IMPROVING DEFECT PREDICTION MODELS BY COMBINING CLASSIFIERS PREDICTING DIFFERENT DEFECTS

JEAN PETRIĆ

School of Computer Science
University of Hertfordshire

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

June 2018

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements. This dissertation contains fewer than 80,000 words including appendices, bibliography, footnotes, tables and equations and has fewer than 150 figures.

JEAN PETRIĆ
June 2018

Acknowledgements

My first massive thank you goes to my parents and my brother. I'm very aware I wasn't the easiest kid to deal with, so thank you for your perpetual support and care during those days and today. Thank you for supporting me in my education. One special thanks to my brother here, who was always pushing me with my maths homework. It's been a long and enjoyable journey for me, but I hope you're as proud and happy as I am.

My second enormous thank you goes to David Bowes. It's been a great pleasure having you as a mentor, something you taught me, extends for life. I'm very grateful that you're this person. Thank you for all the kind deeds that have always made me feel like at home. Thank you for sharing your knowledge and advice with me. Thank you for believing in me!

I must extend my gratitude to the rest of my supervision team, Tracy Hall, Nathan Baddoo and Bruce Christianson. Working with Tracy has been a pleasantly rewarding experience filled with rigour and hard-work. Nathan, who was always there to help, and who spent numerous hours making my work stronger. Bruce, one of the wisest men I know, who diligently went through all my work and provided invaluable advice. It's difficult to imagine that any better *ensemble* of supervisors could exist. I was remarkably fortunate to have you on my supervision team!

My former lab mates Ankur and Alex also deserve one huge thank you. It was a pleasure sharing the same space and time with you for the last couple of years. I hope you enjoyed our conversations as much as I did. Big thanks also go to Kheng Lee and Giuseppe who made my last several months super enjoyable. Having you guys as friends is an additional reward in itself.

And last, but foremost, I want to dedicate this dissertation to my cousin Santos Vrbat, who inspired me for life.

Abstract

Background: The software industry spends a lot of money on finding and fixing defects. It utilises software defect prediction models to identify code that is likely to be defective. Prediction models have, however, reached a performance bottleneck. Any improvements to prediction models would likely yield less defects-reducing costs for companies.

Aim: In this dissertation I demonstrate that different families of classifiers find distinct subsets of defects. I show how this finding can be utilised to design ensemble models which outperform other state-of-the-art software defect prediction models.

Method: This dissertation is supported by published work. In the first paper I explore the quality of data which is a prerequisite for building reliable software defect prediction models. The second and third papers explore the ability of different software defect prediction models to find distinct subsets of defects. The fourth paper explores how software defect prediction models can be improved by combining a collection of classifiers that predict different defective components into ensembles. An additional, non-published work, presents a visual technique for the analysis of predictions made by individual classifiers and discusses some possible constraints for classifiers used in software defect prediction.

Result: Software defect prediction models created by classifiers of different families predict distinct subsets of defects. Ensembles composed of classifiers belonging to different families outperform other ensemble and standalone models. Only a few highly diverse and accurate base models are needed to compose an effective ensemble. This ensemble can consistently predict a greater number of defects compared to the increase in incorrect predictions.

Conclusion: Ensembles should not use the majority-voting techniques to combine decisions of classifiers in software defect prediction as this will miss correct predictions of classifiers which uniquely identify defects. Some classifiers could be less successful for software defect prediction due to complex decision boundaries of defect data. Stacking based ensembles can outperform other ensemble and stand-alone techniques. I propose new possible avenues of research that could further improve the modelling of ensembles in software defect prediction. Data quality should be explicitly considered prior to experiments for researchers to establish reliable results.

Table of contents

List of Published Papers

Paper 1. Petrić J, Bowes D, Hall T, Christianson B, Baddoo N. The jinx on the NASA software defect data sets. In Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering 2016 Jun 1 (p. 13). ACM.

Paper 2. Bowes D, Hall T, Petrić J. Different classifiers find different defects although with different level of consistency. In Proceedings of the 11th International Conference on Predictive Models and Data Analytics in Software Engineering 2015 Oct 21 (p. 3). ACM.

Paper 3. Bowes D, Hall T, Petrić J. Software defect prediction: do different classifiers find the same defects?. Software Quality Journal. 2017:1-28.

Paper 4. Petrić J, Bowes D, Hall T, Christianson B, Baddoo N. Building an ensemble for software defect prediction based on diversity selection. InProceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement 2016 Sep 8 (p. 46). ACM.

List of figures

List of tables

Chapter 1

Introduction

A software defect is an imperfection or deficiency in a work product where that work product does not meet its requirements or specifications [IEEE Std 2009]. Traditionally, software functionality has been manually or automatically validated and defects fixed upon discovery. However, with the ever-growing size and complexity of software systems, manual inspections have become infeasible, whilst automatic testing is limited when resources are scarce.

The aim of software defect prediction modelling is to identify software units likely to be defective. Software defect prediction models are cheaper to produce and evaluate than software testing, because they require minimal human effort. Predictions are made based on software metrics that are typically gathered from the system source code or the development process. The models can help practitioners save costs [Tassey 2002] by focusing their efforts on code units predicted to be defect-prone. In the last three decades many software defect prediction models have been developed [Catal and Diri 2009, Hall et al. 2012, Malhotra 2015], yet only few have been used in industry (e.g. Zimmermann et al. [2009]). Some of the likely reasons for the limited use of software defect prediction models in industry are their low rate of true and high rate of false positive predictions.

A systematic literature review by Hall et al. [2012] synthesised software defect prediction models between 2000 and 2010. They reported that some simple modelling techniques (i.e. Naïve Bayes and Logistic Regression) perform well compared to more complex techniques, such as Support Vector Machines. Lessmann et al. [2008a] compared 22 classifiers on ten public datasets and reported no significant difference amongst the top 17 classifiers. More recent studies have also shown no clear winner amongst classifiers used for software defect prediction [Elish 2014, Malhotra 2015].

To mitigate the performance bottleneck, researchers have applied more advanced machine learning techniques to software defect prediction. Recent literature reviews by Malhotra [2015] and Wahono [2015] reported an increase in the use of ensemble techniques in software

defect prediction. Their results suggest that ensembles can improve performances of software defect prediction models. Mısırlı et al. [2011a] demonstrated a decrease in false positives when applying ensembles compared to single models. Chen et al. [2018] successfully demonstrated the effectiveness of ensemble models on nine highly-imbalanced defect datasets. Despite the positive outcomes of ensembles used in software defect prediction, it is likely that performance can be further improved.

The studies by Elish [2014], Lessmann et al. [2008a], Malhotra [2015] suggest that the choice of classifier for software defect prediction is in most cases irrelevant. If all defects were the same, the choice of classifier would not matter. However, defects differ in their nature. Some defects are configuration related [Xia et al. 2013], install-ability/requirements/usability issues [Hernandez-Gonzalez et al. 2018], whilst others link to structural problems [Petrić and Grbac 2014]. As defects differ in their nature, an investigation of the success of different machine learning techniques to predict distinct defective components would be worthwhile. Some classifiers could be more successful than others in finding particular defects.

Panichella et al. [2014] were amongst the first to consider the ability of different classifiers to predict distinct defective components. They established that different classifier types, which identify different subsets of defects, have the ability to complement each other when combined. In a cross-project defect prediction set-up[1], Panichella et al. [2014] proposed Combined Defect Predictor (CODEP) where predictions from the first layer of classifiers are used as an input to the second (meta) layer classifier[2]. Their results demonstrated that the CODEP approach can outperform stand-alone classifiers. No work, however, has explored in detail how ensembles for software defect prediction should be constructed so that they can identify the defects predicted by different types of classifiers.

In this work I analyse ensemble designs that have the potential to improve the performance of software defect prediction models. The aim is to establish whether models created by different classifiers can find different defective components, and if they do, what is a suitable way to combine them in order to obtain models that outperform state-of-the-art techniques. I also develop visual techniques to better understand the constraints of classifiers that constitute the ensemble. I ensure the models used in this dissertation are trained on high-quality datasets and use state-of-the-art approaches.

[1] Cross-project defect prediction uses several different software projects when building prediction models and validates the model on the project of interest, which is different from my work which considers within-project defect prediction. Within-project defect prediction uses the same project to build and validate prediction models.

[2] This technique is more commonly known as stacking in the machine learning context.

1.1 Thesis Statement

I show that software defect prediction models created by classifiers from different machine learning families can predict distinct subsets of defects. This property can be used to construct composite models that detect more defects than state-of-the-art models. I show that it matters how the decisions of composite models are combined. The popular majority-voting technique may not be the most appropriate as it may miss defects predicted by individual classifiers. Stacking ensembles can be used instead. Combined classifiers need to be diverse as this increases their ability to find more defects. Stacking ensembles need not be composed of many classifiers. Three accurate and diverse classifiers are sufficient to obtain acceptable results. I visualise the individual predictions of different classifiers and discuss potential characteristics that could limit certain classifiers from discovering more defects.

1.2 Research Questions

This dissertation aims to answer two research questions. The first research question focuses on the ability of classifiers to identify different defective components. The second research question, which is broken into three parts, focuses on modelling techniques whose aim is to improve software defect prediction models. More details about the research questions are given below.

RQ1. Do models created by different classifiers find different defective components?

Motivation. In a large experimental analysis Lessmann et al. [2008a] established no significant difference amongst the top 17 best performing classifiers. Similar performances of various defect models suggest that the choice of the modelling technique is irrelevant. However, by their nature, defects are different. As the underlying mechanisms of prediction models differ between models, it is likely that some models have better chance of picking up diverse subsets of defects that others cannot. An empirical analysis is needed to establish whether models created by different classifiers find distinct subsets of defects.

RQ2. How can software defect prediction models be improved by combining classifiers predicting different defective components?

RQ2a. Can stacking ensembles based on explicit diversity improve prediction performance compared to other software defect prediction models?

Motivation. Diversity and accuracy play key parts in the success of ensemble learners in various scientific disciplines. This question investigates whether an ensemble for which

diversity and accuracy are explicitly considered improves prediction performance when compared to other ensembles and standalone techniques.

RQ2b. How many classifiers combined into stacking ensembles are needed to provide good software defect prediction models?

Motivation. Combining multiple classifiers increases the training time of ensemble models. Very large ensembles would be slower to train compared to their smaller counterparts. This question analyses what would be an ideal size of ensemble (i.e. the size after which adding new classifiers would not improve the performance).

RQ2c. How much diversity and which base classifiers need to be combined to provide good ensemble models?

Motivation. If different families of classifiers find different defective components, it should matter which type of classifiers are combined in the ensemble. I look at the frequency of the type of classifiers typically selected into ensembles.

1.3 Contributions

I make the following contributions to knowledge in this dissertation:

Practical Contributions – I find that although different classifiers achieve similar prediction performances, they find distinct subsets of defects. I propose an ensemble model which is composed of classifiers that complement each other and show that this ensemble outperforms other defect predictors. A consistent increase in correctly predicted defects compared to the increase in false positives is observed across datasets using the proposed ensemble. Companies whose priority is to comprehensively detect defects would benefit from this ensemble model. I also develop dynamic visual techniques which provide the possibility for analysing individual predictions made by different classifiers. These visual techniques suggest that some classifiers achieve poor prediction performances but find defects with unusual code characteristics, whilst others find more defects for the price of increasing false positive predictions. Future researchers can use these visualisation techniques to improve understanding of the advantages and limitations of classifiers for predicting distinct subsets of defects.

Methodological Contributions – I show that a successful ensemble model needs only a few diverse and accurate classifiers to perform well. The diversity of those classifiers should come by selecting from different classifier families. Ensembles should not use the majority-voting technique for decision making. Majority-voting misses those defects correctly predicted by individual classifiers.

Data Contributions – My final major contribution is about data quality. I propose five novel integrity constraints for cleaning defect datasets which I add to the existing state-of-the-art list of data integrity checks. I use the novel and state-of-the-art integrity checks to clean the commonly used NASA defect datasets and report significant problems within them. I also clean the PROMISE datasets which I use in this dissertation. My additional integrity constraints improve data quality and increase the strength of conclusions I make. My findings should urge researchers to explicitly consider data quality in the future. Data quality underpins the confidence we can have in the results of studies using this data.

1.4 Dissertation Outline

The following is an overview of each subsequent chapter in this dissertation:

- **Chapter 2** is an introduction to the current state-of-the-art knowledge in software defect prediction.
- **Chapter 3** is an overview of machine learning techniques used in software defect prediction. This chapter provides details of the techniques used in this dissertation.
- **Chapter 4** describes the work I did on data quality. The first part describes the background knowledge about data cleaning and reports on new issues which I found. This is followed by **Paper 1** that specifically details the methodology and results of data cleaning applied to the NASA Metrics Data Program datasets.
- **Chapter 5** demonstrates the ability of different classifiers to find distinct subsets of defects. This work is supported by two papers, **Paper 2** and **Paper 3**, where the latter is an extension to the former containing more datasets.
- **Chapter 6** introduces an ensemble model which exploits the findings from Chapter 5 to produce a more effective software defect prediction model. This work is supported by **Paper 4**.
- **Chapter 7** shows a novel approach for visualising predictions made by classifiers using confusion matrices as density plots.
- **Chapter 8** finalises this dissertation by presenting the overall conclusion. It also presents the significance of my work and its implications for future directions of research.

Chapter 2

A Review of Software Defect Prediction

Software defects impact industries and people who rely on software. A good example of this is the shut down of London airspace on a Friday afternoon in the run up to Christmas 2014. This shut down, which affected some 240,000 passengers, was caused by a latent defect that had been in the code since the system was deployed in 1994. Defects can have various impacts on industries and people. Reputation damage, huge costs to companies, and life threatening risks are examples of this impact. The field of software defect prediction develops techniques to prevent defects before they manifest. In this chapter I summarise the aims of software defect prediction and how they are achieved. In particular, I detail the components needed to conduct a defect prediction study. Within this chapter I also provide definitions of the terms used.

2.1 Definition of a Defect

In this dissertation the term 'defect' will correspond to a static code anomaly in software as defined in the IEEE standard classification for software anomalies [IEEE Std 2009]. A defect may or may not manifest itself during the program execution. When a defect manifests itself during the program execution, it becomes a fault which then causes a failure. A defect that is discovered by inspection or static code analysis and removed from the system is not a fault. Figure 2.1 depicts a complete relationship of the terms as defined by [IEEE Std 2009]. In this dissertation, I will use the terms 'bug' and 'fault' interchangeably to denote a defect, which is in accordance with the literature where the term defect is described using any of the terms above.

There are different types of defects. Some defects appear due to configuration problems [Arshad et al. 2013, Xia et al. 2013], structural problems in code [Petrić and Grbac 2014], installability, requirements and usability Hernandez-Gonzalez et al. [2018], and so forth. In

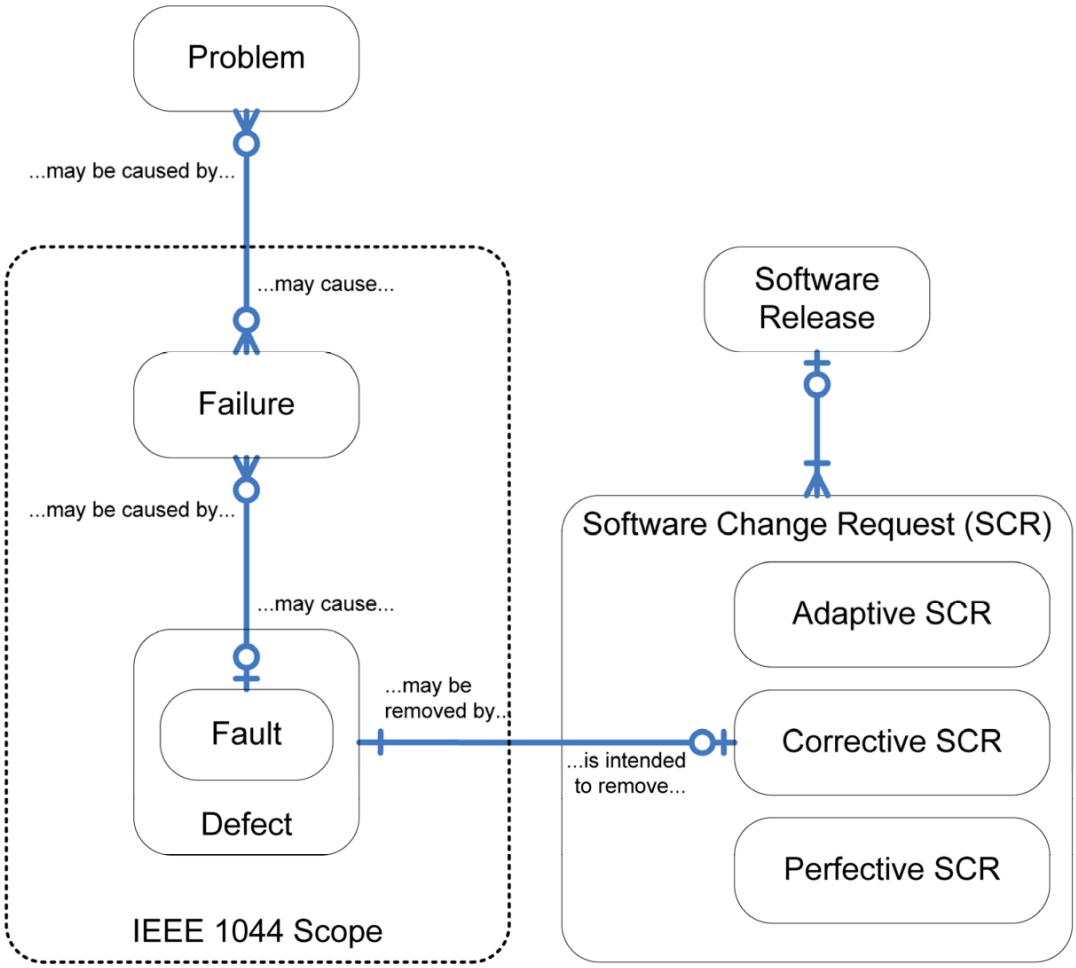


Fig. 2.1 The IEEE classification of software anomalies [IEEE Std 2009]

this dissertation I am interested in defects found exclusively in the source code, which is common in software defect prediction. Most studies rely on datasets containing known code defects. The majority of datasets I use in this dissertation are Java based projects with several C and C++ systems. Defects appearing outside of the source code are out of the scope of this work.

2.2 The Aim of Software Defect Prediction

Software engineering researchers develop techniques which assist practitioners to find defects early. One such approach is software testing, which checks the correctness of software functionalities. Software testing is widely used in software companies. However, testing is typically limited by predefined goals to limit resource expenditure [Yusifoglu et al. 2015].

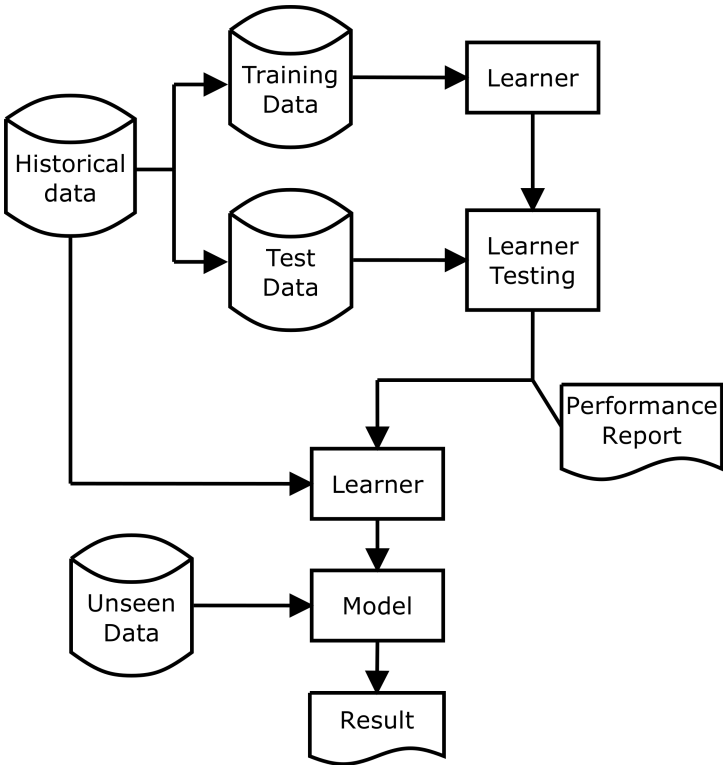


Fig. 2.2 The standard defect prediction framework

For example, one goal in testing could be reaching a fixed threshold for coverage. Where resources are scarce, testing has limited use for uncovering defects. Software defect prediction serves as an alternative and addition to testing which makes forecasts about units of code likely to contain defects.

Software defect prediction relies on prediction models which are typically based on machine learners. Machine learners learn from existing defect data to make informed decisions about yet unseen data. Decisions are either continuous or categorical. The former predicts the number of defects in a unit, whereas the later indicates whether a unit is defective or not. Historical defect data is usually collected on previous versions and contains various measurements of the software. Most frequently used measurements in software defect prediction are static code metrics. They hold basic characteristics of software code such as; lines of code, complexity, the number of operators and operands, and so forth [Hall et al. 2012, Malhotra 2015]. Defect data is typically cheap to collect using automated tools. The historical data is used to build prediction models. Once models are built, they can be used by practitioners at a fraction of the cost of testing to take further actions about units predicted as likely to be defective.

Prediction models are bound to make errors if they have not been trained on data which represents the whole model. Models' performance determines how useful they are at predicting defects. Software defect prediction models rarely achieve performances beyond 80% recall [Hall et al. 2012]. The imbalanced nature of datasets [Mahmood et al. 2015], nondescript metrics, and inappropriate modelling techniques limit the performance of software defect prediction models. Inaccurate models are likely to incur wasted effort by inspecting non-defective units, which is conceivably one of the reasons why those models are not widely used by the industry. However, where the reliability of software is crucial, software defect prediction models can be valuable.

Figure 2.2 depicts the stages in software defect prediction modelling. Software defect prediction models are usually constructed by using historical software data, which is split into training and testing sets allowing the model to be validated. The training set, made of independent and dependent variables, is then fed into some mathematical model, usually a machine learner, which aims to find relevant patterns in the data. The model can then be used to predict the defectiveness of new unseen data. Prudent improvements to any stage of the framework can potentially reduce the errors of software defect prediction models and make them useful for a wider software industry.

2.3 Independent and Dependent Variables in Software Defect Prediction

Software defect prediction requires two sets of variables – the independent and dependent variables. The former measures different aspects of software code (such as lines of code), whilst the latter (either the number of defects or the binary defective/not–defective label) are to be predicted. There is an assumption that the dependent variable(s) are somehow associated with the independent variables. In other words there is a link between a dependant and an independent variable such that the dependent variable may respond to a change of the independent variable. In the remaining of this section I describe commonly used independent and dependent variables in software defect prediction.

2.3.1 Independent Variables in Software Defect Prediction

An independent variable typically measures a certain phenomenon which is then used for determining what effect it has on a dependent variable. In software defect prediction we use terms metrics, attributes, and features to denote the independent variables. In this dissertation I use the words metrics, attributes, and features interchangeably as these terms

are typically used in the literature to denote the same concept. There are two major types of the independent variables in software defect prediction: source code metrics and process metrics. Source code metrics directly measure source code itself (an example is lines of code). Process metrics measure some aspect of the environment in which the software is being produced. An example of the process metric is the number of changes made to a particular unit in the system. Other types of metrics have also been used in software defect prediction, such as source code text, socio-technical network [Hall et al. 2012], and developer oriented metrics [Di Nucci et al. 2018, Posnett et al. 2013].

Lines of code counts (LOC) are commonly used in software defect prediction to account for software size. The LOC counts can be broken down into executable, comment, and blank lines of code. The usefulness of LOC counts have extensively been debated in the literature. Zhang [2009] discovered that the number of defects can be predicted with reasonable precision and recall using only lines of code as the input. He et al. [2015] demonstrated that LOC in combination with the coupling between objects and lack of cohesion metrics can achieve satisfactory results independent from the model used. Contrary, Fenton and Neil [1999b] demonstrated that alternative explanations between the size and defect density relationship can be given. They concluded that the LOC counts are not sufficient to create appropriate software defect prediction models. The LOC counts remain extensively used in software defect prediction, often accompanied by other metrics [Zhang et al. 2017].

Cyclomatic complexity (CC) and Halstead metrics are a popular set of product metrics based on software size. CC measures the complexity of a program by calculating how many independent paths exist in the program [McCabe 1976]. The number of independent paths of a program is particularly useful for establishing coverage in software testing, as coverage is calculated based on the independent paths exercised. Despite its usefulness in software testing, CC has been criticised as a possible proxy for LOC metrics [Shepperd 1988]. Therefore, Shepperd [1988] and Fenton and Bieman [2014] argue that CC should be interpreted with caution in the context of software defect prediction. Halstead metrics are a set of metrics derived from the unique number of operators/operands and their total numbers. They were introduced by Halstead [1977] with the aim to measure code complexity and developer effort. For example, *Halstead Vocabulary* is merely the sum of the unique operators and operands, whilst *Halstead Bugs* is equal to *Halstead Vocabulary* divided by 3000. Halstead metrics, which particularly rely on constants (e.g. *Halstead Bugs*), have been a subject of considerable criticism due to constants that cannot be entirely justified [Fenton and Bieman 2014].

With an increased popularity of object-oriented languages (OO), Chidamber and Kemerer [1994] proposed the most commonly used set of OO specific metrics [Malhotra 2015].

Initially, the Chidamber and Kemerer [1994] set had consisted of six OO metrics, which was later extended with several additional metrics. Amongst commonly used OO metrics are WMC (weighted method per class), CBO (coupling between objects), DIT (depth of inheritance), LCOM1 (lack of cohesion of methods), and so forth [Okutan and Yildiz 2014, Yan et al. 2017].

Another large group of metrics used in software defect prediction are process metrics. Moser et al. [2008] established that the number of previous defects is a good indicator of future bugs. Similarly, D'Ambros et al. [2009] demonstrated that previous defect reports are the best predictors. However, in their systematic literature review, Hall et al. [2012] showed that process metrics are no better than the LOC counts. Stanić and Afzal [2017] confirmed that process are as good as static code metrics, however particular process metrics such as number of distinct committers and number of defects in the previous version are more discriminative than other process metrics. The combination of static code and process metric sets, on the other hand, tends to improve prediction performances [Stanić and Afzal 2017].

Although the product and process metrics have mostly been used in defect prediction, much effort has been put to engineering new metrics for defect prediction. Zimmermann and Nagappan [2008] used graph theory, showing that software modules with a greater degree of centrality tend to be more defective. Similarly, Petrić and Grbac [2014] used graph theory to show that some graph structures are more related to defects than others. Shippey [2015] presented a new metric based on the Java abstract syntax tree and demonstrated usefulness in predicting specific subsets of defects. Bowes et al. [2016] demonstrated that the combination of static code and test related metrics work well for predicting defect-proneness. More recently, Kirbas et al. [2017] showed how evolutionary coupling, which is defined as an implicit relationship between two or more software artefacts that are frequently changed together, is positively correlated with defectiveness.

Overall, there is no single independent variable which is always associated with the top performing software defect prediction models. Malhotra [2015]'s systematic review suggests some OO metrics to be particularly useful for software defect prediction, such as coupling between objects and response for a class. Contrary, the number of children and depth of inheritance metrics have not shown to be useful for software defect prediction. It is likely that a combination of different types of independent variable will yield better predictors compared to a single group of metrics [Hall et al. 2012].

2.3.2 Dependent Variables in Software Defect Prediction

Software defect prediction can be divided into two groups given the dependent variable. One group of software defect prediction studies is concerned with whether a module is defective or not (binary approach), whilst the other focuses on the number of defects per module. The two groups differ in their approaches towards prediction, as the former uses classification, whilst the latter uses regression techniques.

According to Wahono [2015], classification is more prominent than regression within software defect prediction. In classification, the dependent variable is the status of a module – defective or non-defective. The defectiveness status has historically been extracted in two different ways. The first way is to use one of the algorithms for extracting defects (more details in Section 2.4.2). Where a module is found to be defective in the past, it is assigned a defective label, otherwise a non-defective label. The second way is based on the transformation from the number of defects to the binary values. The number of defects is typically established by counting how many distinctive bugs appeared in a particular unit. To make the transformation from the continuous to binary form, the formula by Menzies et al. [2007b] has predominantly been used:

$$defective? = (error\_count \geq 1).$$

"Although such a binary labelling toward module defectiveness is clearly a simplification of the real world, it is hoped that such a classification system could be an effective aid at determining which modules require further attention during testing" [Gray et al. 2010]. By binarising the dependent variable, defect quantity is permanently lost and there is no distinction between modules containing substantially different numbers of defects. Another problem is the loss of severity of the problem, as modules with more defects are more likely to contain a critical defect [Gray et al. 2012].

