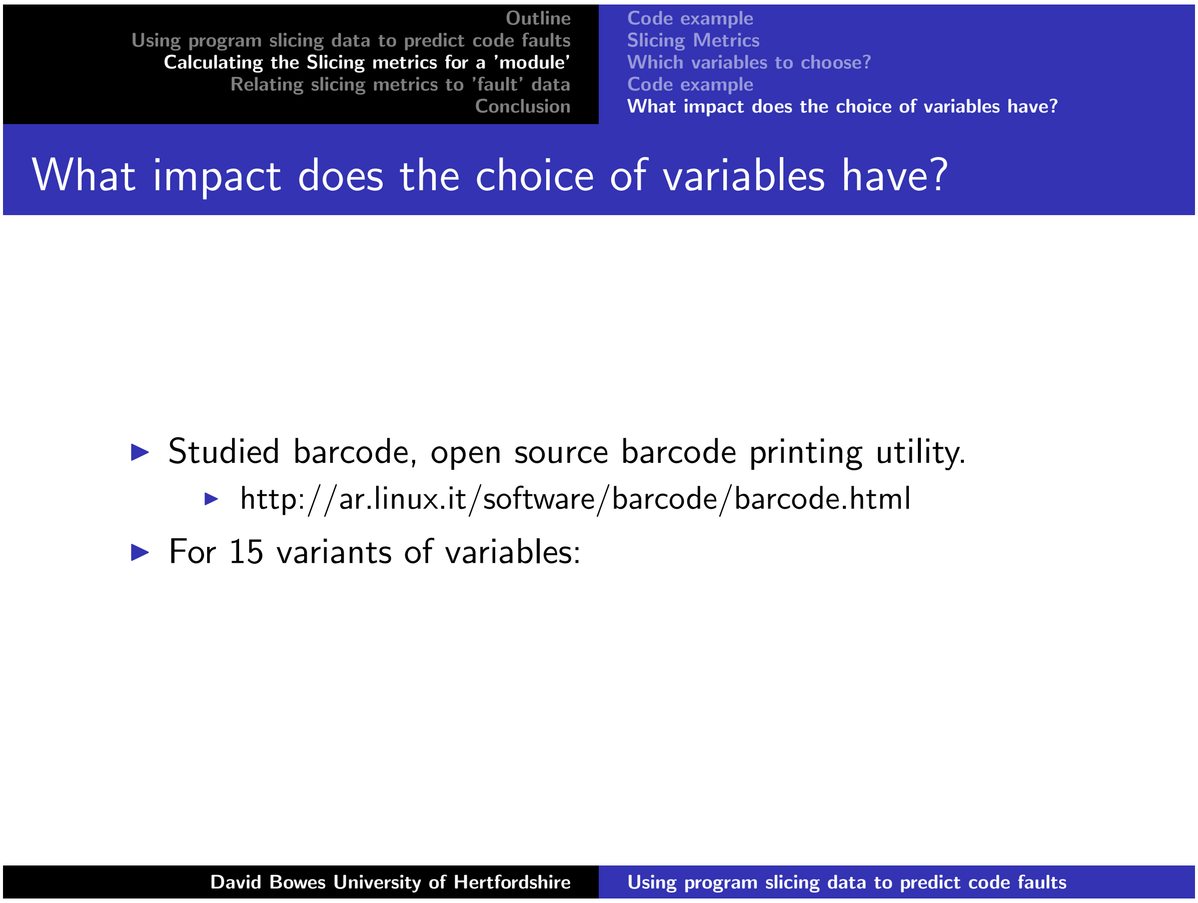
Outline
Using program slicing data to predict code faults
Calculating the Slicing metrics for a 'module'
Relating slicing metrics to 'fault' data
Conclusion

Code example
Slicing Metrics
Which variables to choose?
Code example
What impact does the choice of variables have?

Code example

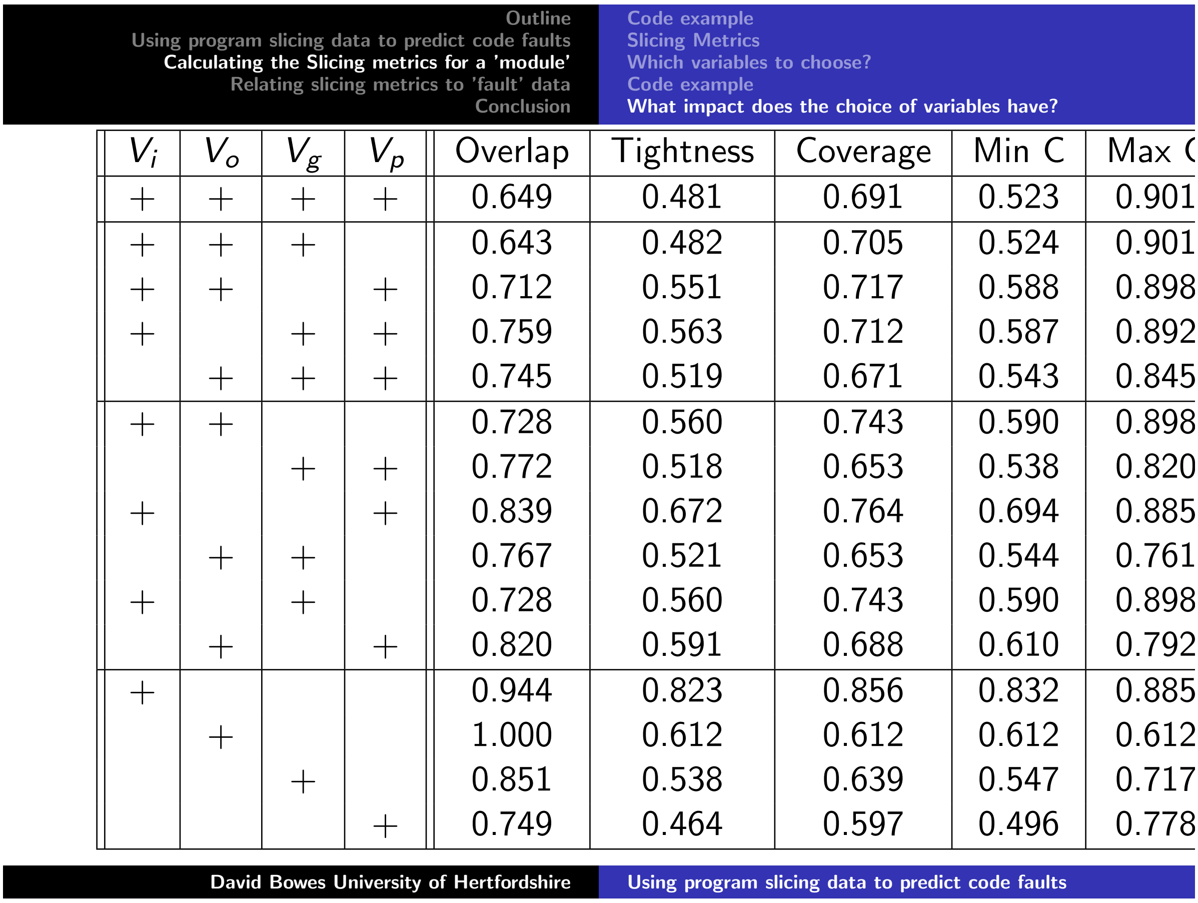
```
public class Fib {

    int start=1;//may be err?

    public static void main(String[] args) {
        Fib f = new Fib();
        for (int i = 1; i < 10; i++) {
            System.out.println(i+"    "+f.fib(i));
        }
    }

    public int fib(int n) {
        int a = 0, b = 1;
        int c = start, d = 1;//fix me?
        while (c < n) {
            System.out.printf(" debug %d\r\n", d);
            d = a + b;
            a = b;
            b = d;
            c++;
        }
        return b;
    }
}
```

David Bowes University of Hertfordshire
Using program slicing data to predict code faults



What impact does the choice of variables have?

- Studied barcode, open source barcode printing utility.
  - http://ar.linux.it/software/barcode/barcode.html
- For 15 variants of variables:

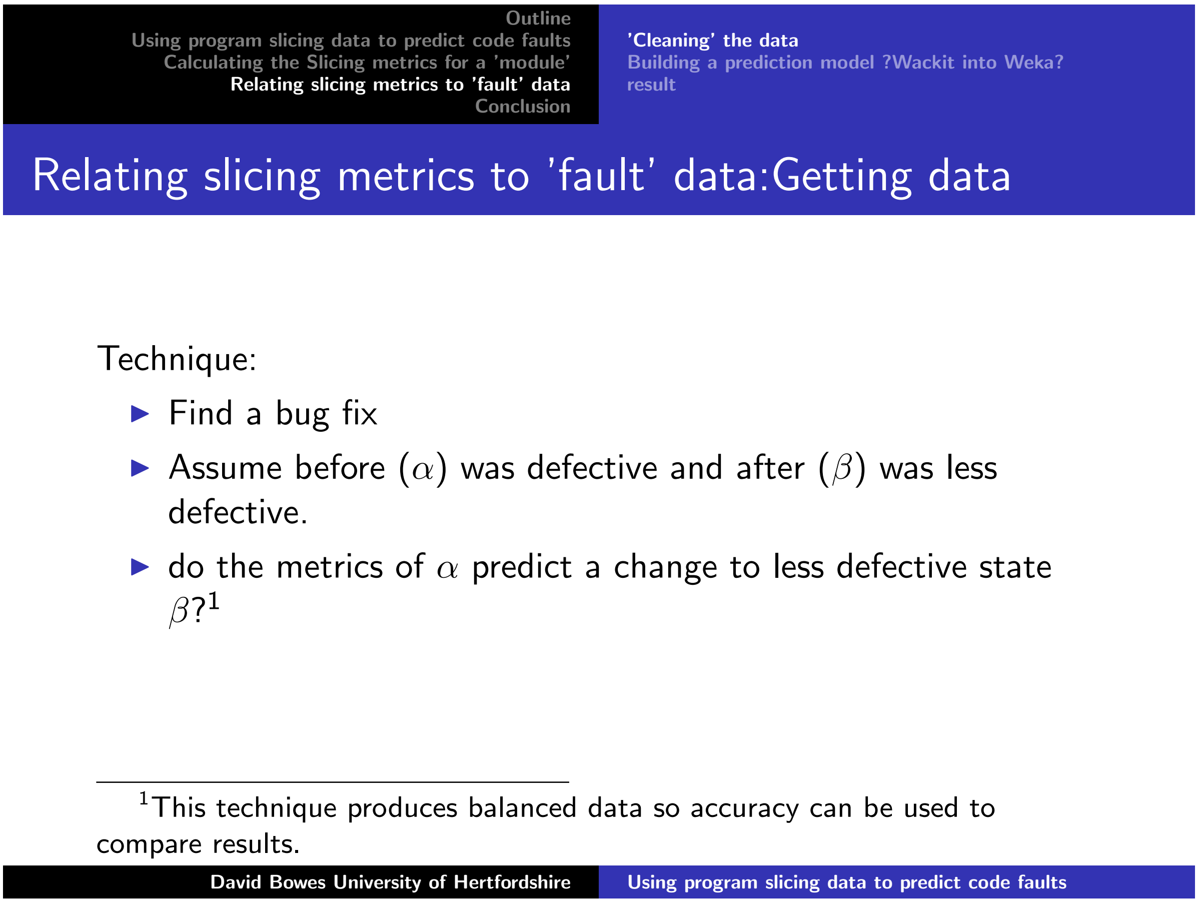


Outline
Using program slicing data to predict code faults
Calculating the Slicing metrics for a 'module'
Relating slicing metrics to 'fault' data
Conclusion

Code example
Slicing Metrics
Which variables to choose?
Code example
What impact does the choice of variables have?

V_i	V_o	V_g	V_p	Overlap	Tightness	Coverage	Min C	Max C
+	+	+	+	0.649	0.481	0.691	0.523	0.901
+	+	+		0.643	0.482	0.705	0.524	0.901
+	+		+	0.712	0.551	0.717	0.588	0.898
+		+	+	0.759	0.563	0.712	0.587	0.892
	+	+	+	0.745	0.519	0.671	0.543	0.845
+	+			0.728	0.560	0.743	0.590	0.898
		+	+	0.772	0.518	0.653	0.538	0.820
+			+	0.839	0.672	0.764	0.694	0.885
	+	+		0.767	0.521	0.653	0.544	0.761
+		+		0.728	0.560	0.743	0.590	0.898
	+		+	0.820	0.591	0.688	0.610	0.792
+				0.944	0.823	0.856	0.832	0.885
	+			1.000	0.612	0.612	0.612	0.612
		+		0.851	0.538	0.639	0.547	0.717
			+	0.749	0.464	0.597	0.496	0.778

David Bowes University of Hertfordshire
Using program slicing data to predict code faults



Outline
Using program slicing data to predict code faults
Calculating the Slicing metrics for a 'module'
Relating slicing metrics to 'fault' data
Conclusion

'Cleaning' the data
Building a prediction model ?Wackit into Weka?
result

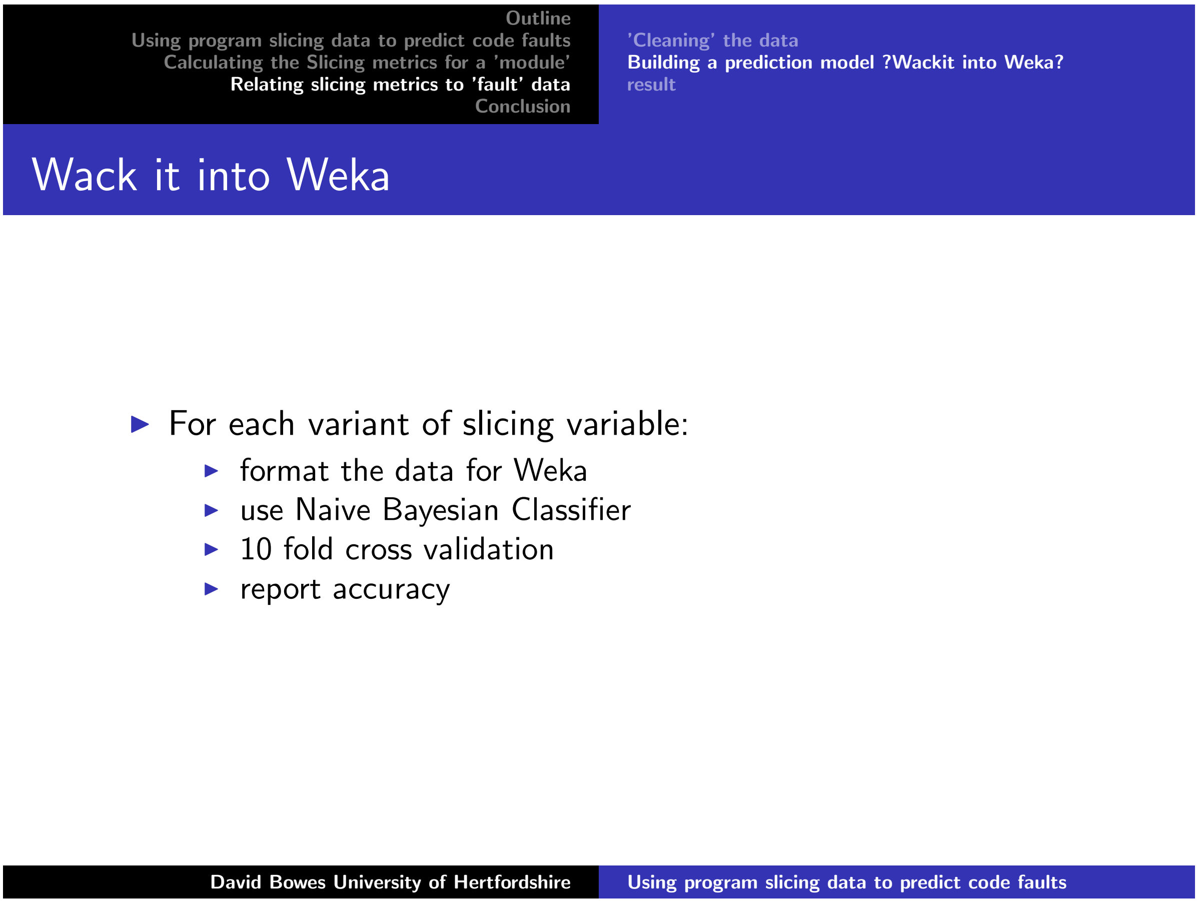
Relating slicing metrics to 'fault' data:Getting data

Technique:

- Find a bug fix
- Assume before (α) was defective and after (β) was less defective.
- do the metrics of α predict a change to less defective state β?[1]

[1] This technique produces balanced data so accuracy can be used to compare results.