Learning to rank approaches, which are regression-based, are an alternative to the classification system. In a learning to rank approach, the information about the number of defects is kept. Software units are sorted from the ones which require immediate attention towards the units which are less likely to be problematic [Yang et al. 2015, Yu et al. 2017]. Using a ranking approach allows developers to focus their efforts to the most problematic units first. However, despite the shortcomings of the binarisation process, the classification approach remains the most commonly used in software defect prediction.

2.4 Data and Quality

Until the early 2000s datasets in software defect prediction were largely not available to the wider community. Companies would generally report their results without providing the datasets on which the models were built [Kamei and Shihab 2016]. The scarcity in data availability posed two problems for software defect prediction. First, researchers had a very limited pool of data to choose from, making the comparison of various modelling techniques harder. Second, replications and reproducibility of studies, which are an essential part of every scientific discipline, become challenging when only parts of study details are disclosed [Mahmood et al. 2018].

In the rest of this section I report on the most commonly used software defect prediction datasets and describe how defect data is collected. I then briefly talk about some of the problems with defect data, and the techniques used for improving the quality of defect data.

2.4.1 Software Defect Prediction Datasets

The NASA Metrics Data Program (MDP) was amongst the first to publicly share 14 datasets for software defect prediction[1] [Menzies et al. 2015]. Since their release, the datasets have remained a popular choice amongst researchers. According to Hall et al. [2012], the NASA MDP datasets have been used in 62 out of 208 software defect prediction studies from 2000 to 2010. Subsequent to the NASA MDP program, other researchers have shared defect data publicly [D'Ambros et al. 2010, Jureczko and Madeyski 2010, Zimmermann et al. 2007a]. Most of those datasets are today available at the tera-PROMISE website[2], and the Eclipse website[3].

2.4.2 Data Collection and Algorithms

One way to identify defective code is by analysing the code's version control system (VCS) and the projects bug tracking system (e.g. BugZilla). A VCS records any changes done to files, by keeping information about added, removed, or modified lines in each commit. A commit is a point in time when a developer decides to submit their changes. At that point the changes to files are recorded, submitted to a VCS, and given a unique identifier. At the point of submitting a commit to a VCS, the commit can be given a comment. Git, SVN, and Mercurial are examples of well-known version control systems [Lanubile et al. 2010].

[1] Initially, the datasets were shared at http://mdp.ivv.nasa.gov. Eventually the datasets have been transferred to http://openscience.us/repo/.

[2] http://openscience.us/repo/defect/

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

A bug tracking system is a repository of defect reports which are reported about the system. When a defect is discovered, a developer typically opens a defect report which keeps the relevant information about the defect. Ideally, the defect then gets fixed by some developer and committed to a VCS. At that point, a good practice is to give a descriptive comment about the commit by providing a link to the relevant defect report. When the link between a defect report in a bug tracking system, and a commit in a VCS is established, it is possible to obtain defect data [Mauša et al. 2014, Shippey 2015].

Most of the publicly available datasets[4] are collected by using either the SZZ or BugInfo algorithm. Both algorithms rely on the link between a VCS and a bug tracking system. The SZZ algorithm matches the fix described in the bug tracking system with the corresponding commit in the version control system that removed the defect. By backtracking through the version control records, it is possible to identify earlier code changes which ended up being fixed. It is assumed that the code changes which had appeared prior to the fault report introduced a defect. The module of code is therefore labelled as defective between the time the fault was inserted and the time it was fixed. Using this technique it is possible to identify, for a particular snapshot of the code, which methods were faulty and which were not. The SZZ algorithm was described by Śliwerski et al. [2005], and it is based on the early work of Čubranić and Murphy [2003] and Fischer et al. [2003] who used links between a bug tracking system and a VCS. BugInfo is a simplified version of the SZZ algorithm introduced by Jureczko and Spinellis [2010]. BugInfo first identifies links between a bug tracking system and commits in a VCS. It then labels all files in a commit which are linked to a defect report as defective.

The current approaches in extracting defect data have some shortcomings. Both, SZZ and BugInfo depend on a reliable linkage between a VCS and bug tracking system. Where a practice of assigning unique defect report identifiers to commit messages is not followed, both approaches are incapable of inferring the link. Mauša et al. [2014] report on the effectiveness of bug linking techniques. They establish that traditional approaches can work well in certain environments, however particular parts of the algorithm might need adjustments for each project (for example, adjusting a regular expression for matching commits and defect reports).

Tracking back changes to a defect insertion point is another challenge for obtaining defect data. Text- and dependence-based are two common approaches for determining the origins of defects [Davies et al. 2014]. The text approach uses only changes in text code and returns all relevant files, whilst the dependence approach examines changes between control and data returning the first relevant file. Davies et al. [2014] manually analysed over 170 bugs and

[4] Available in the tera-PROMISE and Eclipse repositories

showed that both approaches were partially successful in retrieving correct defect insertion points (29-79% precision, 40-70% recall). Despite these limitations, most commonly used algorithms for extracting defects, SZZ and BugInfo, use the text-based approach.

2.4.3 Defect Data Issues

Many studies have used openly available datasets for software defect prediction. Boucher and Badri [2018], Di Nucci et al. [2017], Li et al. [2017], and Gao et al. [2015] are a small subset of recent studies that use datasets from public repositories. According to Wahono [2015], about 65% of the studies in software defect prediction are based on public datasets.

However, public datasets impose a risk to the community. As many researchers rely on the same datasets, any errors in the data could endanger the validity of the whole body of studies. The work of Ghotra et al. [2015] demonstrated how the quality of data can affect the conclusions researchers make in software defect prediction studies. The authors replicated an earlier work of Lessmann et al. [2008a] who had used erroneous NASA datasets in their study. After these datasets were cleaned by Shepperd et al. [2013], Ghotra et al. [2015] compared whether there were any differences in results. They concluded that four statistically distinct ranks of classification techniques emerged as opposed to the two distinct groups that emerged from the earlier study by Lessmann et al. [2008a].

Any issues with the publicly shared data should be eradicated early. The availability of such data makes it possible to partially verify their correctness and quality. For example, Kaminsky and Boetticher [2004] reported repeated tuples, whilst Boetticher [2006] reported tuples that share the same attribute values but distinctive labels within the NASA MDP datasets. Gray et al. [2011] carried out the first systematic cleaning of the NASA MDP datasets. Shepperd et al. [2013] extended their work by providing a comprehensive list of integrity checks to clean the NASA datasets. Subsequently, Petrić et al. [2016] expanded the Shepperd et al. [2013]'s list by providing two additional integrity checks. The problems reported about the NASA MDP datasets alone have demonstrated how difficult is to collect high-quality data. A detailed description of the issues of defect datasets and their remedy is described in Section 4.2.

2.4.4 Data Cleansing Techniques

The issues with public defect datasets yielded an extensive set of cleansing techniques. Kaminsky and Boetticher [2004] reported on the repeated data instances in the NASA MDP datasets. Bezerra et al. [2007] additionally cleaned inconsistent data instances, where two modules share equal values of the independent, but opposite values for the dependent

variables. Despite the possibility of repeated and inconsistent instances occurring, it poses a problem in the context of machine learning. As Witten and Frank [2002] describe, the data points on which learners train need to be distinct from the data points on which the models are validated (observe the separation of train and test data in Figure 2.2). Therefore, repeated data points should be removed.

Other issues specific to the machine learning context occur in public defect datasets. Constant and repeated attributes and missing values are amongst those reported by Gray et al. [2012]. Gray et al. [2012] developed a series of steps to systematically clean the NASA datasets. Their technique is not necessarily limited to the NASA data. They describe five stages through which the quality of data can be improved. The first two stages remove constant and repeated attributes. These two stages do not add benefits to software defect prediction in the machine learning context. The following stage replaces missing values. The fourth stage removes data points which do not follow domain specific rules. The final stage removes repeated and inconsistent data points. The order of these stages is essential, as any change in ordering could lead to a different final dataset.

Violations of the domain-specific rules have repeatedly been reported in public datasets Gray et al. [2012], Shepperd et al. [2013]. Shepperd et al. [2013] reported twenty integrity checks to mitigate the issues of low-quality datasets. Effects of individual violation rules are different amongst public datasets. Chapter 4 describes the effect of data cleansing when applied to the NASA datasets.

2.5 Software Defect Prediction Frameworks

Menzies et al. [2007b], Lessmann et al. [2008a], and Song et al. [2011] are the three most influential software defect prediction frameworks according to Wahono [2015]. Menzies et al. [2007b] selected OneR, C4.5 decision tree, and Naïve Bayes classifiers and tested their performance on ten NASA MDP datasets. Their framework is based on the 10×10 cross validation, where a dataset is divided into ten folds. At any one time, nine folds are used for training and one fold for testing. Cross validation is typically repeated ten times to minimise order effects which can affect classifiers' performance [Fisher et al. 1992]. Menzies et al. [2007b] used *InfoGain*, a feature selection method to select the most relevant features. The performance of the models was established using two measures, the probability of detection (*pd*) and the probability of false alarm (*pf*).

Lessmann et al. [2008a] was a follow-up to Menzies et al. [2007b]. They proposed additional results by investigating more classifiers and suggested certain improvements for Menzies et al. [2007b]'s methodological framework. A major improvement to the framework

was using the Area-Under-Curve measure (AUC) for assessing classifiers' prediction performance. This improvement was necessary as Zhang and Zhang [2007] had demonstrated that *pd* and *pf* are not suitable performance measures for highly imbalanced datasets (see Section 3.1.1 for data imbalance and Section 3.4.3 for measuring performance in software defect prediction). Imbalanced datasets are common to software defect prediction where the non-defective class is typically over-represented compared to the defective class. Lessmann et al. [2008a]'s experimental design also used a different partitioning of the data with respect to Menzies et al. [2007b], where 2/3 was used for training and 1/3 for testing each classifier. They argued that the split-sample set-up enables easy replication and offers an unbiased estimate of a classifier's generalisation performance.

The current state-of-the-art framework was suggested by Song et al. [2011]. Their framework offers an improved approach to feature selection with respect to Menzies et al. [2007b]. Menzies et al. [2007b] ranked features on the entire dataset, violating the intention of the holdout strategy (i.e. separating testing data points during the training). As the potential result "they overestimate the performance of their learning model and thereby report a potentially misleading result" [Song et al. 2011]. In addition, Menzies et al. [2007b] evaluated each feature separately and selected features with the highest scores. Such strategy "cannot consider features with complementary information and does not account for attribute dependence" Song et al. [2011]. On the other hand, Lessmann et al. [2008a] omitted feature selection. Song et al. [2011] considered these shortcomings and provided their own framework which has wider application. Song et al. [2011]'s framework is depicted in Figure 2.2. This framework is the basis of the software defect prediction models used in this dissertation.

2.6 Summary of Software Defect Prediction

A defect is an unwanted anomaly in a working product which can cause a program to fail. The field of software defect prediction builds prediction models whose aim is to find locations in code where defects are likely to reside. The earlier a defect is identified and fixed the lower the cost of its effect is [Karg et al. 2011]. Software defect prediction models use two key elements to make predictions: independent and dependent variables. The independent variables or metrics describe software or the process used in developing software. Those metrics typically characterise individual software units. Metrics about each unit are typically obtained using automated tools. Each unit is further characterised with a dependent variable. The dependent variable is either the number of defects that occurs in a unit or a binary label indicating whether a unit is defective or not. Defects are obtained from bug tracking and

version control systems by using algorithms such as SZZ and BugInfo. Defect datasets are constructed by combining the independent and dependent variables. The quality of defect datasets has been a subject of much concern. The usefulness of models depends on the quality of the data with which is provided. Several frameworks have been proposed to standardise the construction of software defect prediction models.

Chapter 3

Predictive Modelling for Software Defect Prediction

Predictive modelling in software defect prediction is predominantly accomplished using machine learning techniques. The field of machine learning involves algorithms that use a set of mathematical rules which enable computers to build models. In software defect prediction, machine learners are used to find predictable patterns in software's historical data which can be used to locate potentially defective code units. Machine learning is in general divided into supervised and unsupervised (semi-supervised learning is a combination of both). Depending on the dependent variable, machine learning can further be divided on classification and regression. The most common type of machine learning in software defect prediction is supervised classification [Malhotra 2015].

In this Chapter, I primarily focus on supervised classification machine learning algorithms. Supervised machine learners in classification are typically called classifiers, the terminology I use throughout this dissertation. In this Chapter I describe the following: first, I detail the most common issues regarding the use of machine learning in software defect prediction. With respect to the underlying algorithm of a classifier, each belongs to a particular category. I, therefore, explain four basic groups of classifiers (families of classifiers). These classifiers are used in this work. I then describe various types of ensembles of machine learners which are also part of this work. Finally, I explain how models are evaluated in software defect prediction.

3.1 An Introduction to Machine Learning

Machine learning is a method that automates the process of learning and identifying patterns in data to make informed decisions. Almost any non-random data contains underlying patterns [Segaran 2007]. The two most popular areas of machine learning are supervised and unsupervised learning. Supervised learning requires both independent and dependent variables to be known during the learning stage. The independent variables describe a data instance, usually in terms of metrics. The dependent variables are typically nominal or continuous variables we try to predict. Once a machine learner is trained on a sufficient dataset, the resultant model can be used to predict the dependent variable from yet unseen data. Unsupervised learning, on the other hand, contains only independent variables (metrics) during the learning stage. Various clustering algorithm can then use data instances described by the independent variables to demonstrate any patterns in the data. Software defect prediction studies are predominately supervised learning [Wahono 2015].

Supervised learning can further be broken down to classification and regression methods. In classification methods the dependent variable is required to be categorical (also called nominal). Categorical variables contain two or more categories. In software defect prediction these categories are typically a binary label indicating whether a unit is defective or non-defective. Contrary, in regression modelling the dependent variables are discrete values. In a regression set-up the dependent variable is typically the number of defects per unit. According to Wahono [2015], more than 70% of software defect prediction studies are classification methods.

Machine learners are used to build prediction models. Once a learner learns from sufficient examples in the data, a prediction model can be built (or predictor). This predictor can then be deployed in the classification set-up and used to label an unseen instance for which the label is not known as defective or non-defective. Similarly, in the regression set-up, a predictor can be used to estimate the number of defects for an unseen instance.

Three factors of machine learning have been widely explored in software defect prediction studies: data imbalance, feature selection, and model optimisation. Defect datasets are typically imbalanced. Most instances belong to non-defective units. The under-representation of defective instances introduces a problem to machine learners which tend to optimise for recall [Chawla et al. 2004] (i.e. predicting more true positives for the price of introducing more false positives). Feature selection is another factor influencing software defect prediction models. Correlated features (the independent variables described in Section 2.3.1) are known to negatively affect the performance of a model [Hall 1999] and result in one variable being over-represented [Gray et al. 2012]. Finally, the ability of some learners to perform well depend on their parameters. A good example is the support vector machine family of

classifiers which need substantial tuning of the parameters to perform well [Sarro et al. 2012]. Techniques which deal with these three factors are discussed next.

3.1.1 Data Imbalance

Data imbalance occurs when one class is under-represented compared to other classes in classification learning [He and Garcia 2009]. For example, in software defect prediction it is common that most of data instances belong to the non-defective class. This problem can hamper the performance of machine learners as many learners tend to maximise the predictive accuracy by ignoring the minority class [Chawla et al. 2004]. The data imbalance problem can be mitigated in three ways [Gray 2013]. First, a performance metric measuring classifiers' success rate can be replaced with another metric. The second approach is to modify classifier parameters by, for example, increasing the cost of a minority class misclassification. The third option is to re-sample the training data by adding or removing instances to obtain a more balanced distribution. Often, the combination of multiple approaches is used.

Data can be re-sampled in several ways. Over-sampling is a technique where data balance is obtained by adding minority class instances. New class instances are often added by duplicating the existing minority instances, or by synthesising similar instances to the existing ones (the synthesising technique called SMOTE is introduced by Chawla et al. [2002]). Under-sampling changes data balance by removing instances belonging to the majority class. A simple under-sampling technique will randomly drop instances of the majority class until the balance is obtained. Clustering methods, where similar instances belonging to the majority class are removed, have also been proposed [Yen and Lee 2009]. Another typical approach to deal with class imbalance is to use an ensemble of learners. Ensembles have been repeatedly reported to successfully deal with class imbalance [Rodríguez et al. 2012, Wang and Yao 2013].

Mahmood et al. [2015] demonstrated that training on defect datasets with balance under 20% results in weak software defect prediction models. They used the MCC performance measure to evaluate the models. Bennin et al. [2017] reported significant improvements of software defect prediction models with large effect sizes by applying re-sampling techniques on highly imbalanced defect datasets. Interestingly, Bennin et al. [2017] noticed the improvements only when the models are evaluated using performance measures based on the confusion matrix (i.e. re-sampling had no effect on AUC). Pelayo and Dick [2012] performed re-sampling of both classes to obtain data balance and demonstrated a 23% improvement in the average geometric mean accuracy across four datasets. Rodriguez et al. [2014] compared four re-sampling techniques grouped into sampling, cost-sensitive, ensembles, and hybrid

approaches to find that re-sampling can significantly improve the correct classification of the minority class if data is preprocessed.

3.1.2 Feature Selection

The objective of feature selection is to remove irrelevant and/or redundant features and retain only relevant features [Maimon and Rokach 2010]. There are two fundamentally different approaches when selecting relevant features, namely filter and wrapper. The former assesses the amount of information each attribute carries towards predicting the dependent variable. The latter evaluates various subsets of features using machine learning before deciding which features are most valuable [Witten and Frank 2002]. Redundant features can be removed by establishing the correlation between features.

Both approaches described by Witten and Frank [2002] have appeared in the software defect prediction literature. According to Malhotra [2015] the correlation based feature selection (CFS) was used the most. The CFS is a filter based technique which retains features highly correlated with the dependent variable and removes all others. Menzies et al. [2007b] demonstrated that typically two or three features are sufficient to build competitive prediction models. The authors used the Information Gain (InfoGain) filter technique. InfoGain works on a principle of Shanon's entropy, where the features with most information with respect to the dependent variable are preserved [Mitchell et al. 1997]. Wrappers have also successfully been applied in software defect prediction [Bowes et al. 2016, Cahill et al. 2013, Rodríguez et al. 2012]. The systematic literature review by Hall et al. [2012] reported that "the use of feature selection on sets of independent variables seems to improve the performance of models".

In software defect prediction features are typically correlated. Highly correlated, and especially repeated features, can result in a single feature being over-represented [Gray et al. 2011]. Hall [1999] and Howley et al. [2006] demonstrated that highly correlated features can harm classification performances of many different classifiers. Although not useful for training learners, correlated features are beneficial for validating data integrity. Some feature selection techniques are better than others for software defect prediction. Xu et al. [2016] empirically compared 32 feature selection techniques in software defect prediction and found that filter and wrapper techniques work best. Filter-based techniques select features regardless of the model typically using correlation between independent and dependent variables. Wrapper-based, on the other hand, select best features for each particular model. A similar study comparing 30 feature selection techniques by Ghotra et al. [2017] established that filter-based approach outperforms other techniques across different projects and classification techniques. Ghotra et al. [2017] validated results on 21 commonly

used machine learning techniques in software defect prediction, compared to Xu et al. [2016] who used Random Forest alone.

3.1.3 Model Optimisation

Tuning involves searching for the parameters that produce the most accurate classifier. The search is typically performed in a systematic manner. A grid search, for example, systematically selects best parameters by performing an exhaustive search [Hsu et al. 2003]. Hill climb uses heuristics to find most appropriate parameters [Jacobson and Yücesan 2004]. The search is performed on the training data, which is split into folds. In each iteration of the search, one fold is used in turn as a validation fold, with the remaining folds being used for training a classifier with a set of parameters. The parameters that achieve the highest performance are used to train the classifier on the entire training set.

Some classifiers require tuning to perform well. For example, the SVM classifier is known to perform poorly if not tuned [Soares et al. 2004]. Koru and Liu [2005], Mende and Koschke [2009] and Mende [2010] reported that parameter tuning affects the performance of software defect prediction models. Even though Jiang et al. [2008b] and Tosun and Bener [2009] have shown that the default parameters of the frequently used machine learning tools (e.g. Weka, R, Scikit-learn) are suboptimal, the default parameters remain a favourite choice of many software defect prediction studies. For example, Mende et al. [2009] used the default number of trees in a random forest classifier, Weyuker et al. [2008] used the default parameters for C4.5, and Jiang et al. [2008a] and Bibi et al. [2006] left the default number of nearest neighbours in k-NN. Sarro et al. [2012] used a genetic algorithm (GA) to select the optimal parameters for SVMs. They argue that commonly used grid-search is coarse grained compared to GA and may miss best parameters. Their technique improved recall and F-measure (see Table 3.2) compared to grid-search, and performed better than random-search on all datasets. According to Mahmood et al. [2018] the number of studies performing tuning is low. One out of 13 studies performed parameter tuning. When these 13 studies were replicated, the tuning appeared in two out of 21 studies.

Fu et al. [2016] suggest that tuning is relatively simple to perform and often surprisingly fast. Their results show that tens of attempts to tune the parameters of a model can drastically improve its prediction performances. In certain cases, they show, the models' precision went from 0% to 60%. Similarly, Tantithamthavorn et al. [2016] demonstrate that parameter tuning can increase "the likelihood of producing a top-performing classifier by as much as 83%". However, parameter tuning could potentially increase the risk of over-fitting. Over-fitting occurs when a classifier specialises for one dataset and it cannot generalise to other datasets. The Tantithamthavorn et al. [2016]'s study which uses 26 classifiers on 18 different datasets

shows that classifier tuning does not cause greater over-fitting compared to the equivalent classifiers with their default parameters.

3.2 Basic Machine Learning Techniques

Basic learning classifiers can be divided into several categories. In this section I will particularly discuss four types used in my experiments:

- instance-based classifiers
- tree-based classifiers
- Bayesian classifiers
- linear separation classifiers

3.2.1 Instance-based Classifiers

Instance-based classifiers belong to a lazy learning technique where for each individual prediction the whole training dataset needs to be re-considered [Witten and Frank 2005]. The benefit of this lazy approach is that data instances can be added or removed dynamically. The down side occurs for large datasets where recalculations for predicting new instances can be expensive. Predictions are typically determined by calculating the distance of neighbour instances [Martin 1995]. The assumption is that instances nearby each other will share the same class. The distance between instances are established using a distance function. The choice of a distance function depends on the type of attributes. Amongst others, the Euclidean distance can be used for numeric attributes and the Hamming distance for categorical attributes as a distance function.

A commonly used instance-based classifier in software defect prediction is k-nearest neighbour [Ghotra et al. 2015, Tantithamthavorn et al. 2016]. K-nearest neighbour determines the class by taking the rounded average class of the nearest instances. The most significant parameter is the number of nearby instances to consider (k). For $k = 1$ the classifier will clone the same class of its nearest neighbour. For $k = 3$ the classifier will take the average class of its three nearest neighbours. The k parameter is typically an odd number for a binary classification, as an even k can potentially cause an undetermined class.

3.2.2 Tree-based Classifiers

Tree-based or decision tree classifiers use a decision tree as a model to make predictions. The tree structure contains leaves representing class labels, and branches representing paths of attributes that lead to those class labels. A tree model is created once at the training stage and then used to predict new, yet unseen, instances. A top-down approach is typically used to construct a tree. Witten and Frank [2005] describes this as the divide-and-conquer approach. The crucial part of the algorithm is to chose suitable nodes. Each node is selected based on its ability to best split data into homogeneous subsets with the same class. For this purpose many methods can be used, however the most commonly used are information entropy [Quinlan 1993] and Gini index [Ceriani and Verme 2012].

Different implementation of tree-based classifiers exist. C4.5 is a commonly used tree-based classifier in software defect prediction, which is an extension to an earlier ID3 algorithm [Quinlan 1986]. C4.5 uses information entropy to construct a tree. To best split data into homogeneous subsets, C4.5 uses information gain, which is the difference in entropy. The attribute with the highest information gain is then chosen as a node. J48 is a popular variant of C4.5 available in the Weka tool [Witten and Frank 2002]. Recursive PARTitioning (RPart) is another tree-based classifier which is similar to Classification and Regression Trees (CART) [Therneau et al. 1997]. The RPart programs build classification or regression models of a very general structure where the resulting models can be represented as binary trees. The decision tree is built by finding the variable which best splits the data into two groups. The same process is then applied to each of the groups. The algorithm recursively continues to split the data into groups until no improvement can be made.

Decision trees perform comparatively to other classification techniques in defect prediction [Malhotra 2015]. However, particular implementations of decision trees need to be optimised to work well. For example, Tantithamthavorn et al. [2016] showed that the C5.0 implementation of decision trees can gain 27 percentage points improvement if the parameters are optimised. Decision trees are used to construct Random Forest, an ensemble classifier known to perform well in software defect prediction [Laradji et al. 2015].

3.2.3 Bayesian Classifiers

A Bayesian classifier is a linear classification technique which works on the principle of applying Bayes' theorem. Bayes' theorem describes the probability of an event occurring given that conditional events happen. Equation 3.1 depicts Bayes' theorem.

$$Prob(A|B) = \frac{Prob(A) \times Prob(B|A)}{Prob(B)} \tag{3.1}$$

Naïve Bayes classifiers are a family of simple Bayesian classifiers with a strong but false assumption of the independence between attributes. Even though attributes are seldom fully independent, Naïve Bayes classifiers often perform comparable to other more sophisticated classifiers [Huang et al. 2003]. A model is created by calculating conditional probabilities based on the attribute values. Numeric attributes are typically converted to nominal values by splitting them into bins. This process is called discretisation [Yang and Webb 2009].

A Naïve Bayes classifier uses training data to determine how much is each attribute associated to different classes. The benefit of this approach is that probabilities of a model are explicitly available which simplifies the model's understanding. Similar to decision-tree techniques, in Naïve Bayes, it is enough to train once and use it to predict any number of instances for which a class is unknown.

Despite its simplicity, Naïve Bayes performs well across different datasets in software defect prediction [Hall et al. 2012]. A recent systematic literature review by Malhotra [2015] showed that Naïve Bayes performs competitively compared to other commonly used software defect prediction models. Naïve Bayes has been extensively studied in software defect prediction [Wahono 2015].

3.2.4 Linear Separation Classifiers

Linear separation classifiers work by finding a linear equation that effectively separates classes. There are typically an infinite number of solutions to find an effective separator. Two separators are commonly used in the literature: soft- [Cortes and Vapnik 1995] and hard-margin [Boser et al. 1992]. Soft-margin separators are established by finding a linear separator which ensures the highest prediction performance on training data. A hard-margin separator positions a linear separator to ensure a large distance between classes despite the possibility that this could hinder the performance. The benefit of using a hard-margin over soft-margin separator is in generalisability. As soft-margin separators are perfectly adjusted to training data, their generalisability may be limited.

Support Vector Machines (SVM) are classifiers which use a linear separator. SVMs build models by producing a hyper-plane which can separate the training data into two classes. The items (vectors) which are closest to the hyper-plane are used to modify the model with the aim of producing a hyper-plane which has the greatest average distance from the supporting vectors. Sequential Minimal Optimization (SMO) is a commonly used implementation of SVMs, which solves the quadratic programming problem that arises during the training of SVMs [Platt 1998].

Support Vector Machines have obtained conflicting results in software defect prediction, likely due to their need for extensive parameter optimisation. Gray [2013] argues that the

performance of SVM classifiers is very sensitive to the parameter values which need to be carefully tuned. When the best parameters are selected, SVM can achieve 70% accuracy on average [Gray et al. 2009]. On the other hand, Ghotra et al. [2015] showed that SVM perform less well than some other commonly used techniques such as Naïve Bayes. However, no details of the model optimisation were given in [Ghotra et al. 2015].

3.3 Ensembles of Machine Learners

An ensemble of machine learners is a collection of classifiers which are combined in a certain way to make the final prediction. The assumption is that opinions from multiple experts (classifiers) should reduce the noise in training data and improve the prediction performances. Occasionally, ensembles appear in the literature under various names such as: mixture of experts, multiple classifier systems, committee of classifier, etc. There are several sound reasons which favour the use of ensemble systems [Polikar 2006]:

- statistical reasons: the average decision made by multiple classifiers may reduce the poor selection of a single classifier with weak generalisability performances
- lots of data: a problem could potentially be broken down and the decision be given to multiple classifiers
- little data: re-sampling techniques can be used to create sets of data from which classifier can learn the underlying distribution
- complex boundary decisions: certain complex boundaries cannot be solved by a single classifier, but potentially can be solved with a combination of them
- combining data from multiple sources: occasionally, data for predicting a single phenomenon can be gathered from multiple sources which requires ensembles to be used

Ensembles have repeatedly been shown to produce more favourable performances over single classifiers [Polikar 2006]. One of the first applications of ensembles for a classification problem is Dasarathy and Sheela [1979]. They discussed using multiple classifiers for a partitioned attribute space. Tukey [1977] combined the outputs of two linear regression models. Ensembles have particularly gained in popularity in the nineties when the well-known bagging, boosting, and stacking algorithms were introduced. Freund et al. [1996] proposed an award winning ensemble algorithm called AdaBoost. They showed that AdaBoost can improve the prediction performance by using classifiers which perform only marginally

better than random (so-called *weak* classifiers). The predictions of these classifiers are then combined using majority-voting. Schapire [1990] had previously laid the foundations of AdaBoost. Around the same time, Breiman [1996] suggested the bagging ensemble classifier, where multiple classifiers are trained on slightly different data. Various other ensemble systems appeared in response to different problems researchers and practitioners were facing: stacked generalisation [Wolpert 1992], consensus aggregation [Benediktsson and Swain 1992], dynamic classifier selection [Woods et al. 1997], and so forth.

Ensembles have occasionally been investigated in the context of software defect prediction. Random Forest has been repeatedly used with mixed results. Lessmann et al. [2008a] demonstrated that Random Forest performs well within software defect prediction, whilst Tantithamthavorn et al. [2016] showed that Random Forest is not always in the range of the top performing classifiers. Tosun et al. [2008] used Naïve Bayes, neural networks and Voting Feature Intervals to create an ensemble of classifiers. They demonstrated that the ensemble performs considerably better than Naïve Bayes. At the time, Naïve Bayes was considered as an effective classifier for software defect prediction [Menzies et al. 2007b], and not significantly worse than most of other classifiers [Lessmann et al. 2008a]. Tosun et al. [2008] conducted a similar study in 2011 confirming that ensembles of classifiers can improve software defect prediction Mısırlı et al. [2011b]. Huanjing et al. [2010] compared various ensembles for their ability of selecting attributes for software defect prediction. Laradji et al. [2015] used ensembles to demonstrate their strengths in selecting attributes and managing data imbalance to improve software defect prediction.

According to Surowiecki [2004] there are four criteria needed for achieving superior performances over single individuals:

- opinion diversity – every individual should have private information
- independence – an opinion of an individual is not influenced by others
- decentralisation – individuals should make informed decisions based on their local knowledge about the problem
- aggregation – the opinions should be somehow aggregated into a joint decision

Although Surowiecki [2004] argued about ensembles of humans in decision making, the same concepts apply to ensembles of machine learners [Rokach 2009]. The assumption is, as Surowiecki [2004] argues, that under special circumstances, aggregated decisions can offer superior results compared to individuals (even if those individuals are experts). Ensembles of human decisions is the basis of a democracy system.

There are two key components to consider when constructing ensembles of machine learners: generating individual components (classifiers) and combining their decisions [Polikar 2006]. An ensemble's classifiers should be chosen in such a way to promote diversity amongst its components. If an ensemble consisted of classifiers making errors on the same instances, then it could not perform better than an individual classifier. Therefore, each classifier should ideally make errors on different data points. Combining decisions of multiple classifiers is another key component. Majority-voting is a popular technique to combine predictions, where the class being predicted by the majority of classifiers in an ensemble is being chosen as a final decision. Numerous other combining techniques exist. Weighted majority-voting favours predictions from some classifiers more than from others. Behavioural Knowledge Space (BKS) tracks the frequency of how often each class combination was produced by the classifiers [Huang and Suen 1995]. The class combination appearing most often during training for a particular class is chosen as a prediction class when the same combination appears during testing.

In the rest of this section I will explain why diversity and combining decisions are the key concepts for ensembles. I will then describe two popular ensemble approaches: bagging and boosting. Finally, I will introduce the stacking ensembles approach used in my experiments.

The cornerstones of ensembles: diversity and combining decisions

Ensemble systems prefer diversity amongst their classifiers. Classifiers are diverse if their decision boundaries differ. Those different decision boundaries ensure that classifiers make mistakes on different instances. The assumption is that if each classifier makes different mistakes, their decisions can be combined in a way that could potentially reduce the total error [Polikar 2006].

There are at least five ways to achieve diversity that are used by popular ensemble techniques. Amongst the easiest is data re-sampling, where each data subset is randomly drawn from the entire training set and then used for training a single classifier. Bagging and boosting use this approach. Tuning classifiers' parameters is another common approach to change decision boundaries of classifiers. Diversity can also be achieved by selecting classifiers belonging to different classifier families. Finally, by selecting different attributes, different decision boundaries can be achieved.

Diversity is typically measured using a certain quantity indicator. Pair-wise measures, which count the differences between each pair of classifiers in the ensemble, are frequently used [Polikar 2006]. Examples of pair-wise measures are correlation, Q-statistics, and entropy measure. Kuncheva and Whitaker [2003] report that no diversity measure consistently yields

ensembles with the greatest performance. They suggest the use of Q-statistics due to its simplicity and intuitive meaning.

Weighted Accuracy Diversity (WAD) is a novel pair-wise measure introduced by Zeng et al. [2014]. This method favours most accurate classifiers given a high diversity between them. WAD's definition is given in Equation 3.2.

$$WAD_{\alpha,\beta}(Acc, Div) = \frac{Acc \times Div}{\beta \times Acc + \alpha \times Div} \tag{3.2}$$

Where $\alpha + \beta = 1$ and Acc and Div are accuracy and diversity of the ensemble, calculated according to Equation 3.3 and 3.4, respectively.

$$Acc = \frac{\sum_{i=1}^{k} O(x_i)}{k} \tag{3.3}$$

Where k is the number of classifiers in the ensemble, and $O(x_i)$ is 1 if the prediction of a classifier i is correct, 0 otherwise.

$$Div = \frac{2}{m(m-1)} \sum_{i=1}^{m-1} \sum_{j=i+1}^{m} \frac{N^{10} + N^{01}}{N^{00} + N^{11} + N^{00} + N^{01}} \tag{3.4}$$

Where m represents the number of classifiers in the ensemble, and N the number of disagreements between a pair of classifiers C_i and C_j. The disagreements are established based on the confusion matrix of two classifiers depicted in Table 3.1. I use a variation of the WAD measure for creating a stacking ensemble classifier described in Chapter 6.

Table 3.1 The confusion matrix of disagreements between a pair of classifiers

	C_j (correct)	C_j (incorrect)
C_i (correct)	N^{11}	N^{10}
C_i (incorrect)	N^{01}	N^{00}

3.3.1 Bagging and Boosting

Each ensemble technique achieves diversity and combines prediction in a specific way. Bagging is amongst the simplest algorithms whose pseudocode is depicted in Algorithm 1. Diversity is achieved by creating different subsets of data on which bagging classifiers are trained. The training of the algorithm works as follows. In the first iteration a bootstrapped version of a dataset with known labels is created (line 6). Namely, a specific number of instances is randomly selected from the dataset with replacement. A classifier is then trained

on the bootstrapped dataset (line 7) and added to the ensemble (line 8). These steps are repeated a predefined number of times N. To evaluate a new unlabelled instance, each trained classifier in the ensemble makes a prediction. The counter is incremented each time the defective class gets predicted by a classifier (lines 11-13). Finally, if the majority of classifiers in the ensemble predicted the instance $defective$, the final prediction of the ensemble becomes $defective$. Otherwise, the $non-defective$ class is predicted.

TRAIN:
1: Training data D with labels $l_1 = defective$ and $l_2 = non-defective$
2: Classifier C
3: Integer N specifying the number of iterations
4: Percentage I of instances to create a bootstrapping training dataset from
5: **for** $i = 1$ **to** N **do**
6: Create a subset D_i by randomly selecting $I\%$ of D with replacement
7: Create a classifier C_i using D_i
8: Add C_i to the ensemble, E
TEST:
9: Evaluate E on unlabelled instance x
10: Integer T initialised to 0
11: **for** $i = 1$ **to** N **do**
12: **if** $E_i = l_1$ **then**
13: $T = T + 1$
14: **if** $T > N/2$ **then**
15: Predict l_1
16: **else**
17: Predict l_2

Algorithm 1 A simplified pseudocode for the bagging algorithm [Polikar 2006]

Boosting is another popular ensemble algorithm whose pseudocode is given in Algorithm 2. The algorithm achieves diversity in a similar way to bagging, by creating an ensemble of classifiers whose data is resampled. The final decision is made by majority-voting. However, unlike bagging, in boosting data resampling is not random. The basic boosting algorithm is made of three classifiers. The first classifier is trained on the randomly drawn subset of the training data (lines 3-4). The second classifier gets half of the training data which is correctly predicted by the first classifier and the other half are misclassifications (lines 5-6). The third classifier is trained on the instances for which the first and second classifier disagree (lines 7-8). Finally, the final decision is made by majority-voting amongst the three classifiers (lines 9-10). Schapire [1990] showed that the performance of a boosting ensemble is never worse than the best classifier in the ensemble, given that each classifier is better than random.

INPUT:
1: Training data D of size M with labels $l_1 = defective$ and $l_2 = non-defective$
2: Classifiers C_1, C_2, C_3
TRAIN:
3: Choose $M_1 < M$ instances from D without replacement to create D_1
4: Create C_1 by training on D_1
5: Create D_2 by selecting half of correctly predicted instances from C_1 and another half misclassifications applying the following rules:
a: Flip a fair coin. If Head, choose samples from D and classify them using C_1. Stop when the first instance is misclassified and add it to D_2
b: If Tail, choose samples from D and classify them using C_1. Stop when the first instance is correctly predicted and add it to D_2
c: Repeat 5 until no more instances can be added to D_2
6: Train C_2 on D_2
7: Create D_3 by selecting instances on which C_1 and C_2 disagree
8: Train C_3 on D_3
TEST:
9: Classify each test instance x with C_1 and C_2. If the classifications agree this is the prediction
10: Otherwise, use C_3 and choose its prediction as final

Algorithm 2 A simplified pseudocode for the boosting algorithm [Polikar 2006]

Bagging and boosting are widely shown to be effective compared to single classifiers [Chan and Paelinckx 2008, Prusa et al. 2015]. The key difference between bagging and boosting is in the way each samples the data. Whilst bagging allows replacement of samples, boosting does not. One of the most popular bagging algorithms, Random Forest, typically performs well across different domains: medicine [Shaikhina et al. 2017], 3D face recognition [Fanelli et al. 2013], and chemistry [Cano et al. 2017]. Boosting ensembles have also demonstrated success compared to other techniques across fields, e.g. heart disease classification [de Menezes et al. 2017] and emotion recognition from speech [Kim et al. 2015].

3.3.2 Stacking Ensembles

Stacking is an ensemble technique which combines predictions of multiple base classifiers via a meta-classifier. A pseudocode of the stacking ensemble classifier is depicted in Algorithm 3. Stacking sets off by training all of its base classifiers on the whole training dataset (lines 4-5). The decisions of the base classifiers are recorded (line 6) as they are used to construct a new dataset D' (line 7-8) whose attributes are prediction outputs of each base classifier. A

new meta-classifier is then trained on the D' dataset which results in the ensemble classifier E (line 9). At this point new unlabelled instances can be fed into the ensemble. For each test instance the base classifiers of the ensemble make the predictions. Their output is fed into the meta-classifier which makes the final decision (lines 10-11).

INPUT:
1: Training data $D = \{x_i, y_i\}$, where x_i is an ith instance and y_i its label (defective or non-defective)
2: Unlabelled test data T of size P
3: A number O of base classifiers $C_{1..O}$
TRAIN:
4: **for** $i = 1$ to O **do**
5: Train C_i on D
6: Record decision p_i of C_i
7: **for** $j = 1$ to N **do**
8: Construct meta-data $D'_j = \{p_{1..O}, y_i\}$, where $p_{1..N} = \{C_1(x_i), ..., C_O(x_i)\}$
9: Train meta-classifier on D' and construct E
TEST:
10: **for** $i = 1$ to P **do**
11: Predict T_i using E

Algorithm 3 A simplified pseudocode for the stacking algorithm

Stacking's architecture substantially differs from the bagging and boosting algorithms. Unlike bagging and boosting where diversity is achieved via different training datasets, stacking trains its base classifiers using the same dataset [Rokach 2009]. Diversity, therefore, has to be achieved by varying the base classifiers. This is typically done in two ways. One option is to tune parameters of the same classifier, creating a set of models with different decision boundaries. Another option is to use classifiers of different type, each making a unique decision boundary. Unbounded to the diversity choice, the goal is to create a new dataset from the predictions of the base classifiers. Those predictions are a training dataset for the meta-classifier, which can be of the same or different type as the base classifiers.

The stacking ensemble has shown superior performances compared to other ensemble learners. It has been successfully applied on problems such as malware detection [Yan et al. 2018] and a wide variety of the UCI machine learning datasets [Džeroski and Ženko 2004]. Most notably, Sill et al. [2009] reported stacking to be a top performing system in the Netflix competition. I use the stacking approach to create a model which can outperform other state-of-the-art models in software defect prediction.

3.3.3 Algorithm Recommendation

Algorithm recommendation is a variation of the stacking technique designed to select the most appropriate algorithm based on the characteristics of data [Kalousis 2002]. The technique is similar to stacking as both approaches require meta-data and meta-classifiers. Stacking uses predictions from the base classifiers to construct meta-data from which the meta-classifier learns. In contrast, algorithm recommendation constructs bespoke meta-data depending on the meta-target. The meta-target is the dependent variable which the meta-classifier in algorithm recommendation aims to predict. Typically, the meta-target is the solution achieving highest performance or lowest cost [Porto et al. 2018]. Examples of meta-targets are predictions of appropriate base learners [Di Nucci et al. 2017] and selecting the most appropriate attribute selection method [Parmezan et al. 2017]. Algorithm 4 depicts the algorithm recommendation technique.

INPUT:
1: Training data $D = \{x_i, y_i\}$, where x_i is an ith instance and y_i its label
2: Unlabelled test data T of size J
3: A number K of base classifiers $C_{1..K}$
TRAIN:
4: **for** $i = 1$ to J **do**
5: Train C_i on D
6: Record decision p_i of C_i
7: Assemble meta-data D' of size K according to the meta-learning goal
a: Construct meta-attributes $a_{1..z}$ by extracting information from D
b: Construct meta-target $t_{1..z}$ by applying bespoke transformations to $p_{i..N}$
8: Train meta-classifier on D' and construct E
TEST:
9: **for** $i = 1$ to J **do**
10: Transform T_i to a meta-attribute a'_i
11: Feeding a'_i to E to predict a suitable classifier C'_i
12: Predict T_i using C'_i

Algorithm 4 A simplified pseudocode for algorithm recommendation

Algorithm recommendation has recently been applied in the domain of software defect prediction. das Dôres et al. [2016] argue that since there is no single algorithm which performs best overall, "research efforts should be directed towards improving algorithm recommendation instead of building more robust approaches". They suggest a novel algorithm recommendation approach, where meta-data is extracted from the basic information about the datasets (i.e. number of instances, number of binary attributes, etc.). The meta-target is to produce the rank of the most suitable classifier given the characteristics of data. They com-

pare their novel approach with random and majority ranking to establish that their approach achieves better results.

Di Nucci et al. [2017] use algorithm recommendation to build an ensemble which predicts the most suitable classifier for predicting a data instance in software defect prediction. Their approach is different from das Dôres et al. [2016] in the granularity and construction of the meta-data. The meta-data in Di Nucci et al. [2017] uses the same independent variables as the ones given to the base level of the ensemble. However, the meta-target is replaced with the label depicting the classifier which correctly predicts a given instance. This meta-data is fed to a meta-level classifier which is Random Forest. Each new instance for which the defectiveness label is unknown is given to their ensemble. The meta-prediction is the best suitable classifier that can potentially correctly predict defectiveness. The assumption is that the selected classifier will be successful in predicting similar instances. The results of Di Nucci et al. [2017]'s study suggest that their approach is performing better in the majority of cases compared to single classifiers and voting ensembles. Future work is needed to establish how the stacking approach used in this work compare with Di Nucci et al. [2017]'s algorithm recommendation technique.

3.4 Defect Prediction Models

3.4.1 Regression and Classification Models

Two types of experiments are typically employed in defect prediction: regression and classification [Wahono 2015]. In software defect prediction, a variable capturing the number of defects in a unit is typically the dependent variable. This number can be directly used in regression techniques to estimate the number of defects. The predicted number of defects need not be accurate for a model to be useful. The estimated number of defects per unit can be sorted in descending order, prioritising the units with most defects [Khoshgoftaar and Allen 2003]. Gray et al. [2011] argue that a descending ordered list could be of a greater practical value as it would allow practitioners to prioritise fixing as resources allow. Other exemplar studies which use regression techniques to predict the number of defects per unit are Bibi et al. [2006], Kanmani et al. [2004], Xu et al. [2000], and Ostrand et al. [2005].

Classification studies require the dependent variable to be nominal. In software defect prediction the dependent variable is typically a binary label describing whether a unit is defective or not. In the classification set-up all instances predicted as defective have an equal priority. This is a weakness of the classification approach. However, even though some well-known public repositories contain the number of defects per unit, those data are

usually binarised. For example, Rahman and Devanbu [2013] binarise their data to perform a classification task. Zimmermann et al. [2007b] collect their own data from Eclipse projects by counting the number of pre- and post-release defects. However, they binarise the number of defects to *hasdefect=1* and *hasdefect=0* to perform classification. Other studies, such as Jing et al. [2014], Kim et al. [2007], Wang et al. [2016], also use classification techniques to predict defects.

3.4.2 Testing the Generalisability of Prediction Models

A model is useful when it accurately predicts new, as yet unseen, data. Typically, a dataset is split into a training and a test set, where a classifier is built using the training and validated using the test set. If a classifier consistently achieves similar performances for both sets, we can establish that the classifier has the ability to generalise.

Train and test sets can be assembled in multiple ways. The simplest approach is to use one version of software as a train set, and its next version as a test set. When this is not possible, a single dataset can be split into train and test sets (e.g. 70% train and 30% test). The former approach is impossible for new systems, as they do not have historic versions. The latter is often challenging due to the imbalanced nature of software defect prediction datasets. The imbalanced nature leaves few defective instances to train on. Therefore, a common approach in software defect prediction is some form of cross-validation. A cross-validation approach reserves a certain amount for testing (a holdout), and uses the rest for training. This process can then be repeated several times, where each time part of data is randomly sampled for testing, and the rest is used for training Witten and Frank [2002].

Software defect prediction studies typically use N-fold stratified cross-validation [Mısırlı et al. 2011a, Rathore and Kumar 2017, Xia et al. 2014]. Stratified cross-validation ensures that an equal proportion of the defective and non-defective classes is represented in each holdout. This approach is needed in highly imbalanced datasets. However, the commonly used 10-fold stratified cross validation violates the cross-validation heuristics [Krstajic et al. 2014]. In particular, to perform stratification the cross-validation algorithm needs to know the total proportion of the defective and non-defective classes beforehand (i.e. during training cross-validation needs to see the entire dataset). Leave-one-out cross validation is a special form of N-fold cross validation, where N is equal to the number of data instances. Leave-one-out does not suffer from the bias introduced by the 10-fold stratified cross validation.

To fairly evaluate the generalisability of a prediction model, any form of test data needs to be hidden from a classifier during training [Witten and Frank 2002]. Menzies et al. [2007b] conducted a software defect prediction study where attribute selection was performed on the entire dataset (training and test set) violating the holdout strategy. As a consequence,

the authors could have potentially "overestimated the performance of their learning model and thereby reported a potentially misleading result" [Song et al. 2011]. Two more minor methodological issues related to attribute selection occurred in the experiment (see Song et al. [2011]). The methodological issues motivated Song et al. [2011] to develop a generalised state-of-the-art software defect prediction framework. The framework aids in eliminating most of the mistakes made in earlier software defect prediction studies.

3.4.3 Performance Metrics for Evaluating Defect Prediction Models

To assess the effectiveness of prediction models we measure their performances. Table 3.2 depicts commonly reported performance measures in software defect prediction. Regression models are typically evaluated on the difference between the value of the original instance and its prediction. For example, Mean Magnitude of Relative Error (MMRE) can be used for regression purposes [Shepperd and Schofield 1997]. MMRE measures the relative error on values from a ratio scale. The performance of classification models are usually derived from the confusion matrix (for an example of a confusion matrix see Section 5.3). Several performance measures can be derived from a confusion matrix (refer to Section 5.3).

In classification, which is the most common type of software defect prediction, it is difficult to choose a correct performance measure. In the Menzies et al. [2007b] study, the authors used the probability of detection (pd) and the probability of false alarm (pf) performance measures. Zhang and Zhang [2007] replied via a comment paper that the models built in Menzies et al. [2007b] are of no practical use, as the precision of models were low. The precision figures were not reported in the original study, however Zhang and Zhang [2007] derived them using the available performance metrics and class distribution data. Menzies et al. [2007a] argued that models in software defect prediction seldom yield high precision, yet can still be useful in practice.

Gray [2013] demonstrated how data balance can misleadingly impact the pd and pf performance measures of defect predictors. For highly imbalanced datasets, it is possible to achieve the same values of pd and pf even if the minority class is entirely misclassified. Gray [2013] suggested the use of precision along with pd and pf for a suitable evaluation of a model.

However, precision is not an ideal performance measure as it does not account for the true negative quadrant of the confusion matrix. In 2014 Shepperd et al. [2014] proposed a more suitable performance measure for software defect prediction, namely Mathews Correlation Coefficient (MCC). Unlike precision, MCC is based on all four quadrants of the confusion matrix. The authors reported that MCC "is a balanced measure and handles situations where the ratio of class sizes are highly imbalanced which is typical of software defect

data (classes containing defects are often relatively rare)" [Shepperd et al. 2014]. More recently, Tantithamthavorn et al. [2018] demonstrated that Area Under the Curve (AUC) is another performance measure not affected during data rebalancing and should be used for comparing the performance of software defect prediction models. Therefore, MCC and AUC are state-of-the-art performance measures for defect prediction.

Table 3.2 Composite Performance Measures

Construct	Defined as	Description
Recall pd (probability of detection) Sensitivity True positive rate	$TP/(TP+FN)$	Proportion of defective units correctly classified
Precision	$TP/(TP+FP)$	Proportion of units correctly predicted as defective
pf (probability of false alarm) False positive rate	$FP/(FP+TN)$	Proportion of non-defective units incorrectly classified
Specificity True negative rate	$TN/(TN+FP)$	Proportion of correctly classified non defective 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
Matthews Correlation Coefficient	$\frac{TP \times TN - FP \times FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}}$	Combines all quadrants of the binary confusion matrix to produce a value in the range -1 to +1 with 0 indicating random correlation between the prediction and the recorded results. MCC can be tested for statistical significance, with $\chi^2 = N \cdot MCC^2$ where N is the total number of instances.

Chapter 4

A Methodology for Improving Data Quality in Software Defect Prediction

Data quality has attracted significant attention in software defect prediction. Researchers have reported dozens of integrity constraints that can help to improve the quality of software defect prediction datasets [Gray et al. 2012, Shepperd et al. 2013]. Earlier research has shown that data quality affects the conclusions of software defect prediction studies [Ghotra et al. 2015]. Therefore, to establish reliable results, researchers need to consider preprocessing their data before use.

The aim of this analysis is to ensure that the models used in this work are built on high-quality data. High quality data will lead to conclusions which are more reliable compared to conclusions derived from data of poor quality [Bowes 2013]. I have used state-of-the-art integrity violation checks to clean all datasets that I use in this work. However, during the cleaning process, I discovered additional integrity violations in publicly available software defect prediction datasets. Using the integrity violations I find, and the already existing integrity violations, I create a comprehensive list of integrity checks to clean all datasets used in this work. In this Chapter, I describe the findings and report all integrity violation checks which are part of the methodology used in this dissertation.

4.1 Prelude

I show that four public datasets are highly affected by the integrity checks. These problematic datasets are from the NASA corpus, which I use in some of my studies. After cleaning, these datasets lose most of their tuples rendering them poor candidates for software defect prediction. Refer to Section 4.5 for detailed findings. Two public datasets also change when

the constraints are applied, e.g. *xalan* and *xerces* from the PROMISE repository, but the majority of their tuples remain intact. The commercial datasets I use in this work do not suffer from the integrity violations, indicating that the tools used to extract metrics from these sources did not produce inconsistent values. Section 4.4 details the extent of the integrity violations on the commercial and 14 public datasets. The next section summarises data cleaning procedures reported in the literature. Section 4.3 demonstrates the extended set of integrity constraints, followed by the section reporting on the impact of applying them.

4.2 Data Cleansing and Its Impact on Defect Studies

Numerous researchers have addressed data quality. Kaminsky and Boetticher [2004] removed duplicates from the KC2 NASA dataset. Boetticher [2006] cleaned duplicates and tuples with questionable values (e.g. 1.1 LOC) from several NASA datasets. Similarly, Bezerra et al. [2007] removed duplicates and inconsistent instances in five NASA datasets. Inconsistent rows occur when repeated instances have different labels. Removal of inconsistent cases had earlier been carried out by Khoshgoftaar and Seliya [2004].

More systematic approach to data cleaning was first carried by Gray et al. [2011]. They used the NASA datasets to perform a five-stage cleaning. Gray et al. [2011] suggested removal of constant attributes (e.g. rows identifiers and attributes with zero variance) and removal of repeated data points as they do not benefit software defect prediction. Their systematic cleaning also deals with missing values, enforces domain specific integrity checks, and removes duplicates and inconsistent instances. Shepperd et al. [2013] extended Gray et al. [2011]'s work. They formalised Gray et al. [2011]'s approach and compiled a list of 18 integrity checks specific to the domain. Petrić et al. [2016] fulfilled that list with two additional integrity checks.

Many more studies have used publicly available datasets in software defect prediction (e.g. [Jing et al. 2014, Laradji et al. 2015, Tong et al. 2017]). Not all studies used the same versions of the datasets. For example, Lessmann et al. [2008a] used ten NASA datasets to compare the performance of 22 learners. Ghotra et al. [2015] repeated a similar study to Lessmann et al. [2008a]'s using both, the datasets reported in Lessmann et al. [2008a] and Shepperd et al. [2013]'s cleaned versions of those datasets. They found a statistically significant impact of models trained on the cleaned versions compared to the models trained on the non-cleaned datasets. Gray et al. [2011] also demonstrated how repeated instances "can have a huge influence on the performance of classifiers". Poor quality data affects the conclusions researchers make in software defect prediction studies.

4.3 Subsequent Data Cleansing

I use the NASA datasets in two of my studies (refer to Tables A.1 and A.2 in Appendix A for the context of each dataset released by NASA). As the reliability of my studies rely on their quality, I use Gray et al. [2011]'s and Shepperd et al. [2013]'s integrity checks for cleaning. Further analysis of the cleaned datasets discovered two additional rules violating domain specific constraints. Section 4.5 describes how the two integrity checks are derived.

I also use 14 datasets from the commonly used PROMISE repository [Wahono 2015] and three commercial systems (refer to Tables A.3 and A.4, and Tables A.5 and A.6 in Appendix A for the context of the PROMISE and commercial systems, respectively). All 17 systems contain object-oriented metrics. These metrics differ from the attributes in the NASA systems, so different integrity checks apply. I identified a possible set of domain knowledge integrity constraints occurring in the PROMISE datasets which are:

- $CC_{avg} < CC_{max}$
- $NOC < LOC$
- $NPM < WMC$

Where CC is cyclomatic complexity [McCabe 1976], NOC is the number of children, LOC is the number of lines of code, NPM is the number of public methods for a class and WMC is weighted method per class (refer to Table A.4 in Appendix A for more details about the metrics in the PROMISE datasets). These three violations are reported in Petrić et al. [2016] and Bowes et al. [2017].