David Bowes University of Hertfordshire | Using program slicing data to predict code faults



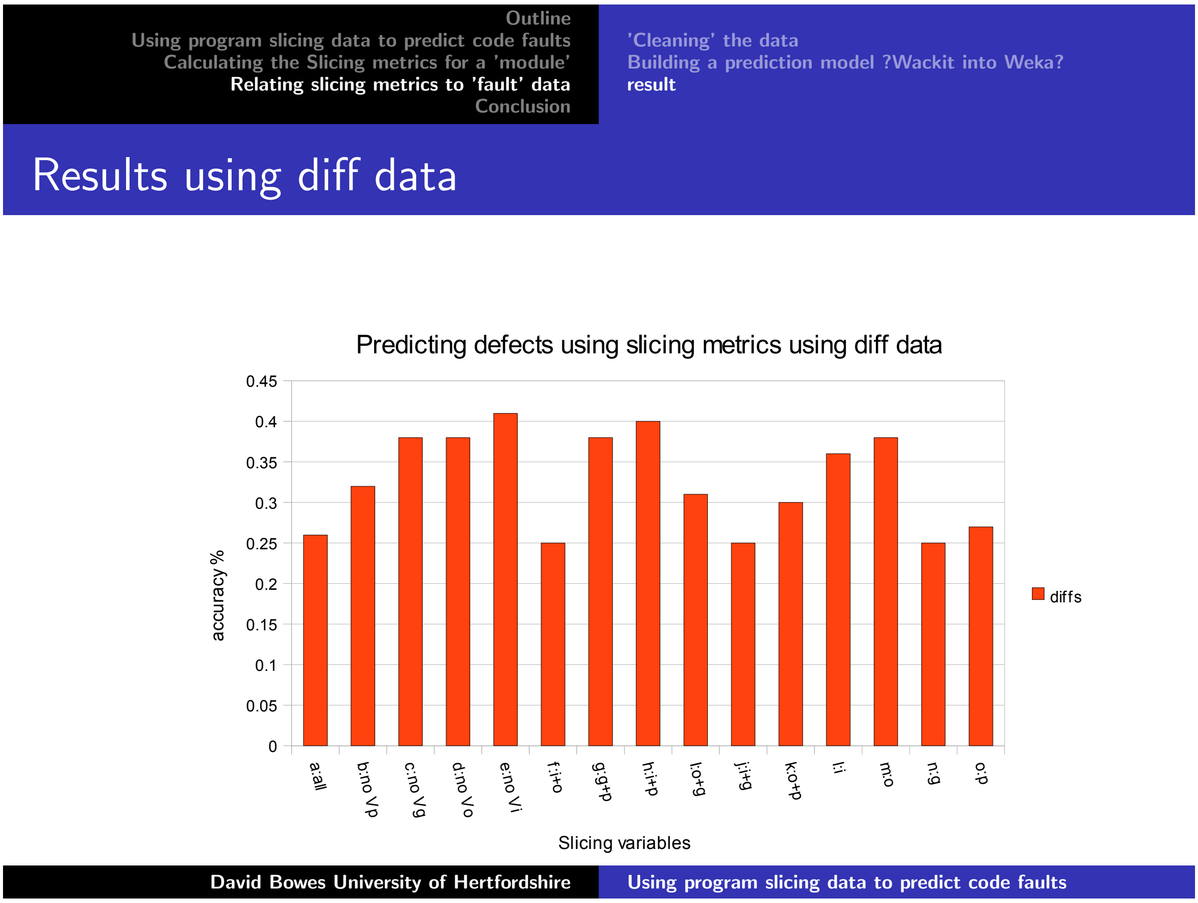
Outline
Using program slicing data to predict code faults
Calculating the Slicing metrics for a 'module'
Relating slicing metrics to 'fault' data
Conclusion

'Cleaning' the data
Building a prediction model ?Wackit into Weka?
result

Wack it into Weka

- For each variant of slicing variable:
  - format the data for Weka
  - use Naive Bayesian Classifier
  - 10 fold cross validation
  - report accuracy

David Bowes University of Hertfordshire | Using program slicing data to predict code faults

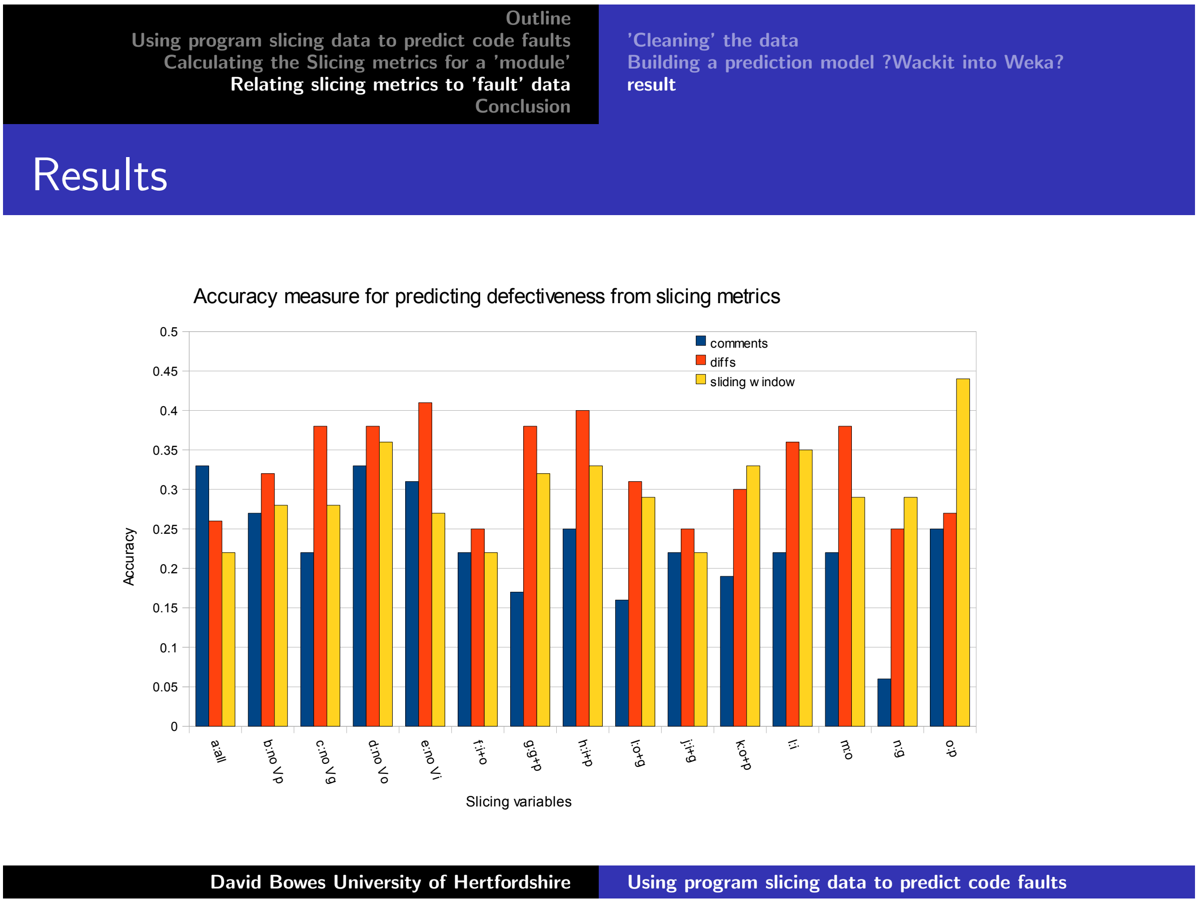


Outline
Using program slicing data to predict code faults
Calculating the Slicing metrics for a 'module'
Relating slicing metrics to 'fault' data
Conclusion

'Cleaning' the data
Building a prediction model ?Wackit into Weka?
result

Results using diff data

David Bowes University of Hertfordshire
Using program slicing data to predict code faults



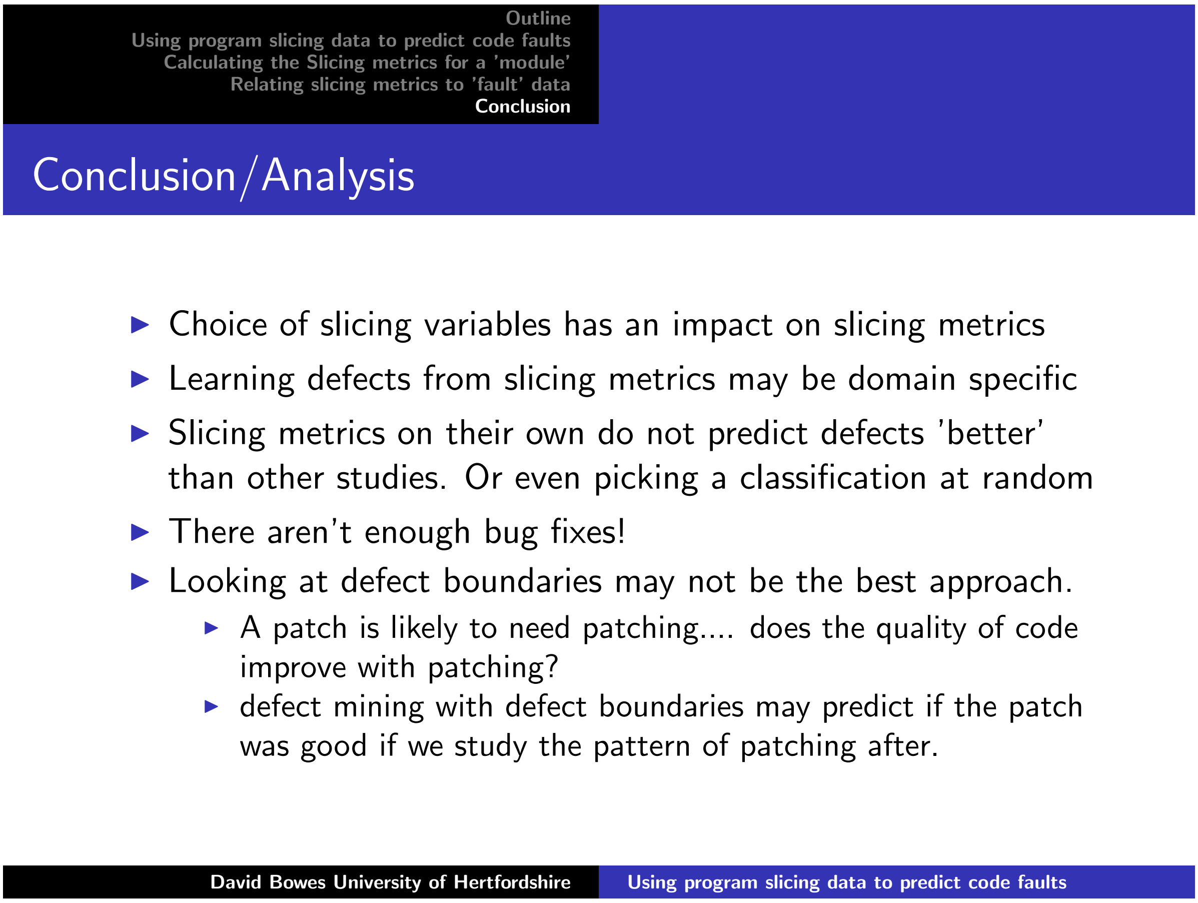
Outline
Using program slicing data to predict code faults
Calculating the Slicing metrics for a 'module'
Relating slicing metrics to 'fault' data
Conclusion

'Cleaning' the data
Building a prediction model ?Wackit into Weka?
result

Results

Accuracy measure for predicting defectiveness from slicing metrics

David Bowes University of Hertfordshire | Using program slicing data to predict code faults

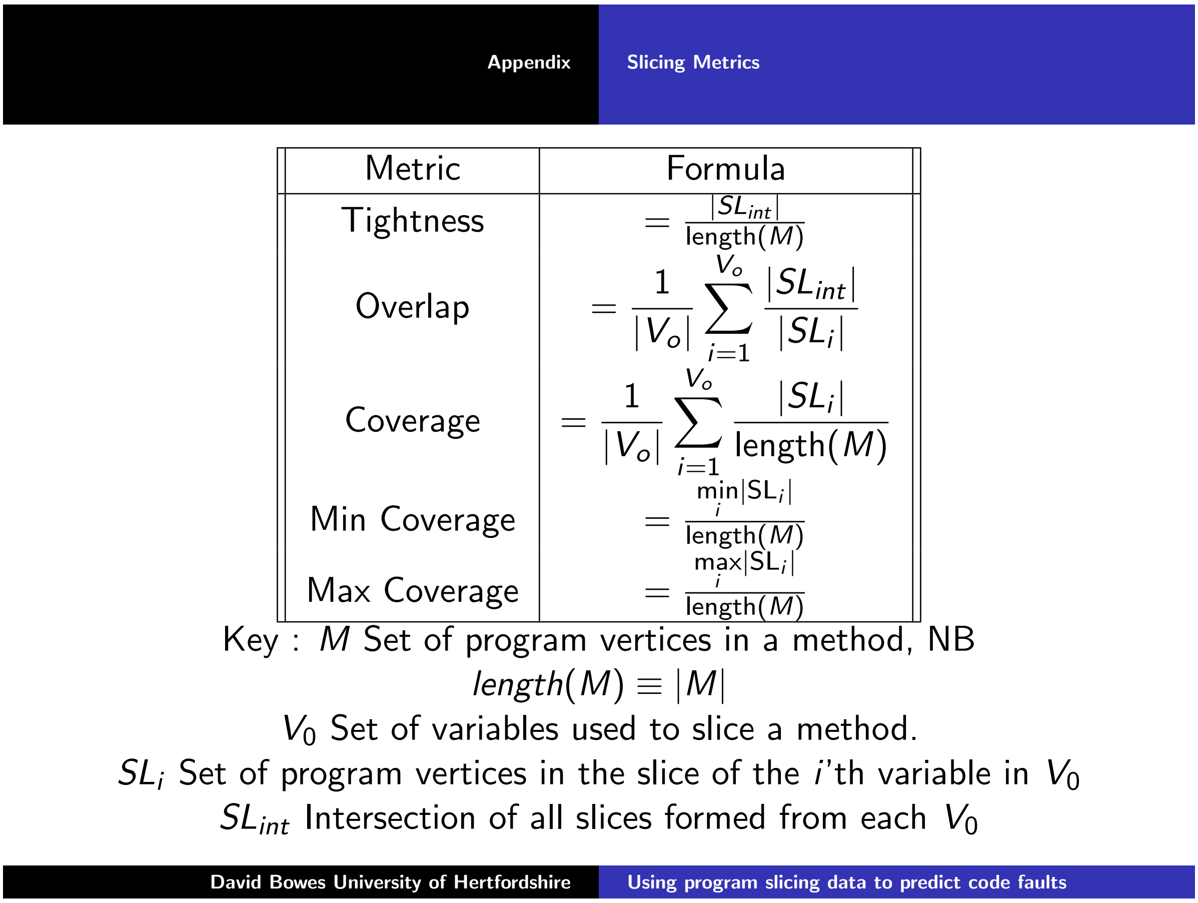


Outline
Using program slicing data to predict code faults
Calculating the Slicing metrics for a 'module'
Relating slicing metrics to 'fault' data
Conclusion

Conclusion/Analysis

- Choice of slicing variables has an impact on slicing metrics
- Learning defects from slicing metrics may be domain specific
- Slicing metrics on their own do not predict defects 'better' than other studies. Or even picking a classification at random
- There aren't enough bug fixes!
- Looking at defect boundaries may not be the best approach.
  - A patch is likely to need patching.... does the quality of code improve with patching?
  - defect mining with defect boundaries may predict if the patch was good if we study the pattern of patching after.

David Bowes University of Hertfordshire | Using program slicing data to predict code faults



Appendix Slicing Metrics

Metric	Formula
Tightness	$= \frac{\lvert SL_{int}\rvert}{\text{length}(M)}$
Overlap	$= \frac{1}{\lvert V_o\rvert}\sum_{i=1}^{V_o}\frac{\lvert SL_{int}\rvert}{\lvert SL_i\rvert}$
Coverage	$= \frac{1}{\lvert V_o\rvert}\sum_{i=1}^{V_o}\frac{\lvert SL_i\rvert}{\text{length}(M)}$
Min Coverage	$= \frac{\min\limits_i \lvert \mathrm{SL}_i\rvert}{\text{length}(M)}$
Max Coverage	$= \frac{\max\limits_i \lvert \mathrm{SL}_i\rvert}{\text{length}(M)}$

Key : M Set of program vertices in a method, NB
$length(M) \equiv |M|$
V_0 Set of variables used to slice a method.
SL_i Set of program vertices in the slice of the i'th variable in V_0
SL_{int} Intersection of all slices formed from each V_0

David Bowes University of Hertfordshire Using program slicing data to predict code faults

B.3 Program slicing-based cohesion measurement: the challenges of replicating studies using metrics.

Bowes D, Hall T, Kerr A (2011) Program slicing-based cohesion measurement: the challenges of replicating studies using metrics. In: Proceeding of the 2nd International Workshop on Emerging Trends in Software Metrics, ACM, pp 75–80

Program Slicing-Based Cohesion Measurement: The Challenges of Replicating Studies Using Metrics

David Bowes
University of Hertfordshire
College Lane
Hatfield, UK
d.h.bowes@herts.ac.uk

Tracy Hall
Brunel University
Kingston Lane
Uxbridge, UK
tracy.hall@brunel.ac.uk

Andrew Kerr
University of Hertfordshire
College Lane
Hatfield, UK
a.kerr@herts.ac.uk

ABSTRACT

It is important to develop corpuses of data to test out the efficacy of using metrics. Replicated studies are an important contribution to corpuses of metrics data. There are few replicated studies using metrics reported in software engineering.

To contribute more data to the body of evidence on the use of novel program slicing-based cohesion metrics.

We replicate a very well regarded study by Meyers and Binkley [15, 16] which analyses the cohesion of open source projects using program slicing-based metrics.

Our results are very different from Meyers and Binkley's original results. This suggests that there are a variety of opportunities for inconsistently to creep into the collection and analysis of metrics data during replicated studies.

We conclude that researchers using metrics data must present their work with sufficient detail for replication to be possible. Without this detail it is difficult for subsequent researchers to accurately replicate a study such that consistent and reliable data can be added to a body of evidence.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics – *product metrics*

General Terms

Measurement, Experimentation.

Keywords

Empirical software engineering, data quality, slicing metrics

1. INTRODUCTION

In this paper we present our replication of Meyers and Binkley's [15, 16] study of slice based cohesion metrics. These metrics are little known measures of code structure based on the program slices underlying that code (program slicing and slice-based metrics are described in Section 3). Meyers and Binkley [15, 16] conducted a large empirical study of slice-based metrics covering 63 open source C programs. They also show how cohesion changes as two open source programs evolve.

The aim of our replication was to ensure that we could generate identical results using the same cohesion metrics, the same techniques and the same open source data as originally used by Meyers and Binkley [15, 16]. Once we had ensured our approach collected valid metrics data we intended to add to the body of evidence on the use of slice based metrics by using the same techniques to collect data from additional software projects.

Replication plays an important role in building large corpuses of metrics data. Such large bodies of evidence are currently rare in software engineering. However the development of these is fundamental to generating significant results on which rigorous conclusions can be drawn in software engineering. Other more mature disciplines have used replicated studies to build significant evidence bases (e.g. the replication of cold fusion techniques by Lonchampt [13]).

However it proved much more difficult than we anticipated to generate the same results as the original Meyers and Binkley [15, 16] study. In this paper we report on the difficulties associated with what we anticipated would be a small and simple replication study. We focus particularly on the problems related to specifying precisely and implementing consistently the definition of metrics being used to collect data.

In Section Two we present the background to performing replicated studies. In Section Three we provide an overview of program slicing and slice-based metrics. Section Four describes the Meyers and Binkley's [15, 16] original study. In Section Five we report our replicated results. In Section Six we discuss our findings. We conclude and summarise in Section Seven.

2. BACKGROUND

Shull et al [20] define replication as "a study that is run, based on the design and results of a previous study, whose goal is to either verify or broaden the applicability of the results of the initial study". Our study can be further defined as an exact replication which Shull et al [21] describe as "one in which the procedures of an experiment are followed as closely as possible to determine whether the same results can be obtained".

An increasing number of replicated studies are now being conducted in software engineering (e.g. Andersson & Runeson, [1]). This growth is partly in response to the increasing amount of publically available software engineering metrics data. This data

WETSoM' 11, May 24, 2011, Waikiki, Honolulu, HI, USA
Copyright 2011 ACM 978-1-4503-0593-8/11/05 ...$10.00

includes the PROMISE repository of data as well as the huge number of open source system (OSS) projects.

Conducting replicated studies is difficult. Several previous studies report on these difficulties. For example Shull et al. [20] report that during their study of defect prediction they struggled to accurately replicate the original study. The main reason was that tacit knowledge from the original study could not be built into the replication. This was despite the replicated study adopting a detailed laboratory package designed to facilitate replication (the laboratory package was based on [3]).

Anderson and Runeson [1] replicated Fenton and Ohlsson's [4] study of software defects. They tested the original study's hypotheses using the original study's techniques but applied to different data sets. They report different results from the original study, but are one of the few studies not to report difficulties associated with performing the actual replication.

We could find only one other previous study focused on replicated studies using OSS metrics data. Robles [19] report their review of the 171 papers published at the Mining Software Repositories Workshop between 2004 and 2009. They analyse each paper in terms of its potential for replication. They report that only two of these papers could be easily replicated, while 64 would be possible to replicate. Robles [19] reports that for a variety of reasons the majority of studies could not be replicated. Lack of detail in the method description, especially in relation to the processing of data, was a major reason for studies not being repeatable. The unavailability of tools, scripts and processed data was also reported as an impediment to repeatability. Robles [19] concludes that publically available raw data is not sufficient to allow for replication and calls for a replication framework to be used by original studies.

3. PROGRAM SLICING AND SLICING DATA

The Meyers and Binkley's [15, 16] study that we replicate is an analysis of program slicing data, consequently in this section we describe program slicing and slice-based metrics

3.1 Program Slicing

Program slicing was introduced by Weiser [22, 23]. It is a technique for decomposing programs into slices where each slice preserves the effects on a set of variables at a given point. Informally a slice of a program at a point *p* for a set of variables *V* is an executable subset of that program that preserves the state changes of *V* up to *p*. The pair <*p*, *V*> is the slicing criterion. Weiser's proposed applications for slicing were debugging and parallel execution [23] and slicing was intended to be useful for programmers maintaining code [22].