The three commercial systems contain yet another set of object-oriented metrics compared to the PROMISE systems. Table 4.1 depicts these metrics. I used the JHawk tool[1] to collect them. I identified the following domain specific constraints that should hold for the JHawk metrics:

- $LOOP <= NLOC$
- $LOOP <= NOS$
- $VDEC <= NOS$

To these 3, an additional 2 integrity checks defined by Shepperd et al. [2013] apply to the JHawk metrics:

- $NLOC > 0$

[1] http://www.virtualmachinery.com/products.htm#JHAWK

Table 4.1 JHawk method level metrics

Metric	Short Description
CAST	Number of class casts in the method
COMP	Cyclomatic Complexity
CREF	Number of different classes referenced in the method
EXCR	Number of exceptions referenced by this method
EXCT	Number of exceptions thrown by this method
HBUG	Estimated Halstead Bugs in the method
HDIF	The Halstead Difficulty of a method is an indicator of method complexity
HEFF	The Halstead Effort for the method is an indicator of the amount of time that it will take a programmer to implement the method
HLTH	The Halstead Length of the method
HVOC	The Halstead Vocabulary of the method
HVOL	The Halstead Volume of a method is an indicator of method size
LMET	Number of calls to local methods i.e. methods that are defined in the class of the method
LOOP	Number of loops in the method
MDN	Maximum Depth of Nesting
MOD	Number of modifiers in the method declaration
NAME	Name of method
NAND	Number of operands in the method
NEXP	Number of Java Expressions in the method
NLOC	Number of Lines of Code in the method
NOA	Number of arguments in method signature
NOC	Number of comments
NOCL	Number of comment Lines
NOPR	Number of operators in the method
NOS	Number of Java statements in the method
TDN	Total Depth of Nesting
VDEC	Number of variables declared in the method
VREF	Number of variable references in the method
XMET	Number of calls to methods that are not defined in the class of the method

- $HEFF = HDIF * HVOL$

When performing the last integrity check, I used the following equation to avoid the floating point comparison problem:

$$ROUND(HEFF) = ROUND(HDIF * HVOL)$$

4.4 The Impact of Data Cleansing on Defect Datasets

Shepperd et al. [2013]'s integrity constraints demonstrate that for the 2 NASA datasets, KC3 and PC2, there are over 50% of erroneous cases in the data. The PC2 and PC5 datasets contain even higher number of tuples which do not benefit software defect prediction (above 80%). Ghotra et al. [2015] showed that data quality affects the conclusions we reach in software defect prediction studies. Our novel integrity checks show that 4 more NASA datasets suffer severely from problematic instances. JM1 and MC2 datasets remain with insufficient data points after cleaning, becoming poor candidates for defect prediction. MC1 and PC4 datasets remain with no defective instances making them unusable for defect prediction. Future work which further investigates the effect of Petrić et al. [2016] constraints remains an open research avenue.

Table 4.2 The impact of the extended set of integrity checks on the PROMISE datasets

Dataset	# of modules pre cleaning	# of modules post cleaning	% loss due to cleaning	% defective methods post cleaning
ant 1.7	745	722	3.1	23.0
arc	234	210	10.3	12.4
camel 1.6	965	877	9.1	21.0
ivy 2.0	352	345	2.0	11.6
jedit 4.2	367	363	1.1	13.2
log4j 1.2	205	202	1.5	92.6
lucene 2.4	340	335	1.5	59.1
poi 3.0	442	397	10.2	64.5
redaktor	176	169	4.0	14.8
synapse 1.2	256	244	4.7	35.2
tomcat	858	791	7.8	9.7
velocity 1.6	229	209	8.7	36.4
xalan 2.6	885	724	18.2	44.6
xerces 1.4	588	482	18.0	77.0

Table 4.3 The impact of the extended set of integrity checks on the commercial systems

Dataset	# of modules pre-cleaned	# of modules post-cleaned	% loss due to cleaning
PA	4996	4996	0.0
KN	4314	4314	0.0
HA	9062	8998	0.7

Table 4.2 depicts the impact of data cleansing rules reported in Section 4.3 when applied to the 14 PROMISE datasets. Refer to Section 5.4 to find the selection criteria for these 14 datasets. Two systems, *xalan* and *xerces*, are the most affected by the extended set of integrity checks amongst the PROMISE datasets with 18% of erroneous instances. Two more datasets, *arc* and *poi*, are affected by 10% of erroneous tuples. Each dataset is somewhat affected, with *jedit* being the least affected with 1%. We also collected data for 3 commercial systems from our big UK-based telecommunication industry collaborator. Table 4.3 demonstrates the impact of the extended set of cleaning rules applied to the commercial systems. As the table shows, PA and KN systems are not affected by the cleansing rules. The HA system is minimally affected by 0.7%. These figures suggest that the 3 datasets are of high quality.

4.5 The paper: "The Jinx on the NASA Software Defect Data Sets"

This section contains the following paper:

Paper 1. Petrić J, Bowes D, Hall T, Christianson B, Baddoo N. The jinx on the NASA software defect data sets. In Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering 2016 Jun 1 (p. 13). ACM.

This paper was published in the Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering. The paper was awarded the "Best Short Paper and Work in Progress Award". The study reports two novel integrity checks affecting most of the NASA MDP datasets.

The Jinx on the NASA Software Defect Data Sets

Jean Petrić
Science and Technology Research Institute
University of Hertfordshire
Hatfield, Hertfordshire
AL10 9AB, UK
j.petric@herts.ac.uk

David Bowes
Science and Technology Research Institute
University of Hertfordshire
Hatfield, Hertfordshire
AL10 9AB, UK
d.h.bowes@herts.ac.uk

Tracy Hall
Department of Computer Science
Brunel University London
Uxbridge, Middlesex
UB8 3PH, UK
tracy.hall@brunel.ac.uk

Bruce Christianson
Science and Technology Research Institute
University of Hertfordshire
Hatfield, Hertfordshire
AL10 9AB, UK
b.christianson@herts.ac.uk

Nathan Baddoo
Science and Technology Research Institute
University of Hertfordshire
Hatfield, Hertfordshire
AL10 9AB, UK
n.baddoo@herts.ac.uk

ABSTRACT

Background: The NASA datasets have previously been used extensively in studies of software defects. In 2013 Shepperd et al. presented an essential set of rules for removing erroneous data from the NASA datasets making this data more reliable to use.

Objective: We have now found additional rules necessary for removing problematic data which were not identified by Shepperd et al.

Results: In this paper, we demonstrate the level of erroneous data still present even after cleaning using Shepperd et al.'s rules and apply our new rules to remove this erroneous data.

Conclusion: Even after systematic data cleaning of the NASA MDP datasets, we found new erroneous data. Data quality should always be explicitly considered by researchers before use.

Keywords

Data quality, software defect prediction, machine learning

1. INTRODUCTION

Software defect prediction (SDP) uses historical software data to predict locations in code likely to have defects before the software is released. Defects may cause software behave in unintended ways deviating from requirements. To find defects, SDP researchers use quantitative measures of software, which are considered to correlate with parts of the software that are likely to be defective. Various, usually automated, approaches are used to learn from historical data and make predictions on new data.

Most automated defect prediction is performed using various machine learning techniques [7, 9]. Regression and classification are the two most commonly used approaches in defect prediction. Regression techniques predict the density or number of possible defects for each module, whilst classification techniques only give categorical information (i.e. a module is defective or non-defective). Gray et al. argue that regression techniques should be preferred over classification techniques since they can provide a priority list of potentially defective modules [2]. However, according to Wahono, 77% of studies have used classification techniques, compared to 14% that used regression techniques [9].

Historical software data is usually fed into machine learners for the purpose of their training. The training data contain quantitative measures for each module, along with the number or label depicting whether a module is defective or not. Machine learners then search for patterns in data and derive mathematical rules for predicting defectiveness. Therefore, the accuracy of predictions highly relies on the quality of historical data. Lessmann et al. conducted a comprehensive study benchmarking over 20 different machine learning algorithms on the NASA MDP datasets [6]. The authors concluded that the top performing 17 classifiers do not result in significantly different prediction performances. However, after Gray et al.'s [3] and Shepperd et al.'s [8] efforts on data cleansing, Ghotra et al. [1] performed a similar study to Lessmann et al. on the cleaned version of the NASA MDP datasets. In this case, the authors concluded that the classifiers' prediction performances are significantly different, and therefore that the choice of a classifier matters. Hence, poor data quality may significantly affect the conclusions that researchers derive.

In the early 2000s, the lack of data availability was a great challenge [5]. However, NASA stepped in and published the NASA Metrics Data Program datasets for researchers to use. These datasets soon became very popular among researchers, since the data in its original form can easily be used for doing defect prediction. According to Hall et al., the NASA MDP datasets have been used in 62 out of 208 software prediction studies from 2000 to 2010 [4]. However,

EASE '16, June 01-03, 2016, Limerick, Ireland

DOI: http://dx.doi.org/10.1145/2915970.2916007

not much contextual information was released along with the datasets. Without articulating sufficient contextual information it becomes more difficult for researchers to understand their results. Ambiguity in the metric definitions makes it difficult or even impossible to verify the consistency of these metrics. Therefore, future effort in collecting defect data should ensure that enough contextual information is also retrieved.

Shepperd et al.'s paper: "Data Quality: Some Comments on the NASA Software Defect Datasets" was published in the September 2013 issue of the IEEE Transactions on Software Engineering journal [8]. Their paper, which extends the work of Gray et al. [2, 3], presents a set of cleaning steps for removing erroneous data from the *NASA Metrics Data Program* datasets. The aim of this paper is to point out additional inconsistency in some of the NASA MDP datasets. The contribution of this paper is two-fold. First, we introduce additional integrity checks essential for cleaning the NASA MDP data before use. Second, we show to what extent the NASA MDP data is affected by applying these additional integrity checks.

In the next section we present related work aimed at cleaning NASA MDP datasets. Following the related work, we show our findings and point out some additional integrity checks that should be addressed when cleaning the defect prediction data. Our conclusions are set out in the final section.

2. RELATED WORK

Gray et al. systematically questioned the quality of the NASA MDP datasets [2, 3]. Gray et al. pointed out a series of problems in the data and offered a solution for dealing with those problems. They identified several concerns that can be summarised as:

- *repeated or inconsistent instances*: multiple software modules contain the same attribute (i.e. static code metric) and class values (i.e. defective label). This situation is clearly possible in the real world, however repeated data points can cause over-optimistic performances when used in the machine learning context. Inconsistent instances happen when multiple software modules have the same attribute values, but different class values. Again, such situation is possible in the real world context, however it can potentially have a negative impact on machine learners;
- *data integrity*: for example, 1.1 lines of code is a clear example of such integrity check;
- *constant and repeated attributes*: attributes that do not contain any variance are of no use for machine learners. Repeated attributes, on the other hand, contain the same values for each instance, which can harm predictions having some attribute over-represented;
- *missing values*: can either be harmful or completely ignored by classification methods.

Shepperd et al. have built on the work of Gray et al. producing a comprehensive set of rules for data cleansing [8]. Their most significant extension from Gray et al.'s rules was in data integrity. They compiled a list of 18 different referential integrity checks that can test the validity of data instances. Overall, they found a significant number of erroneous data points, which they divided into two domains. The first domain contains only the problematic data, i.e. the data with impossible values. The other domain deals with the data that is not problematic, but does not help defect prediction (e.g. repeated attributes). All instances that fit in either of these two categories were removed. The authors provided the cleaned versions of the NASA datasets for both domains to the scientific community.

3. INVESTIGATION AND RESULTS

Our analysis is based on the Shepperd et al.'s cleaned versions of the NASA datasets from the tera-PROMISE repository[1], namely DS′ and DS″. DS′ denotes data with conflicting attribute values and implausible values removed, whilst DS″ is a dataset from which data have been removed that are not problematic but which do not help improve defect prediction (e.g., attributes with constant values, as defined by Shepperd et al. [8]). The cleaned versions of the NASA datasets were not publicly available from the Shepperd et al. original site[2] at the time of our analysis. Consequently, we used the Shepperd et al.'s cleaned version of the NASA datasets from the tera-PROMISE repository. The tera-PROMISE repository did not contain all 14 datasets initially published by the NASA MDP. Datasets KC1 and KC4 were not available in the tera-PROMISE repository, and dataset KC2 did not contain Shepperd et al.'s cleaned versions of the data. The remaining 11 NASA datasets used in this study were at revision number 7[3] in the tera-PROMISE SVN repository. Although the Shepperd et al. and tera-PROMISE versions of the cleaned datasets may differ, we used the tera-PROMISE version as this was the only version now available. However, the tera-PROMISE version is frequently used in software defect prediction studies.

Table 1 provides definitions and acronyms of the *lines of code* (LOC) metrics used in this study. These definitions were available on the now defunct MDP site[4] in the original *NASA Metrics Data Program* documents. For the sake of simplicity and space, we use the letters a to e to denote the LOC metrics used and replace the *number of lines* metric with the letter N (as shown in Table 1). We introduce a new variable, called ξ, which quantifies the missing lines in a module (also shown in Table 1).

Table 1 shows that N counts all LOC between open and close brackets in a module. Because all of these 11 NASA systems were written in either C/C++ or JAVA, only a limited number of code structures are allowed to occur between open and close brackets. In particular, a module may contain a number of: blank lines (variable d), comment lines (variable b) and lines containing code. In the NASA data sets lines containing code are divided into either code and comment on the same line (variable a) or only executable code on a line (variable c). Consequently N is equal to:

$$N = a + b + c + d + \xi \qquad \text{(IC1)}$$

where we should have $\xi = 0$ provided that the variable a is

[1] http://openscience.us/repo/
[2] http://j.mp/scvvIU
[3] The most recent revision at the time of conducting our analysis
[4] http://mdp.ivv.nasa.gov

Table 1: Definition of lines of code metrics in NASA MDP datasets

Acronym	Metric	Definition
a	LOC_CODE_AND_COMMENT	The number of lines which contain both code & comment in a module.
b	LOC_COMMENTS	The number of lines of comments in a module.
c	LOC_EXECUTABLE	The number of lines of executable code for a module (not blank or comment)
d	LOC_BLANK	The number of blank lines in a module.
e	LOC_TOTAL	The total number of lines for a given module.
N	NUMBER_OF_LINES	Number of lines in a module. Pure, simple count from open bracket to close bracket. Includes every line in between, regardless of character content.
ξ	IRREGULARITY_COUNT	The number of unexpected missing lines in a module. We add this metric to support a novel rule for removing erroneous data.

Table 2: Results of erroneous data in the NASA defect datasets violating two new integrity checks

Dataset	DS'						DS"					
	IC1 partition				IC1 violation (P1 + P4)	IC2 violation	IC1 partition				IC1 violation (P1 + P4)	IC2 violation
	P1	P2	P3	P4			P1	P2	P3	P4		
CM1	5	1	311	27	32 (9.3%)	0%	4	1	295	27	31 (9.48%)	0%
JM1	-	-	-	-	-	9555 (99.6%)	-	-	-	-	-	7753 (99.63%)
KC3	0	0	128	72	72 (36%)	0%	0	0	123	71	71 (36.6%)	0%
MC1	0	0	3552	5725	5725 (61.71%)	0%	0	0	115	1873	1873 (94.22%)	0%
MC2	0	0	1	126	126 (99.21%)	0%	0	0	1	124	124 (99.2%)	0%
MW1	0	0	264	0	0 (0%)	0%	0	0	253	0	0 (0%)	0%
PC1	12	1	711	35	47 (6.19%)	0%	12	1	660	32	44 (6.24%)	0%
PC2	-	-	-	-	-	0%	-	-	-	-	-	0%
PC3	2	3	1087	33	35 (3.11%)	0%	2	3	1040	32	34 (3.16%)	0%
PC4	0	0	275	1124	1124 (80.34%)	0%	0	0	228	1059	1059 (82.28%)	0%
PC5	0	0	1504	15497	15497 (91.15%)	0%	0	0	94	1617	1617 (94.51%)	0%

not subsumed in c, and similarly the variable b is not subsumed in a. To verify that the variable a is not subsumed in c we found multiple data points where $a > c$. The same verification test was used to confirm that the variable b is not subsumed in a. Both verification tests were performed after the datasets had been cleaned using our integrity checks. C/C++ programming languages allow the use of preprocessor directives that could have been ignored by the metric extraction tool and not counted in any of the LOC metrics described above. However, we confirmed that this is not the case since dataset KC3 is written in Java and some of KC3 instances also violate the (IC1) rule.

Table 3: All possible outcomes of violating the IC1 rule

Partition	Rule	Description
P1	$\xi < 0$	Implausible values. It is impossible to have more lines of code than total number of lines in a module.
P2	$\xi = 0$	Expected values. Number of lines in a module matches the sum of lines of the code metrics.
P3	$\xi = 1$	Out by one. This is the most common of these issues in the NASA datasets.
P4	$\xi > 1$	The ξ are the missing lines in the dataset.

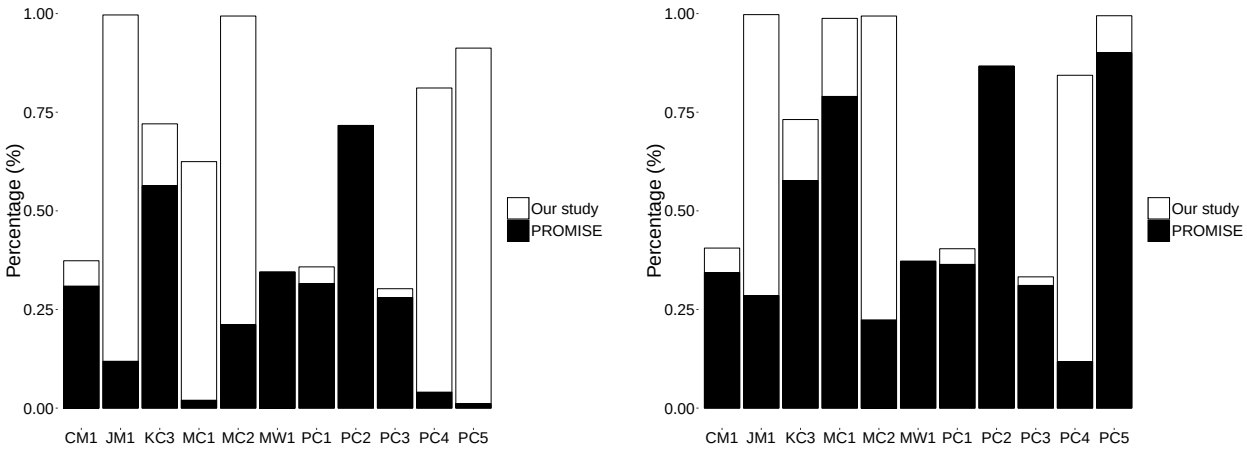
Although all data should obey equation (IC1) with $\xi = 0$, this is not the case for all of the 11 NASA datasets we analysed. Table 3 presents all possible outcomes of calculating the integrity check (IC1). Partition *P2* is the ideal situation where the data complies with (IC1), and therefore to the equation (IC1). Partition *P3* is where the result is out by one. This is a commonly occurring situation in these datasets and could be explained by tools not counting the last line in a module. If the last line of a module is just a bracket, the tool may count this as an additional line (N) but not as a blank line (d), because the new line character comes after the close bracket. This is not the case for the beginning of a module, because the new line character comes after the open bracket. However we cannot verify our suspicion as no code is available with the NASA datasets. We do not remove these out by one instances and accept the data that are present in the partition *P3* set. Partition *P1* presents an impossible situation, because N must always be equal to or greater than the sum of metrics from which it is derived. In *P4* missing lines of code occur, which we capture with the ξ variable. But, because we cannot know for certain the source of ξ and what effect it may have on defect prediction, we should probably avoid such occurrences. Therefore, our opinion is that the data in the *P1* and *P4* sets are violating the (IC1) rule.

The total number of lines in a module (e) for all investigated NASA datasets was equal to:

$$e = a + c \qquad \text{(IC2)}$$

except for the JM1 dataset. We derived the IC2 rule by checking its validity on 10 out of 11 NASA datasets we used. Unfortunately, due to non-availability of the NASA source code and information about the metric tools used, we could not check the reason for this anomaly. However, it is likely that problems were encountered during the collection of metrics for the JM1 dataset because it is the only instance of the NASA datasets that violates the (IC2) rule.

Figure 1: The erroneous instances discarded by Shepperd et al. (PROMISE data) and by (Our study)



(a) Problematic data (DS′)

(b) Data that do not add to defect prediction (DS″)

We also encountered problems with data availability. Datasets JM1 and PC2 were missing N and d metrics, respectively, so we could not check the (IC1) rule for these two NASA datasets.

Table 2 presents the overall results of applying the (IC1) and (IC2) rules to Shepperd et al. cleaned versions of the NASA datasets. Table 2 shows that some datasets are affected dramatically by applying the (IC1) rule. For example, more than 60% of the data in MC1, MC2, PC4 and PC5 breaks the (IC1) rule. Table 2 also shows that there is more erroneous data in the DS″ dataset than the DS′ dataset. This is because the DS″ cleaning procedure reduces the amount of data affected by Shepperd et al.'s rules, whilst not removing the data affected by our rules. Table 2 shows that JM1 and MC2 datasets are particularly problematic. After cleaning, insufficient data remains in the JM1 and MC2 datasets, rendering them poor candidates for defect prediction. Additionally, the post-cleaning of MC1 and PC4 datasets removed all defective data points, making them unusable for defect prediction.

Finally, Figure 1 compares the amount of instances discarded by applying the rules from Shepperd et al. and by our rules. The black bar denotes the instances removed from the tera-PROMISE version of the NASA MDP datasets by using Shepperd et al.'s cleaning rules. Similarly, the white bar denotes the instances eliminated by using our (IC1) and (IC2) rules. The figures clearly show that JM1, MC1, MC2, PC4 and PC5 NASA MDP datasets are highly affected by our rules and not by [8]. Furthermore, in the case of DS″ the JM1, MC1, MC2 and PC5 datasets remain with less than 5% of the original data points.

4. CONCLUSION

Software defect prediction models rely on the quality of the datasets on which they are built. NASA data has been used frequently in previous defect prediction studies. The quality of the NASA data underpins the confidence that we can have in the results of studies using this data. Because data collection mistakes are inevitable, it is essential that the quality of data is explicitly considered and that data is cleaned before use. It is critical that the data cleaning presented by Shepperd et al. and extended here by us is applied to the NASA data before it is used in future studies.

5. ACKNOWLEDGEMENTS

The authors would like to thank Dr David Gray and Professor Martin Shepperd on their valuable comments and recommendations. This work was partly funded by a grant from the UK's Engineering and Physical Sciences Research Council under grant number: EP/L011751/1.

6. REFERENCES

[1] B. Ghotra, S. McIntosh, and A. E. Hassan. Revisiting the impact of classification techniques on the performance of defect prediction models. In *37th Int. Conf. on Software Engineering (ICSE)*, 2015.

[2] 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. In *Evaluation Assessment in Software Engineering (EASE 2011)*, pages 96–103, 2011.

[3] D. Gray, D. Bowes, N. Davey, Y. Sun, and B. Christianson. Reflections on the NASA MDP data sets. *Software, IET*, 6(6):549–558, Dec 2012.

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

[5] Y. Kamei and E. Shihab. Defect prediction: Accomplishments and future challenges. In *Software Analysis, Evolution and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, 2016.

[6] 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 2008.

[7] R. Malhotra. A systematic review of machine learning techniques for software fault prediction. *Applied Soft Computing*, 27:504 – 518, 2015.

[8] M. Shepperd, Q. Song, Z. Sun, and C. Mair. Data quality: Some comments on the NASA software defect datasets. *Software Engineering, IEEE Transactions on*, 39(9):1208–1215, Sept 2013.

[9] R. S. Wahono. A systematic literature review of software defect prediction: Research trends, datasets, methods and frameworks. *Journal of Software Engineering*, 1(1):1–16, 2015.

4.6 Summary of My Contributions

I discovered the following data integrity rules:

- I defined the IC1 and IC2 integrity constraints by analysing the NASA datasets as reported in Section 4.5
- I defined the $LOOP <= NLOC$, $LOOP <= NOS$ and $VDEC <= NOS$ constraints by analysing the Commercial datasets as reported in Section 4.3
- I defined the $CC_{avg} < CC_{max}$, $NOC < LOC$ and $NPM < WMC$ by analysing the PROMISE datasets as reported in Section 4.3

These rules extend the list of integrity constraints identified by Gray et al. [2011] and Shepperd et al. [2013]. Combined together, the rules described in this Chapter and the rules reported by Gray et al. [2011] and Shepperd et al. [2013], show that the NASA datasets contain a high number of problematic data. In this chapter I have also shown that the public PROMISE data contains a fraction of erroneous data. Commercial data we collected is not affected by the extended set of integrity checks.

The integrity constraints demonstrate that publicly available defect data are not immune to issues and need to be cleaned before use. As data quality can affect conclusions in software defect prediction [Ghotra et al. 2015, Gray et al. 2011], I use the comprehensive list of integrity checks described in this Chapter to improve the reliability of results reported in this work. This list can also be used by other researchers who attempt doing software defect prediction. The validation of the impact of the integrity constraints reported in this Chapter are left for other researchers to study as future work.

4.7 Summary of the Contributions to the Paper

I conducted the initial analysis of the NASA datasets and discovered the issues with the lines of code count (the IC1 and IC2 rules). Consequently, I devised initial definitions of the IC1 and IC2 rules and wrote the first full draft of the paper. David Bowes re-checked the rules and suggested the final partitioning of the IC1 rule into four partitions. David Bowes and Tracy Hall provided most of the refinements to the paper by improving its readability. Bruce Christianson improved the definition of the IC1 rule by suggesting that `LOC_CODE_AND_COMMENT` is not subsumed in `LOC_EXECUTABLE` and `LOC_COMMENTS` is not subsumed in `LOC_CODE_AND_COMMENT`. Bruce Christianson and Nathan Baddoo provided minor comments to improve paper's readability.

4.8 Threats to Validity

The new integrity checks rely on domain specific knowledge and have a solid theoretical background. However, due to unavailability of the NASA source code I could not track the root cause of the NASA MDP datasets problems. In case of the JHawk tool, I could not establish the root cause of the erroneous data points, as the source code of the tool is not publicly available. Although the nature of the datasets and tooling issues do not allow researchers to establish the root cause, the impact of these findings is not imperilled as it contains domain specific knowledge.

Chapter 5

Classifiers' Ability to Predict Unique Subsets of Defects

Current trends suggest that the choice of classifier to build a prediction model is irrelevant for software defect prediction as most predictors achieve similar performances. If all classifiers find the same defects, then it would not matter which of them is chosen as a predictor. However, if this is not the case, the choice of a classifier would drive which defects get or do not get predicted. The aim of this chapter is to present empirical evidence which supports the case that models created by different classifiers find different defective components. This is important as more appropriate models which account for diversity in predictions could be built. In addition, some defects matter more than others. I show that models created by different classifiers find different subsets of defects, by using distinct classifier families. Despite the similar predictive performances that classifiers achieve, each classifier detects different sets of defects. This needs to be taken into account when creating software defect prediction models.

5.1 Prelude

The aim of this chapter is to answer the following research question (initially posed in Chapter 1):

RQ1. Do models created by different classifiers find different defective components?

I conduct an empirical analysis to compare the performance of Random Forest, Naïve Bayes, RPart, PART and SVM classifiers when predicting defects in 12 NASA, 17 PROMISE, and three commercial datasets. I find that although prediction performance results of different models are similar, each learner finds a unique subset of defects missed by the others. The

results are reported in three separate studies. Two studies are already published as conference and journal papers and are an integral part of this chapter (refer to Section 5.3 for the papers):

Paper 2. Bowes D, Hall T, Petrić J. Different classifiers find different defects although with different level of consistency. In Proceedings of the 11th International Conference on Predictive Models and Data Analytics in Software Engineering 2015 Oct 21 (p. 3). ACM.

Paper 3. Bowes D, Hall T, Petrić J. Software defect prediction: do different classifiers find the same defects?. Software Quality Journal. 2017:1-28.

One further study reported in Section 5.4 was subsequently conducted to account for the erroneous data reported in Chapter 4, which was not known about at the time of initial publication. In particular, **Paper 2** and **Paper 3** did not consider the comprehensive set of integrity constraints reported in Chapter 4. For this reason, I repeated the studies from **Paper 2** and **Paper 3** using datasets which were cleaned by the comprehensive set of integrity constraints. Section 5.4 describes in more detail the need for extending **Paper 2** and **Paper 3** and reports on the differences I found.

The rest of this chapter is organised as follows. In the next section I explain the motivation for investigating the predictions of individual defects. This is followed by the section presenting the papers. Section 5.4 is a subsequent study which repeats the analysis reported in the papers but takes into account problems with the erroneous data. The rest of the chapter details the contributions of my findings, the contributions to the papers, and threats to validity.

5.2 The Motivation for Investigating the Predictions of Individual Defects

The motivation of this analysis is stated in **Paper 2** and **Paper 3** which I reiterate here. I aim to find whether models created by different classifier techniques find distinct subsets of defects. The focus is on within-project prediction by using the classification approach. Within-project defect prediction builds and tests models using one or multiple versions of datasets from the same project. In contrast, cross-project defect prediction trains a model using datasets from different projects before the model makes predictions on the project of interest. I focus on within-project defect prediction as many eminent researchers before me have also aimed to do this (e.g. [Briand et al. 2002, D'Ambros et al. 2012, Lessmann et al. 2008b]). In addition, within-project defect prediction is preferred when enough data is

available, which is the case in this work, because of the heterogeneity of data represented in the cross-project strategy [Di Nucci et al. 2017] (i.e. data composed of different populations). Classification is used over regression as it enables easier comparison with other studies in software defect prediction which are predominately based on classification [Wahono 2015].

Those before me have differentiated predictive performance using some form of measurement scheme. Such schemes typically calculate performance values (e.g. precision, recall, F-measure etc.) to calculate an overall number representing how well models correctly predict defective and non-defective code taking into account the level of incorrect predictions made (see Table 3.2 for the description of performance measures). I go beyond this by looking at the individual defects that specific classifiers detect and do not detect. I show that, despite the overall figures suggesting similar predictive performances, there is a marked difference between classifiers in terms of the specific defects each detects and does not detect.

These findings suggest that assessing predictive performance using conventional measures gives only a basic picture of the performance of models. Models built using only one classifier are not likely to comprehensively detect defects. The results suggest that the way forward in building high performance prediction models is by using ensembles [Kim et al. 2011]. Ensembles are collections of individual classifiers trained on the same data and combined to perform a prediction task. I investigate the application of ensembles in software defect prediction in Chapter 6.

5.3 Different Classifiers Find Different Defects

This section presents the two papers described at the beginning of this chapter. **Paper 2** was published at the 11th International Conference on Predictive Models and Data Analytics in Software Engineering in 2015. This paper won the best paper award. **Paper 3** was an invited paper which was published in Software Quality Journal. I wrote both papers together with David Bowes and Tracy Hall.

The aim of **Paper 2** is to answer the question: "Do models created by different learners find different defective components?". **Paper 3** builds on the work presented in **Paper 2** by adding more datasets into the experiment and validating the conclusions previously made. To the NASA datasets used in **Paper 2**, 6 new datasets (3 open source, and 3 industrial datasets) were added to **Paper 3**. Introducing more datasets to the analysis increases the diversity of code and defects in those systems. Confirming the previous findings with more and diverse datasets provide stronger evidence that the results are generalisable.

Different Classifiers Find Different Defects Although With Different Level of Consistency

David Bowes
Science and Technology Research Institute,
University of Hertfordshire,
Hatfield, Hertfordshire,
AL10 9AB, UK.
Email: d.h.bowes@herts.ac.uk

Tracy Hall
Department of Computer Science
Brunel University
Uxbridge, Middlesex
UB8 3PH, UK
Email: tracy.hall@brunel.ac.uk

Jean Petrić
Science and Technology Research Institute,
University of Hertfordshire,
Hatfield, Hertfordshire,
AL10 9AB, UK.
Email: j.petric@herts.ac.uk

***Abstract*—BACKGROUND – During the last 10 years hundreds of different defect prediction models have been published. The performance of the classifiers used in these models is reported to be similar with models rarely performing above the predictive performance ceiling of about 80% recall.**
OBJECTIVE – We investigate the individual defects that four classifiers predict and analyse the level of prediction uncertainty produced by these classifiers.
METHOD – We perform a sensitivity analysis to compare the performance of Random Forest, Naïve Bayes, RPart and SVM classifiers when predicting defects in 12 NASA data sets. The defect predictions that each classifier makes is captured in a confusion matrix and the prediction uncertainty is compared against different classifiers.
RESULTS – Despite similar predictive performance values for these four classifiers, each detects different sets of defects. Some classifiers are more consistent in predicting defects than others.
CONCLUSIONS – Our results confirm that a unique sub-set of defects can be detected by specific classifiers. However, while some classifiers are consistent in the predictions they make, other classifiers vary in their predictions. Classifier ensembles with decision making strategies not based on majority voting are likely to perform best.

I. Introduction

Defect prediction models can be used to direct test effort to defect-prone code[1]. Latent defects can then be detected in code before the system is delivered to users. Once found these defects can be fixed pre-delivery, at a fraction of post-delivery fix costs. Each year defects in code cost industry billions of dollars to find and fix. Models which efficiently predict where defects are in code have the potential to save companies large amounts of money. Because the costs are so huge, even small improvements in our ability to find and fix defects can make a significant difference to overall costs. This potential to reduce costs has led to a proliferation of models which predict where defects are likely to be located in code. Hall et al. [14] provide an overview of several hundred defect prediction models published in 208 studies.

The aim of this paper is to identify classification techniques which perform well in software defect prediction. We focus on within-project prediction as this is a very common form of defect prediction. Many eminent researchers before us have also aimed to do this (e.g.[5], [17]). Those before us have differentiated predictive performance using some form of measurement scheme. Such schemes typically calculate performance values (e.g. precision, recall, etc. (see Table III)) to calculate an overall number representing how well models correctly predict truly defective and truly non-defective code taking into account the level of incorrect predictions made. We go beyond this by looking underneath the numbers and at the individual defects that specific classifiers detect and do not detect. We show that, despite the overall figures suggesting similar predictive performances, there is marked difference between four classifiers in terms of the specific defects each detects and does not detect. We also investigate the effect of prediction 'flipping' among these four classifiers. Although different classifiers can detect different sub-sets of defects, we show that the consistency of predictions vary greatly among the classifiers. In terms of prediction consistency, some classifiers tend to be more stable when predicting a specific software unit as defective or non-defective, hence 'flipping' less between experiment runs.

Identifying the defects that different classifiers detect is important as it is well known [10] that some defects matter more than others. Identifying defects with critical effects on a system is more important than identifying trivial defects. Our results offer future researchers an opportunity to identify classifiers with capabilities to identify sets of defects that matter most. Panichella et al. [27] previously investigated the usefulness of a combined approach to identifying different sets of individual defects that different classifiers can detect. We build on [27] by further investigating whether different classifiers are equally consistent in their predictive performances. Our results confirm that the way forward in building high performance prediction models in the future is by using ensembles [16]. Our results also show that researchers should repeat their experiments a sufficient number of times to avoid the 'flipping' effect that may skew prediction performance.

We compare the predictive performance of four classifiers: Naïve Bayes, Random Forest, RPart and Support Vector Machines (SVM). These classifiers were chosen as they are widely used by the machine learning community and have been commonly used in previous studies. These classifiers offer an opportunity to compare the performance of our classification models against those in previous studies. These classifiers also use distinct predictive techniques and so it is reasonable

[1]Defects can occur in many software artefacts, but here we focus only on defects found in code.

to investigate whether different defects are detected by each and whether the prediction consistency is distinct among the classifiers. We apply these classifiers to each of 12 NASA data sets[2]. NASA data sets provide a standard set of independent variables (static code metrics) and dependent variables (defect data labels) for each module in the data set. We chose these NASA data sets because they are also commonly used in software defect prediction and, again, allow our results to be benchmarked against previous results. Unfortunately NASA datasets provide no code. This means that any feature analysis of particular defect sets is not possible.

The following section is an overview of defect prediction. Section Three details our methodology. Section Four presents results which are discussed in Section Five. We identify threats to validity in Section Six and conclude in Section Seven.

II. Background

Classifiers are mathematical techniques for building models which can then predict dependent variables (defects). Defect prediction has frequently used trainable classifiers. Trainable classifiers build models using training data which has items composed of both independent and dependant variables. There are many classification techniques that have been used in previous defect prediction studies. Witten and Frank [32] explain classification techniques in detail and Lessmann et al. [17] summarise the use of 22 such classifiers for defect prediction. Ensembles of classifiers are also used in prediction [23], [31]. Ensembles are collections of individual classifiers trained on the same data and combined to perform a prediction task. An overall prediction decision is made by the ensemble based on the predictions of the individual models. Majority voting is a decision-making strategy commonly used by ensembles. Although not yet widely used in defect prediction, ensembles have been shown to significantly improve predictive performance. For example, Misirli et al. [25] combine the use of Artificial Neural Networks, Naïve Bayes and Voting Feature Intervals and report improved predictive performance over the individual models. Ensembles have been more commonly used to predict software effort estimation (e.g. [24]) where their performance has been reported as sensitive to the characteristics of data sets ([6], [28]).

Many defect prediction studies individually report the comparative performance of the classification techniques they have used. Mizuno and Kikuno [26] report that, of the techniques they studied, Orthogonal Sparse Bigrams Markov models (OSB) are best suited to defect prediction. Bibi et al. [2] report that Regression via Classification works well. Khoshgoftaar et al. [15] report that modules whose defect proneness is predicted as uncertain, can be effectively classified using the TreeDisc technique. Our own analysis of the results from 19 studies [14] suggests that Naïve Bayes and Logistic regression techniques work best. However overall there is no clear consensus on which techniques perform best. Several influential studies have performed large scale experiments using a wide range of classifiers to establish which classifiers dominate. In Arisholm et al.'s [1] systematic study of the impact that classifiers, metrics and performance measures have on predictive performance, eight classifiers were evaluated. Arisholm et al. [1] report that classifier technique had limited impact on predictive performance. Lessmann et al's [17] large scale comparison of predictive performance across 22 classifiers over 10 NASA data sets showed no significant performance differences among the top 17 classifiers.

In general, defect prediction studies do not consider individual defects that different classifiers predict or do not predict. Panichella et al. [27] is an exception to this reporting a comprehensive empirical investigation into whether different classifiers find different defects. Although predictive performances among the classifiers in their study were similar, they showed that different classifiers detect different defects. Panichella et al. proposed CODEP which uses an ensemble technique (i.e. stacking [33]) to combine multiple learners in order to achieve better predictive performances. The CODEP model showed superior results when compared to single models. However, Panichella et al. conducted a cross-project defect prediction study which differs from our study. Cross-project defect prediction has an experimental set-up based on training models on multiple projects and then tested on one project. Consequently, in cross-project defect prediction studies the multiple execution of experiments is not required. Contrary, in within-project defect prediction studies, experiments are frequently done using cross-validation techniques. To get more stabilised and generalised results experiments based on cross-validation are repeated multiple times. As a drawback of executing experiments multiple times, the prediction consistency may not be stable resulting in classifiers 'flipping' between experimental runs. Therefore, in within-project analysis prediction consistency should also be taken into account.

Our paper further builds on Panichella et al. in a number of other ways. Panichella et al. conducted analysis at a class-level while our study is at a module level (i.e. the smallest unit of functionality, usually a function, procedure or method). Panichella et al. also consider regression analysis where probabilities of a module being defective are calculated. Our study deals with classification where a module is labelled either as defective or non-defective. Therefore, the learning algorithms used in each study differ. We also show full performance figures by presenting the numbers of true positives, false positives, true negative and false negatives for each classifier.

Predictive performance in all previous studies is presented in terms of a range of performance measures (see the following sub-section for more details of such measures). The vast majority of predictive performances were reported to be within the current performance ceiling of 80% recall identified by Menzies et al. [22]. However, focusing only on performance figures, without examining the individual defects that individual classifiers detect, is limiting. Such an approach makes it difficult to establish whether specific defects are consistently missed by all classifiers, or whether different classifiers detect different sub-sets of defects. Establishing the set of defects each classifier detects, rather than just looking at the overall performance figure, allows the identification classifier ensembles most likely to detect the largest range of defects.

Studies present the predictive performance of their models using some form of measurement scheme. Measuring model performance is complex and there are many ways in which the performance of a prediction model can be measured. For example, Menzies et al. [21] use pd and pf to highlight

[2] http://promisedata.googlecode.com/svn/trunk/defect/

standard predictive performance, while Mende and Koschke [20] use $Popt$ to assess effort-awareness. The measurement of predictive performance is often based on a confusion matrix (shown in Table II). This matrix reports how a model classified the different defect categories compared to their actual classification (predicted versus observed). Composite performance measures can be calculated by combining values from the confusion matrix (see Table III).

There is no one best way to measure the performance of a model. This depends on the 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. Zhou et al. [34] 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. [1] also discuss how model performance varies depending on how it is measured. The different performance measurement schemes used mean that directly comparing the performance reported by individual studies is difficult and potentially misleading. Comparisons cannot compare like with like as there is no adequate point of comparison. To allow such comparisons we previously developed a tool to transform a variety of reported predictive performance measures back to a confusion matrix [4].

III. Methodology

We have chosen four different classifiers for this study: Naïve Bayes, RPart, SVM and Random Forest. These four classifiers were chosen because they build models based on different mathematical properties. Naïve Bayes produces models based on the combined probabilities of a dependent variable being associated with the different categories of the dependent variables. Naïve Bayes requires that both the dependent and independent variables are categorical. RPart is an implementation of a technique for building Classification and Regression Trees (CaRT). RPart builds a decision tree based on the information entropy (uniformity) of the sub sets of training data which can be achieved by splitting the data using different independent variables. SVMs build models by producing a hyper-plane which can separate the training data into two classes. The items (vectors) which are closest to the hyper-plane are used to modify the model with the aim of producing a hyper-plane which has the greatest average distance from the supporting vectors. Random Forest is an ensemble technique. It is built by producing many CaRTs, each with samples of the training data having a sub-set of features. Bagging is also used to improve the stability of the individual trees by creating training sets produced by sampling the original training data with replacement. The final decision of the ensemble is determined by combining the decisions of each tree and computing the modal value.

The different methods of building a model by each classifier may lead to differences in the items predicted as defective. Naïve Bayes is purely probabilistic and each independent variable contributes to a decision. RPart may use only a subset of independent variables to produce the final tree. The decisions at each node of the tree are linear in nature and collectively put boundaries around different groups of items in the original training data. RPart is different to Naïve Bayes in that the thresholds used to separate the groups are different at each node compared to Naïve Bayes which decides the threshold to split continuous variables before the probabilities are determined. SVMs use mathematical formulae to build non linear models to separate the different classes. The model is therefore not derived from decisions based on individual independent variables, but on the ability to find a formula which separates the data with the least amount of false negatives and false positives.

Classifier tuning is an important part of building good models. As described above, Naïve Bayes requires all variables to be categorical. Choosing arbitrary threshold values to split a continuous variable into different groups may not produce good models. Choosing good thresholds may require many models to be built on the training data using different threshold values and determining which produces the best results. Similarly for RPart, the number of items in the leaf nodes of a tree should not be so small that a branch is built for every item. Finding the minimum number of items required before branching is an important process in building good models which do not over fit on the training data and then do not perform as well on the test data. Random Forest can be tuned to determining the most appropriate number of trees to use in the forest. Finally SVMs are known to perform poorly if they are not tuned [30]. SVMs can use different kernel functions to produce the complex hyper-planes needed to separate the data. The radial based kernel function has two parameters: C and γ, which need to be tuned in order to produce good models.

In practice, not all classifiers perform significantly better when tuned. Both Naïve Bayes and RPart can be tuned, but the default parameters and splitting algorithms are known to work well. Random Forest and particularly SVMs do require tuning. For Random Forest we tuned the number of trees from 50 to 200 in steps of 50. For SVM using a radial base function we tuned γ from 0.25 to 4 and C from 2 to 32. In our experiment tuning was carried out by splitting the training data into 10 folds, 9 folds were combined together to build models with the parameters and the 10th fold was used to measure the performance of the model. This was repeated with each fold being held out in turn. The parameters which produced the best average performance we used to build the final model on the entire training data.

We used the NASA data sets first published on the now defunct MDP website[3]. This repository consists of 13 data sets from a range of NASA projects. A summary of each dataset can be found in Table I. While carrying out our SLR [14] into defect prediction, we extracted the predictive performance of many studies including those we analyse in this paper. In this study we use 12 of the 13 NASA data sets. JM1 was not used because during cleaning, 29% of data was removed suggesting that the quality of the data may have been poor. We carried out a series of pre-processing steps. Initially we allocated each item in each original dataset a unique RowID. The unique RowIDs allow us to map items to their predictions for each cross validation run. We then discretised the data into two categories by assigning the defect label of false for any item with zero defects and a defect label of true where the number of reported defect is greater than zero.

[3] http://mdp.ivv.nasa.gov---unfortunatelynownotaccessible

TABLE I. SUMMARY STATISTICS FOR NASA DATA SETS BEFORE AND AFTER CLEANING

Project	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)
Spacecraft Instrumentation	CM1	C	20	505	505	0.0	9.5	9.5
Ground Data Storage Management	KC1	C++	43	2109	2096	0.6	15.4	15.5
	KC3	Java	18	458	458	0.0	9.4	9.4
	KC4	Perl	25	125	125	0.0	48.8	48.8
Combustion Experiment	MC1	C & C++	63	9466	9277	2.0	0.7	0.7
	MC2	C	6	161	161	1.2	32.3	32.3
Zero Gravity Experiment	MW1	C	8	403	403	0.0	7.7	7.7
Flight Software for Earth Orbiting Satellites	PC1	C	40	1107	1107	0.0	6.9	6.9
	PC2	C	26	5589	5460	2.3	0.4	0.4
	PC3	C	40	1563	1563	0.0	10.2	0.0
	PC4	C	36	1458	1399	4.0	12.2	12.7
	PC5	C++	164	17186	17001	1.1	3.0	3.0
Real-time Predictive Ground System	JM1	C	315	10878	7722	29.0	19.0	21.0

The data quality of the original MDP data sets can be improved [3], [12], [29]. [12], [11], [29] describe techniques for cleaning the data. Shepperd has provided a 'cleaned' version of the MDP data sets[4], however full traceability back to the original items is not provided. Consequently we did not use Shepperd's cleaned data sets. Instead we cleaned the data sets ourselves. We carried out the following data cleaning stages described by [12]: Each independent variable was tested to see if all values were the same, if they were, this variable was removed because they contained no information which allows us to discriminate defective items from non defective items. The correlation for all combinations of two independent variables was found, if the correlation was 1 the second variable was removed. Where the dataset contained the variable 'DECISION DENSITY' any item with a value of 'na' was converted to 0. The 'DECISION_DENSITY' was also set to 0 if 'CONDITION_COUNT'=0 and 'DECISION_COUNT'=0. Items were removed if

1) HALSTEAD_LENGTH!= NUM_OPERANDS+NUM_OPERATORS
2) CYCLOMATIC_COMPLEXITY> 1+NUM_OPERATORS
3) CALL_PAIRS> NUM_OPERATORS

Our method for data cleaning also differs from [29] because we do not remove items where the executable lines of code is zero. We did not do this because we have not been able to determine how the metrics were computed and it is possible to have zero executable lines in Java interfaces.

A. Experimental Set-Up

The following experiment was repeated 100 times. Experiments are more commonly repeated 10 times. We chose 100 repeats because Mende [19] reports that using 10 experiment repeats results in an unreliable final performance figure. Each dataset was split into 10 stratified folds. Each fold was held out in turn to form a test set and the other folds were combined and randomised (to reduce ordering effects) to produce the training set. Such stratified cross validation ensures that there are instances of the defective class in each test set, so reduces the likelihood of classification uncertainty. Re-balancing of the training set is sometimes carried out to provide the classifier with a more representative sample of the infrequent defective instances. Re-balancing was not carried out because not all classifiers benefit from this step. For each training/testing pair four different classifiers were trained using the same training set. Where appropriate a grid search was performed to identify optimal meta-parameters for each classifier on the training set. The model built by each classifier was used to classify the test set. The RowID, DataSet, runid, foldid and classified label was recorded for each item in the test set for each classifier.

We calculate predictive performance values using two different measures: f-measure and MCC (see Table III). F-measure was selected because it is very commonly used by published studies and allows us to easily compare the predictive performance of our models against previous models. It has a range of 0 to 1. MCC was selected because it is relatively easy to understand with a range from -1 to +1. MCC has the added benefit that it encompasses all four components of the confusion matrix whereas f-measure ignores the proportion of true negatives. The results for each combination of classifier and dataset were further analysed by calculating for each item the frequency of being classified as defective. The results were then categorised by the original label for each item so that we can see the difference between how the models had classified the defective and non defective items.

TABLE II. CONFUSION MATRIX

	Predicted defective	Predicted defect free
Observed defective	True Positive (TP)	False Negative (FN)
Observed defect free	False Positive (FP)	True Negative (TN)

The confusion matrix is in many ways analogous to residuals for regression models. It forms the fundamental basis from which almost all other performance statistics are derived .

IV. RESULTS

We aim to investigate variation in the individual defects and prediction consistency produced by the four classifiers. To ensure the defects that we analyse are reliable we first checked that our four models were performing satisfactorily. Figure 1 compares the MCC performance of our models against 600 defect prediction performances reported in published studies

[4]http://nasa-softwaredefectdatasets.wikispaces.com

TABLE III. COMPOSITE PERFORMANCE MEASURES

Construct	Defined as	Description
Recall pd (probability of detection) Sensitivity True positive rate	$TP/(TP+FN)$	Proportion of defective units correctly classified
Precision	$TP/(TP+FP)$	Proportion of units correctly predicted as defective
pf (probability of false alarm) False positive rate	$FP/(FP+TN)$	Proportion of non-defective units incorrectly classified
Specificity True negative rate	$TN/(TN+FP)$	Proportion of correctly classified non defective 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
Matthews Correlation Coefficient	$\frac{TP \times TN - FP \times FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}}$	Combines all quadrants of the binary confusion matrix to produce a value in the range -1 to +1 with 0 indicating random correlation between the prediction and the recorded results. MCC can be tested for statistical significance, with $\chi^2 = N \cdot MCC^2$ where N is the total number of instances.

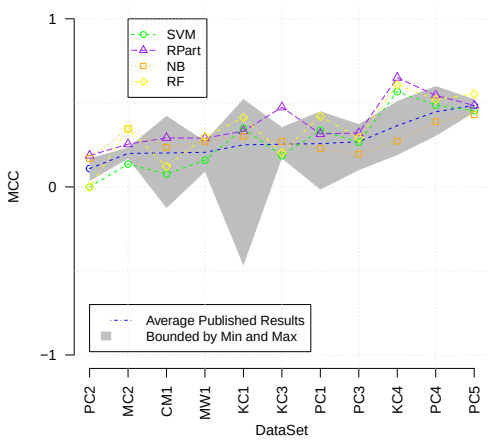


Fig. 1. Our Results Compared to Results Published by other Studies.

using these NASA data sets [14][5]. We re-engineered MCC from the performance figures reported in these previous studies using DConfusion. This is a tool we developed for transforming a variety of reported predictive performance measures back to a confusion matrix. DConfusion is described in [4]. Figure 1 shows that the performances of our four classifiers are generally in keeping with those reported by others. Figure 1 confirms that some data sets are notoriously difficult to predict. For example few performances for PC2 are better than random. Whereas very good predictive performances are generally reported for PC5 and KC4.

We investigated classifier performance variation with Table IV showing the overall performance of our four classifiers across all 12 data sets. Table IV shows little overall difference in average MCC performance across the four classifiers. However these overall performance figures mask a range of different performances by classifiers when used on individual NASA data sets. For example Table V shows Naïve Bayes and RPart performing generally best when used only on the CM1 data set. Whereas Table VI shows much lower performance figures for Naïve Bayes when used only on the KC4 data set[6].

TABLE IV. PERFORMANCE MEASURES ALL DATA SETS BY CLASSIFIER

	MCC	
Classifier	Average	StDev
SVM	0.291	0.188
RPart	0.331	0.162
NB	0.269	0.083
RF	0.356	0.184

TABLE V. PERFORMANCE MEASURES FOR CM1

Classifier	MCC	F-Measure
SVM	0.077	0.094
RPart	0.291	0.349
NB	0.236	0.315
RF	0.121	0.131

TABLE VI. PERFORMANCE MEASURES FOR KC4

Classifier	MCC	F-Measure
SVM	0.567	0.795
RPart	0.650	0.825
NB	0.272	0.419
RF	0.607	0.809

Having established that our models were performing acceptably we next wanted to identify the particular defects that each of our four classifiers predicts so that we could identify variations in the defects predicted by each. We needed to be able to label each module as either containing a predicted defect (or not) by each classifier. As we used 100 repeated 10-fold cross validation experiments, we needed to decide on a prediction threshold at which we would label a module as either predicted defective (or not) by each classifier, i.e. how many of these 100 runs must have predicted that a module was defective before we labelled it as such. We analysed the labels that each classifier assigned to each module for each

[5]Data set MC1 is not included in the figure because none of the studies we had identified previously used this dataset.

[6]Performance tables for all data sets are available from https://bugcatcher.stca.herts.ac.uk/nasa/sensitivity/.

of the 100 runs. There was a surprising amount of prediction 'flipping' between runs. On some runs a module was labelled as defective and other runs not. There was variation in the level of prediction flipping amongst the classifiers. Table VII shows the overall label 'flipping' between the classifiers.

TABLE VII. FREQUENCY OF ALL ITEMS FLIPPING IN ALL DATA SETS

	Non Defective Items			Defective Items		
Classifier	Never	<5%	<10%	Never	<5%	<10%
SVM	0.983	0.985	0.991	0.717	0.746	0.839
RPart	0.972	0.972	0.983	0.626	0.626	0.736
NB	0.974	0.974	0.987	0.943	0.943	0.971
RF	0.988	0.991	0.993	0.748	0.807	0.859

Table VII divides predictions between the actual defective and non-defective labels (i.e. the known labels for each module). For each of these two categories Table VII shows three levels of label flipping: never, 5% and 10%. For example a value of defective items flipping $Never = 0.717$ would indicate that 71.7% of defective items never flipped, a value of defective items flipping $<5\% = 0.746$ would indicate that 74.6% of defective items flipped less than 5% of the time. Table VII suggests that non defective items had a more stable prediction than defective items. This is probably because of the imbalance of data. Although Table VII does not seem to indicate much flipping between modules being predicted as defective or non defective, this table includes all data sets together and so the low flipping in large data sets masks the flipping that occurs in individual data sets.

Tables VIII and IX show the label flipping variations during the 100 runs between data sets[7]. For some data sets using particular classifiers results in a high level of flipping (prediction uncertainty). For example Table VIII shows that using RPart on CM1 results in prediction uncertainty, with 54% of the predictions for known defective modules flipping at least once between being predicted defective to predicted non defective between runs. Table IX shows the prediction uncertainty of using SVM on the KC4 data set with only 26% of known defective modules being consistently predicted as defective or not defective across all cross validation runs. Figure 2 shows the flipping for SVM on KC4 in more detail[8]. As a result of analysing these labelling variations between runs, we decided to label a module as having been predicted as either defective or not defective if it had a been predicted as such on more than 50 runs. Using a threshold of 50 is the equivalent of choosing the label based on the balance of probability.

TABLE VIII. FREQUENCY OF FLIPPING FOR CM1

	Non defective items			Defective Items		
Classifier	Never	<5%	<10%	Never	<5%	<10%
SVM	0.952	0.954	0.972	0.812	0.917	0.958
RPart	0.783	0.783	0.873	0.458	0.458	0.542
NB	0.961	0.961	0.980	0.958	0.958	1.000
RF	0.980	0.987	0.991	0.917	0.917	0.938

TABLE IX. FREQUENCY OF FLIPPING FOR KC4

	Non Defective Items			Defective Items		
Classifier	Never	<5%	<10%	Never	<5%	<10%
SVM	0.719	0.734	0.828	0.262	0.311	0.443
RPart	0.984	0.984	1.000	0.902	0.902	0.984
NB	0.938	0.938	0.984	0.885	0.885	0.934
RF	0.906	0.938	0.953	0.803	0.820	0.918

Having labelled each module as being predicted or not as defective by each of the four classifiers, we constructed

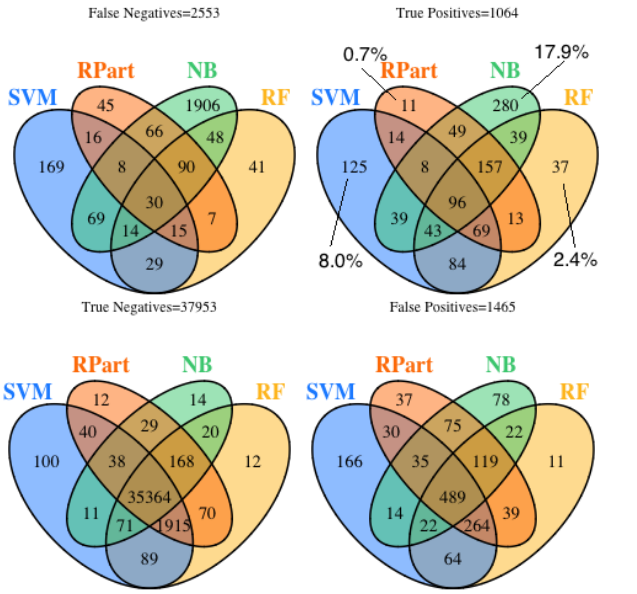


Fig. 3. Sensitivity Analysis for all Data Sets using Different Classifiers. n= 37987 p= 1568

set diagrams to show which defects were identified by which classifiers. Figure 3 shows a set diagram for the 12 frequently used NASA data sets together. The Figure is divided into the four quadrants of a confusion matrix. The performance of each individual classifier is shown in terms of the numbers of predictions falling into each quadrant. Figure 3 shows similarity and variation in the actual modules predicted as either defective or not defective by each classifier. Figure 3 shows that 96 out of 1568 defective modules are correctly predicted as defective by all four classifiers (only 6.1%). Very many more modules are correctly identified as defective by individual classifiers. For example Naïve Bayes is the only classifier to correctly find 280 (17.9%) defective modules and SVM is the only classifier to correctly locate 125 (8.0%) defective modules (though such predictive performance must always be weighed against false positive predictions). Our results suggest that using only a Random Forest classifier would fail to predict many (526 (34%)) defective modules.