Horwitz et al. [10] introduced the system dependence graph (SDG), a representation of a program which includes procedure dependence graphs (PDGs) and both direct dependence and transitive dependence edges connecting them. The SDG was created for the purpose of computing interprocedural slices. They described a two phase algorithm for computing interprocedural slices using an SDG. Horwitz et al. [10] used a slightly different concept of a slice to Weiser's: "The [backward] slice of a program with respect to program point *p* and variable *x* consists of all statements and predicates of the program that might affect the value of *x* at point *p*" [10] with the additional constraint of only allowing slicing criteria *(p,x)* where *x* is used or declared at *p*.

Horwitz et al. [10] also introduced the forward slice: "the forward slice of a program with respect to a program point *p* and variable *x* consists of all statements and predicates of the program that might be affected by the value of *x* at point *p*".

3.2 Slice-based Cohesion

The cohesion of a module is "the extent to which its individual components are needed to perform the same task" ([5], p. 312). Higher cohesion is considered preferable, indicating a module tightly focused on one task

Metrics for the cohesion of a module can be calculated from the sizes of slices, the intersection of slices and the size of the module. Weiser [22] suggested five metrics based on program slicing: coverage, overlap, clustering, parallelism and tightness. Longworth [14] investigated the use of Weiser's metrics as measures of cohesion, redefining overlap so that it, coverage and tightness have values varying between 0.0 and 1.0. He rejected clustering as a useful cohesion metric and noted that parallelism is of a different nature to the others.

Ott and Thuss [17] again changed the definition of overlap, and introduced the metrics mincoverage and maxcoverage. Table 1 describes all their metric definitions excluding parallelism.

Table 1. Cohesion metric definition [17]

$Coverage(M) = \frac{1}{\lvert V_0\rvert} \sum_{i=1}^{\lvert V_0\rvert} \frac{\lvert SL_i\rvert}{length(M)}$	Average ratio of the size of a slice to the size of the module
$MinCoverage(M) = \frac{1}{length(M)} \min_i \lvert SL_i\rvert$	Smallest ratio of the size of a slice to the size of the module
$MaxCoverage(M) = \frac{1}{length(M)} \max_i \lvert SL$	Largest ratio of the size of a slice to the size of the module
$Overlap(M) = \frac{1}{\lvert V_0\rvert} \sum_{i=1}^{\lvert V_0\rvert} \frac{\lvert SL_{int}\rvert}{\lvert SL_i\rvert}$	Average ratio of the size of the intersection to the size of a slice
$Tightness(M) = \frac{\lvert SL_{int}\rvert}{length(M)}$	Ratio of the size of the intersection to the size of the module

Green et al. [9] describe in detail the evolution of slice-based cohesion metrics.

4. THE ORIGINAL STUDY

Few previous studies report the empirical study of program slice-based metrics. Pan et al [18] use slice-based metrics to investigate bug classification but there are few other examples of such studies. Consequently little is known about how they perform. However the advent of tools to automatically collect slice-based metrics data (e.g. CodeSurfer [8]) means that using these metrics has recently become more practical.

Meyers and Binkley [15, 16] are the first to conduct a large empirical study of slice-based metrics covering 63 open source C programs, including Barcode. As part of this they produce averages of the cohesion metrics for the different release versions of Barcode and Gnugo. These show how cohesion changes as programs evolve. In addition they tested for correlation between the metrics and lines of code but found no evidence of correlation. However they did find that some cohesion metrics are strongly correlated, while others are weakly correlated or not correlated at all.

Our study focuses on replicating Meyers and Binkley's results for the Barcode open source project. Barcode contains a total of 65 distinct functions and 49 revisions being studied, numbered 000 to 050 with 032 and 042 missed. There is one data point for each combination of function and revision for a total of 3185 points. Those where the function has yet to be defined (769 points) were discarded leaving 2416 data points. Additionally one point was deleted accidentally leaving 2415.

Meyers and Binkley used CodeSurfer [7] to automatically collect slice-based metric data as we also do.

CodeSurfer is a sophisticated commercial static code analyser for C code which can be configured to collect slicing metrics data. This configuration impacts on the data collected by CodeSurfer and so it is important that we configure CodeSurfer to collect data in the same way as Meyers and Binkley did. Although Meyers and Binkley [15, 16]do not report the CodeSurfer configuration they used, we had lengthy discussions with David Binkley regarding their configurations. Important aspects of our CodeSurfer configuration are that our:

- build script was initially built using the ./configure command,
- make file was cleaned using sed to remove –O2 options from any line which called the gcc compiler.
- command used on Barcode to build a CodeSurfer project for each release was: `csurf hook-build barcode -preset-build-options highest --- make`

The `-preset-build-options highest` causes CodeSurfer to produce the most detailed analysis of the program being analysed. Although we used the `highest` preset of CodeSurfer it is possible to use a lower preset such as `high.` Doing so requires the individual CodeSurfer Build Options relevant to slicing metrics to be specified. Many of the build options in `highest` do not affect the computation of slicing metrics data. Consequently the decision regarding which preset to use is a trade-off between using unnecessary processing (`highest)` or using more manual CodeSurfer expertise (`high)`. Appendix B shows the CodeSurfer presets for each build option. Unfortunately Meyers and Binkley do not report on the preset that they used.

The data collection and analysis process we adopted attempts to accurately replicate that reported by Meyers and Binkley [15, 16]. Full details of our method and the scripts which we used are available in Kerr [11] which should enable any future replication of this study. Kerr [11] is available on request from the first author.

5. RESULTS

Meyers and Binkley [15, 16] charted average cohesion metrics (Figure 1) for each released version of Barcode.

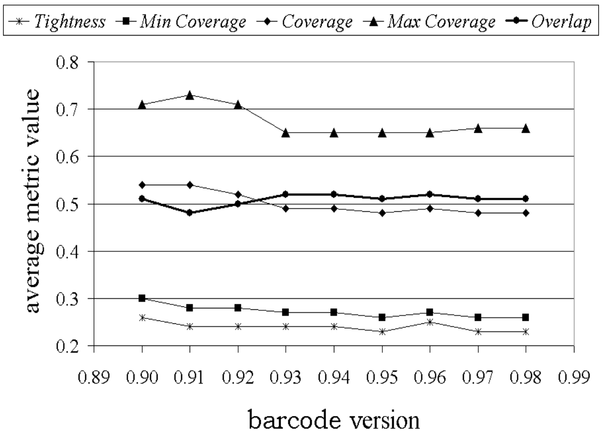


Figure 1. Longitudinal cohesion study [16]

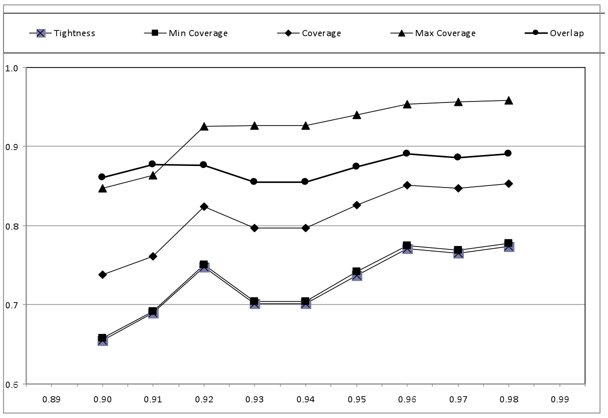


Figure 2. Longitudinal study (our data)

Figure 2 shows our replication of Meyers and Binkley's original cohesion data. It shows that the data we collected for all five cohesion metrics are different from Meyers and Binkley's original data. Trends in each metric differ between the studies with our cohesion data slowly increasing (improving) rather than decreasing as Meyers and Binkley's data does. Again we expected the pattern to be identical.

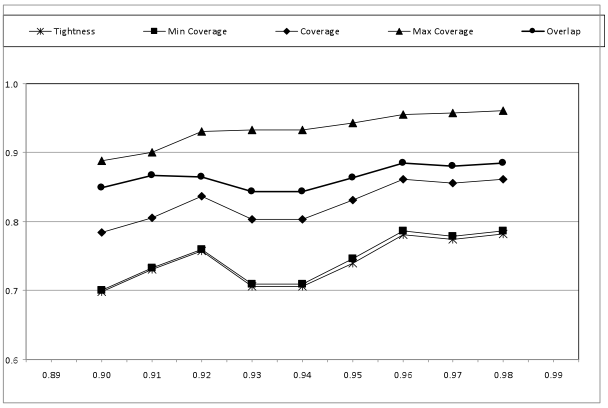


Figure 3. Longitudinal cohesion study, full vertex removal (our data)

In trying to understand why our data was so different from Meyers and Binkley's we realized that we must have collected the data for the slices slightly differently from them. Consequently we tried to improve our replication of Meyers and Binkley's cohesion data by altering the way we specified the slices on which the metrics data is based. The first change to the way we collected the slice-based data was to remove all non coding vertices, such as the body and entry vertices associated with '{'. These are common to all methods and tend to unilaterally increase the metrics values. This issue may underpin why our values are consistently higher than those reported by Meyers and Binkley. This was also done in Green et al.'s [9] study of slice-based metrics.

Figure 3 shows our new data once these vertices have been removed from the underlying slices. Not only is this data very different from our first set of Barcode cohesion data but it remains very different from Meyers and Binkley's original data. This means that the removal of these vertices cannot explain the greater values for cohesion metrics that we report compared to Meyers and Binkley [15, 16].

To understand how our data collection differed from Meyers and Binkley's and to try to move our results towards theirs we decided to explore in detail one data point for each metric from Figure 3 (i.e. replicate the metric values for just one version). We collected data from version 0.98. There are a variety of ways in which the lines of code affected by a variable can be included in a specific slice. The decision made as to which variables will be included in slices affects the values of the slice-based metrics data. These variables include:

1. Formal Ins: Input parameters for the function specified in the module declaration.
2. Formal Outs: Return variables.
3. Globals: Variables used by or affected by the module.
4. Printf: Variables which appear as Formal Outs in the list of parameters in an output statement.

In addition code can be sliced on a file or project basis. As Meyers and Binkley [15, 16] did not specify which of these variables they used in their original study we looked at the affect of them all on the metric values produced for version 0.98. We combined these variable settings to collect data in 30 different ways for each of the five cohesion metrics (Appendix A). We discuss the impact of these combinations on fault prediction in [2]. The closest match with the original data came when we sliced files individually with only printf variables. Even so, the values for tightness continued to be greatly different from the values obtained by Meyers and Binkley.

6. DISCUSSION

Our study shows that there are many ways in which the data collected by a metric can vary. For example, as we have shown here the way that program slices are set-up has a large impact on the data collected by slice-based metrics. This finding is not new as consistently defining and collecting even lines of code data is an age old challenge. However we suspect the problem has been significantly underestimated across empirical studies. This is because it is by replicating previous empirical studies that the problem comes to light, yet relatively few replicated studies are performed. Consequently we need to treat cautiously the findings of studies using metrics that compare their empirical findings to others.

Addressing the problem of consistency in metrics definition and metrics data collection sufficient for replication is not easy. The problem has many facets, including the:

- precision of the original metric definition
- interpretation of this definition (it is not easy to formalise every aspect of a definition)
- implementation of this definition in a data collection tool – the actual practical way in which data is collected has an impact on the data collected.
- tools used to collect data. Different versions of tools may (invisibly) collect data slightly differently.
- data used – open source projects constantly evolve and it is essential to ensure that the correctly matching data snapshot has been used.

The consequence of variation in metrics data is that inconsistent data can be collected. This makes it difficult to replicate previous studies and therefore amass a reliable body of evidence on a topic.

However the variation in data collected by a metric may also have beneficial side effects, as a particular metric variant may be more useful than another variant. Indeed our previous study suggests that some slice-base metric variants are more useful than others for fault prediction in [2].

Despite having several detailed conversations with one of the authors of the original study (David Binkley) we failed to get to the bottom of why our data was different to the original data. We concluded that it is likely to be the result of us using a newer version of CodeSurfer. In the newer version that we subsequently discovered that there are differences in the project build options and the compiler and build environment. This may have affected the SDGs produced, and thus affected the metric values. This issue was probably confounded by other issues that we have not yet identified.

7. CONCLUSIONS

Software engineering needs to move towards a more scientific presentation of studies using metrics data. Studies reporting data in some other scientific disciplines are presented in a great deal more methodological detail than studies in software engineering. Such detail enables replication and thereby the growth of large corpuses of metrics data.

Such calls for methodological detail are not new. Kitchenham et al [12] were probably the first to explicitly call for this in empirical software engineering. However several subsequent studies also recommend this increased detail in relation to allowing replication (e.g. [19]). Indeed some studies present replication frameworks for use by empirical researchers (e.g. [3]). However our findings suggest that studies are not reporting sufficient methodological detail to enable replication of metrics studies. We suspect that the necessary technical complexity of some metrics studies makes it difficult to include such lengthy detail within a normal published paper. As a consequence we may need to treat cautiously studies which use metrics and compare their empirical findings to other studies.

Our conclusion is that we need to revisit how methodologies are presented in studies using metrics data and consider how important (and potentially extensive) methodological detail can be systematically made available to subsequent researchers. In particular researchers need to be encouraged to publish supporting documentation regarding experimental set-ups for published papers.

8. ACKNOWLEDGMENTS

An academic program licence for CodeSurfer® was provided by GrammaTech, http://www.grammatech.com/

9. REFERENCES

[1] Andersson, C. Runeson, P (2007) A Replicated Quantitative Analysis of Fault Distributions in Complex Software Systems. Software Engineering, IEEE Transactions on, 33(5), p273 - 286

[2] Bowes D, Counsell S, Hall T. (2008) Calibrating program slicing metrics for practical use, TAIC PART Conference, Windsor, Computer Society Press

[3] Brooks A, Roper M et al (2007) Replication's role in software engineering. In: Shull F, Singer J, Sjoberg DIK (eds) Guide to advanced empirical software engineering. Springer, London, pp 365–379

[4] Fenton, N.E.; Ohlsson, N. (2000) "Quantitative analysis of faults and failures in a complex software system," *Software Engineering, IEEE Transactions on* , vol.26, no.8, pp.797-814

[5] Fenton, N. E., Pfleeger, S. L. (1997). *Software Metrics: A Rigorous and Practical Approach* (2nd ed.). Boston, MA: PWS Publishing Company.

[6] Garcia Campos, C. (2009, April). *CVSAnalY Manual.* Retrieved May 16, 2010, from http://gsyc.es/~carlosgc/files/cvsanaly.pdf

[7] GrammaTech Inc. (2007). *CodeSurfer User Guide and Technical Reference* .

[8] GrammaTech Inc. (2010). *CodeSurfer product page*. Retrieved May 16, 2010, from GrammaTech Web site: http://www.grammatech.com/products/codesurfer/overview.html

[9] Green, P., Lane, P., Rainer, A., Scholz, S.-B. (2009). *An Introduction to Slice-Based Cohesion and Coupling Metrics.* Technical Report No. 488, University of Hertfordshire, School of Computer Science.

[10] Horwitz, S., Reps, T., Binkley, D. (1990). Interprocedural Slicing Using Dependence Graphs. *ACM Transactions on Programming Languages and Systems , 12* (1), 26-60.

[11] Kerr, M (2010). Can Slice-Based Metrics be Used to Predict the Amount of Maintenance that will be Required on Each Module of a Program?

[12] Kitchenham, B.A.; Pfleeger, S.L.; Pickard, L.M.; Jones, P.W.; Hoaglin, D.C.; El Emam, K.; Rosenberg, J.; , "Preliminary guidelines for empirical research in software engineering," *Software Engineering, IEEE Transactions on* , vol.28, no.8, pp. 721- 734, Aug 2002

[13] Lonchampt, G., L. Bonnetain, Hieter P (1996). Reproduction of Fleischmann and Pons Experiments. in Sixth International Conference on Cold Fusion, Progress in New Hydrogen Energy. 1996. Lake Toya, Hokkaido, Japan: New Energy and Industrial Technology Development Organization, Tokyo Institute of Technology, Tokyo, Japan.

[14] H. Longworth. (1984) Slice based program metrics. Master's thesis, Computer Science, Michigan Technical University, Michigan, USA.

[15] Meyers, T. M., Binkley, D. (2004) A Longitudinal and Comparative Study of Slice-Based Metrics. International Software Metrics Symposium, Chicargo, USA, IEEE Procs

[16] Meyers, T. M., Binkley, D. (2007). An Empirical Study of Slice-Based Cohesion and Coupling Metrics. *ACM Transactions on SoftwareMaintenance , 17*(1), pp. 1-25.

[17] Ott, L. M., & Thuss, J. J. (1993). Slice Based Metrics for Estimating Cohesion. In Proceedings of Internationl Software Metrics Symposium, Proceedings of the IEEE-CS, 71—81

[18] Pan K, Kim S, Whitehead J. Bug classification using program slicing metrics. In SCAM '06: Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation, Washington, DC, USA, 2006. IEEE Computer Society, p31–42

[19] Robles, G. (2010) Replicating MSR: A study of the potential replicability of papers published in the Mining Software Repositories Proceedings. Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on, Computer Society Press, 171 – 180

[20] Shull, F.; Basili, V.; Carver, J.; Maldonado, J.C.; Travassos, G.H.; Mendonca, M.; Fabbri, S. (2002) Replicating software engineering experiments: addressing the tacit knowledge problem. Empirical Software Engineering, Proceedings in International Symposium. p7 - 16

[21] Shull, F, Carver J, Vegas S, Juristo N, "The role of replications in Empirical Software Engineering," *Empirical Software Engineering*, vol. 13, no. 2, pp. 211–218, 2008.

[22] Weiser, M. (1981). Program Slicing. *Proceedings of the 5th International Conference on Software Engineering* (pp. 439-449). San Diego: IEEE Press.

[23] Weiser, M. (1984). Program Slicing. *IEEE Transon Software Engineering , SE-10* (4), 352-357.

APPENDIX A: COMBINATIONS OF SLICING SETTINGS

	Individual slicing settings selected					
	Formal Ins	Formal outs	Globals	Printf	File basis	Project basis
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	•					•
31				•		•
32	•	•	•	•		•

Note: Rows 4 and 20 are not possible to implement as at least one variable must be sliced on. Consequently there are 30 rather than the expected 32 rows.

APPENDIX B: CODESURFER SETTINGS

	Functionality	
Build Option	**high**	**highest**
Pointer Analysis	a	af
Pointer Analysis Hint Mask	75	75
Pointer Analysis Inlining	yes	yes
Pointer Analysis Inline Mask	513	537
Pointer Analysis Inline Max Growth (%)	0	0
String Constant Pointer Targets	one-string	many-strings
Variable Use/Def Sets	yes	yes
Create Declaration Vertices	yes	yes
Compute GMOD	yes	yes
Compute Data Dependence	yes	yes
Compute Control Dependence	yes	yes
Compute Summary Edges	yes	yes
Control-Flow Edges	both	both
CodeSurfer Library Models	normal	normal
AST Database	yes	yes
PDG_VERTEX -> AST mapping	yes	yes

Some of the built-in presets are defined as follows [7]

B.4 Developing fault-prediction models: What the research can show industry.