There is much more agreement between classifiers about non-defective modules. In the true negative quadrant Figure 3 shows that all four classifiers agree on 35364 (93.1%) out of 37987 true negative modules. Though again, individual non defective modules are located by specific classifiers. For example, Figure 3 shows that SVM correctly predicts 100 non defective modules that no other classifier predicts. The pattern of module predictions across the classifiers varies slightly between the data sets. Figure 4 shows a set diagram for the CM1 data set and Figure 5 shows a set diagram for the KC4 data set[9]. KC4 is an interesting data set. It is unusually balanced between defective and non-defective modules (64 v 61). It is also a small data set (only 125 modules). Figure 5 shows that for KC4 Naïve Bayes behaves differently compared to how it behaves for the other data sets. In particular for KC4 Naïve Bayes is much less optimistic (i.e. it predicts only 17

[7]Label flipping tables for all data sets are available from https://bugcatcher.stca.herts.ac.uk/nasa/sensitivity/.

[8]Violin plots for all data sets are available from https://bugcatcher.stca.herts.ac.uk/nasa/sensitivity/

[9]Set diagrams for all data sets can be found at https://bugcatcher.stca.herts.ac.uk/nasa/sensitivity/

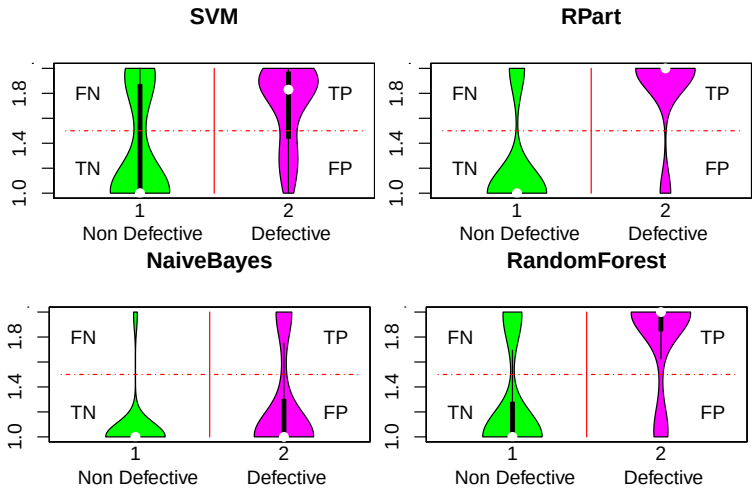


Fig. 2. Violin Plot of Frequency of Flipping for KC4 Data Set

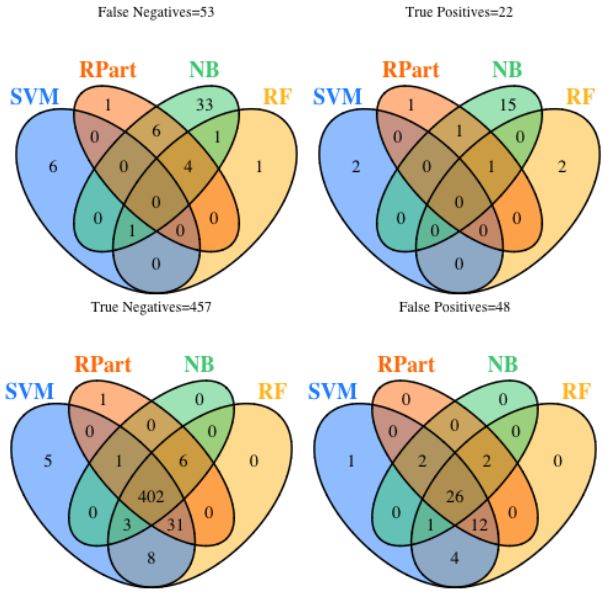


Fig. 4. CM1 Sensitivity Analysis using Different Classifiers. n= 457 p= 48

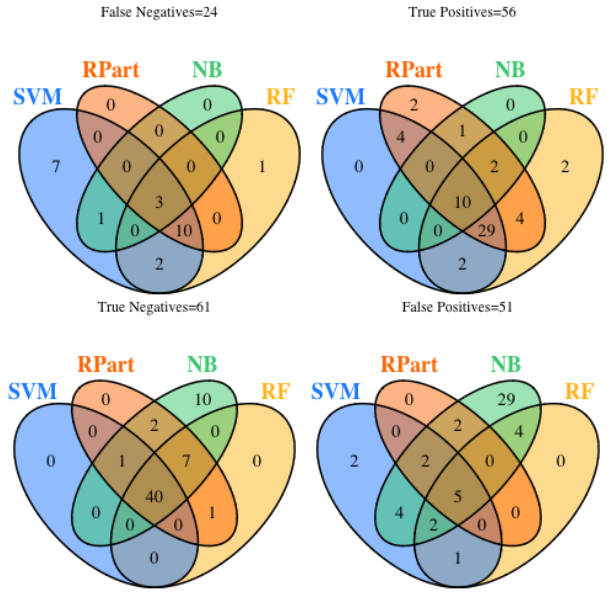


Fig. 5. Sensitivity Analysis for KC4 using Different Classifiers. n= 64 p= 61

out of 125 modules as being defective) in its predictions than it is for the other data sets.

V. Discussion

Our results suggest that there is uncertainty in the predictions made by classifiers. We have demonstrated that there is a surprising level of prediction flipping between cross validation runs by classifiers. This level of uncertainty is not usually observable as studies normally only publish average final prediction figures. Few studies concern themselves with the results of individual cross validation runs. Elish and Elish [8] is a notable exception to this, where the mean and the standard deviation of the performance values across all runs are reported. Few studies run experiments 100 times. More commonly experiments are run only 10 times (e.g. [17], [21]). This means that the level of prediction flipping between runs is likely to be artificially reduced. We suspect that prediction flipping by a classifier for a data set is caused by the random generation of the folds. The items making up the individual folds determine the composition of the training data and the model that is built. The larger the data set the less prediction flipping occurs. This is likely to be because larger data sets may have training data that is more consistent with the entire data set. Some classifiers are more sensitive to the composition of the training set than other classifiers. RPart is particularly sensitive for CM1 where 21% of non defective items flip at least once and 54% of defective items flip. Although RPart performs relatively well (MCC = 0.227), the items it predicts as being defective are not consistent across different cross validation runs. Future research is needed to use our results on flipping to identify the threshold at which overall defective or not defective predictions should be determined.

The level of uncertainty among classifiers may be valuable for practitioners in different domains of defect predictions. For instance, where stability of prediction plays a significant role our results suggest that Naïve Bayes would be the most suitable selection. On the other hand, learners such as RPart may be avoided in applications where higher prediction consistency is needed. The reasons for this prediction inconsistency are yet to be established. More classifiers with different properties should also be investigated to establish the extent of uncertainty in predictions.

Other large scale studies comparing the performance of defect prediction models show that there is no significant difference between classifiers [1], [17]. Our overall MCC values for the four classifiers we investigate also suggest performance similarity. Our results show that specific classifiers are sensitive to data set and that classifier performance varies according to data set. For example, our SVM model performs poorly on CM1 but performs much better on KC4. Other studies have also reported sensitivity to data set (e.g. Lessmann et al. [17]).

Similarly to Panichella et al. [27], our results also suggest that overall performance figures hide a variety of differences in the defects that each classifier predicts. While overall performance figures between classifiers are similar, very different subsets of defects are actually predicted by different classifiers. So it would be wrong to conclude that, given overall performance values for classifiers are similar, it does not matter which classifier is used. Very different defects are predicted by different classifiers. This is probably not surprising given that the four classifiers we investigate approach the prediction task using very different techniques. Future work is needed to investigate whether there is any similarity in the characteristics of the set of defects that each classifier predicts. Currently it is not known whether particular classifiers specialise in predicting particular types of defect.

Our results strongly suggest the use of classifier ensembles. It is likely that a collection of heterogeneous classifiers offer the best opportunity to predict defects. Future work is needed to extend our investigation and identify which set of classifiers perform the best in terms of prediction performance and consistency. This future work also needs to identify whether a global ensemble could be identified or whether effective ensembles remain local to the data set. Our results also suggest that ensembles should not use the popular majority voting approach to deciding on predictions. Using this decision making approach will miss the unique subsets of defects that individual classifiers predict. Again, future work is needed to establish a decision making approach for ensembles that will exploit our findings.

VI. Threats to Validity

Although we implemented what could be regarded as current best practice in classifier-based model building, there are many different ways in which a classifier may be built. There are also many different ways in which the data used can be pre-processed. All of these factors are likely to impact on predictive performance. As Lessmann et al. [17] say *classification is only a single step within a multistage data mining process [9]. Especially, data preprocessing or engineering activities such as the removal of non informative features or the discretisation of continuous attributes may improve the performance of some classifiers (see, e.g., [7], [13]). Such techniques have an undisputed value.* Despite the likely advantages of implementing these many additional techniques, as Lessmann et al. we implemented only a basic set of these techniques. Our reason for this decision was the same as Lessmann et al. *...computationally infeasible when considering a large number of classifiers at the same time*. The experiments we report here each took several days of processing time. We did implement a set of techniques that are commonly used in defect prediction of which there is evidence they improve predictive performance. We went further in some of the techniques we implemented e.g. running our experiments 100 times rather than the 10 times that studies normally do. However we did not implement a technique to address data imbalance (e.g. SMOTE). This was because data imbalance does not affect all classifiers equally. We implemented only partial feature reduction. The impact of the model building and data pre-processing approaches we used are not likely to significantly affect the results we report. In addition the range of approaches we used are comparable to current defect prediction studies.

Our studies are also limited in that we only investigated four classifiers. It may be that there is less variation in the defect subsets detected by classifiers that we did not investigate. We believe this to be unlikely, as the four classifiers we chose are representative of discrete groupings of classifiers in terms of the prediction approaches used. However future work will have to determine whether additional classifiers behave as we report these four classifiers to. We also used only NASA data in our study. Again, it is possible that other data sets behave differently. We believe this will not be the case, as the 12 NASA data sets we investigated were wide ranging in their features and produced a variety of results in our investigation. A further limitation to using only NASA data is that no code accompanies this data. Consequently it is not possible to inspect or further analyse the particular defects being predicted. Extending this study to data sets which include code is now being planned.

Our analysis is also limited by only measuring predictive performance using f-measure and MCC metrics. Such metrics are implicitly based on the cut-off points used by the classifiers themselves to decide whether a software component is defective or not. All software components having a defective probability above a certain cut-off point (in general it is equal to 0.5) are labelled as defective, or as non-defective otherwise. For example, Random Forest not only provides a binary classification of data points, but also provides the probabilities for each component belonging to defective or non-defective categories. D'Ambros et al. [18] investigated the effect of different cut-off points on the performances of classification algorithms in the context of defect prediction and proposed other performance metrics that are independent from the specific (and also implicit) cut-off points used by different classifiers. Future work includes consideration of the different cut-off points to the individual performances of the four classifiers used in this paper.

VII. Conclusion

We report a surprising amount of prediction variation within experimental runs. We repeated our cross validation runs 100 times. Between these runs we found a great deal of inconsistency in whether a module was predicted as defective or not by the same model. This finding has important implications for defect prediction as many studies only repeat experiments 10 times. This means that the reliability of some previous results may be compromised. In addition the prediction flipping that we report has implications for practitioners. Although practitioners may be happy with the overall predictive performance of a given model, they may not be so happy that the model predicts different modules as defective depending on the training of the model.

Performance measures can make it seem that defect prediction models are performing similarly. However, even where similar performance figures are produced, different defects are identified by different classifiers. This has important implications for defect prediction. First, assessing predictive performance using conventional measures such as f-measure, precision or recall gives only a basic picture of the performance of models ([10]). Second, models built using only one classifier are not likely to comprehensively detect defects. Ensembles of classifiers need to be used. Third, current approaches to ensembles need to be re-considered. In particular the popular 'majority' voting decision approach used by ensembles will miss the sizeable sub-sets of defects that single classifiers correctly predict. Ensemble decision-making strategies need to be enhanced to account for the success of individual classifiers in finding specific sets of defects. As Panichella et al. suggested, techniques such as "local prediction" may be suitable for within-project defect prediction as well.

The feature selection techniques for each classifier could also be explored in future. Since different classifiers find different sub-set of defects it is reasonable to explore whether some particular features better suit specific classifiers. Perhaps some classifiers work better when combined with specific subsets of features.

We suggest new ways of building enhanced defect prediction models and opportunities for effectively evaluating the performance of those models in within-project studies. These opportunities could provide future researchers with the tools with which to break through the performance ceiling currently being experienced in defect prediction.

VIII. Acknowledgements

This work was partly funded by a grant from the UKs Engineering and Physical Sciences Research Council under grant number: EP/L011751/1

References

[1] 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.

[2] S. Bibi, G. Tsoumakas, I. Stamelos, and I. Vlahvas. Software defect prediction using regression via classification. In *Computer Systems and Applications. IEEE International Conference on.*, 2006.

[3] G. Boetticher. *Advanced machine learner applications in software engineering*, chapter Improving credibility of machine learner models in software engineering, pages 52 – 72. Idea Group Publishing, Hershey, PA, USA, 2006.

[4] D. Bowes, T. Hall, and D. Gray. DConfusion: a technique to allow cross study performance evaluation of fault prediction studies. *Automated Software Engineering*, pages 1–27, 2013.

[5] 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, jul 2002.

[6] H. Chen and X. Yao. Regularized negative correlation learning for neural network ensembles. *Neural Networks, IEEE Transactions on*, 20(12):1962–1979, 2009.

[7] J. Dougherty, R. Kohavi, and M. Sahami. Supervised and unsupervised discretization of continuous features. In *ICML*, pages 194–202, 1995.

[8] K. Elish and M. Elish. Predicting defect-prone software modules using support vector machines. *Journal of Systems and Software*, 81(5):649–660, 2008.

[9] U. Fayyad, G. Piatetsky-Shapiro, and P. Smyth. From data mining to knowledge discovery in databases. *AI magazine*, 17(3):37, 1996.

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

[11] D. Gray. *Software Defect Prediction Using Static Code Metrics : Formulating a Methodology*. PhD thesis, Computer Science, University of Hertfordshire, 2013.

[12] D. Gray, D. Bowes, N. Davey, Y. Sun, and B. Christianson. Reflections on the nasa mdp data sets. *Software, IET*, 6(6):549 –558, dec. 2012.

[13] M. A. Hall and G. Holmes. Benchmarking attribute selection techniques for discrete class data mining. *Knowledge and Data Engineering, IEEE Transactions on*, 15(6):1437–1447, 2003.

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

[15] 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.

[16] S. Kim, H. Zhang, R. Wu, and L. Gong. Dealing with noise in defect prediction. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 481–490, New York, NY, USA, 2011. ACM.

[17] 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.

[18] M. L. M. DAmbros and R. Robbes. Evaluating defect prediction approaches: a benchmark and an extensive comparison. *Empirical Software Engineering*, 17(4):531577, 2012.

[19] T. Mende. On the evaluation of defect prediction models. In *The 15th CREST Open Workshop*, Oct. 2011.

[20] 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.

[21] 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.

[22] T. Menzies, B. Turhan, A. Bener, G. Gay, B. Cukic, and Y. Jiang. Implications of ceiling effects in defect predictors. In *Proceedings of the 4th international workshop on Predictor models in software engineering*, pages 47–54, 2008.

[23] L. L. Minku and X. Yao. Ensembles and locality: Insight on improving software effort estimation. *Information and Software Technology*, 2012.

[24] L. L. Minku and X. Yao. Software effort estimation as a multi-objective learning problem. *ACM Transactions on Software Engineering and Methodology, to appear*, 2013.

[25] A. T. Mısırlı, A. B. Bener, and B. Turhan. An industrial case study of classifier ensembles for locating software defects. *Software Quality Journal*, 19(3):515–536, 2011.

[26] O. Mizuno and T. Kikuno. Training on errors experiment to detect fault-prone software modules by spam filter. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC-FSE '07, pages 405–414, New York, NY, USA, 2007. ACM.

[27] A. Panichella, R. Oliveto, and A. De Lucia. Cross-project defect prediction models: L'union fait la force. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week - IEEE Conference on*, pages 164–173, Feb 2014.

[28] M. Shepperd and G. Kadoda. Comparing software prediction techniques using simulation. *Software Engineering, IEEE Transactions on*, 27(11):1014 –1022, nov 2001.

[29] M. Shepperd, Q. Song, Z. Sun, and C. Mair. Data quality: Some comments on the nasa software defect datasets. *Software Engineering, IEEE Transactions on*, 39(9):1208–1215, 2013.

[30] C. Soares, P. B. Brazdil, and P. Kuba. A meta-learning method to select the kernel width in support vector regression. *Machine learning*, 54(3):195–209, 2004.

[31] Z. Sun, Q. Song, and X. Zhu. Using coding-based ensemble learning to improve software defect prediction. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 42(6):1806–1817, 2012.

[32] I. Witten and E. Frank. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2005.

[33] D. H. Wolpert. Stacked generalization. *Neural Networks*, 5(2):241 – 259, 1992.

[34] Y. Zhou, B. Xu, and H. Leung. On the ability of complexity metrics to predict fault-prone classes in object-oriented systems. *Journal of Systems and Software*, 83(4):660–674, 2010.

Software Qual J
DOI 10.1007/s11219-016-9353-3

Software defect prediction: do different classifiers find the same defects?

David Bowes[1] · Tracy Hall[2] · Jean Petrić[1,2]

Abstract During the last 10 years, hundreds of different defect prediction models have been published. The performance of the classifiers used in these models is reported to be similar with models rarely performing above the predictive performance ceiling of about 80% recall. We investigate the individual defects that four classifiers predict and analyse the level of prediction uncertainty produced by these classifiers. We perform a sensitivity analysis to compare the performance of Random Forest, Naïve Bayes, RPart and SVM classifiers when predicting defects in NASA, open source and commercial datasets. The defect predictions that each classifier makes is captured in a confusion matrix and the prediction uncertainty of each classifier is compared. Despite similar predictive performance values for these four classifiers, each detects different sets of defects. Some classifiers are more consistent in predicting defects than others. Our results confirm that a unique subset of defects can be detected by specific classifiers. However, while some classifiers are consistent in the predictions they make, other classifiers vary in their predictions. Given our results, we conclude that classifier ensembles with decision-making strategies not based on majority voting are likely to perform best in defect prediction.

Keywords Software defect prediction · Prediction modelling · Machine learning

✉ Jean Petrić
j.petric@herts.ac.uk

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

Tracy Hall
tracy.hall@brunel.ac.uk

[1] Science and Technology Research Institute, University of Hertfordshire, Hatfield, Hertfordshire, AL10 9AB, UK

[2] Department of Computer Science, Brunel University London, Uxbridge, Middlesex, Uxbridge UB8 3PH, UK

Published online: 07 February 2017

1 Introduction

Defect prediction models can be used to direct test effort to defect-prone code.[1] Latent defects can then be detected in code before the system is delivered to users. Once found, these defects can be fixed pre-delivery, at a fraction of post-delivery fix costs. Each year, defects in code cost industry billions of dollars to find and fix. Models which efficiently predict where defects are in code have the potential to save companies large amounts of money. Because the costs are so huge, even small improvements in our ability to find and fix defects can make a significant difference to overall costs. This potential to reduce costs has led to a proliferation of models which predict where defects are likely to be located in code. Hall et al. (2012) provide an overview of several hundred defect prediction models published in 208 studies.

Traditional defect prediction models comprise of four main elements. First, the model uses independent variables (or predictors) such as static code features, change data or previous defect information on which to base its predictions about the potential defect proneness of a unit of code. Second, the model is based on a specific modelling technique. Modelling techniques are mainly either machine learning (classification) or regression methods[2] (Wahono 2015). Third, dependent variables (or prediction outcomes) are produced by the model which are usually either categorical predictions (i.e. a code unit is predicted as either defect prone or not defect prone) or continuous predictions (i.e. the number of defects are predicted in a code unit). Fourth, a scheme is designed to measure the predictive performance of a model. Measures based on the confusion matrix are often used for categorical predictions and measures related to predictive error are often used for continuous predictions.

The aim of this paper is to identify classification techniques which perform well in software defect prediction. We focus on within-project prediction as this is a very common form of defect prediction. Many eminent researchers before us have also aimed to do this (e.g. Briand et al. (2002) and Lessmann et al. (2008)). Those before us have differentiated predictive performance using some form of measurement scheme. Such schemes typically calculate performance values (e.g. precision and recall ; see Table 3) to calculate an overall number representing how well models correctly predict truly defective and truly non-defective codes taking into account the level of incorrect predictions made. We go beyond this by looking underneath the numbers and at the individual defects that specific classifiers detect and do not detect. We show that, despite the overall figures suggesting similar predictive performances, there is a marked difference between four classifiers in terms of the specific defects each detects and does not detect. We also investigate the effect of prediction 'flipping' among these four classifiers. Although different classifiers can detect different subsets of defects, we show that the consistency of predictions vary greatly among the classifiers. In terms of prediction consistency, some classifiers tend to be more stable when predicting a specific software unit as defective or non-defective, hence 'flipping' less between experiment runs.

[1] Defects can occur in many software artefacts, but here, we focus only on defects found in code.

[2] In this paper, we concentrate on classification models only. Hall et al. (2012) show that about 50% of prediction models are based on classification techniques. We do this because a totally different set of analysis techniques is needed to investigate the outcomes of regression techniques. Such an analysis is beyond the scope of this paper.

Identifying the defects that different classifiers detect is important as it is well known (Fenton and Neil 1999) that some defects matter more than others. Identifying defects with critical effects on a system is more important than identifying trivial defects. Our results offer future researchers an opportunity to identify classifiers with capabilities to identify sets of defects that matter most. Panichella et al. (2014) previously investigated the usefulness of a combined approach to identifying different sets of individual defects that different classifiers can detect. We build on (Panichella et al. 2014) by further investigating whether different classifiers are equally consistent in their predictive performances. Our results confirm that the way forward in building high-performance prediction models in the future is by using ensembles (Kim et al. 2011). Our results also show that researchers should repeat their experiments a sufficient number of times to avoid the 'flipping' effect that may skew prediction performance.

We compare the predictive performance of four classifiers: Naïve Bayes, Random Forest, RPart and Support Vector Machines (SVM). These classifiers were chosen as they are widely used by the machine-learning community and have been commonly used in previous studies. These classifiers offer an opportunity to compare the performance of our classification models against those in previous studies. These classifiers also use distinct predictive techniques, and so, it is reasonable to investigate whether different defects are detected by each and whether the prediction consistency is distinct among the classifiers.

We apply these four classifiers to twelve NASA datasets,[3] three open source datasets,[4] and three commercial datasets from our industrial partner (see Table 1). NASA datasets provide a standard set of independent variables (static code metrics) and dependent variables (defect data labels). NASA data modules are at a function level of granularity. Additionally, we analyse the open source systems: Ant, Ivy, and Tomcat from the PROMISE repository (Jureczko and Madeyski 2010). Each of these datasets is at the class level of granularity. We also use three commercial telecommunication datasets which are at a method level. Therefore, our analysis includes datasets with different metrics granularity and from different software domains.

This paper extends our earlier work (Bowes et al. 2015). We build on our previous findings by adding more datasets into our experimental set-up and validating the conclusions previously made. To the NASA datasets used in Bowes et al. (2015), we add six new datasets (three open source, and three industrial datasets). Introducing more datasets to our analysis increases the diversity of code and defects in those systems. Confirming our previous findings with increased and diverse datasets provides evidence that our results may be generalisable.

The following section is an overview of defect prediction. Section 3 details our methodology. Section 4 presents results which are discussed in Section 5. We identify threats to validity in Section 6 and conclude in Section 7.

2 Background

Many studies of software defect prediction have been performed over the years. In 1999, Fenton and Neil critically reviewed a cross section of such studies (Fenton and Neil 1999). Catal and Diri (2009) mapping study identified 74 studies, and in our more recent study

[3] http://promisedata.googlecode.com/svn/trunk/defect/

[4] http://openscience.us/repo/defect/ck/

Table 1 Summary statistics for datasets before and after cleaning

Project	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)
Spacecraft instrumentation	CM1	C	20	505	505	0.0	9.5	9.5
Ground data	KC1	C++	43	2109	2096	0.6	15.4	15.5
Storage	KC3	Java	18	458	458	0.0	9.4	9.4
Management	KC4	Perl	25	125	125	0.0	48.8	48.8
Combustion	MC1	C & C++	63	9466	9277	2.0	0.7	0.7
Experiment	MC2	C	6	161	161	1.2	32.3	32.3
Zero gravity experiment	MW1	C	8	403	403	0.0	7.7	7.7
Flight software	PC1	C	40	1107	1107	0.0	6.9	6.9
for Earth	PC2	C	26	5589	5460	2.3	0.4	0.4
orbiting	PC3	C	40	1563	1563	0.0	10.2	0.0
satellites	PC4	C	36	1458	1399	4.0	12.2	12.7
	PC5	C++	164	17186	17001	1.1	3.0	3.0
Real-time predictive ground system	JM1	C	315	10878	7722	29.0	19.0	21.0
Telecommunication	PA	Java	21	4996	4996	0.0	11.7	11.7
Software	KN	Java	18	4314	4314	0.0	7.5	7.5
	HA	Java	43	9062	9062	0.0	1.3	1.3
Java build tool	Ant	Java	209	745	742	0.0	22.3	22.4
Dependency manager	Ivy	Java	88	352	352	0.0	11.4	11.4
Web server	Tomcat	Java	301	858	852	0.0	9.0	9.0

(Hall et al. 2012), we systematically reviewed 208 primary studies and showed that predictive performance varied significantly between studies. The impact that many aspects of defect models have on predictive performance have been extensively studied.

The impact that various independent variables have on predictive performance has been the subject of a great deal of research effort. The independent variables used in previous studies mainly fall into the categories of product (e.g. static code data) metrics and process (e.g. previous change and defect data) as well as metrics relating to developers. Complexity metrics are commonly used (Zhou et al. 2010), but LOC is probably the most commonly used static code metric. The effectiveness of LOC as a predictive independent variable remains unclear. Zhang (2009) reports LOC to be a useful early general indicator of defect proneness. Other studies report LOC data to have poor predictive power and is out-performed by other metrics (e.g. Bell et al. (2006)). Malhotra (2015) suggests that object-oriented metrics such as coupling between objects and response for a class are useful for defect prediction.

Several previous studies report that process data, in the form of previous history data, performs well (e.g. D'Ambros et al. (2009), Shin et al. (2009), Nagappan et al. (2010), and Madeyski and Jureczko (2015)). D'Ambros et al. (2009) specifically report that previous bug reports are the best predictors. More sophisticated process measures have also been reported to perform well (e.g. Nagappan et al. (2010)). In particular, Nagappan et al. (2010) use 'change burst' metrics with which they demonstrate good predictive performance. The few studies using developer information in models report conflicting results. Ostrand et al. (2010) report that the addition of developer information does not improve predictive performance much. Bird et al. (2009b) report better performances when developer information is used as an element within a socio-technical network of variables. Madeyski and Jureczko (2015) show that some process metrics are particularly useful for predictive modelling. For example, the number of developers changing a file can significantly improve defect prediction (Madeyski and Jureczko 2015). Many other independent variables have also been used in studies, for example Mizuno et al. (2007) and Mizuno and Kikuno (2007) use the text of the source code itself as the independent variable with promising results.

Lots of different datasets have been used in studies. However, our previous review of 208 studies (Hall et al. 2012) suggests that almost 70% of studies have used either the Eclipse dataset.[5,6] Wahono (2015) and Kamei and Shihab (2016) suggest that the NASA datasets remain the most popular for defect prediction, and also report that the PROMISE repository is used increasingly. Ease of availability mean that these datasets remain popular despite reported issues of data quality. Bird et al. (2009a) identifies many missing defects in the Eclipse data. While Gray et al. (2012), Boetticher (2006), and Shepperd et al. (2013), and Petrić et al. (2016b) raise concerns over the quality of NASA datasets in the original PROMISE repository.[7] Datasets can have a significant effect on predictive performance. Some datasets seem to be much more difficult than others to learn from. The PC2 NASA dataset seems to be particularly difficult to learn from. Kutlubay et al. (2007) and Menzies et al. (2007) both note this difficulty and report poor predictive results using these datasets. As a result, the PC2 dataset is more seldom used than other NASA datasets. Another example of datasets that are difficult to predict from are those used by Arisholm and Briand (2007) and Arisholm et al. (2010). Very low precision is reported in both of these Arisholm et al. studies (as shown in Hall et al. (2012)). Arisholm and Briand (2007) and Arisholm et al. (2010) report many good modelling practices and in some ways are exemplary studies. But these studies demonstrate how the data used can impact significantly on the performance of a model.

It is important that defect prediction studies consider the quality of data on which models are built. Datasets are often noisy. They often contain outliers and missing values that can skew results. Confidence in the predictions made by a model can be impacted by the quality of the data used while building the model. For example, Gray et al. (2012) show that defect predictions can be compromised where there is a lack of data cleaning with Jiang et al. (2009) acknowledging the importance of data quality. Unfortunately, Liebchen and Shepperd (2008) report that many studies do not seem to consider the quality of the data they use, but that small problems with data quality can have a significant impact on results.

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

[6] https://code.google.com/p/promisedata/(Menzies et al. 2012)

[7] http://promisedata.org

The features of the data also need to be considered when building a defect prediction model. In particular, repeated attributes and related attributes have been shown to bias the predictions of models. The use of feature selection on sets of independent variables seems to improve the performance of models (e.g. Shivaji et al. (2009), Khoshgoftaar et al. (2010), Bird et al. (2009b), and Menzies et al. (2007)).

Data balance is also an important factor in defect prediction and has been considered by previous studies. This is important as substantially imbalanced datasets are commonly used in defect prediction studies (i.e. there are usually many more non-defective units than defective units) (Bowes et al. 2013; Myrtveit et al. 2005). An extreme example of this is seen in NASA dataset PC2, which has only 0.4% of datapoints belonging to the defective class (23 out of 5589 datapoints). Imbalanced data can strongly influence both the training of a model, and the suitability of performance metrics. The influence data imbalance has on predictive performance varies from one classifier to another. For example, C4.5 decision trees have been reported to struggle with imbalanced data (Chawla et al. 2004; Arisholm and Briand 2007; Arisholm et al. 2010), whereas fuzzy-based classifiers have been reported to perform robustly regardless of class distribution (Visa and Ralescu 2004). Data balancing has shown positive effects when used with Random Forest (Chen et al. 2014); however, there is a risk of over-fitting (Gray et al. 2012). On the other hand, data balancing has demonstrated no significant effect on performance when used with some other techniques (e.g. Naïve Bayes (Rodriguez et al. 2014)). Studies specifically investigating the impact of defect data balance and proposing techniques to deal with it include, for example, Khoshgoftaar et al. (2010), Shivaji et al. (2009), and Seiffert et al. (2009). Gao et al. (2015) investigate the combination of feature selection and data balancing techniques. Particularly, Gao et al. (2015) experiment with changing the order of the two techniques, using feature selection followed by data balancing (separately using sampled and unsampled data instances), and vice versa. They show that sampling performed prior to feature selection by keeping the unsampled data can boost prediction performance more than the other two approaches (Gao et al. 2015).

Classifiers are mathematical techniques for building models which can then predict dependent variables (defects). Defect prediction has frequently used trainable classifiers (Wahono 2015). Trainable classifiers build models using training data which has items composed of both independent and dependant variables. There are many classification techniques that have been used in previous defect prediction studies. Witten (2005) explain classification techniques in detail and Lessmann et al. (2008) summarise the use of 22 such classifiers for defect prediction. Ensembles of classifiers are also used in prediction (Minku and Yao 2012; Sun et al. 2012; Laradji et al. 2015; Petrić et al. 2016a). Ensembles are collections of individual classifiers trained on the same data and combined to perform a prediction task. An overall prediction decision is made by the ensemble based on the predictions of the individual models. Majority voting is a decision-making strategy commonly used by ensembles. Although not yet widely used in defect prediction, ensembles have been shown to significantly improve predictive performance. For example, Mısırlı et al. (2011) combine the use of Artificial Neural Networks, Naïve Bayes and Voting Feature Intervals and report improved predictive performance over the individual models. Ensembles have been more commonly used to predict software effort estimation (e.g. Minku and Yao (2013)) where their performance has been reported as sensitive to the characteristics of datasets (Chen and Yao 2009; Shepperd and Kadoda 2001).

Many defect prediction studies individually report the comparative performance of the classification techniques they have used. Mizuno and Kikuno (2007) report that, of the techniques they studied, Orthogonal Sparse Bigrams Markov models (OSB) are best suited to defect prediction. Bibi et al. (2006) report that Regression via Classification works well.

Khoshgoftaar et al. (2002) report that modules whose defect proneness is predicted as uncertain, can be effectively classified using the TreeDisc technique. Our own analysis of the results from 19 studies (Hall et al. 2012) suggests that Naïve Bayes and Logistic regression techniques work best. However, overall, there is no clear consensus on which techniques perform best. Several influential studies have performed large-scale experiments using a wide range of classifiers to establish which classifiers dominate. In Arisholm et al. (2010) systematic study of the impact that classifiers, metrics and performance measures have on predictive performance, eight classifiers were evaluated. Arisholm et al. (2010) report that the classifier technique had limited impact on predictive performance. Lessmann et al. (2008) large-scale comparison of predictive performance across 22 classifiers over 10 NASA datasets showed no significant performance differences among the top 17 classifiers.

In general, defect prediction studies do not consider individual defects that different classifiers predict or do not predict. Panichella et al. (2014) is an exception to this reporting a comprehensive empirical investigation into whether different classifiers find different defects. Although predictive performances among the classifiers in their study were similar, they showed that different classifiers detect different defects. Panichella et al. proposed CODEP which uses an ensemble technique (i.e. stacking Wolpert (1992)) to combine multiple learners in order to achieve better predictive performances. The CODEP model showed superior results when compared to single models. However, Panichella et al. conducted a cross-project defect prediction study which differs from our study. Cross-project defect prediction has an experimental set-up based on training models on multiple projects and then tested on one project (explanatory studies on cross-project defect prediction were done by Turhan et al. (2009) and Zimmermann et al. (2009)). Consequently, in cross-project defect prediction studies, the multiple execution of experiments is not required. Contrary, in within-project defect prediction studies, experiments are frequently done using cross-validation techniques. To get more stabilised and generalised results, experiments based on cross validation are repeated multiple times. As a drawback of executing experiments multiple times, the prediction consistency may not be stable resulting in classifiers 'flipping' between experimental runs. Therefore, in a within-project analysis, prediction consistency should also be taken into account.

Our paper further builds on Panichella et al. in a number of other ways. Panichella et al. conducted an analysis only at a class level while our study is additionally extended to a module level (i.e. the smallest unit of functionality, usually a function, procedure or method). Panichella et al. also consider regression analysis where probabilities of a module being defective are calculated. Our study deals with classification where a module is labelled either as defective or non-defective. Therefore, the learning algorithms used in each study differ. We also show full performance figures by presenting the numbers of true positives, false positives, true negative and false negatives for each classifier.

Predictive performance in all previous studies is presented in terms of a range of performance measures (see the following sub-sections for more details of such measures). The vast majority of predictive performances were reported to be within the current performance ceiling of 80% recall identified by Menzies et al. (2008). However, focusing only on performance figures, without examining the individual defects that individual classifiers detect, is limiting. Such an approach makes it difficult to establish whether specific defects are consistently missed by all classifiers, or whether different classifiers detect different subsets of defects. Establishing the set of defects each classifier detects, rather than just looking at the overall performance figure, allows the identification classifier ensembles most likely to detect the largest range of defects.

Table 2 Confusion matrix

The confusion matrix is in many ways analogous to residuals for regression models. It forms the fundamental basis from which almost all other performance statistics are derived.

	Predicted defective	Predicted defect free
Observed defective	True positive (TP)	False negative (FN)
Observed defect free	False positive (FP)	True negative (TN)

Studies present the predictive performance of their models using some form of measurement scheme. Measuring model performance is complex and there are many ways in which the performance of a prediction model can be measured. For example, Menzies et al. (2007) use pd and pf to highlight standard predictive performance, while Mende and Koschke (2010) use $Popt$ to assess effort-awareness. The measurement of predictive performance is often based on a confusion matrix (shown in Table 2). This matrix reports how a model classified the different defect categories compared to their actual classification (predicted versus observed). Composite performance measures can be calculated by combining values from the confusion matrix (see Table 3).

There is no one best way to measure the performance of a model. This depends on the 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. Zhou et al. (2010) 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. (2010) also discuss how model performance varies depending on how it is measured. The different performance measurement schemes used mean that directly comparing the performance reported by individual studies is difficult and potentially misleading. Comparisons cannot compare like with like as there is no adequate point of comparison. To allow such comparisons, we previously developed a tool to transform a variety of reported predictive performance measures back to a confusion matrix (Bowes et al. 2013).

Table 3 Composite performance measures

Construct	Defined as	Description
Recall pd (probability of detection) sensitivity true positive rate	$TP/(TP+FN)$	Proportion of defective units correctly classified
Precision	$TP/(TP+FP)$	Proportion of units correctly predicted as defective
F-measure	$\frac{2 \cdot Recall \cdot Precision}{Recall+Precision}$	Most commonly defined as the harmonic mean of precision and recall
Matthews correlation coefficient	$\frac{TP \times TN - FP \times FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}}$	Combines all quadrants of the binary confusion matrix to produce a value in the range -1 to +1 with 0 indicating random correlation between the prediction and the recorded results. MCC can be tested for statistical significance, with $\chi^2 = N \cdot MCC^2$ where N is the total number of instances.

3 Methodology

3.1 Classifiers

We have chosen four different classifiers for this study: Naïve Bayes, RPart, SVM and Random Forest. These four classifiers were chosen because they build models based on different mathematical properties. Naïve Bayes produces models based on the combined probabilities of a dependent variable being associated with the different categories of the dependent variables. Naïve Bayes requires that both the dependent and independent variables are categorical. RPart is an implementation of a technique for building Classification and Regression Trees (CaRT). RPart builds a decision tree based on the information entropy (uniformity) of the subsets of training data which can be achieved by splitting the data using different independent variables. SVMs build models by producing a hyper-plane which can separate the training data into two classes. The items (vectors) which are closest to the hyper-plane are used to modify the model with the aim of producing a hyper-plane which has the greatest average distance from the supporting vectors. Random Forest is an ensemble technique. It is built by producing many CaRTs, each with samples of the training data having a subset of features. Bagging is also used to improve the stability of the individual trees by creating training sets produced by sampling the original training data with replacement. The final decision of the ensemble is determined by combining the decisions of each tree and computing the modal value.

The different methods of building a model by each classifier may lead to differences in the items predicted as defective. Naïve Bayes is purely probabilistic and each independent variable contributes to a decision. RPart may use only a subset of independent variables to produce the final tree. The decisions at each node of the tree are linear in nature and collectively put boundaries around different groups of items in the original training data. RPart is different to Naïve Bayes in that the thresholds used to separate the groups are different at each node compared to Naïve Bayes which decides the threshold to split continuous variables before the probabilities are determined. SVMs use mathematical formulae to build nonlinear models to separate the different classes. The model is therefore not derived from decisions based on individual independent variables, but on the ability to find a formula which separates the data with the least amount of false negatives and false positives.

Classifier tuning is an important part of building good models. As described above, Naïve Bayes requires all variables to be categorical. Choosing arbitrary threshold values to split a continuous variable into different groups may not produce good models. Choosing good thresholds may require many models to be built on the training data using different threshold values and determining which produces the best results. Similarly for RPart, the number of items in the leaf nodes of a tree should not be so small that a branch is built for every item. Finding the minimum number of items required before branching is an important process in building good models which do not overfit on the training data and then do not perform as well on the test data. Random Forest can be tuned to determine the most appropriate number of trees to use in the forest. Finally SVMs are known to perform poorly if they are not tuned (Soares et al. 2004). SVMs can use different kernel functions to produce the complex hyper-planes needed to separate the data. The radial-based kernel function has two parameters: C and γ, which need to be tuned in order to produce good models.

In practice, not all classifiers perform significantly better when tuned. Both Naïve Bayes and RPart can be tuned, but the default parameters and splitting algorithms are known to work well. Random Forest and particularly SVMs do require tuning. For Random Forest we tuned the number of trees from 50 to 200 in steps of 50. For SVM using a radial base

function, we tuned γ from 0.25 to 4 and C from 2 to 32. In our experiment, tuning was carried out by splitting the training data into 10 folds, 9 folds were combined together to build models with the parameters and the 10th fold was used to measure the performance of the model. This was repeated with each fold being held out in turn. The parameters which produced the best average performance we used to build the final model on the entire training data.

3.2 Datasets

We used the NASA datasets first published on the now defunct MDP website.[8] This repository consists of 13 datasets from a range of NASA projects. In this study, we use 12 of the 13 NASA datasets. JM1 was not used because during cleaning, 29% of data was removed suggesting that the quality of the data may have been poor. We extended our previous analysis (Bowes et al. 2015) by using 6 additional datasets, 3 open source and 3 commercial. All 3 open source datasets are at class level, and originate from the PROMISE repository. The commercial datasets are all in the telecommunication domain and are at method level. A summary of each dataset can be found in Table 1. Our choice of datasets is based on several factors. First, the NASA and PROMISE datasets are frequently used in defect prediction. Second, three open source datasets in our analysis (*Ant*, *Ivy*, and *Tomcat*) are very different in nature, and they could have a variety of different defects. Commercial datasets also add to the variety of very different datasets. We use these factors to enhance the possibility of generalising our results, which is one of the major contributions of this paper.

The data quality of the original NASA MDP datasets can be improved (Boetticher 2006; Gray et al. 2012; Shepperd et al. 2013). Gray et al. (2012), Gray (2013), and Shepperd et al. (2013) describe techniques for cleaning the data. Shepperd has provided a 'cleaned' version of the MDP datasets,[9] However, full traceability back to the original items is not provided. Consequently we did not use Shepperd's cleaned NASA datasets. Instead we cleaned the NASA datasets ourselves. We carried out the following data cleaning stages described by Gray et al. (2012): Each independent variable was tested to see if all values were the same; if they were, this variable was removed because they contained no information which allows us to discriminate defective items from non-defective items. The correlation for all combinations of two independent variables was found; if the correlation was 1, the second variable was removed. Where the dataset contained the variable 'DECISION DENSITY', any item with a value of 'na' was converted to 0. The 'DECISION_DENSITY' was also set to 0 if 'CONDITION_COUNT'=0 and 'DECISION_COUNT'=0. Items were removed if:

1. HALSTEAD_LENGTH!= NUM_OPERANDS+NUM_OPERATORS
2. CYCLOMATIC_COMPLEXITY> 1+NUM_OPERATORS
3. CALL_PAIRS> NUM_OPERATORS

Our method for cleaning the NASA data also differs from Shepperd et al. (2013) because we do not remove items where the executable lines of code is zero. We did not do this because we have not been able to determine how the NASA metrics were computed and it is possible to have zero executable lines in Java interfaces. We performed the same cleaning to our commercial datasets. We performed cleaning of the open source datasets for which

[8] http://mdp.ivv.nasa.gov – unfortunately now not accessible

[9] http://nasa-softwaredefectdatasets.wikispaces.com

we defined a similar set of rules as described above, for data at a class level. Particularly, we removed items if:

1. AVERAGE_CYCLOMATIC_COMPLEXITY> MAXIMAL_CYCLOMATIC_COMPLEXITY
2. NUMBER_OF_COMMENTS> LINES_OF_CODE
3. PUBLIC_METODS_COUNT> CLASS_METHODS_COUNT

Since all the NASA and open source datasets are publicly available, the aforementioned cleaning steps can be applied to them. We do not provide pre-cleaned data as we believe it is vital that researchers do their own cleaning. Problems with poor-quality data being used have prolificated because researchers have taken pre-cleaned data without questioning their quality (for example NASA datasets (Gray et al. 2011; Shepperd et al. 2013)). Cleaning the data we use is straightforward to future researchers as the cleaning steps are easy to understand and implement.

3.3 Experimental Set-Up

The following experiment was repeated 100 times. Experiments are more commonly repeated 10 times. We chose 100 repeats because Mende (2011) reports that using 10 experiment repeats results in an unreliable final performance figure. Each dataset was split into 10 stratified folds. Each fold was held out in turn to form a test set and the other folds were combined and randomised (to reduce ordering effects) to produce the training set. Such stratified cross validation ensures that there are instances of the defective class in each test set, so reduces the likelihood of classification uncertainty. Re-balancing of the training set is sometimes carried out to provide the classifier with a more representative sample of the infrequent defective instances. Data balancing may have very different effects on an experiment depending on the classifier used (as explained in Section 2). To reduce the confounding factors of data balance, we do not apply this technique in our experiment. Specifically, it would be difficult to control the impact of data balance on performance across the range of classifiers we used. Also, our experiment is focused on the dispersion of individual predictions based on real data across classifiers, rather than investigating whether and how re-balancing affects defect prediction results. Furthermore, data balancing is infrequently used in defect prediction studies. For each training/testing pair, four different classifiers were trained using the same training set. Where appropriate, a grid search was performed to identify optimal meta-parameters for each classifier on the training set. The model built by each classifier was used to classify the test set.

To collect the data showing individual predictions made by individual classifiers, the RowID, DataSet, runid, foldid and classified label (defective or not defective) was recorded for each item in the test set for each classifier and for each cross-validation run.

We calculate predictive performance values using two different measures: f-measure and MCC (see Table 3). F-measure was selected because it is very commonly used by published studies and allows us to easily compare the predictive performance of our models against previous models. Additionally, f-measure gives the harmonic mean of both measures, precision and recall. It has a range of 0 to 1. MCC was selected because it is relatively easy to understand with a range from -1 to +1. MCC has the added benefit that it encompasses all four components of the confusion matrix whereas f-measure ignores the proportion of true negatives. Furthermore, Matthews' Correlation Coefficient (MCC) has been demonstrated to be a reliable measure of predictive model performance (Shepperd et al. 2014). The results for each combination of classifier and dataset were further analysed by calculating for each

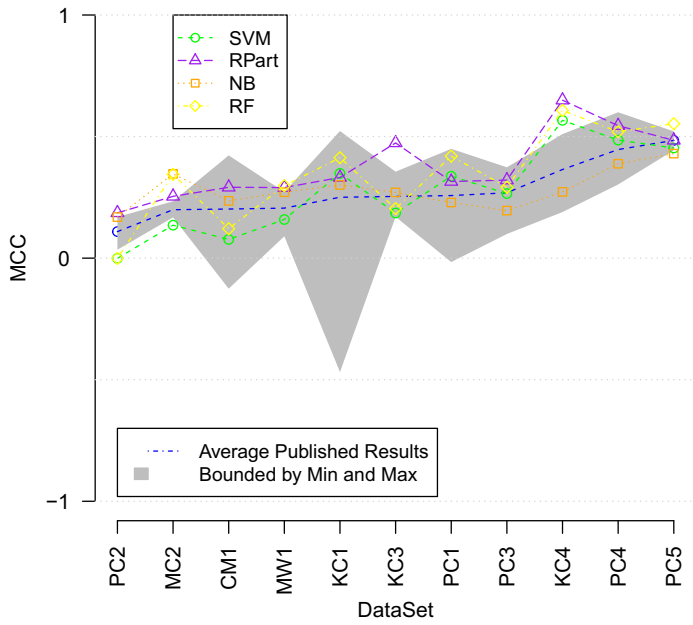


Fig. 1 Our results compared to results published by other studies

item the frequency of being classified as defective. The results were then categorised by the original label for each item so that we can see the difference between how the models had classified the defective and non-defective items.

4 Results

We aim to investigate variation in the individual defects and prediction consistency produced by the four classifiers. To ensure the defects that we analyse are reliable, we first checked that our models were performing satisfactorily. To do this, we built prediction models using the NASA datasets. Figure 1 compares the MCC performance of our models against 600 defect prediction performances reported in published studies using these NASA datasets Hall et al. (2012).[10] We re-engineered MCC from the performance figures reported in these previous studies using DConfusion. This is a tool we developed for transforming a variety of reported predictive performance measures back to a confusion matrix. DConfusion is described in (Bowes et al. 2013). Figure 1 shows that the performances of our four classifiers are generally in keeping with those reported by others. Figure 1 confirms that some datasets are notoriously difficult to predict. For example, few performances for PC2 are better than random. Whereas, very good predictive performances are generally reported for PC5 and KC4. The RPart and Naïve Bayes classifiers did not perform as well on the NASA datasets as on our commercial datasets (as shown in Table 4). However, all our commercial datasets are highly imbalanced, where learning from a small set of defective items becomes more difficult, so this imbalance may explain the difference in the way these two classifiers perform. Similarly the SVM classifier performs better on the open source datasets than it does on the NASA datasets. The SVM classifier seems to perform particularly poorly when

[10]Dataset MC1 is not included in the figure because none of the studies we had identified previously used this dataset.

Table 4 MCC performance for all datasets by classifier

	NASA datasets		OSS datasets		Commercial datasets		All datasets	
Classifier	Average	StDev	Average	StDev	Average	StDev	Average	StDev
SVM	0.291	0.188	0.129	0.134	0.314	0.140	0.245	0.154
RPart	0.331	0.162	0.323	0.077	0.166	0.148	0.273	0.129
NB	0.269	0.083	0.322	0.089	0.101	0.040	0.231	0.071
RF	0.356	0.184	0.365	0.095	0.366	0.142	0.362	0.140

used on extremely imbalanced datasets (especially the case when datasets have less than 10% faulty items).

We investigated classifier performance variation across all the datasets. Table 4 shows little overall difference in average MCC performance across the four classifiers, except Random Forest, which usually performs best (Lessmann et al. 2008). By using Friedman's non-parametric test at the significance level 0.05, we formally confirmed no statistically significant difference in the MCC performance values across all datasets when applied amongst SVM, Naïve Bayes, and RPart classifiers (p-value: 0.939). Similarly, by using the same procedure, we established that there is a statistically significant difference in terms of MCC performance when Random Forest is added to the statistical test (p value: 0.021). However, these overall performance figures mask a range of different performances by classifiers when used on individual datasets. For example, Table 5 shows Naïve Bayes performing relatively well when used on the Ivy and KC4 datasets, however, much worse on the KN dataset. On the other hand, SVM achieves the highest MCC performance across all classifiers on the KN dataset, but poor performance values on the Ivy dataset.[11] By repeating Friedman's non-parametric statistical test on the three datasets reported in Table 5, across all runs, we confirmed a statistically significant difference amongst the four classifiers, with the p value less than 0.0001 in the cases where Random Forest was included or excluded from the test.

Having established that our models were performing acceptably (comparable to the 600 models reported in Hall et al. (2012) and depicted in Fig. 1), we next wanted to identify the particular defects that each of our four classifiers predicts so that we could identify variations in the defects predicted by each. We needed to be able to label each module as either containing a predicted defect (or not) by each classifier. As we used 100 repeated 10-fold cross-validation experiments, we needed to decide on a prediction threshold at which we would label a module as either predicted defective (or not) by each classifier, i.e. how many of these 100 runs must have predicted that a module was defective before we labelled it as such. We analysed the labels that each classifier assigned to each module for each of the 100 runs. There was a surprising amount of prediction 'flipping' between runs. On some runs, a module was labelled as defective and other runs not. There was variation in the level of prediction flipping amongst the classifiers. Table 7 shows the overall label 'flipping' between the classifiers.

Table 6 divides predictions between the actual defective and non-defective labels (i.e. the known labels for each module) for each of our dataset category, namely NASA, commercial (Comm), and open source dataset (OSS), respectively. For each of these two categories, Table 6 shows three levels of label flipping: never, 5% and 10%. For example, a value of

[11] Performance tables for all datasets are available from https://sag.cs.herts.ac.uk/?page_id=235

Table 5 Performance measures for KC4, KN and Ivy

	KC4		KN		Ivy	
Classifier	MCC	F-measure	MCC	F-measure	MCC	F-measure
SVM	0.567	0.795	0.400	0.404	0.141	0.167
RPart	0.650	0.825	0.276	0.218	0.244	0.324
NB	0.272	0.419	0.098	0.170	0.295	0.375
RF	0.607	0.809	0.397	0.378	0.310	0.316

defective items flipping $Never = 0.717$ would indicate that 71.7% of defective items never flipped, a value of defective items flipping $< 5\% = 0.746$ would indicate that 74.6% of defective items flipped less than 5% of the time. Table 7 suggests that non-defective items had a more stable prediction than defective items across all datasets. Although Table 7 shows the average numbers of prediction flipping across all datasets, this statement is valid for all of our dataset categories as shown in Table 6. This is probably because of the imbalance of data. Since there is more non-defective items to learn from, predictors could be better trained to predict them and hence flip less. Although the average numbers do not indicate much flipping between modules being predicted as defective or non-defective, these tables show datasets together, and so, the low flipping in large datasets masks the flipping that occurs in individual datasets.

Table 8 shows the label flipping variations during the 100 runs between datasets.[12] For some datasets, using particular classifiers results in a high level of flipping (prediction uncertainty). For example, Table 8 shows that using Naïve Bayes on KN results in prediction uncertainty, with 73% of the predictions for known defective modules flipping at least once between being predicted defective to predicted non-defective between runs. Table 8 also shows the prediction uncertainty of using SVM on the KC4 dataset with only 26% of known defective modules being consistently predicted as defective or not defective across all cross-validation runs. Figure 2 presents violin plots showing the flipping for the four different classifiers on KC4 in more detail.[13] The violin plots show the flipping that occurs for each quadrant of the confusion matrix. The y-axis represents the probability of a module to flip, where the central part of a 'violin' represents the 50% chance of flipping, reducing towards no flipping at the ends of the 'violin'. The x-axis demonstrates the proportion of modules that flip, where a wide 'violin' indicates a high proportion of modules, and a narrow 'violin' represents a small number of modules. For example, Fig. 2 shows that SVM is particularly unstable when predicting both, defective and non-defective modules for KC4, compared to the other classifiers. The reason for that is the wider 'violin' body around the 50% probability of flipping. On the other hand, Naïve Bayes shows the greatest stability when predicting non-defective instances since the majority of modules are concentrated closer to the ends of the 'violin'. RPart provides relatively stable predictions for defective instances when used on the KC4 dataset. As a result of analysing these labelling variations between runs, we decided to label a module as having been predicted as either defective or not defective if it had been predicted as such on more than 50 runs. Using a threshold of 50 is the equivalent of choosing the label based on the balance of probability.

[12] Label flipping tables for all datasets are available from https://sag.cs.herts.ac.uk/?page_id=235.

[13] Violin plots for all datasets are available from https://sag.cs.herts.ac.uk/?page_id=235

Table 6 Frequency of all items flipping across different dataset categories

		Non-defective items			Defective items		
	Classifier	Never	<5 %	<10 %	Never	<5 %	<10 %
NASA	SVM	0.983	0.985	0.991	0.717	0.746	0.839
	RPart	0.972	0.972	0.983	0.626	0.626	0.736
	NB	0.974	0.974	0.987	0.943	0.943	0.971
	RF	0.988	0.991	0.993	0.748	0.807	0.859
Comm	SVM	0.959	0.967	0.974	0.797	0.797	0.797
	RPart	0.992	0.992	0.995	0.901	0.901	0.901
	NB	0.805	0.805	0.879	0.823	0.823	0.823
	RF	0.989	0.992	0.995	0.897	0.897	0.897
OSS	SVM	0.904	0.925	0.942	0.799	0.799	0.799
	RPart	0.850	0.850	0.899	0.570	0.570	0.570
	NB	0.953	0.953	0.971	0.924	0.924	0.924
	RF	0.958	0.970	0.975	0.809	0.809	0.809

Having labelled each module as being predicted or not as defective by each of the four classifiers, we constructed set diagrams to show which defects were identified by which classifiers. Figures 3–5 show set diagrams for all dataset categories, divided in groups for NASA datasets, open source datasets, and commercial datasets, respectively. Figure 3 shows a set diagram for the 12 frequently used NASA datasets together. Each figure is divided into the four quadrants of a confusion matrix. The performance of each individual classifier is shown in terms of the numbers of predictions falling into each quadrant. Figures 3–5 show similarity and variation in the actual modules predicted as either defective or not defective by each classifier. Figure 3 shows that 96 out of 1568 defective modules are correctly predicted as defective by all four classifiers (only 6.1%). Very many more modules are correctly identified as defective by individual classifiers. For example, Naïve Bayes is the only classifier to correctly find 280 (17.9%) defective modules and SVM is the only classifier to correctly locate 125 (8.0%) defective modules (though such predictive performance must always be weighed against false positive predictions). Our results suggest that using only a Random Forest classifier would fail to predict many (526 (34%)) defective modules. Observing Figs. 4 and 5 we came to similar conclusions. In the case of the open source datasets, 55 out of 283 (19.4%) unique defects were identified by either Naïve Bayes or SVM. Many more unique defects were found by individual classifiers in the commercial datasets, precisely 357 out of 1027 (34.8%).