Hall T, Beecham S, Bowes D, Gray D, Counsell S (2011a) Developing fault-prediction models: What the research can show industry. Software, IEEE 28(6):96 –99

VOICE OF EVIDENCE

Editor: **Tore Dybå**
SINTEF
tore.dyba@sintef.no

Editor: **Helen Sharp**
The Open University, London
h.c.sharp@open.ac.uk

Developing Fault-Prediction Models:

What the Research Can Show Industry

Tracy Hall, Sarah Beecham, David Bowes, David Gray, and Steve Counsell

CODE FAULTS ARE a daily reality for software development companies. Finding and removing them costs the industry billions of dollars each year. The lure of potential cost savings and quality improvements has motivated considerable research on fault-prediction models, which try to identify areas of code where faults are most likely to lurk. Developers can then focus their testing efforts on these areas. Effectively focused testing should reduce the overall cost of finding faults, eliminate them earlier, and improve the delivered system's quality. Problem solved.

> Analysis of 206 fault-prediction models reported in 19 mature research studies reveals key features to help industry developers build models suitable to their specific contexts.

So why do so few companies seem to be developing fault-prediction models?

One reason is probably the sheer number and complexity of studies in this field. Before companies can start to develop the models, they must understand questions such as which metrics to include, which modeling techniques perform best, and how context affects fault prediction.

To answer these questions, we conducted a systematic review of the published studies of fault prediction in code from the beginning of 2000 to the end of 2010.[1] On the basis of this review and subsequent analysis of 206 models, we present key features of successful models here.

Context Is Key

We found 208 studies published on predicting code faults over the 11-year period covered in our review. These studies contain many hundreds of individual models, whose construction varies considerably.

Although it would be nice if companies could simply select one of these published models and use it in their own environment, the evidence suggests that fault-prediction models perform less well when transferred to different contexts.[2] This means that practitioners must understand how existing models might or might not relate to their own model development. In particular, they must understand the specific context in which an existing model was developed, so they can base their own models on those that are contextually compatible.

However, we found three challenges to this apparently simple requirement. First, many studies presented insufficient information about the development context. Second, most studies reported models built using open source data, which can limit their compatibility with commercial systems. Third, it remains unclear which context variables (application domain, programming language, size, system maturity, and so on) are tied to a fault-prediction model's performance.

Establishing Confidence in Existing Models

Practitioners need a basic level of confidence in an existing model's performance. Such confidence is based on

0740-7459/11/$26.00 © 2011 IEEE

understanding how well the model has been constructed, its development context, and its performance relative to other models.

Our review suggests that few of the published models provide sufficient information to adequately support this understanding. We developed a set of criteria based on research (for example, see Kai Petersen and Claes Wohlin[3]), to assess whether a fault-prediction study reports the basic information to support confidence in a model. Figure 1 shows a checklist based on these criteria (details are available elsewhere [1]).

When we applied these criteria to the 208 studies we reviewed, only 36 passed all criteria. Each of these 36 studies presents clear models and provides the information necessary to understand the model's relevance to a particular context.

We quantitatively analyzed the performance of the models presented in 19 of the 36 studies. These 19 studies all report categorical predictions, such as whether a code unit was likely to be faulty or not (see the sidebar). Such predictions use performance measures that usually stem from a confusion matrix (see Figure 2). This matrix supports a performance comparison across categorical studies.

We omitted 13 studies from further analysis because they reported continuous predictions, such as how many faults are likely to occur in each code unit. Such studies employ a wide variety of performance measures that are difficult to compare. Likewise, we omitted four categorical studies that reported performance data based on the area under a curve.

The 19 studies reporting categorical predictions contained 206 individual models. For each model, we extracted performance data for

- *precision* = TP/(TP + FP): proportion of units predicted as faulty that were faulty;
- *recall* = TP/(TP + FN): proportion of faulty units correctly classified; and
- *f-measure* = (2 × recall × precision)/(recall + precision): the harmonic mean of precision and recall.

For studies that didn't report these measures, we recomputed them from the available confusion-matrix-based data. This let us compare the performance of all 206 models across the 19 studies and to draw quantitative conclusions.

Phase One: Prediction Criteria

Is a prediction model reported?

Is the prediction model tested on unseen data?

Phase Two: Context Criteria

Is the source of data reported?

Is the maturity of data reported?

Is the size of data reported?

Is the application domain of data reported?

Is the programming language of data reported?

Phase Three: Model Criteria

Are the independent variables clearly reported?

Is the dependent variable clearly reported?

Is the granularity of the dependent variable reported?

Is the modeling technique used reported?

Phase Four: Data Criteria

Is the fault data acquisition process described?

Is the independent variables data acquisition process described?

Is the faulty/nonfaulty balance of data reported?

FIGURE 1. Checklist of criteria for establishing confidence in a model. Only 36 of 208 studies from the systematic literature review met all criteria.

	Number of code units	
	Predicted faulty	Predicted not faulty
Actually faulty	True positive (TP)	False negative (FN)
Actually not faulty	False positive (FP)	True negative (TN)

FIGURE 2. Confusion matrix. The two columns and two rows capture the ways in which a prediction can be correct or incorrect.

FAULT-PREDICTION MODEL ELEMENTS

There are three essential elements to a fault-prediction model.

PREDICTOR VARIABLES

These independent variables are usually metrics based on software artifacts, such as static code or change data. Models have used a variety of metrics—from simple LOC metrics to complex combinations of static-code features, previous fault data, and information about developers.

OUTPUT VARIABLES

A model's output, or independent variable, is a prediction of fault proneness in terms of faulty versus nonfaulty code units. This output typically takes the form of either *categorical* or *continuous* output variables.

Categorical outputs predict code units as either faulty or nonfaulty. Continuous outputs usually predict the number of faults in a code unit. Predictions can address varying units of code—from high-granularity units, such as plug-in level, to low-granularity units, such as method level.

MODELING TECHNIQUES

Model developers can use one or more techniques to explore the relationship between the predictor (or independent) variables and the output (or dependent) variables. Among the many available techniques are statistically based regression techniques and machine-learning techniques such as support vector machines. Ian Witten and Eibe Frank provide an excellent guide to using machine-learning techniques.[1]

Reference

1. I.H. Witten and E. Frank, *Data Mining: Practical Machine Learning Tools and Techniques*, Morgan Kaufmann, 2005.

What Works in Existing Models

When compared with each other, most of the 206 models peaked at about 70 percent recall—that is, they correctly predict about 70 percent of actual real faults. Some models performed considerably higher (for example, see Shivkumar Shivaji and his colleagues[4]), while others performed considerably lower (see Erik Arishholm and his colleagues[5]).

Models based on techniques such as naïve Bayes and logistic regression seemed to perform best. Such techniques are comparatively easy to understand and simple to use.

Additionally, the models that performed relatively well tended to combine a wide range of metrics, typically including metrics based on the source code, change data, and data about developers (see Christian Bird and his colleagues[6]). Often, the models performing best had optimized this set of metrics (for example, by using Principal Component Analysis or Feature Selection as in Shivaji[4]). Models using source-code text directly as a predictor yielded promising results (see Osamu Mizuno and Tohru Kikuno[7]).Models using static-code or change-based metrics alone performed least well. Models using LOC metrics performed surprisingly competitively.

Our systematic literature review suggests first that successful fault-prediction models are built or optimized to specific contexts. The 36 mature studies we identified can support this task with clearly defined models that include development contexts and methodologies. Our quantitative analysis of the 19 categorical studies from this set further suggests that successful models are based on both simple modeling techniques and a wide combination of metrics. Practitioners can use these results in developing their own fault-prediction models.

Acknowledgments
We thank the UK's Engineering and Physical Science Research Council, which supported this research at Brunel University under grant EPSRC EP/E063039/1, and the Science Foundation Ireland, which partially supported this work at Lero under grant 3/CE2/I303_1. We are also grateful to Sue Black and Paul Wernick for providing input to the early stages of the work reported.

References

1. T. Hall et al., "A Systematic Review of Fault Prediction Performance in Software Engineering," accepted for publication in *IEEE Trans. Software Eng.*; preprint available at http://bura.brunel.ac.uk/handle/2438/5743.
2. N. Nagappan, T. Ball, and A. Zeller, "Mining Metrics to Predict Component Failures," *Proc. 28th Int'l Conf. Software Eng.*, (ICSE 06), ACM Press, 2006, pp. 452–461.
3. K. Petersen and C. Wohlin, "Context in Industrial Software Eng. Research," *Proc. 3rd Int'l Symp. Empirical Software Eng. and Measurement* (ESEM 09), IEEE CS Press, 2009, pp. 401–404.
4. S. Shivaji et al., "Reducing Features to Improve Bug Prediction," *Proc. 24th IEEE/ACM Int'l Conf. Automated Software Eng.* (ASE 09), IEEE CS Press, 2009, pp. 600–604.
5. E. Arisholm, L.C. Briand, and E.B. Johannessen, "A Systematic and Comprehensive Investigation of Methods to Build and Evaluate Fault Prediction Models," *J. Systems and Software*, vol. 83, no. 1, 2010, pp. 2–17.
6. C. Bird et al., "Putting It All Together: Using Socio-Technical Networks to Predict

Failures," *Proc. 20th Int'l Symp. Software Reliability Eng.* (ISSRE 09), IEEE CS Press, 2009, pp. 109–119.

7. O. Mizuno and T. Kikuno, "Training on Errors Experiment to Detect Fault-Prone Software Modules by Spam Filter," *Proc. 6th Joint Meeting European Software Eng. Conf. and ACM SIGSOFT Symp. Foundations of Software Eng.* (ESEC-FSE 07), ACM Press, 2007, pp. 405–414.

TRACY HALL is a reader in software engineering at Brunel University, UK. Contact her at tracy.hall@brunel.ac.uk.

SARAH BEECHAM is a research fellow at Lero, the Irish Software Engineering Research Centre. Contact her at sarah.beecham@lero.ie.

DAVID BOWES is a senior lecturer at the University of Hertfordshire, UK. Contact him at d.h.bowes@herts.ac.uk.

DAVID GRAY is a PhD student at the University of Hertfordshire, UK. Contact him at d.gray@herts.ac.uk.

STEVE COUNSELL is a reader in software engineering at Brunel University, UK. Contact him at steve.counsell@brunel.ac.uk.

IEEE SOFTWARE CALL FOR PAPERS

Lean Software Development

Submission deadline: 1 February 2012 • Publication: September/October 2012

The lean product development paradigm entails an end-to-end focus on delivering to customer needs, minimized rework, efficient work streams, empowered teams, and continuous improvement.

We are interested to learn from industry experiences and academic empirical studies what principles deliver value and how organizations introduce lean. This issue will emphasize lean issues that influence software design, development, and management, and thus the success or failure of software projects. Our target is commercial and industry software, and issues of broad interest across software products and services, embedded software, and end-user-developed software.

We solicit articles in the following areas, among others:

- managing the transition from traditional development to lean;
- applying lean to critical (such as safety-critical) environments;
- experiences with combining lean and agile techniques;
- lean methods and experiences in commercial software, e.g., Kanban, value stream analysis, options thinking, queuing theory, and pull systems;
- systems thinking;
- case studies of notable successes or failures;
- empirical studies on adoption and use of lean principles in software engineering; and
- tool support for lean development.

Questions?

For more information about the special issue, contact the corresponding guest editor:

- Christof Ebert, Vector Consulting Services; Christof.Ebert@vector.com

Editorial team: Pekka Abrahamsson, Christof Ebert, Nilay Oza, Mary Poppendieck

For full call for papers: www.computer.org/software/cfp5
For full author guidelines: www.computer.org/software/author.htm
For submission details: software@computer.org

B.5 SLuRp: a tool to help large complex systematic literature reviews deliver valid and rigorous results.

Bowes D, Hall T, Beecham S (2012a) SLuRp: a tool to help large complex systematic literature reviews deliver valid and rigorous results. In: Proceedings of the 2nd International Workshop on Evidential Assessment of Software Technologies, ACM, pp 33–36

SLuRp – A tool to help large complex systematic literature reviews deliver valid and rigorous results

David Bowes
Science and Technology Research Institute
University of Hertfordshire
Hatfield, Herts
d.h.bowes@herts.ac.uk

Tracy Hall
Department of Information Systems and Computing
Brunel University
Uxbridge, UK
tracy.hall@brunel.ac.uk

Sarah Beecham
Lero – The Irish Software Engineering Research Centre
University of Limerick
Limerick, Ireland
sarah.beecham@lero.ie

ABSTRACT

Background: Systematic literature reviews are increasingly used in software engineering. Most systematic literature reviews require several hundred papers to be examined and assessed. This is not a trivial task and can be time consuming and error-prone.

Aim: We present SLuRp - our open source web enabled database that supports the management of systematic literature reviews.

Method: We describe the functionality of SLuRp and explain how it supports all phases in a systematic literature review.

Results: We show how we used SLuRp in our SLR. We discuss how SLuRp enabled us to generate complex results in which we had confidence.

Conclusions: SLuRp supports all phases of an SLR and enables reliable results to be generated. If we are to have confidence in the outcomes of SLRs it is essential that such automated systems are used.

Categories and Subject Descriptors

H.4.0 [**Information Systems Applications**]: *General*

General Terms

Management, Measurement, Documentation, Standardization.

Keywords

Systematic literature reviews, SLRs, collaboration tool, Open Source.

1. INTRODUCTION

Systematic literature reviews (SLRs) are increasingly established as an important aspect of software engineering research. The Journal of Information and Software Technology (IST) has a special section devoted to publishing SLRs. Top rated software engineering journals, such as IEEE Transactions on Software Engineering (TSE) and ACM's Transactions on Software Engineering and Methodology (TOSEM) have published numerous SLRs. These papers tend to have a high impact where as many as 334 citations have been recorded (e.g. [1]).

Although conducting SLRs has become popular they are difficult to execute well [2]. Performing an SLR in software engineering is a large, time consuming and complex task. Many hundreds or even thousands of papers can be identified as potentially relevant in the early stages. An extreme example of this is [3] where reviewers sourced over 3,000 papers, and only used 7 of them in their final review. In our own most recent SLR of 208 studies on fault prediction performance in software engineering [4], we initially identified 2,073 papers. Many pieces of information about all of these papers need to be accurately recorded, maintained, analysed and reported.

In addition to managing a large number of papers, performing an SLR requires many steps. All of these steps need to be implemented accurately for every paper.

In this paper we introduce a tool, 'SLuRp', to support the complex task of managing large numbers of papers, sharing tasks amongst a research team and following the arduous and rigorous SLR methodology recommended by Kitchenham and Charters [12]. Our contribution is to provide the research community with a support tool that increases the rigor and validity of our hitherto manual methods, while simplifying and shortening the time required to implement the many steps required to conduct a SLR.

Our own SLR [4], implemented twelve distinct steps. In most cases several reviewers are involved in an SLR. Our own SLR involved five different reviewers. The data each reviewer collects needs to be reliably stored and collated. This administrative complexity puts the quality of SLRs at risk, as the reliability and credibility of SLR conclusions are dependent on the quality of the SLR process used [5].

The recording and management of SLR data is often not reported in papers. Where the process is reported manual records are common [6, 7], as is the use of Endnote [8]. Both of these approaches are potentially problematic. Manual record keeping is time consuming and fault-prone. The functionality of Endnote supports only a limited number of SLR steps.

Other researchers have recognised the need to automate and simplify the SLR methodology for improved accuracy and speed

ESEM '12, September 19-20, 2012, Lund University, Sweden.

of execution. According to Felizardo et al [13], their Systematic Mapping Visual Text Mining tool (SM-VTM) reduces the effort and time required to categorize and classify data in systematic mapping studies. However the SM-VTM approach has questionable usability as it requires some prior experience and knowledge of text mining and visualization techniques. Malheiros et al [14] developed a similar VTM tool to specifically support the selection of primary studies in the systematic review process. VTM was shown to speed-up the selection process and improve quality of the selection process. While providing a potentially useful way to mine and cluster information from primary studies in a SLR, VTM does not provide the holistic, management of the whole SLR process, that requires tracking progress and storing of related data as in our SLuRp approach. Previous automated approaches focus on one aspect of the complex SLR process, leaving researchers to manage and integrate the new methods with their manual approaches. Eppi-reviewer[1] has been extensively used to support SLRs in other domains. It imports articles using reference manager databases and has a heavy focus on the synthesis aspect of literature reviews. However Eppi-reviewer is very generic and it is not obvious how to perform quality checks.

We could find no previous studies that look in detail at the way in which papers and data are managed and processed in existing SLRs. This is an important omission as noted by [9] who call for effective information retrieval tools to support performing systematic reviews given the “growth of the number of available papers and results published in the empirical software engineering field”.

In response to the difficulties we experienced in managing and recording information in our first three SLRs [8, 10, 11] we developed SLuRp (Systematic Literature unified Review program). This is our own web enabled database for recording and managing all data necessary to perform an SLR. SLuRp is written in Java with an SQL database. It is open source and available for other reviewers to adapt and use[3]. SLuRp has a client side Java component which semi-automates the process of extracting information when querying online databases.

In the next section we describe the functionality of SLuRp. In Section Three discuss the advantages of SLuRp using examples from our own SLR. We conclude in Section Four.

2. THE FUNCTIONALITY OF SLuRp

SLuRp is designed around the SLR tasks required to conduct a review according to Kitchenham and Charters’s [12] guidelines. How SLuRp supports each SLR task is discussed in this section.

2.1 Identify Relevant Research

All definitions required in the SLR must be established by the SLR research team. These include definitions of research questions, search terms, inclusion/exclusion criteria, quality check criteria etc. Once established all of these definitions are stored centrally on SLuRp.

2.2 Select Primary Studies.

a) SLuRp can apply pre-defined search terms to online databases (this is not permitted by some online databases e.g. ACM Portal).

b) SLuRp can semi-automatically extract papers from databases and save these.

c) SLuRp can semi-automatically store [PDF] copies of all papers locally if appropriate permissions exist.

d) SLuRp can record bibliographic details by importing BibTeX/RIS files from other citation management systems.

e) SLuRp prompts users to assign two+ reviewers to each paper. Each of the two+ reviewers independently applies the previously defined and stored inclusion and exclusion criteria to their assigned papers in turn.

f) SLuRp records assessment of each reviewer against the inclusion/exclusion criteria; and

g) SLuRp Records reason for rejection / acceptance.

h) SLuRp will identify differences in reviewer selections.

i) The need to reconcile disagreements between reviewers will be flagged by SLuRp.

j) Where reviewers cannot agree, SLuRp allows the user to assign an arbitrator. The paper remains in a ‘needs moderation’ state until a consensus is reached.

k) Once a decision has been recorded, SLuRp will remove rejected papers and record the reason (though SLuRp will change the status of papers to rejected, rather than remove them from the database). The frequency of disagreements for including or excluding papers is automatically generated by SLuRp. This allows inter-rater reliability scores to be easily produced.

l) Where reviewers have agreed to include the paper, SLuRp will store full copies of accepted papers (though SLuRp can be set up to store full copies of all papers).

2.3 Assess Study Quality

a) SLuRp allows reviewers to define a set of quality criteria. The application of which SLuRp supports in the same way as it deals with the application of inclusion/exclusion criteria. Steps e) to k) are repeated in the application of the quality criteria. SLuRp will: allocate each accepted paper to two reviewers; record results of quality assessment; flag differences in reviewer assessments; invite reviewers to reconcile differences between assessments; flag that a paper needs arbitration where no agreement is reached and record the results of quality assessment.

b) SLuRp allows reviewers to record all data extracted from included papers which have passed the quality check. It can record data about the study and the context of the study. For example dates of study, type of study, etc. It can also record data relating to the SLR’s research questions. Quantitative and qualitative can be recorded. This is achieved by each reviewer completing a SLuRp form answering project defined questions. Answers can be in the form of categorical, numerical or free flow textual data. Usually drop down type answers are linked to recording categorical data, whereas free textual data is required for qualitative data extraction.

We also wanted to ensure the validity to the data extracted from included papers. Consequently SLuRp also allows two+ reviewers to extract data from each included paper. The results of these independent data extractions are compared and reconciled using the same process as applied during the

[1] http://eppi.ioe.ac.uk/cms/Default.aspx?alias=eppi.ioe.ac.uk/cms/er4

[3] https://bugcatcher.stca.herts.ac.uk/SLuRp

application of inclusion/exclusion criteria and during the quality check (i.e. moderation between reviewers, followed by independent arbitration).

2.4 Synthesize Data

a) SLuRp aggregates quantitative findings and displays these in tables, or graphical form, e.g. box plot. All data is stored in a database which links the papers to the data extracted. This allows the data to be aggregated and statistically analysed using SQL statements. SLuRp allows the results of the data analysis to be presented in two main forms:
   - Tabular: results can be cross-tabulated and presented in HTML or LaTeX format for inclusion in papers.
   - Graphical: SLuRp uses JFreeChart[4] to produce: box plots, pie charts, scatter plots, bar charts.

   Each chart can be produced as: pdf, jpg, svg and png.

b) These graphical displays can highlight trends in results and allow the team to consider bias in results.

c) SLuRp supports the research team to synthesise qualitative data, according to the research questions (e.g report cross cutting themes across papers), by recording qualitative information which can then be aggregated and analysed in tabular and graphical format.

2.5 The Advantages of Using SLuRp

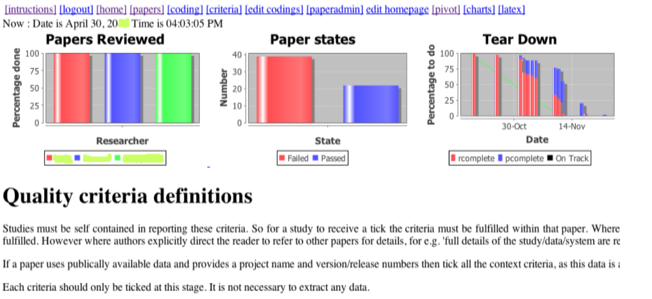
We have shown how SLuRp directly supports the implementation of SLR tasks. SLuRp also offers reviewers additional benefits we now discuss.

Centrally storing the SLR protocol and all associated instructions and criteria. The validity of SLR results is undermined if the protocol and associated instructions and criteria are not properly understood or applied consistently. SLuRp allows all SLR documentation to be centrally stored. This means that current versions of documents are always easily accessible to the whole team. This also means that current versions of all guidelines, instructions, definitions and criteria are those being used by everyone. In our previous SLRs such definitions change regularly and we have had problems ensuring that the right versions of definitions and instructions are being used by everyone. SLuRp's centralized storage of instructions, definitions and criteria ensures consistent application across papers and across reviewers.

Figure 1 is a screen shot of the first page SLuRp generates every time a user logs onto our own fault prediction SLR. At the top of the screen is the menu list, which has various options which include allowing access to the: instructions for all aspects of the SLR, full list of papers, editing facilities for any criteria used and the codes defined for data extraction etc.

This menu allows full access to all documentation (instructions, guidelines, definitions, criteria, code schemes etc.). All of this documentation can be edited via this menu. Editing access is determined by a variety of SLuRp user permissions.

Figure 1 is a screen shot of the first page SLuRp generates every time a user logs on; using our own fault prediction SLR as an example. The charts we implemented in our SLR dynamically track the progress of: individual reviewers, the current status of papers and the number of papers completed over time. The progress charts produced can be varied according to the specific needs of the SLR. These charts give immediate insight into the progress of individual reviewers and the team as whole.


[intructions] [logout] [home] [papers] [coding] [criteria] [edit codings] [paperadmin] edit homepage [pivot] [charts] [latex]
Papers Reviewed
Percentage done
100
75
50
25
0
Researcher
🟥 [redacted] 🟦 [redacted] 🟩 [redacted]
Paper states
Number
40
30
20
10
0
State
🟥 Failed 🟦 Passed
Tear Down
Percentage to do
100
75
50
25
0
30-Oct
14-Nov
Date
🟥 rcomplete 🟦 pcomplete ■ On Track

Now : Date is April 30, 20[redacted] Time is 04:03:05 PM

Quality criteria definitions

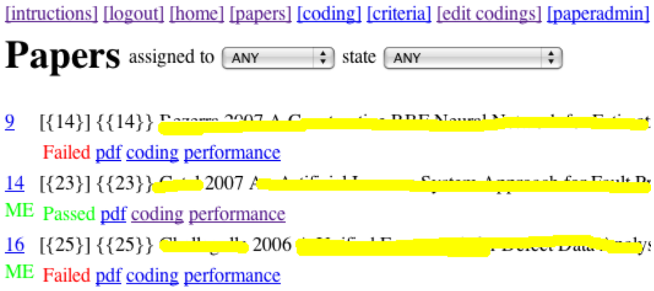
Studies must be self contained in reporting these criteria. So for a study to receive a tick the criteria must be fulfilled within that paper. Where fulfilled. However where authors explicitly direct the reader to refer to other papers for details, for e.g. 'full details of the study/data/system are re

If a paper uses publically available data and provides a project name and version/release numbers then tick all the context criteria, as this data is a

Each criteria should only be ticked at this stage. It is not necessary to extract any data.

Figure 1. Screenshot of opening screen

Maintaining a central list of papers and their current status. A central list of all papers is always maintained by SLuRp. This list can be filtered according to the particular reviewer assigned to papers. This allows a reviewer to easily identify the papers assigned to them and access all relevant information regarding that paper, including the pdf of the full paper. This central list of papers can also be filtered according to the status of each paper. This allows sets of papers to be easily generated, for example, all those papers which have been included, or need to be moderated. Overall this central list of papers makes accessing the right papers easy and less error-prone.


[intructions] [logout] [home] [papers] [coding] [criteria] [edit codings] [paperadmin]

Papers assigned to ANY state ANY

9 [{14}] {{14}} [redacted]
Failed pdf coding performance

14 [{23}] {{23}} [redacted] 2007 [redacted]
ME Passed pdf coding performance

16 [{25}] {{25}} [redacted] 2006 [redacted]
ME Failed pdf coding performance

Figure 2. List of papers screenshot

Figure 2 shows the central list of papers maintained by SLuRp for our own fault prediction SLR. Figure 2 shows that every paper is listed. The details provided for each paper include:

- paper id number

- publication details (blanked out in Figure 2 to keep anonymous papers that did not pass our quality check)

- link to the pdf

- status in terms of passed or failed the quality check (colour coded, red for Failed, and green for Passed)

- links to pages for extracted data (coding) and performance data (a second set of data we extracted during our SLR).

Figure 2 also shows the filters that can be applied to this list of papers. For example, the list can be filtered according to the papers assigned to a particular reviewer (assigned to), and/or filtered to list only those papers which are included or which have passed the quality check (state). This allows a specific list of papers to be generated according to the particular task a particular reviewer is currently working on.

[4] http://www.jfree.org/jfreechart/

Controlling and managing the SLR process. SLuRp produces information showing the current status of every paper. For example, papers are labelled as either accepted, rejected, passed or failed the quality check, undecided or in need of moderation or arbitration. SLuRp also produces information showing the current status of every reviewer. SLuRp quickly identifies if a reviewer is falling behind and needs extra support with their allocated reviewing tasks. This allows progress to be monitored and schedules to be managed. SLR process information is displayed in bar charts. These are dynamically updated and always current (examples of these are given in Figure 2).

Ensuring data validity. SLuRp allows at least two reviewers to be allocated to review each paper. Each reviewer 'votes' on whether the paper meets the inclusion and exclusion criteria. Each reviewer then 'votes' on whether the paper meets the quality criteria. Finally each reviewer 'votes' on what data is extracted from an included paper. SLuRp will highlight any conflicts. All conflicts will then go into the moderation process. This is where the reviewers themselves first try to resolve a disagreement. Should this prove impossible the paper will be labeled as needing arbitration. A new reviewer will then cast a deciding vote. This rigorous process of voting at three different SLR stages, secures the validity of the findings reported.

Reporting SLR results. The graphical charts and tables produced by SLuRp can be used directly in the final report and write up of the review. SLuRP can act as a LateX editor which incorporates the graphical results and tables directly into the final report. This allows changes to the raw data to be automatically included in the latest version of the SLR.

Producing data on the SLR process. In reporting an SLR a variety of data about the SLR itself needs to be produced. This may include the number of papers that meet the inclusion/exclusion criteria, the number of papers failing particular inclusion/exclusion criteria, the proportion of papers that failed the quality check, the number of decisions that required moderation. The inter rater reliability score for extracted data. SLuRp can produce all of this data easily.

3. CONCLUSION

Performing a rigorous SLR which reports reliable results is difficult but essential. Many of the difficulties are related to the administrative complexities involved with managing and controlling any large complex project. Our experience is that in order to produce reliable valid results, more than one reviewer is required. Maintaining large amounts of data in a team with several reviewers is time-consuming and error-prone. These errors are difficult to identify and eliminate without the use of a specific SLR tool like SLuRp.

4. ACKNOWLEDGMENTS

We thank the UK's Engineering and Physical Science Research Council, which supported this research at Brunel University under grant EPSRC EP/E063039/1. We also thank the Science Foundation Ireland, which partially supported this work at Lero under grant 3/CE2/I303_1. Finally we thank Thomas Shippey for writing the code to extract bibtex files from online databases.

5. REFERENCES

[1] Dybå, T. and T. Dingsøyr (2008). *Empirical studies of agile software development: A systematic review.* Information and Software Technology, 50(9–10): pp.833-859.

[2] Cruzes, D.S. and T. Dybå (2011). *Research synthesis in software engineering: A tertiary study.* Information and Software Technology, 53(5): pp.440-455

[3] Morais, Y., T. Burity, and G. Elias. *A Systematic Review of Software Product Lines Applied to Mobile Middleware.* in Sixth Int. Conf. on Information Technology. 2009.

[4] Hall, T., S. Beecham, D. Bowes, D. Gray, and S. Counsell (in press). *A Systematic Literature Review on Fault Prediction Performance in Software Engineering.* IEEE Transactions on Software Engineering (TSE).

[5] Brereton, P., B.A. Kitchenham, D. Budgen, M. Turner, and M. Khalil (2007). *Lessons from applying the systematic literature review process within the software engineering domain.* Journal of Systems and Software, 80(4): pp.571-583.

[6] Stol K, Ali Babar M, Avgeriou P, and Fitzgerald B, *A comparative study of challenges in integrating Open Source Software and Inner Source Software.* Information and Software Technology, 2011. 53(12): p. 1319-1336.

[7] Lane, S. and I. Richardson (2011). *Process models for service-based applications: A systematic literature review.* Information and Software Technology, 53(5): pp.424-439.

[8] Beecham, S., N. Baddoo, T. Hall, H. Robinson, and H. Sharp (2008). *Motivation in Software Engineering: A Systematic Literature Review.* Information and Software Technology (IST), Elsevier, 50 (9-10): pp.860–878.

[9] Ramampiaro, H., D. Cruzes, R. Conradi, and M. Mendona. *Supporting evidence-based Software Engineering with collaborative information retrieval* in Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom), 2010 6th International Conference on. 2010, 9-12 Oct. 2010.

[10] Hall, T., S. Beecham, N. Baddoo, H. Sharp, and H. Robinson (2009). *A Systematic Review of Theory Use in Studies Investigating the Motivations of Software Engineers.* ACM Transactions on Software Engineering and Methodology (TOSEM), 18(3).

[11] Zhang, M., T. Hall, and N. Baddoo (2011). *Code bad smells: a review of current knowledge.* Journal of Software Maintenance and Evolution: research and practice, 23(3): pp.179- 202.

[12] Kitchenham, B. and S. Charters (2007), *Guidelines for performing systematic literature reviews in software engineering (version 2.3)*, in *EBSE Technical Report*, Keele University, UK.

[13] Felizardo R K, Nakagawa E Y, Feitosa D, Minghim R, and Maldonado D C. *An Approach Based on Visual Text Mining to Support Categorization and Classification in the Systematic Mapping*. in *14th International Conference on Evaluation and Assessment in Software Engineering (EASE)*. 2010. Keele University, UK.

[14] Malheiros V D, Hohn E, Pinho R,. Mendonça M, and Maldonado J C. *A Visual Text Mining approach for Systematic Reviews. In: Empirical Software Engineering and Measurement*. in *First International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 2007. Madrid, Spain: IEEE Computer Society.

B.6 DConfusion: A technique to allow cross study performance evaluation of fault prediction studies.

Bowes D, Hall T, Gray D (2013) DConfusion: A technique to allow cross study performance evaluation of fault prediction studies. Automated Software Engineering Journal (Accepted subject to minor changes)

Special Issue of ASEJL:Next Generation of Empirical SE manuscript No.
(will be inserted by the editor)

DConfusion: A technique to allow cross study performance evaluation of fault prediction studies.

David Bowes · Tracy Hall · David Gray

the date of receipt and acceptance should be inserted later

Abstract There are many hundreds of fault prediction models published in the literature. The predictive performance of these models is often reported using a variety of different measures. Most performance measures are not directly comparable. This lack of comparability means that it is often difficult to evaluate the performance of one model against another. Our aim is to present an approach that allows other researchers and practitioners to transform many performance measures back into a confusion matrix. Once performance is expressed in a confusion matrix alternative preferred performance measures can then be derived. Our approach has enabled us to compare the performance of 600 models published in 42 studies. We demonstrate the application of our approach on 8 case studies, and discuss the advantages and implications of doing this.

D.Bowes
Science and Technology Research Institute
University of Hertfordshire
College Lane
Hatfield, AL10 9AB
United Kingdom
E-mail: d.h.bowes@herts.ac.uk

T. Hall
Department of Information Systems and Computing
Brunel University
Uxbridge
Middlesex, UB8 3PH
United Kingdom
E-mail: tracy.hall@brunel.ac.uk

D. Gray
Science and Technology Research Institute
University of Hertfordshire
College Lane
Hatfield, AL10 9AB
United Kingdom
E-mail: d.gray@herts.ac.uk

Keywords fault, confusion matrix, machine learning

1 Introduction

Imagine the following simplified scenario:

"You are a practitioner thinking about starting to use fault prediction models. You hope that such models will help you to identify the most fault prone parts of your system. You then plan to target your test effort on those parts of the system. You think that doing this may reduce the faults delivered to your users and reduce the cost of your system. You are not an expert in fault prediction models yourself, but you have seen many such models published in the literature. You identify several published models that have been developed in a similar software development context to your own. You decide to evaluate the performance of these models with a view to trying out the top three models in your project. However when you look at the model performance figures they are reported using a variety of different performance measures. Several studies report Precision[1] *and Recall. Some report Error Rate. Some report pd and pf. Others report p_{opt}. A few report Area Under the Curve of the Receiver Operator Curve. One provides a confusion matrix. It is beyond your expertise to identify a comparative point of reference amongst these different measures. You struggle to understand how the overall performance of a model compares to the others. And so you decide that fault prediction models are too complicated to use and abandon the idea."*

This type of scenario may partially explain why the uptake of fault prediction models is low in industry. This low uptake is important as finding and fixing faults in code costs the software industry many millions of dollars every year. Predicting effectively where faults are in code occupies many researchers and practitioners. Our previous work (Hall et al., 2012) showed that 208 software fault prediction studies were published between January 2000 and December 2010. These 208 studies contained many hundreds of fault prediction models.

Despite this significant research effort it remains difficult or inconvenient to compare the performance of these models. The difficulty in comparing predictive performance means that identifying which fault prediction models perform best in a given context is complex. This complexity in comparing the performance of models is not only a likely barrier to practitioners using fault prediction models, but also makes it difficult for researchers to meta-analyse fault prediction studies (Cruzes and Dybå, 2011). This lack of opportunity to meta-analyse limits the ability of the fault prediction community to mature, as we are not building an evidence base that is as useful as it should be.

One of the difficulties when comparing the performance of fault prediction models stems from the many performance measurement schemes devised, used and reported by studies. Many of the schemes used by studies highlight different aspects of predictive performance. For example, Menzies et al. (2007b)

[1] Definitions of particular measures are given in Section 2.

use pd and pf to highlight standard predictive performance, while Mende and Koschke (2010) use p_{opt} to assess effort-awareness. The different performance measurement schemes used mean that directly comparing the performance reported by individual studies is difficult and potentially misleading. Such comparisons cannot compare like with like as there is no adequate point of comparison.

It is perfectly legitimate for studies to report different performance measures. Studies may be interested in reporting prediction models with particular qualities. Some studies may be interested in reporting models which reduce the amount of effort wasted on code predicted as faulty which turns out not to be faulty. In these cases, measures based on the number of false positives will be of most interest. Other studies may be developing models focused on identifying the maximum number of faults in the system. In which case measures related to the number of true positives are likely to be the performance focus. The qualities needed in a fault prediction model depend on, for example, application domain. Models used in the safety critical domain are likely to need different predictive qualities to those in other domains. However developers and potential users of models may want to compare performance in terms of a particular predictive quality. This requires a conversion of performance figures from those reported to those reflecting the predictive quality of interest. The ability to convert predictive measures in this way allows the predictive performance of a wide range of models to be compared and benchmarked.

We previously found (Hall et al., 2012) that Precision and Recall were the most commonly reported predictive performance measures used with binary[2] fault prediction models (e.g. Arisholm et al. (2007); Catal et al. (2007); Denaro and Pezzè (2002); Koru and Liu (2005)). However, many studies provide only limited predictive performance data, often only reporting performance using their preferred performance measures. This preferred data usually represents the performance of specific models in the most positive light. An issue also highlighted by Zeller et al. (2011). This preferred measurement data may be unusual and rarely reported in other studies. For example, only a few studies report the use of Error Rate (Khoshgoftaar and Seliya, 2004; Seliya et al., 2005; Yi et al., 2010) or p_{opt} (Mende and Koschke, 2010). Without additional performance data that is more commonly reported by studies, it is difficult to satisfactorily compare the predictive performance of such models. A common point of comparison is needed.

The confusion matrix is usually at the centre of measuring the predictive performance of models (the confusion matrix is discussed in detail in Section 2). Most other predictive performance measures are calculated from the confusion matrix. The confusion matrix is a powerful point of comparative reference. All models reporting binary results can have their predictive performance expressed via a confusion matrix (Ostrand and Weyuker, 2007). This

[2] Binary models are those predicting that code units (e.g. modules or classes) are either fault prone (fp) or not fault prone (nfp). Binary models do not predict the number of faults in code units. In this paper we restrict ourselves to considering only binary models that are based on machine learning techniques.

means that it is a relatively universal comparative basis. It is also a simple and understandable way to show predictive performance. More sophisticated measures of predictive performance can be calculated from a confusion matrix. The confusion matrix provides measurement flexibility as specific measures may be derived from the confusion matrix which evaluate particular model qualities. The importance of the confusion matrix is discussed in detail by Pizzi et al. (2002).

Although the confusion matrix is pivotal to reporting the performance of fault prediction studies, some weaknesses exist. These weaknesses are caused by the way in which the confusion matrix is used to produce the final reported performance measures. The performance measures of fault prediction studies tend to be averaged over a number of runs of the same experiment in order to produce a more reliable/stable performance value. It is rare for a paper to give the variation in the performance values but Elish and Elish (2008) demonstrate that the variation is relatively small. Forman and Scholz (2010) go into great detail about the different ways in which the confusion matrix and hence the performance measures are computed. It is hard to anticipate if the small variations in performance measures when combined together will adversely affect the accuracy of re-computing the confusion matrix. In this paper we investigate the accuracy of re-computing the confusion matrix by performing a 10 x 10-fold cross-validation experiment.

The confusion matrix also has limitations because it looses some of the information that a model may have about the faultiness of a module. Some learners (e.g. Naive Bayes) can order the modules from most likely to be faulty to the least likely. Several researchers report their results in such an ordering form (e.g. (Ostrand et al., 2004)). This information may be important to developers who are trying to prioritise their effort in fixing modules which are most likely to be faulty. Arisholm et. al (Arisholm et al., 2007) describe a technique for estimating the return on investment of inspecting progressively larger amounts of code by ranking modules by their predicted defectiveness followed by their length. The curve produced is compared against a model which predicts all modules as having a uniform defect density. Mende and Koske (Mende and Koschke, 2010) expand on this by producing a metric (p_{opt}) which compares the return on investment curve of a model to an optimal model which ranks modules by their defect density. Effort aware metrics are important, however the metric values they produce requires more information than knowing if a module is defective or not. Converting all results to a common confusion matrix format will loose any ordering information. The loss of such information is beyond the scope of this paper which is focused on comparing the results of studies which have reported binary classifications. We do however propose a solution to the problem of loosing information in Section 8 which describes how future studies could report their results in a way that allows all information to be retained.

In this paper we present a process by which we transform a variety of reported predictive performance measures back to a confusion matrix (Bowes et al., 2012). These measures cover most of those reported by the 208 fault prediction studies we previously reviewed (Hall et al., 2012). We demonstrate that the re-computation process on a real fault prediction study produces very small errors by carrying out a study using a variety of datasets and learners in order to establish the accuracy of our process for re-computing the confusion matrix. We illustrate the use of this process by constructing the confusion matrix for 8 published models. From these confusion matrices we compute a range of alternative performance measures and demonstrate how the technique has produced a clearer understanding of the results reported by some studies.

In Section 2 we describe the measurement of predictive performance by discussing in detail the basis of the confusion matrix and related compound measures of performance. In Section 3 we present our method of transforming a variety of performance measures to the confusion matrix and explain how alternative measures can then be derived from this matrix. Section 4 describes an experiment to test the accuracy of re-computing the confusion matrix. Section 5 provides the results of worked examples from published studies in which we transform the performance measures back to the confusion matrix. Section 6 identifies threats to the validity of this study. Section 7 discusses the implications of transforming performance measures. Section 8 describes how this work is relevant to empirical software engineering v2.0. We conclude and summarise in Section 9.

2 Measuring Predictive Performance

This section is based on several previous studies which provide an excellent overview of measuring the predictive performance of fault models (e.g. Ostrand and Weyuker (2007), Jiang et al. (2008) and Lessmann et al. (2008)).

2.1 The Confusion Matrix

The measurement of predictive performance is often based on the analysis of data in a confusion matrix (see Ostrand and Weyuker (2007) and Pizzi et al. (2002)). This matrix reports how a prediction model classified the different fault categories compared to their actual classification (i.e. predicted versus observed). This is represented by four pieces of data:

- True Positive (TP): An item is predicted as faulty and it is faulty
- False Positive (FP): An item is predicted as faulty and it is not faulty
- True Negative (TN): An item is predicted as not faulty and it is not faulty
- False Negative (FN): An item is predicted as not faulty and it is faulty

Table 1 shows the structure of a confusion matrix.

In a confusion matrix, it is normal for the sum of the instances of each possibility to be reported, see Table 2.

Few studies report complete confusion matrices for their experiments. Studies that do include: Pai and Dugan (2007), Zhou and Leung (2006) and Kaur

Table 1: Confusion matrix

	observed true	observed false
predicted true	TP	FP
predicted false	FN	TN

Table 2: Confusion matrix with example summed instances

	observed true	observed false
predicted true	33	2
predicted false	17	98

et al. (2009). Most studies prefer to report a sub-set of the compound performance measures shown in Table 3 and discussed in Section 2.2.

2.2 Compound Measures

Many performance measures are related to components of the confusion matrix. Table 3 shows how some commonly used performance measures are calculated relative to the confusion matrix.

Table 3 shows that Accuracy is the proportion of units correctly classified. Table 3 also shows that Recall (otherwise known as the true positive rate, probability of detection (pd) or Sensitivity) describes the proportion of faulty code units (usually files, modules or packages) correctly predicted as such. Precision describes how reliable a prediction is in terms of what proportion of code predicted as faulty actually was faulty. Both Recall and Precision are important when test sets are imbalanced (see the following sub-section), but there is a trade-off between these two measures (see Jiang et al. (2008) for a more detailed analysis of this trade-off). An additional composite measure is the false positive rate (pf) which describes the proportion of erroneously predicted faulty units. The optimal classifier would achieve a pd of 1, Precision of 1, a pf of 0 and an f-measure of 1. The performance measure balance combines pd and pf. A high Balance value (near 1) is achieved with a high pd and low pf. Balance can also be adjusted to factor in the cost of false alarms which typically do not result in fault fixes. Matthews Correlation Coefficient (MCC) is a measure rarely used in software fault prediction (Baldi et al., 2000). MCC is more commonly used in medical research and bioinformatics e.g. Baldi et al. (2000); Sun et al. (2009). It is a Chi Square based performance measure on which all four quadrants of the confusion matrix are included in the calculation. MCC results are the equivalent of reporting R^2 in regression modelling and results range from -1 to 1 (with 0 indicating random results).

The Receiver Operator Curve (ROC) is an important measure of predictive performance. When the combinations of Recall and pf for a series of experi-

Table 3: Compound Performance Measures

Measures	Defined As
Accuracy (a) / Correct Classification Rate (CCR)	$\frac{TP+TN}{TP+TN+FP+FN}$
Error Rate (er)	$\frac{FP+FN}{TP+TN+FP+FN}$
Recall (r)/ True Positive Rate / Sensitivity / Probability of Detection (pd)	$\frac{TP}{TP+FN}$
True Negative Rate / Specificity (spec)	$\frac{TN}{TN+FP}$
False Positive Rate / Type I Error Rate (t1)/ Probability of False Alarm (pf)	$\frac{FP}{TN+FP}$
False Negative Rate / Type II Error Rate (t2)	$\frac{FN}{FN+TP}$
Precision (p)	$\frac{TP}{TP+FP}$
F-Measure / F-Score	$\frac{2 \times Recall \times Precision}{Recall + Precision}$
Balance	$1 - \frac{\sqrt{(0-pf)^2+(1-pd)^2}}{\sqrt{2}}$
G-mean	$\sqrt{Recall \times Precision}$
Matthews Correlation Coefficient (MCC)	$\frac{TP \times TN - FP \times FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}}$

ments are plotted they produce a ROC. It is usual to report the area under the curve (AUC) as varying between 0 and 1, with 1 being the ideal value. Because the AUC is a result of a series of experiments where the meta-parameters are varied, it is not possible to compute the confusion matrix from AUC and visa versa. AUC can only be computed by either a) repeating an experiment to achieve a range of Recall and pf, or b) by varying the meta-parameters e.g. the decision threshold for Naive Bayes. In the first case, reporting the prediction for every run and every item will allow others to compute any performance measure and the AUC. In the latter case, it is necessary to record both the items in each cross-validation fold and the model produced for each training set in order to be able to generate any performance value including the AUC. To our knowledge, no fault prediction study has every provided such an exhaustive set of data.

Additional metrics which take into consideration the effort required to find/fix a defect have been proposed. Arisholme et. al Arisholm et al. (2007) proposed a measure called 'cost effectiveness' (CE). CE is computed by ordering the modules by their predicted defect density and then plotting the number of actual faults found against the total amount of code inspected starting with the modules with the highest predicted defect density.

Previous studies have critiqued the use of these various measures of performance. For example Zhang and Zhang (2007), Menzies et al. (2007a) and Gray et al. (2011) discuss the use of precision. However such a critique is beyond the scope of the work reported here.

2.3 Imbalanced Data

Substantially imbalanced data sets are commonly used in fault prediction studies (i.e. there are usually many more non-faulty units than faulty units in the data sets used in fault prediction) (Chawla et al., 2004; Zhang and Zhang, 2007). An extreme example of this is seen in the NASA data set PC2, which has only 0.4% of data points belonging to the faulty class (23 out of 5589 data points). This distribution of faulty and non-faulty units has important implications in fault prediction. Imbalanced data can strongly influence the suitability of predictive performance measures. Measures which favour the majority class (such as Accuracy and Error Rate) are not sufficient by themselves (He and Garcia, 2008). More appropriate measures for imbalanced data sets include: precision, recall, f-measure, MCC and G-mean He and Garcia (2008). Consequently data imbalance is an important consideration in our method of re-computing the confusion matrix[3].

3 Our Method of Re-Computing the Confusion Matrix

To compare the results of one study with the results of another we re-compute the confusion matrix for each study and then calculate the preferred compound measures from this. Zhang and Zhang (2007) did something similar to this by re-computing Precision for the study by Menzies et al. (2007b) which originally reported pd and pf. Our approach is motivated by the work of Zhang and Zhang (2007). We now describe the process by which transformation from a variety of compound measures to the confusion matrix can be achieved.

[3] Data imbalance also has serious implications for the training of prediction models. Discussion of this is beyond the scope of this work (instead see Gray et al. (2011), Zhang and Zhang (2007), Turhan et al. (2009), Batista et al. (2004) and Kamei et al. (2007)).

3.1 Creating a Frequency-Based Confusion Matrix

The precise method needed to re-compute the confusion matrix varies slightly depending upon the original measures reported. In most cases the first thing that needs to be done is that we produce a frequency-based confusion matrix. These confusion matrices are different from instance based confusion matrices (an example of which was shown in Table 2). Table 4 shows the frequencies (or proportions for each confusion matrix quadrant) based on the instances in Table 2. These frequencies are derived by dividing the instances in each quadrant by the total number of instances in the matrix. This shows the relative proportion each quadrant represents of the whole confusion matrix. From now on we will append $_f$ to TP, TN, FP and FN to distinguish frequency values from instance based values for example TP_f.

3.2 Calculating Faulty and Non-Faulty Data Distributions

Constructing a frequency based confusion matrix is possible when the class distribution (i.e. the proportion of faulty versus non-faulty units) is known[4]. To do this we use d as the frequency of the faulty units, where:

$$d = \frac{TP + FN}{TN + TP + FP + FN} \tag{1}$$

Applying (1) to the example instances reported in Table 2 would result in:

$$n = 33 + 17 + 2 + 98 = 150, d = \frac{33 + 17}{150} = \frac{50}{150} = 0.3333$$

This shows that given the confusion matrix shown in Table 2, 33% of the units in the data set on which the model was applied, were faulty.

Table 4: Frequency Confusion Matrix

	observed true	observed false
predicted true	0.2200	0.0133
predicted false	0.1133	0.6533

$$TN_f + TP_f + FP_f + FN_f = 1$$
$$d = 0.2200 + 0.1133 = 0.3333$$

[4] When this class distribution is not known it is often possible to calculate the proportion of faulty units in a data set.

Table 5: A Subset of Pre-Requisite Combinations of Performance Measures for Re-Computing the Confusion Matrix.

Fault Frequency (d)	Type I	Type II	Precision	Recall	Accuracy	pf	Error Rate	Specificity
			✔	✔		✔		
				✔	✔			✔
			✔	✔	✔			
✔			✔	✔				
✔				✔				✔
✔				✔	✔			
✔				✔		✔		
		✔			✔	✔		
		✔				✔	✔	
	✔	✔					✔	

NB this is not an exhaustive list. For example, it is possible to calculate fault frequency (d) by dividing the number of faulty instances by the total number of instances.

3.3 Transforming Specific Compound Measures

A wide variety of compound measures are reported by studies. Our approach is successful when a particular sub-set of these measures is reported by studies. Table 5 provides some example pre-requisite combinations of performance measures that must be available.

Each of these combinations of measures requires a specific method by which to re-compute the confusion matrix. Formulae for the most common measures reported are now described.

1. Transforming *Precision*, *Recall* **and** *pf*
We first need to know the frequency of the true class d.

$$1 = TP_f + TN_f + FP_f + FN_f \tag{2}$$

$$d = \frac{TP_f + FN_f}{TP_f + TN_f + FP_f + FN_f} = \frac{TP_f + FN_f}{1} = TP_f + FN_f \tag{3}$$

It then becomes possible to calculate TP_f, FP_f, TN_f and FN_f as follows:
Given pf and d

$$TN_f = (1 - d)(1 - pf) \tag{4}$$

$$FP_f = (1 - d)pf \tag{5}$$

Given $Recall(r)$ and d

$$TP_f = d \times r \tag{6}$$

$$FN_f = d(1-r) \tag{7}$$

Given $FNR(TypeII(t2))$, pf and d we already have (2), (4) and (5)

$$FN_f = t2 \times d \tag{8}$$

$$TP_f = 1 - FN_f - TN_f - FP_f \tag{9}$$

Given $Precision(p)$, $Recall(r)$ and d we already have (2), (6) and (7)

$$FP_f = \frac{TP_f(1-p)}{p} = \frac{d(1-p)r}{p} \tag{10}$$

$$TN_f = 1 - FP_f - FN_f - TP_f \tag{11}$$

2. Transforming $ErrorRate(er)$**,** $TypeII(t2)$ **and** pf

$$d = \frac{er - pf}{t2 - pf} \tag{12}$$

which can be used with (8) to give FN_f and (5) to give FP_f.

3. Transforming $Precision(p)$**,** $Recall(r)$ **and** $Accuracy(a)$

$$d = \frac{p(1-a)}{p - 2pr + r} \tag{13}$$

which can then be used with (6),(7),(10) and (11)

4. Transforming $Accuracy(a)$**,** pf **and** $FNR(TypeII(t2))$

$$er = 1 - a \tag{14}$$

$$d = \frac{er - pf}{t2 - pf} \tag{15}$$

which can be used with (8) to give FN_f and (5) to give FP_f.

$$TP_f = d(1 - t2) \tag{16}$$

which can be used with (11) to give TN_f.

We have automated many conversions by developing a java tool ‘DConfusion’[5]. This tool allows individual performance measurement data to be input and will automatically re-compute the confusion matrix by iterating over the the equations until no extra performance measures can be derived.

4 Methodology for Validating the Accuracy of the Re-Computation of the Confusion Matrix

Validating the techniques and equations for re-computing of the confusion matrix is necessary if other users are to have confidence in the secondary data that DConfusion produces. The re-computation of the confusion matrix involves the solution of a set of simultaneous equations and therefore for a single experiment, the error involved with re-computation should only be caused by rounding errors. There is therefore a threat that the re-computation may not be accurate because of the inclusion of rounding errors. We also recognise that some studies report average performance measures from m x n cross-validation studies which may also add to the possible error of re-computation. Using average values from repeated experiments is known to be problematic (Fenton and Neil, 1999). If our process of re-computing the confusion matrix is to have any validity, we need to test the technique on a controlled real world m x n cross-validation fault prediction study. This enables us to know what the original confusion matrix is and allows us to compare the original

[5] This tool is available at: `https://bugcatcher.stca.herts.ac.uk/DConfusion/`

values with computed values. We can also run the experiment a few times to see how the variation in performance affects the average re-computed values. Without our own fault prediction experiment it is not possible to compare the original confusion matrix against the re-computed confusion matrix based on different sub-sets of performance measure. The rest of this section describes the experiments carried out to assess the magnitude of the error which should be expected when re-computing the frequency confusion matrix (which from now on will be called confusion matrix′).

4.1 Validating the Re-Computation Process

We developed the DConfusion tool to re-compute the confusion matrix′ which incorporates the set of equations described above. DConfusion includes a JUnit test to ensure the validity of the equations. The Junit test is included to allow others to extend the set of re-computation equations and still maintain confidence in the results produced by DConfusion. The JUnit test randomly generates TP, TN, FP and FN values to produce a confusion matrix. This is then used to compute the performance values (e.g. precision, recall and accuracy). The JUnit test then uses all permutations of performance values to recalculate the confusion matrix′ which is then compared against the original confusion matrix to test the system. DConfusion tests that the re-computed performance values (including TP_f etc) are within the correct range (for example $0 \leq precision \leq 1$ and $-1 \leq MCC \leq 1$).

When re-computing the confusion matrix′, DConfusion adds a rounding error to the values provided by the user. For example, if the user enters the following values $a = 0.50, p = 0.50$ and $r = 0.50$, DConfusion will also try $a = 0.50 \pm 0.01, p = 0.50 \pm 0.01$ and $r = 0.50 \pm 0.01$ generating $TN_f = 0.25(+0.02 - 0.03), TP_f = 0.25(+0.02 - 0.01), FP_f = 0.25(+0.02 - 0.01)$ and $FN_f = 0.25(+0.02 - 0.01)$. This gives some indication of the amount of error that can be expected due to rounding errors. Adding rounding error values also overcomes the problem where the performance values are identical. This can be an issue for some equations because it can lead to a division by error in some cases e.g. equation 15.

4.2 Validating the Re-Computation Technique on m x n Cross-Validation Experiments

Many studies report average performance values reported over m runs of the same experiment. It is possible that over a set of m runs, the performance values may vary, for example Elish and Elish (2008) report the standard deviation of their performance values. It is possible that the average values may not allow an accurate re-computation of the confusion matrix′. In order to test the accuracy of re-computing the confusion matrix′ on real data we carried out a m x n cross-validation fault prediction experiment which produced all of the performance measures described in Table 5. The accuracy of the re-computation was assessed by selecting a subset of the performance measures (calculated as averages over the m runs) described in Table 5 as the input for the re-computation process. The re-computed confusion matrix$'$ was then used to re-compute the remaining performance measures. These are then compared against the original values from the fault prediction experiment.

4.3 Fault Prediction Experiment

In order to evaluate the re-computation on as wide a set of values of different performance measures as possible, 12 dataset were used with different learners. The fault prediction experiment used twelve frequently used NASA datasets (Table 6) obtained originally from MDP. The data was cleaned using the protocol described in (Gray et al., 2012) with the exception that duplicate tuples were not removed, this was because the learners chosen perform differently when duplicates are present Gray et al. (2012) and we wanted to produce as wide a range of performance measures as possible. Three learners were used: RPART, Random Forest and Naive Bayes. These were chosen for their simplicity, speed and varying performance (Hall et al., 2012). The statistical programming language R was used to perform our experiments.

Table 6: Summary statistics for NASA datasets before cleaning

Dataset	Language	Total KLOC	No. of Modules	%Faulty Modules
CM1	C	20	505	10
KC1	C++	43	2109	15
KC3	Java	18	458	9
KC4	Perl	25	125	49
MC1	C & C++	63	9466	0.7
MC2	C	6	161	32
MW1	C	8	403	8
PC1	C	40	1107	7
PC2	C	26	5589	0.4
PC3	C	40	1563	10
PC4	C	36	1458	12
PC5	C++	164	17186	3

We used a 10-fold stratified cross-validation experiment. This involved splitting a dataset into 10 sets (folds) of tuples in such a way that each of the 10 sets has the same proportion of faulty to none faulty tuples as the population from which the folds were derived. 10 sub experiments are then performed by using one of the folds as a test set and the rest are combined to form a training set. Some of the datasets have a very small proportion of tuples which are faulty (MC1 and PC2). This lack of tuples for the minority class hinders learners in their ability to learn the features of a faulty tuple. The imbalance can be altered in the training set by oversampling the minority class. SMOTE (Chawla et al., 2002) was used to increase the number of the minority class in the training sets by synthetically generating new examples from existing examples of the minority class from the training set. The training set was then randomised so that the 10 sub experiments do not always present the tuples to the learner in the same order. Each learner was then tuned and trained on the training set which resulted in a new model. Each model was then used to predict the class for every item in the test set. Each 10-fold cross-validation experiment was repeated 10 times for each dataset and learner combination.

This experiment follows the guidelines for performing a fault prediction study described by Gray et al. (2012) and Hall and Bowes (2012). The fault prediction experiment is summarised in Algorithm 1 (Appendix B).

4.4 Computing the Performance Measures

A set of performance measures for each learner on each dataset was then calculated using Algorithm 2 (Appendix C) which is described as follows. After a model was built, each tuple in the test set was categorised as TP if the tuple was faulty and predicted as faulty, TN if the tuple was not faulty and predicted as not faulty, FP is the tuple was not faulty but predicted as faulty and FN if the tuple was faulty but predicted as not faulty. The 10-fold cross validation strategy resulted in all tuples being classified. The total number of TP, TN, FP and FN tuples was then calculated for each cross-validation experiment. These values were then used to calculate the following performance measures: accuracy (a), error rate (er), precision (p), probability of false alarm (pf), recall (r), specificity ($spec$), type I ($t1$) and type II ($t2$) described in Table 3. NB the proportion of faulty instances (d) is also reported as it is also calculated from the confusion matrix$'$ even though it is not a performance measure. The results are presented in Fig 1[6]. This shows that the performance measures are not the same for all datasets and that there is some variation in the performance measures for an individual dataset, probably due to the resampling caused by carrying out the experiment 10 times and by using SMOTE.

The average values of each performance measure was then calculated over the 10 runs of each experiment.

To test the re-computation of the confusion matrix (confusion matrix$'$) each set of pre-requisite performance measures in Table 5 were taken from the average values obtained in the experiment above. The pre-requisite performance measures were then used by DConfusion to compute confusion matrix$'$ which then allowed us to re-compute the remaining performance measures not in the

[6] Full results can be found in the supporting material at `https://bugcatcher.stca.herts.ac.uk/DConfusion/analysis.html`

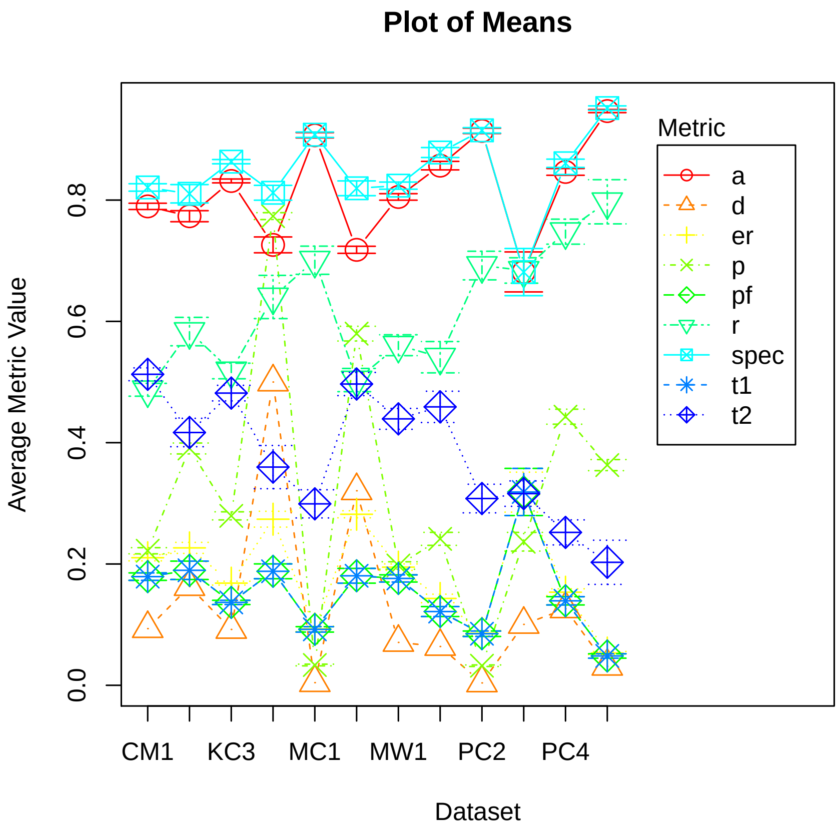


Fig. 1: Performance Measure Values for Different Datasets

set of performance measures used as the input for DConfusion (see Algorithm 2).

4.5 Evaluating the Re-Computed Performance Values

For each dataset/learner combination, we have both the original average performance measure p_i for each performance measure in $\{a, er, p, pf, r, spec, t1, t2\}$ and p'_i, we calculate the difference to get the absolute error $(delta)$. We also divide the delta by p_i to determine the percentage error[7].

$$perr_i = 100 * \frac{p_i - p_i'}{p_i} \tag{17}$$

Table 7 shows that across all datasets, learners and performance measures, the maximum absolute error is 0.003615 and the average average error is -0.0000194. The plot of percentage distribution of the absolute error (Fig 2) also demonstrates that the absolute error is very small with the majority of absolute error values between -0.005 and 0.005. The frequency distribution of the percentage error (Fig 3) shows that the range of percentage error is from -5.0% to 5.0%. The few items where the percentage error is greater than 1.0% from the mean are listed in Table 8. This shows that the very few instances (14 out of 1944) occur for two particular datasets MC1 and PC2 which are the most highly imbalanced datasets. In all cases, p' is always relatively small (< 0.09). Another observation is that MC1 is always associated with Naive Bayes and PC2 is always associated with Random Forest.

Table 7: Distribution of the absolute error ($delta$) for all datasets, learners and performance measures

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
-0.0036	-0.000	0.0000	-0.0000	0.0000	0.0036

[7] Percentage error is not completely correct, as p approaches 1 for many of the performance measures, it becomes increasingly difficult to have a proportionally increasing error, therefore it is to be expected that p values close to 1 will have a small $perr$ value.

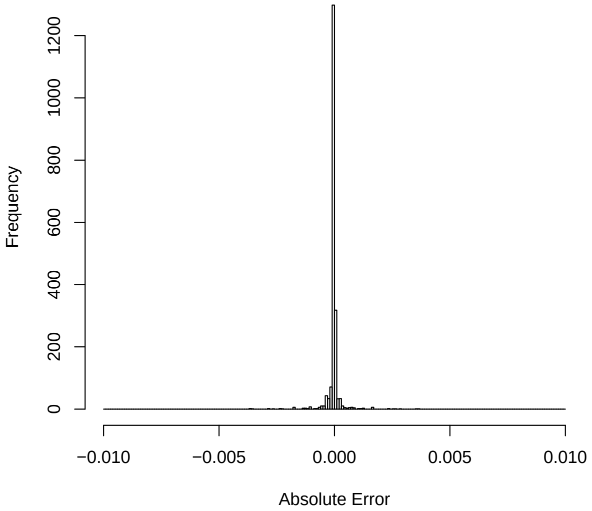


Fig. 2: Frequency Distribution of the Absolute Error (delta) for all Samples.

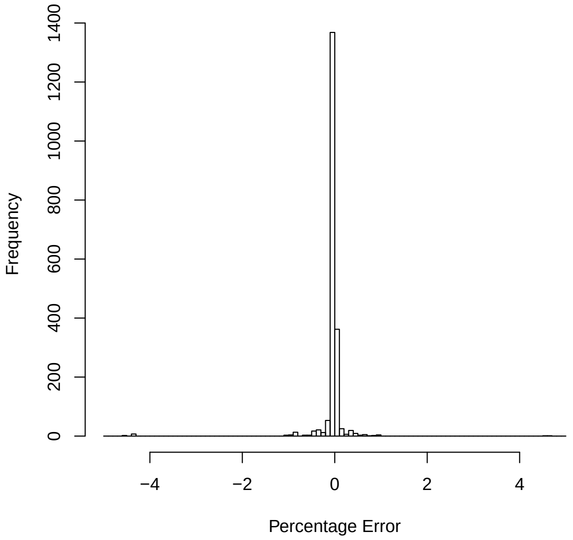


Fig. 3: Percentage Distribution of the Percentage Error for all Samples.

Table 8: Instances where the Percentage Error > 1.0%

Dataset	Learner	Pre-Reqs†	Performance Measure	p	p'	Delta	Percentage Error
MC1	3	3	$t1$	0.0803	0.0767	-0.0036	-4.5011
MC1	3	3	pf	0.0803	0.0767	-0.0036	-4.5011
MC1	3	3	er	0.0820	0.0784	-0.0036	-4.3895
MC1	3	7	p	0.0296	0.0283	-0.0013	-4.3876
MC1	3	8	p	0.0296	0.0283	-0.0013	-4.3876
MC1	3	2	p	0.0296	0.0283	-0.0013	-4.3876
MC1	3	6	p	0.0296	0.0283	-0.0013	-4.3735
MC1	3	4	p	0.0296	0.0283	-0.0013	-4.3735
MC1	3	5	p	0.0296	0.0283	-0.0013	-4.3733
PC2	2	7	p	0.0384	0.0381	-0.0004	-1.0171
PC2	2	8	p	0.0384	0.0381	-0.0004	-1.0171
PC2	2	2	p	0.0384	0.0381	-0.0004	-1.0171
MC1	3	1	d	0.0044	0.0046	0.0002	*4.5911
MC1	3	9	d	0.0044	0.0046	0.0002	*4.6918

* NB the difference in percentage error is due to rounding errors.

† Pre-requisites used to re-compute the confusion matrix′ see Table 5

Keys

Learner: 1=RPART, 2=Random Forest, 3=Naive Bayes

Performance Measure: p=precision,pf=probability of false alarm, $t1$=type I, er=error rate, d=proportion faulty

5 Constructing the Confusion Matrix for some Example Studies

We have transformed the predictive performance data produced by 600 models reported in 42 published studies. A list of these studies is provided in Appendix A. Space restrictions make it is impossible to report the detail for all these transformations. Consequently in this section we present transformations for eight examples. We chose these eight examples to illustrate re-computing the confusion matrix from a range of different original measures.

5.1 Case Studies

Table 9 illustrates the original performance measurement data reported by our eight case study papers. Table 9 shows that a wide range of different measurement data is reported by these eight papers. Given this range it is difficult to evaluate how the performance of these models compares against each other.

We have re-computed the confusion matrix for these eight case studies (shown in Table 10). Based on this confusion matrix data, we have computed the f-measure and MCC data for each case study (also shown in Table 10). It is now possible to comparatively evaluate the predictive performance of these case studies using this common set of data[8].

[8] The aim of this paper is to provide an approach by which others may perform comparative analysis of fault prediction models. A comparative analysis is complex and requires

Table 9: Reported Performance Measurement Data

Study	pd	pf	Error Rate	Type I	Type II	Precision	Recall	Accuracy
1		0.3134	0.3127		0.2826			
2						0.682	0.621	0.641
3	0.471	0.0834						0.8515
4			0.1615	0.1304	0.2830			
5						0.25	0.49	0.805
6						0.717	0.8087	0.7586
7	0.442					0.415		0.918
8		0.4624	0.4442		0.2258			

1. Khoshgoftaar and Seliya (2004)
2. Catal et al. (2007)
3. Kutlubay et al. (2007)
4. Seliya et al. (2005)
5. Ma et al. (2006)
6. Zhou and Leung (2006)
7. Jiang et al. (2008)
8. Yi et al. (2010)

Table 10: Computed Performance Measurement Data

Study	TP_f	TN_f	FP_f	FN_f	f-measure	MCC
1	0.0163	0.6710	0.3063	0.0064	0.0944	0.1288
2	0.3335	0.3075	0.1555	0.2035	0.6501	0.2845
3	0.0575	0.7940	0.0732	0.0646	0.4549	0.3755
4	0.1461	0.6924	0.1038	0.0577	0.6441	0.5457
5	0.7567	0.0483	0.1448	0.0502	0.3311	0.2487
6	0.3762	0.3824	0.1509	0.0905	0.7601	0.5228
7	0.8873	0.0307	0.0433	0.0387	0.4281	0.3842
8	0.4962	0.0596	0.4268	0.0174	0.2114	0.1662

6 Threats to Validity

There are internal and external validity issues that need to be considered when using our approach to re-computing the confusion matrix.

6.1 Internal Validity

Divide by zero problems. Several of our formulas are based on divisions. Where some figures are very similar we encounter divide by zero problems. This division problem is exacerbated by rounding of very small numbers. These small numbers may be very different, but when rounded become the same number. Such numbers suffer from divide by zero issues.

many factors to be taken into account, e.g. the aims of the predictive model. It is beyond the scope of this paper to provide a full comparative analysis of studies against each other.

Data uncertainty. Identifying the balance of faulty and non-faulty units is an important part of our re-computing method. However in a few studies there is inconsistency in the class distribution figures. For example, although an author may have cited a particular class distribution, when we calculate the distribution figure inherent within the results reported (i.e. via the calculation of d), the distribution is different to that stated by the authors. Similarly in some papers where the same data set has been used the distribution varies between experiments. This inconsistency casts some uncertainty over the results in such cases. We suspect that this distribution inconsistency is partly the result of a particular machine learner dealing with the data that it is processing differently to other learners, and partly the result of studies not reporting the data pre-processing that they have applied.

6.2 External Validity

Model tuning. Some models may have been developed to maximise a particular quality (e.g. to reduce false positives). Such models are likely to perform best when their performance is expressed using measures that are sympathetic to the qualities for which the model has been built. Interpreting the performance of such models via alternative performance measures should be treated with caution.

7 Discussion

The process of translating the performance measures reported by studies to the confusion matrix reveals a variety of performance issues with studies that we now discuss.

7.1 Erroneous Results

In some cases our translation to the confusion matrix demonstrated that the original results reported by some studies could not have been possible. For example we found an error in Wang and Li (2010). This error was revealed as our transformations would not work correctly. As a result of this we emailed the authors to clarify the problem. The authors confirmed that a typographical error had crept into their final draft. False Alarms were reported instead of False Positives. It is easy for such errors to creep into published work, especially in an area as complex as fault prediction. Without very careful interpretation such errors can easily be missed and be misleading.

7.2 Definitions of Measures

While performing our transformations we have had difficulty in making sense of the figures reported in some studies. The reason for this was that a number of studies have used non-standard definitions for some well-known performance measures (e.g. Zhou and Leung (2006) does not use a standard definition of Precision and Pai and Dugan (2007) does not use the standard definitions of Sensitivity and Specificity (in both cases, the issues were confirmed by emailing the authors)). Although the definitions used were given in the paper, it is difficult for a reader to pick-up on the nuances of measurement definitions (usually provided via formulae). Consequent mis-understanding could have serious implications for subsequent model users.

7.3 Reporting Performance Based on Predicting Non-Faulty Units

Some papers have reported performance measures based on predicting the majority (non-faulty class) rather than the minority (faulty) class. In some of these cases it is also not made clear that the predictive performance is on the majority class. These issues can be very misleading when trying to evaluate predictive performance. For example, Elish and Elish (2008) has been a very influential fault prediction study using Support Vector Machines (SVM). Their study has been cited more than 60 times and is considered a pivotal paper in the use of SVMs. Table 11 shows the very good Accuracy, Precision and Recall performances reported by Elish and Elish for SVM using datasets cm1, pc1, kc1 and kc3 (taken from Elish and Elish (2008)).

Table 11: Accuracy, Precision and Recall Elish and Elish (2008)

Dataset	Accuracy	Precision	Recall
cm1	0.9069	0.9066	1.0000
pc1	0.9310	0.9353	0.9947
kc1	0.8459	0.8495	0.9940
kc3	0.9328	0.9365	0.9958

Our process to re-compute the confusion matrix would not work on these figures when we assumed that the values for Precision and Recall were based on the non-faulty class. Tables 12 and 13 show our workings for this re-computation. Our workings suggest that Elish and Elish have reported the performance of their SVM models based on predicting non-faulty units rather than faulty units, this was confirmed by e-mailing the authors. Since the vast majority of units in data sets are non-faulty (ranging between 84.6% and 93.7% in their case), predicting the majority class is very easy and so high performance figures are to be expected. Such models are not useful. Our findings are complementary to those of several other authors who report problems reproducing the high predictive performances reported by Elish and Elish when using their SVM settings. For example Arisholm et al. (2010) reports that most papers report a far lower Recall value. de Carvalho et al. (2008) and de Carvalho et al. (2010) used the same SVM settings. de Carvalho et al. (2008)

reported Specificity and Sensitivity values and de Carvalho et al. (2010) reported Precision and Recall for both the faulty and non-faulty classes which are similar to our re-computed values.

Table 12: Frequency of the Class Identified as "True" and the Frequency of the Faulty class

Dataset	Computed d	1 - Computed d	Reported Fault Frequency
cm1	0.9037	0.0963	0.097
pc1	0.9311	0.0689	0.069
kc1	0.8462	0.1538	0.154
kc3	0.9370	0.0630	0.063

Table 13: SVM Confusion Matrix of the Majority Class

Dataset	TP_f	FN_f	FP_f	TN_f
cm1	0.9037	0.0000	0.0931	0.0032
pc1	0.9261	0.0049	0.0641	0.0049
kc1	0.8412	0.0047	0.1490	0.0051
kc3	0.9330	0.0039	0.0633	*-0.0002

* demonstrates that rounding errors occur.

Using our technique it is possible to calculate the Precision and Recall of the faulty units in Elish and Elish's study. Table 14 shows the results of this calculation. Table 14 suggests that the performance of the SVMs in the Elish and Elish study is much less positive. Table 14 shows that f-measure ranges from 0.0 to 0.12. This is compared to their original maximum f-measure of 0.96.

Table 14: Performance Measures for the Faulty Class using the Values from Table 13

Dataset	Accuracy	Precision	Recall	f-measure
cm1	0.9069	1.0000	0.0332	0.0643
pc1	0.9310	0.5000	0.0710	0.1244
kc1	0.8463	0.5204	0.0331	0.0622
kc3	0.9328	-0.0541	-0.0032	-0.0060

8 Empirical SE, V2.0

Re-computing the confusion matrix is a stepping stone to allowing the results of fault prediction studies to be compared and benchmarked. We propose that empirical fault prediction studies v2.0 should provide finer grained results including the prediction for every instance in a dataset for every run. The model for each run should also be stored so that future research can investigate the 'decisions' that the models are making. Reporting the full set of results and models would make the experimental results more transparent and therefore less likely to produce misleading conclusions.

We recognise that a tremendous amount has been achieved by Tim Menzies et al and the PROMISE repository[9] which has provided the data for performing many fault prediction studies. We now suggest that the PROMISE repository is extended for empirical studies v2.0 to include the fine grained results and models from fault prediction studies. This would allow both categorical and continuous studies to be reported and compared which would bring together a larger body of knowledge. Defining how the data and the models should be recorded is an open question, but making it future proof will be the greatest challenge.

9 Conclusion

The predictive performance of published fault prediction models is expressed using a variety of different performance measures. This makes it difficult to compare the performance of published prediction models. We have presented an approach that enables the accurate re-computation of the confusion matrix for studies originally reporting a variety of performance measures. From the confusion matrix a range of other performance measures can be calculated with a good degree of accuracy. Expressing the performance of fault prediction models using a consistent set of measures allows comparative analysis. Our approach has several advantages, including that it:

- allows comparative analysis of a set of fault prediction models in terms of a preferred predictive quality.
- makes meta-analysis possible across the many fault prediction studies published.
- enables the validation of the performance figures reported in published studies.

The advantages of our approach have benefits for fault prediction researchers, practitioners and reviewers. Researchers can use our approach to evaluate predictive performance across sets of models and perform meta-analysis of these models. An evidence base of fault prediction can be built by researchers that will enable more informed future model building research. Practitioners can express model performance to reflect the qualities that they are interested

[9] Now based at `https://code.google.com/p/promisedata/`

in, for example practitioners wanting a model that values finding as many faults as possible might might predominately focus on Recall. Practitioners are then in a more informed position to select a model that is appropriate for their development context. Reviewers of fault prediction studies can use our process as a relatively easy way to check that no errors have crept into fault prediction studies. Without our 'ready reckoner' checking performance figures in studies submitted for review is difficult. Model builders could themselves use our process as a 'ready reckoner' to check their own figures are correct. Model builders and reviewers doing this checking could improve the quality of the fault prediction work that is published.

Overall the approach that we present could significantly improve the quality of fault prediction studies and enable meta-analysis and bechmarking across studies. Achieving this is very important as it will help this research area to mature and grow. Such maturation could ultimately expand the industrial uptake of fault prediction modelling.

References

Arisholm E, Briand LC, Fuglerud M (2007) Data mining techniques for building fault-proneness models in telecom java software. In: Software Reliability, 2007. ISSRE '07. The 18th IEEE International Symposium on, pp 215 –224

Arisholm E, Briand LC, Johannessen EB (2010) A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. Journal of Systems and Software 83(1):2–17

Baldi P, Brunak S, Chauvin Y, Andersen C, Nielsen H (2000) Assessing the accuracy of prediction algorithms for classification: an overview. Bioinformatics 16(5):412–424

Batista G, Prati R, Monard M (2004) A study of the behavior of several methods for balancing machine learning training data. ACM SIGKDD Explorations Newsletter 6(1):20–29

Bowes D, Hall T, Gray D (2012) Comparing the performance of fault prediction models which report multiple performance measures: recomputing the confusion matrix. In: Proceedings of the 8th International Conference on Predictive Models in Software Engineering, ACM, New York, NY, USA, PROMISE '12, pp 109–118

de Carvalho AB, Pozo A, Vergilio S, Lenz A (2008) Predicting fault proneness of classes trough a multiobjective particle swarm optimization algorithm. In: Tools with Artificial Intelligence, 2008. ICTAI '08. 20th IEEE International Conference on, vol 2, pp 387–394

de Carvalho AB, Pozo A, Vergilio SR (2010) A symbolic fault-prediction model based on multiobjective particle swarm optimization. Journal of Systems and Software 83(5):868–882

Catal C, Diri B, Ozumut B (2007) An artificial immune system approach for fault prediction in object-oriented software. In: Dependability of Computer

Systems, 2007. DepCoS-RELCOMEX '07. 2nd International Conference on, pp 238 –245

Chawla N, Bowyer K, Hall L, Kegelmeyer W (2002) Smote: synthetic minority over-sampling technique. Journal of Artificial Intelligence Research 16(1):321–357

Chawla NV, Japkowicz N, Kotcz A (2004) Editorial: special issue on learning from imbalanced data sets. SIGKDD Explorations 6(1):1–6

Cruzes DS, Dybå T (2011) Research synthesis in software engineering: A tertiary study. Inf Softw Technol 53:440–455

Denaro G, Pezzè M (2002) An empirical evaluation of fault-proneness models. In: Proceedings of the 24th International Conference on Software Engineering, ACM, New York, NY, USA, ICSE '02, pp 241–251

Elish K, Elish M (2008) Predicting defect-prone software modules using support vector machines. Journal of Systems and Software 81(5):649–660

Forman G, Scholz M (2010) Apples-to-apples in cross-validation studies: pitfalls in classifier performance measurement. SIGKDD Explor Newsl 12(1):49–57

Gray D, Bowes D, Davey N, Sun Y, Christianson B (2011) Further thoughts on precision. In: Evaluation and Assessment in Software Engineering (EASE)

Gray D, Bowes D, Davey N, Sun Y, Christianson B (2012) Reflections on the nasa mdp data sets. Software, IET 6(6):549 –558

Hall T, Bowes D (2012) The state of machine learning methodology in software fault prediction. In: International Conference on Machine Learning and Applications (ICMLA)

Hall T, Beecham S, Bowes D, Gray D, Counsell S (2012) A systematic literature review on fault prediction performance in software engineering. Software Engineering, IEEE Transactions on 38(6):1276 –1304

He H, Garcia E (2008) Learning from imbalanced data. IEEE Transactions on Knowledge and Data Engineering pp 1263–1284

Jiang Y, Cukic B, Ma Y (2008) Techniques for evaluating fault prediction models. Empirical Software Engineering 13(5):561–595

Kamei Y, Monden A, Matsumoto S, Kakimoto T, Matsumoto K (2007) The effects of over and under sampling on fault-prone module detection. In: Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on, pp 196 –204

Kaur A, Sandhu PS, Bra AS (2009) Early software fault prediction using real time defect data. In: Machine Vision, 2009. ICMV '09. Second International Conference on, accept, pp 242–245

Khoshgoftaar T, Seliya N (2004) Comparative assessment of software quality classification techniques: An empirical case study. Empirical Software Engineering 9(3):229–257

Koru A, Liu H (2005) Building effective defect-prediction models in practice. Software, IEEE 22(6):23 – 29

Kutlubay O, Turhan B, Bener A (2007) A two-step model for defect density estimation. In: Software Engineering and Advanced Applications, 2007. 33rd EUROMICRO Conference on, pp 322 –332

Lessmann S, Baesens B, Mues C, Pietsch S (2008) Benchmarking classification models for software defect prediction: A proposed framework and novel findings. Software Engineering, IEEE Transactions on 34(4):485 –496

Ma Y, Guo L, Cukic B (2006) Advances in Machine Learning Applications in Software Engineering, IGI Global, chap A statistical framework for the prediction of fault-proneness, pp 237–265

Mende T, Koschke R (2010) Effort-aware defect prediction models. In: Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on, pp 107–116

Menzies T, Dekhtyar A, Distefano J, Greenwald J (2007a) Problems with precision: A response to "comments on 'data mining static code attributes to learn defect predictors'". Software Engineering, IEEE Transactions on 33(9):637 –640

Menzies T, Greenwald J, Frank A (2007b) Data mining static code attributes to learn defect predictors. Software Engineering, IEEE Transactions on 33(1):2 –13

Fenton N, Neil M (1999) A critique of software defect prediction models. Software Engineering, IEEE Transactions on 25(5):675–689

Ostrand T, Weyuker E, Bell R (2004) Where the bugs are. In: ACM SIGSOFT Software Engineering Notes, ACM, vol 29, pp 86–96

Ostrand T, Weyuker E (2007) How to measure success of fault prediction models. In: Fourth international workshop on Software quality assurance: in conjunction with the 6th ESEC/FSE joint meeting, ACM, pp 25–30

Pai G, Dugan J (2007) Empirical analysis of software fault content and fault proneness using bayesian methods. Software Engineering, IEEE Transactions on 33(10):675 –686

Pizzi N, Summers A, Pedrycz W (2002) Software quality prediction using median-adjusted class labels. In: Neural Networks, 2002. IJCNN '02. Proceedings of the 2002 International Joint Conference on, vol 3, pp 2405 –2409

Seliya N, Khoshgoftaar T, Zhong S (2005) Analyzing software quality with limited fault-proneness defect data. In: High-Assurance Systems Engineering, 2005. HASE 2005. Ninth IEEE International Symposium on, pp 89 –98

Sun Y, Castellano C, Robinson M, Adams R, Rust A, Davey N (2009) Using pre & post-processing methods to improve binding site predictions. Pattern Recognition 42(9):1949–1958

Turhan B, Kocak G, Bener A (2009) Data mining source code for locating software bugs: A case study in telecommunication industry. Expert Systems with Applications 36(6):9986–9990

Wang T, Li Wh (2010) Naive bayes software defect prediction model. In: Computational Intelligence and Software Engineering (CiSE), 2010 International Conference on, accept, pp 1–4

Yi L, Khoshgoftaar TM, Seliya N (2010) Evolutionary optimization of software quality modelling with multiple repositories. Software Engineering, IEEE Transactions on 36(6):852–864

Zeller A, Zimmermann T, Bird C (2011) Failure is a four-letter word: a parody in empirical research. In: Proceedings of the 7th International Conference on Predictive Models in Software Engineering, ACM, New York, NY, USA, Promise ’11, pp 5:1–5:7

Zhang H, Zhang X (2007) Comments on ”data mining static code attributes to learn defect predictors”. Software Engineering, IEEE Transactions on 33(9):635 –637

Zhou Y, Leung H (2006) Empirical analysis of object-oriented design metrics for predicting high and low severity faults. Software Engineering, IEEE Transactions on 32(10):771 –789

A List of Papers from which we have Re-Computed Confusion Matrices.

E. Arisholm, L. C. Briand, and M. Fuglerud. Data mining techniques for building fault-proneness models in telecom java software. In *ISSRE ’07. The 18th IEEE Intern Symp on*, pages 215 –224, 2007.

E. Arisholm, L. C. Briand, and E. B. Johannessen. A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. *Journal of Systems and Software*, 83(1):2–17, 2010.

C. Bird, N. Nagappan, H. Gall, B. Murphy, and P. Devanbu. Putting it all together: Using socio-technical networks to predict failures. In *20th International Symposium on Software Reliability Engineering*, pages 109–119. IEEE, 2009.

L. Briand, W. Melo, and J. Wust. Assessing the applicability of fault-proneness models across object-oriented software projects. *Software Engineering, IEEE Transactions on*, 28(7):706 – 720, 2002.

B. Caglayan, A. Bener, and S. Koch. Merits of using repository metrics in defect prediction for open source projects. In *FLOSS ’09. ICSE Workshop on*, pages 31–36, 2009.

G. Calikli, A. Tosun, A. Bener, and M. Celik. The effect of granularity level on software defect prediction. In *Computer and Information Sciences, 2009. ISCIS 2009. 24th International Symposium on*, pages 531 –536, 2009.

C. Catal, B. Diri, and B. Ozumut. An artificial immune system approach for fault prediction in object-oriented software. In *Dependability of Computer Systems, 2007. DepCoS-RELCOMEX ’07. 2nd International Conference on*, pages 238 –245, 2007.

C. Cruz and A. Erika. Exploratory study of a uml metric for fault prediction. In *Proceedings of the 32nd ACM/IEEE Intern Conf on Software Engineering*, pages 361–364. 2010.

A. B. de Carvalho, A. Pozo, S. Vergilio, and A. Lenz. Predicting fault proneness of classes trough a multiobjective particle swarm optimization algorithm. In *Tools with Artificial Intelligence, 2008. ICTAI ’08. 20th IEEE International Conference on*, volume 2, pages 387–394, 2008.

A. B. de Carvalho, A. Pozo, and S. R. Vergilio. A symbolic fault-prediction model based on multiobjective particle swarm optimization. *J of Sys & Soft*, 83(5):868–882, 2010.

G. Denaro and M. Pezzè. An empirical evaluation of fault-proneness models. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE ’02, pages 241–251, NY, USA, 2002. ACM.

L. Guo, Y. Ma, B. Cukic, and H. Singh. Robust prediction of fault-proneness by random forests. In *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on*, pages 417 – 428, 2004.

T. Gyimothy, R. Ferenc, and I. Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *Software Engineering, IEEE Transactions on*, 31(10):897 – 910, 2005.

Z. Hongyu. An investigation of the relationships between lines of code and defects. In *Software Maintenance, 2009. IEEE Intern Conf on*, pages 274–283, 2009.

Y. Jiang, B. Cukic, and Y. Ma. Techniques for evaluating fault prediction models. *Empirical Software Engineering*, 13(5):561–595, 2008.

S. Kanmani, V. Uthariaraj, V. Sankaranarayanan, and P. Thambidurai. Object-oriented software fault prediction using neural networks. *Information and Software Technology*, 49(5):483–492, 2007.

A. Kaur and R. Malhotra. Application of random forest in predicting fault-prone classes. In *Advanced Computer Theory and Engineering, 2008, Internl Conf on*, 37–43, 2008.

A. Kaur, P. S. Sandhu, and A. S. Bra. Early software fault prediction using real time defect data. In *Machine Vision, 2009. Intern Conf on*, pages 242–245.

T. Khoshgoftaar and N. Seliya. Comparative assessment of software quality classification techniques: An empirical case study. *Empirical Software Engineering*, 9(3):229–257, 2004.

T. Khoshgoftaar, X. Yuan, E. Allen, W. Jones, and J. Hudepohl. Uncertain classification of fault-prone software modules. *Empirical Software Engineering*, 7(4):297–318, 2002.

A. Koru and H. Liu. Building effective defect-prediction models in practice. *Software, IEEE*, 22(6):23 – 29, 2005.

O. Kutlubay, B. Turhan, and A. Bener. A two-step model for defect density estimation. In *Software Engineering and Advanced Applications, 2007. 33rd EUROMICRO Conference on*, pages 322 –332, 2007.

Y. Ma, L. Guo, and B. Cukic. *Advances in Machine Learning Applications in Software Engineering*, chapter A statistical framework for the prediction of fault-proneness, pages 237–265. IGI Global, 2006.

T. Mende and R. Koschke. Effort-aware defect prediction models. In *Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on*, pages 107–116, 2010.

T. Menzies, J. Greenwald, and A. Frank. Data mining static code attributes to learn defect predictors. *Software Engineering, IEEE Transactions on*, 33(1):2 –13, 2007.

O. Mizuno, S. Ikami, S. Nakaichi, and T. Kikuno. Spam filter based approach for finding fault-prone software modules. In *Mining Software Repositories, 2007. ICSE '07. International Workshop on*, page 4, 2007.

O. Mizuno and T. Kikuno. Training on errors experiment to detect fault-prone software modules by spam filter. In *Procs European Software Engineering Conf and the ACM SIGSOFT symp on The foundations of software engineering*, ESEC-FSE '07, pages 405–414, 2007. ACM.

R. Moser, W. Pedrycz, and G. Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Software Engineering, 2008. ICSE '08. ACM/IEEE 30th Intern Conf on*, pages 181–190.

N. Nagappan, A. Zeller, T. Zimmermann, K. Herzig, and B. Murphy. Change bursts as defect predictors. In *Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on*, pages 309–318.

A. Nugroho, M. R. V. Chaudron, and E. Arisholm. Assessing uml design metrics for predicting fault-prone classes in a java system. In *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, pages 21–30.

G. Pai and J. Dugan. Empirical analysis of software fault content and fault proneness using bayesian methods. *Software Engineering, IEEE Trans on*, 33(10):675–686, 2007.

A. Schröter, T. Zimmermann, and A. Zeller. Predicting component failures at design time. In *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, pages 18–27. ACM, 2006.

N. Seliya, T. Khoshgoftaar, and S. Zhong. Analyzing software quality with limited fault-proneness defect data. In *High-Assurance Systems Engineering, 2005. IEEE Internl Symp on*, pages 89 –98, 2005.

S. Shivaji, E. J. Whitehead, R. Akella, and K. Sunghun. Reducing features to improve bug prediction. In *Automated Software Engineering, 2009. ASE '09. 24th IEEE/ACM International Conference on*, pages 600–604.

Y. Singh, A. Kaur, and R. Malhotra. Predicting software fault proneness model using neural network. *Product-Focused Software Process Improvement*, 5089:204–214, 2008.

A. Tosun and A. Bener. Reducing false alarms in software defect prediction by decision threshold optimization. In *Empirical Software Engineering and Measurement, ESEM 2009. International Symposium on*, pages 477–480.

B. Turhan and A. Bener. A multivariate analysis of static code attributes for defect prediction. In *Quality Software, 2007. Intern Conf on*, pages 231 –237, 2007.

O. Vandecruys, D. Martens, B. Baesens, C. Mues, M. De Backer, and R. Haesen. Mining software repositories for comprehensible software fault prediction models. *Journal of Systems and Software*, 81(5):823–839, 2008.

R. Vivanco, Y. Kamei, A. Monden, K. Matsumoto, and D. Jin. Using search-based metric selection and oversampling to predict fault prone modules. In *Electrical and Computer Engineering, 2010, Canadian Conf on*, pages 1–6.

L. Yi, T. M. Khoshgoftaar, and N. Seliya. Evolutionary optimization of software quality modeling with multiple repositories. *Soft Engin, IEEE Trans on*, 36(6):852–864, 2010.

Y. Zhou and H. Leung. Empirical analysis of object-oriented design metrics for predicting high and low severity faults. *Software Engineering, IEEE Trans on*, 32(10):771–789, 2006.

T. Zimmermann, R. Premraj, and A. Zeller. Predicting defects for eclipse. In *Predictor Models in Software Engineering, 2007. PROMISE'07*, page 9, 2007.

B Fault Prediction Experiment Algorithm

```
Data: n=10 the number of folds
Data: m=10 the number of times to repeat the experiment
Data: dsets<- c(”PC5”, ”KC1”,”CM1”,”KC3”,”KC4”,”MC1”,
      ”MC2”,”MW1”,”PC1”,”PC2”,”PC3”,”PC4”)
Data: learners<- c(rpart,rf,nb)
for runid in 1:m do
    for each dataset in dsets do
        create n stratified folds;
        for each fold in 1:n do
            testing<-fold$_n$;
            training<-dataset-fold$_n$;
            training$'$ <-smote(training);
            randomise(training$'$);
            tp[fold]<-c(training$'$,testing);
        end
        for each learner in learners do
            for each fold in 1:n do
                tuneLearner on tp[fold].training$'$;
                model<-trainLearner on tp[fold].training$'$;
                prediction<-evaluate model on tp[fold].testing;
                result$_{runid,dataset,learner,fold}$ <-prediction;
            end
            result$_{runid,dataset,learner}$ <-aggregate(result$_{runid,dataset,learner,fold}$);
        end
    end
end
```

Algorithm 1: Generating the prediction data using a 10 x 10-fold stratified cross-validation experiment

C Re-Computing the Confusion Matrix′ Algorithm

```
Data: dsets<- c(”PC5”, ”KC1”,”CM1”,”KC3”,”KC4”,”MC1”,
        ”MC2”,”MW1”,”PC1”,”PC2”,”PC3”,”PC4”)
Data: learners<- c(rpart,rf,nb)
Data: performance-measures<- c($p, r, a, pf, d, t1, t2, er, spec$)
Data: pre-requisiteperformance-measures<-
        c({$p, r, pf$},{$r, a, spec$},{$p, r, a$}..{$t1, t2, er$})
Data: m=10 the number of times the experiment was repeated
for each pre-requisite-set in pre-requisite-performance-measures do
    eval-performance-measures¡-performance-measures - pre-requisite-set;
    for each dataset in dsets do
        for each learner in learners do
            for runid in 1:m do
                compute TP,TN,FP,FN;
            end
            for each performance-measure in pre-requisite-set do
                compute mean(performance-measure) over
                result$_{dataset,learner,runid=1:m}$;
                if performance-measure in pre-requisite-set then
                    compute stddev(performance-measure) over
                    result$_{dataset,learner,runid=1:m}$;
                end
            end
            using mean performance-measures re-compute confusion-matrix′ ;
            using confusion-matrix′ re-compute eval-performance-measures′;
            compare mean of eval-performance-measures against
            eval-performance-measures′;
        end
    end
end
```

Algorithm 2: Evaluating the conversion from performance measure to a confusion matrix and then to other performance measures