Table 7 Frequency of all item flipping in all datasets

	Non-defective items			Defective items		
Classifier	Never	<5%	<10%	Never	<5%	<10%
SVM	0.949	0.959	0.969	0.771	0.781	0.812
RPart	0.938	0.938	0.959	0.699	0.699	0.736
NB	0.911	0.911	0.945	0.897	0.897	0.906
RF	0.978	0.984	0.988	0.818	0.838	0.855

Table 8 Frequency of flipping for three different datasets

		Non-defective items			Defective items		
	Classifier	Never	<5 %	<10 %	Never	<5 %	<10 %
KC4	SVM	0.719	0.734	0.828	0.262	0.311	0.443
	RPart	0.984	0.984	1.000	0.902	0.902	0.984
	NB	0.938	0.938	0.984	0.885	0.885	0.934
	RF	0.906	0.938	0.953	0.803	0.820	0.918
KN	SVM	0.955	0.964	0.971	0.786	0.817	0.854
	RPart	0.993	0.993	0.997	0.888	0.888	0.929
	NB	0.491	0.491	0.675	0.571	0.571	0.730
	RF	0.988	0.991	0.994	0.919	0.922	0.957
Ivy	SVM	0.913	0.949	0.962	0.850	0.850	0.875
	RPart	0.837	0.837	0.881	0.625	0.625	0.700
	NB	0.933	0.933	0.952	0.950	0.950	1.000
	RF	0.955	0.974	0.981	0.900	0.925	0.950

There is much more agreement between classifiers about non-defective modules. In the true negative quadrant, Fig. 3 shows that all four classifiers agree on 35364 (93.1%) out of 37987 true negative NASA modules. Though again, individual non-defective modules are located by specific classifiers. For example, Fig. 3 shows that SVM correctly predicts 100 non-defective NASA modules that no other classifier predicts. The pattern of module predictions across the classifiers varies slightly between the datasets. Figures 6, 7 and 8 show set diagrams for individual datasets, KC4, KN and Ivy. Particularly, Fig. 6 shows a set diagram for the KC4 dataset.[14] KC4 is an interesting dataset. It is unusually balanced between defective and non-defective modules (64 v 61). It is also a small dataset (only 125 modules). Figure 6 shows that for KC4 Naïve Bayes behaves differently compared to how it behaves for the other datasets. In particular for KC4 Naïve Bayes is much less optimistic (i.e. it predicts only 17 out of 125 modules as being defective) in its predictions than it is for the other datasets. RPart was more conservative when predicting defective items than non-defective ones. For example, in the KN dataset, RPart is the only classifier to find 17 (5.3%) unique non-defective items as shown on Fig. 8.

5 Discussion

Our results suggest that there is uncertainty in the predictions made by classifiers. We have demonstrated that there is a surprising level of prediction flipping between cross-validation runs by classifiers. This level of uncertainty is not usually observable as studies normally only publish average final prediction figures. Few studies concern themselves with the results of individual cross-validation runs. Elish and Elish (2008) is a notable exception to this, where the mean and the standard deviation of the performance values across all runs are reported. Few studies run experiments 100 times. More commonly, experiments are run only 10 times (e.g. Lessmann et al. (2008); Menzies et al. (2007)). This means that the level of

[14] Set diagrams for all datasets can be found at https://sag.cs.herts.ac.uk/?page_id=235

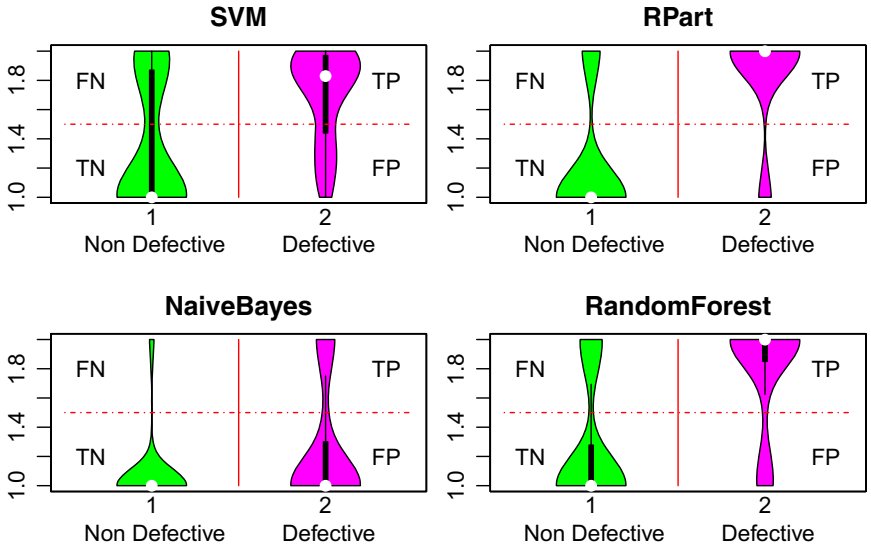


Fig. 2 Violin plot of frequency of flipping for KC4 dataset

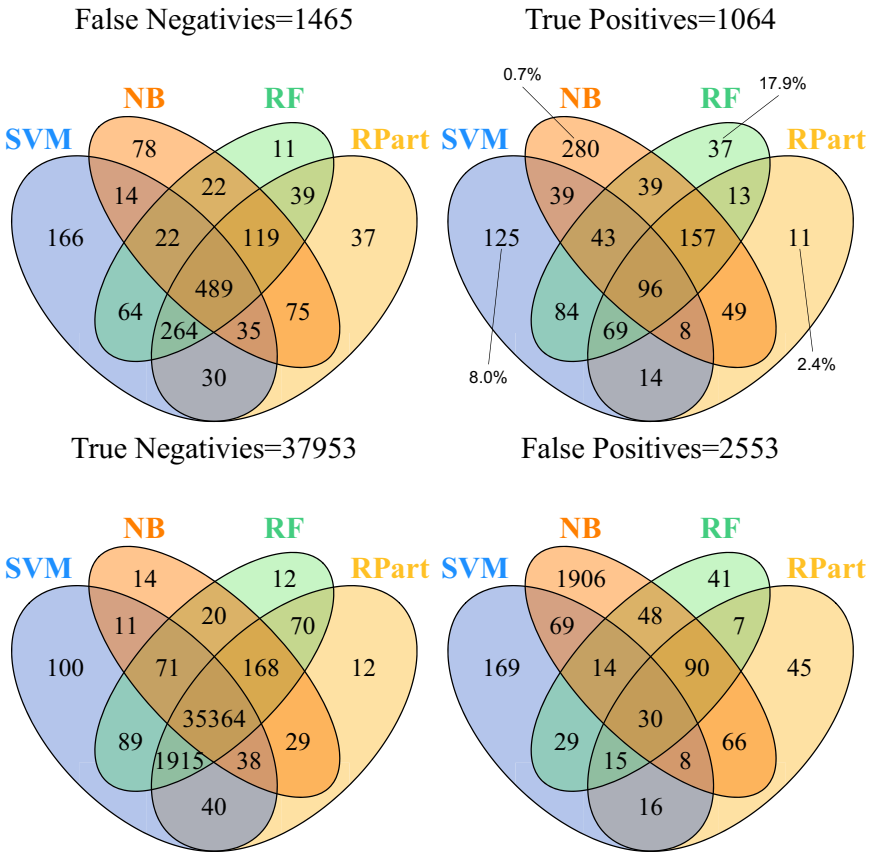


Fig. 3 Sensitivity analysis for all NASA datasets using different classifiers. $n = 37987$; $p = 1568$

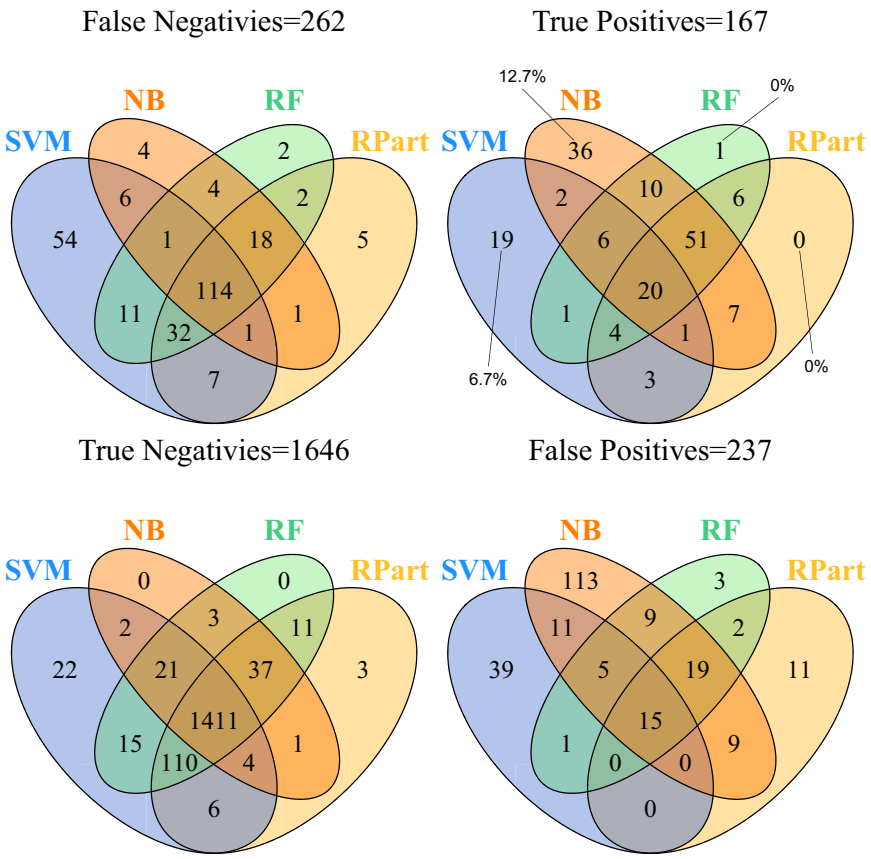


Fig. 4 Sensitivity analysis for all open source datasets using different classifiers. n = 1663; p = 283

prediction flipping between runs is likely to be artificially reduced. We suspect that prediction flipping by a classifier for a dataset is caused by the random generation of the folds. The items making up the individual folds determine the composition of the training data and the model that is built. The larger the dataset, the less prediction flipping occurs. This is likely to be because larger datasets may have training data that is more consistent with the entire dataset. Some classifiers are more sensitive to the composition of the training set than other classifiers. SVM is particularly sensitive for KC4 where 26% of non-defective items flip at least once and 44% of defective items flip. Although SVM performs well (MCC = 0.567), the items it predicts as being defective are not consistent across different cross-validation runs. A similar situation is observed with Ant, where the level of flipping for Rpart is 63% while maintaining a reasonable performance (MCC = 0.398). However, the reasons for such prediction uncertainty remain unknown and investigating the cause of this uncertainty is beyond the scope of this paper. Future research is also needed to use our results on flipping to identify the threshold at which overall defective or not defective predictions should be determined.

The level of uncertainty among classifiers may be valuable for practitioners in different domains of defect predictions. For instance, where stability of prediction plays a significant role, our results suggest that on average, Naïve Bayes would be the most suitable selection. On the other hand, learners such as RPart may be avoided in applications where higher prediction consistency is needed. The reasons for this prediction inconsistency are

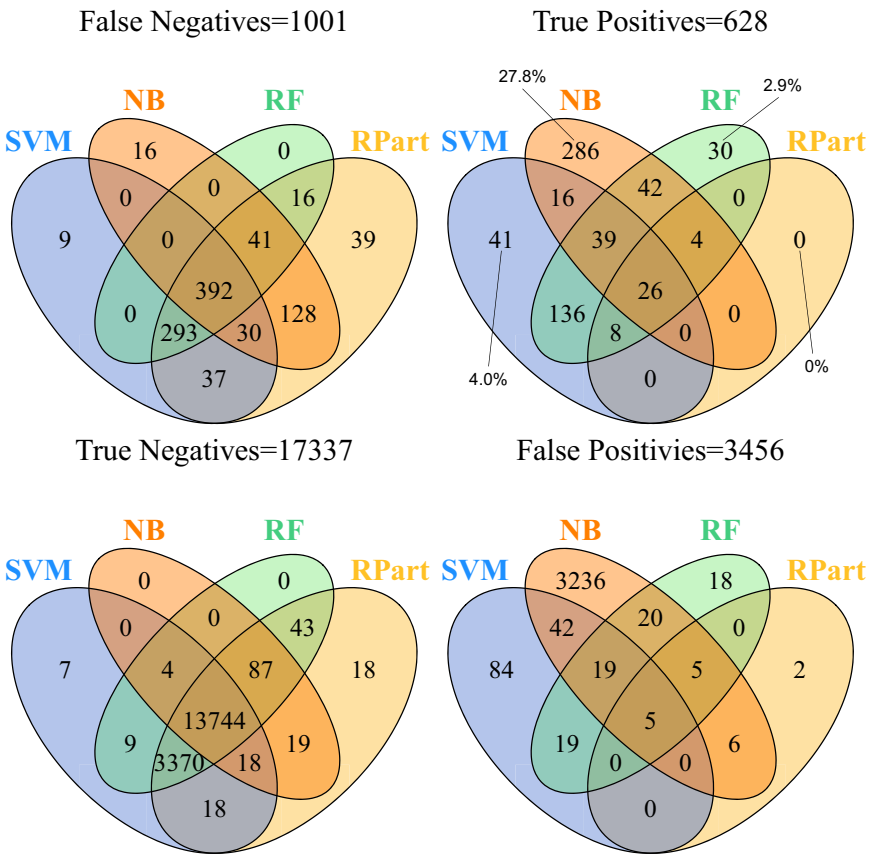


Fig. 5 Sensitivity analysis for all commercial datasets using different classifiers. n = 17344; p = 1027

yet to be established. More classifiers with different properties should also be investigated to establish the extent of uncertainty in predictions.

Other large-scale studies comparing the performance of defect prediction models show that there is no significant difference between classifiers (Arisholm et al. 2010; Lessmann et al. 2008). Our overall MCC values for the four classifiers we investigate also suggest performance similarity. Our results show that specific classifiers are sensitive to dataset and that classifier performance varies according to dataset. For example, our SVM model performs poorly on Ivy but performs much better on KC4. Other studies have also reported sensitivity to dataset (e.g. Lessmann et al. (2008)).

Similarly to Panichella et al. (2014), our results also suggest that overall performance figures hide a variety of differences in the defects that each classifier predicts. While overall performance figures between classifiers are similar, very different subsets of defects are actually predicted by different classifiers. So, it would be wrong to conclude that, given overall performance values for classifiers are similar, it does not matter which classifier is used. Very different defects are predicted by different classifiers. This is probably not surprising given that the four classifiers we investigate approach the prediction task using very different techniques. From each category of system in our analysis, we observe a considerable number of defects predicted by a single classifier. Overall, the NASA category contains 43%, Comm 57% and OSS 34% of unique defects predicted by only one classifier. Future work is needed to investigate whether there is any similarity in the characteristics of

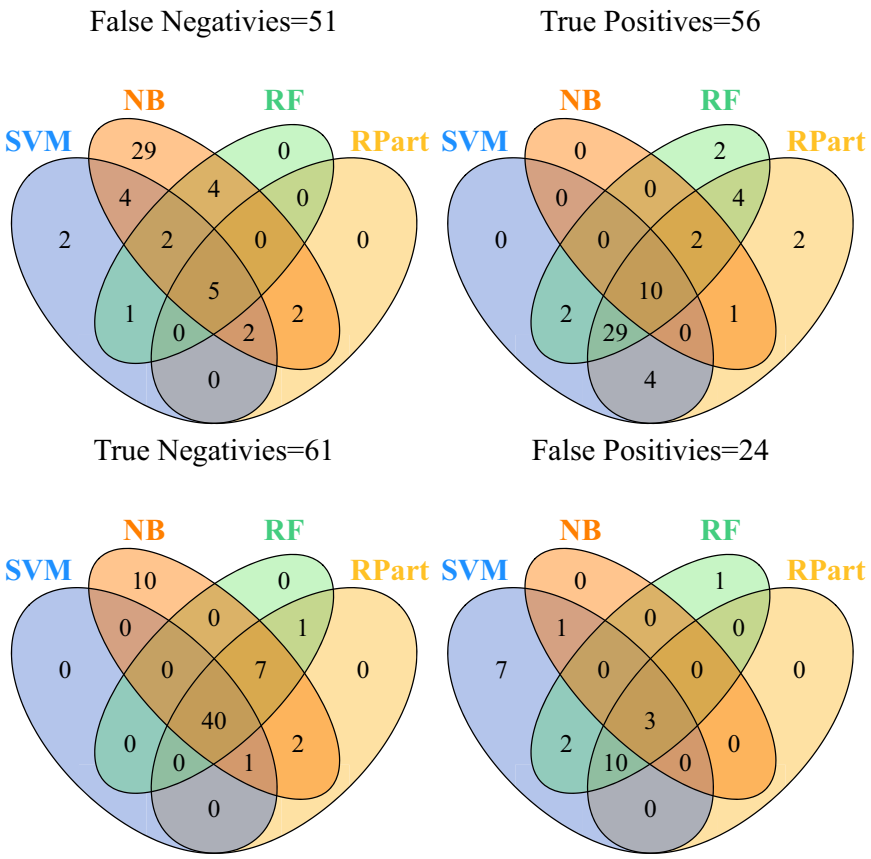


Fig. 6 Sensitivity analysis for KC4 using different classifiers. $n = 64$; $p = 61$

the set of defects that each classifier predicts. Currently, it is not known whether particular classifiers specialise in predicting particular types of defect.

Our results strongly suggest the use of classifier ensembles. It is likely that a collection of heterogeneous classifiers offer the best opportunity to predict defects. Future work is needed to extend our investigation and identify which set of classifiers perform best in terms of prediction performance and consistency. This future work also needs to identify whether a global ensemble could be identified or whether effective ensembles remain local to the dataset. Our results also suggest that ensembles should not use the popular majority voting approach to deciding on predictions. Using this decision-making approach will miss the unique subsets of defects that individual classifiers predict. Such understanding has previously not been obvious since only average overall performance figures for different classifiers have been reported. Our results now support Kim et al. (2011)'s recommendations on the use of classifier ensembles, and we, in addition, provide better understanding about ensemble design. One way forward in building future prediction models could be stacking ensembles. The stacking approach does not base its predictions on voting, but rather uses an additional classifier to make the final prediction. Our recent study has shown that stacking ensembles provide significantly better prediction performance compared to many other classifiers (Petrić et al. 2016a). However, a substantial amount of future work is needed to establish a decision- making approach for ensembles that will fully exploit our findings. Our results further indicate the possible reasons for high false alarms previously attributed to

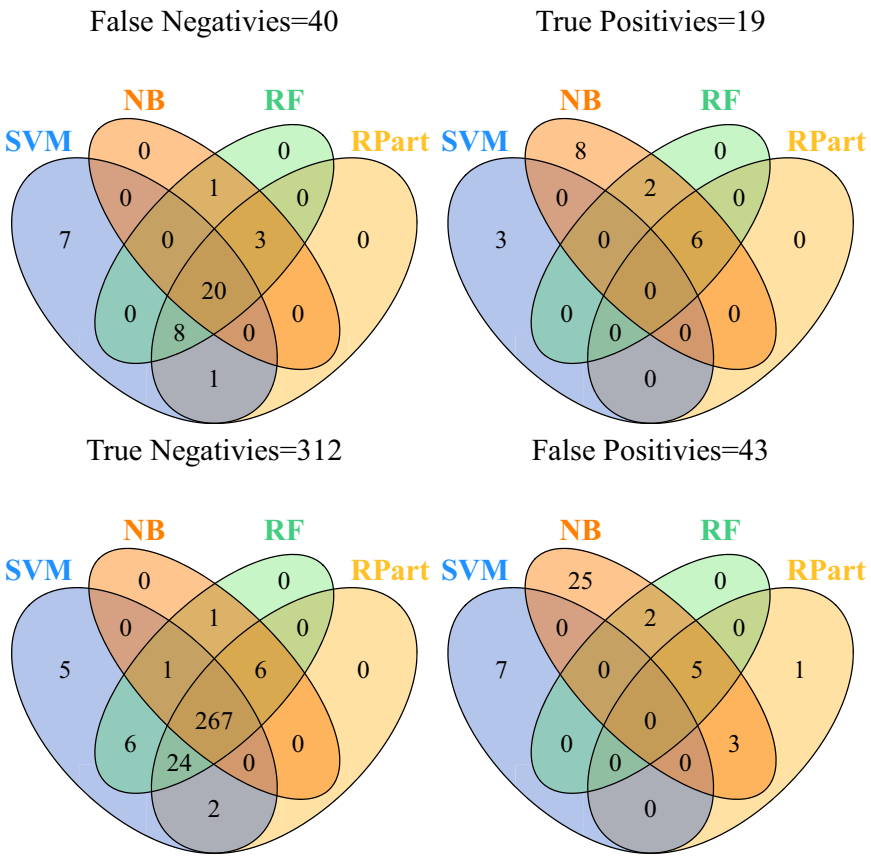


Fig. 7 Ivy sensitivity analysis using different classifiers. $n = 312$; $p = 40$

ensembles. As many true defects are individually predicted by single classifiers, ensembles based on majority voting approach would certainly misclassify such defects.

6 Threats to validity

Although we implemented what could be regarded as current best practice in classifier-based model building, there are many different ways in which a classifier may be built. There are also many different ways in which the data used can be pre-processed. All of these factors are likely to impact on predictive performance. As Lessmann et al. (2008) say *classification is only a single step within a multistage data mining process* (Fayyad et al. 1996). *Especially, data preprocessing or engineering activities such as the removal of non-informative features or the discretisation of continuous attributes may improve the performance of some classifiers* (*see, e.g., Dougherty et al.* (1995) *and Hall and Holmes* (2003)). *Such techniques have an undisputed value.* Despite the likely advantages of implementing these many additional techniques, as Lessmann et al. we implemented only a basic set of these techniques. Our reason for this decision was the same as Lessmann et al. ...*computationally infeasible when considering a large number of classifiers at the same time*. The experiments we report here each took several days of processing time. We did implement a set of techniques that are commonly used in defect prediction of which there is evidence they improve predictive performance. We went further in some of the techniques

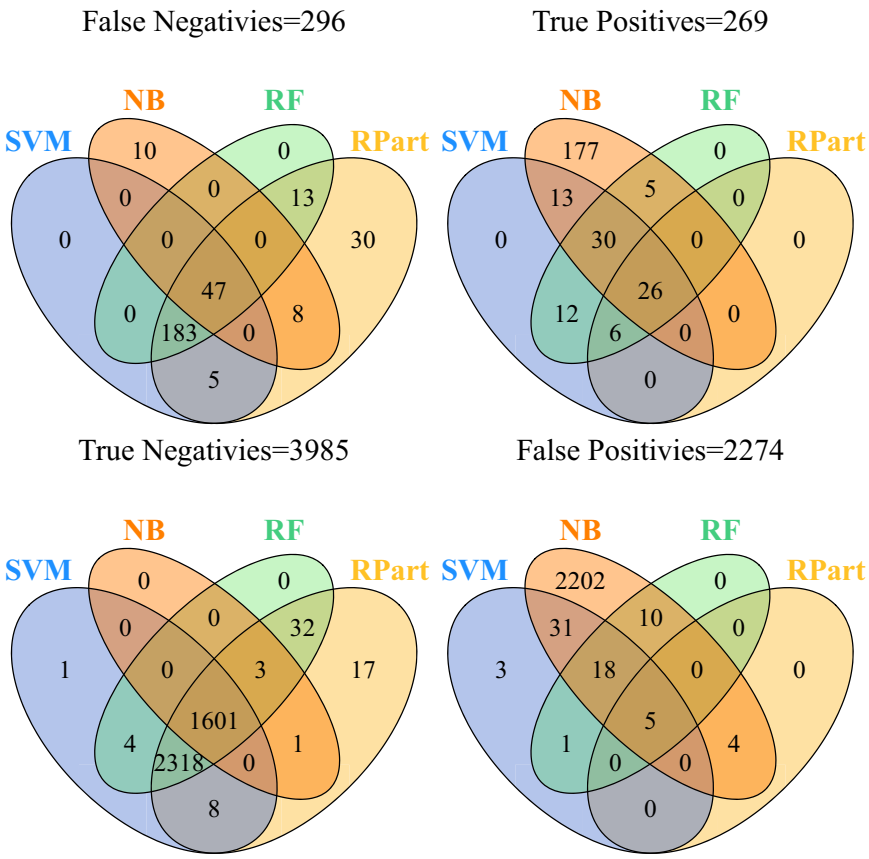


Fig. 8 KN sensitivity analysis using different classifiers. $n = 3992$; $p = 322$

we implemented, e.g. running our experiments 100 times rather than the 10 times that studies normally do. However, we did not implement a technique to address data imbalance (e.g. SMOTE). This was because data imbalance does not affect all classifiers equally. We implemented only partial feature reduction. The impact of the model building and data pre-processing approaches we used are not likely to significantly affect the results we report. This could be due to the ceiling effect reported in 2008, which states that prediction modelling solely based on model building and data pre-processing cannot break through the performance ceiling (Menzies et al. 2008). In addition, the range of steps we applied in our experiments while building prediction models are comparable to current defect prediction studies (e.g. repeated experiments, the use of cross validation, etc.).

Our studies are also limited in that we only investigated four classifiers. It may be that there is less variation in the defect subsets detected by classifiers that we did not investigate. We believe this to be unlikely, as the four classifiers we chose are representative of discrete groupings of classifiers in terms of the prediction approaches used. However, future work will have to determine whether additional classifiers behave as we report these four classifiers to. We also used a limited number of datasets in our study. Again, it is possible that other datasets behave differently. We believe this will not be the case, as the 18 datasets we investigated were wide ranging in their features and produced a variety of results in our investigation.

Our analysis is also limited by only measuring predictive performance using f-measure and MCC metrics. Such metrics are implicitly based on the cut-off points used by the

classifiers themselves to decide whether a software component is defective or not. All software components having a defective probability above a certain cut-off point (in general, it is equal to 0.5) are labelled as 'defective', or as 'non-defective' otherwise. For example, Random Forest not only provides a binary classification of datapoints but also provides the probabilities for each component belonging to 'defective' or 'non-defective' categories. D'Ambros et al. (2012) investigated the effect of different cut-off points on the performances of classification algorithms in the context of defect prediction and proposed other performance metrics that are independent from the specific (and also implicit) cut-off points used by different classifiers. Future work includes consideration of the different cut-off points to the individual performances of the four classifiers used in this paper.

7 Conclusion

We report a surprising amount of prediction variation within experimental runs. We repeated our cross-validation runs 100 times. Between these runs, we found a great deal of inconsistency in whether a module was predicted as defective or not by the same model. This finding has important implications for defect prediction as many studies only repeat experiments 10 times. This means that the reliability of some previous results may be compromised. In addition, the prediction flipping that we report has implications for practitioners. Although practitioners may be happy with the overall predictive performance of a given model, they may not be so happy that the model predicts different modules as defective depending on the training of the model. Our analysis shows that the classifier's inconsistency occurs in a variety of different software domains, including open source and commercial projects.

Performance measures can make it seem that defect prediction models are performing similarly. However, even where similar performance figures are produced, different defects are identified by different classifiers. This has important implications for defect prediction. First, assessing predictive performance using conventional measures such as f-measure, precision or recall gives only a basic picture of the performance of models. Second, models built using only one classifier are not likely to comprehensively detect defects. Ensembles of classifiers need to be used. Third, current approaches to ensembles need to be re-considered. In particular, the popular 'majority' voting decision approach used by ensembles will miss the sizeable subsets of defects that single classifiers correctly predict. Ensemble decision-making strategies need to be enhanced to account for the success of individual classifiers in finding specific sets of defects. Our results support the use of classifier ensembles not based on majority voting.

The feature selection techniques for each classifier could also be explored in the future. Since different classifiers find different subsets of defects, it is reasonable to explore whether some particular features better suit specific classifiers. Perhaps some classifiers work better when combined with specific subsets of features.

We suggest new ways of building enhanced defect prediction models and opportunities for effectively evaluating the performance of those models in within-project studies. These opportunities could provide future researchers with the tools with which to break through the performance ceiling currently being experienced in defect prediction.

Acknowledgments This work was partly funded by a grant from the UK's Engineering and Physical Sciences Research Council under grant number: EP/L011751/1.

References

Arisholm, E., & Briand, L.C. (2007). Fuglerud M 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, L.C., & Johannessen, E.B. (2010). A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. *Journal of Systems and Software*, *83*(1), 2–17.

Bell, R., Ostrand, T., & Weyuker, E. (2006). Looking for bugs in all the right places. In *Proceedings of the 2006 international symposium on Software testing and analysis, ACM, pp 61–72*.

Bibi, S., Tsoumakas, G., Stamelos, I., & Vlahvas, I. (2006). Software defect prediction using regression via classification. In *IEEE international conference on computer systems and applications*.

Bird, C., Bachmann, A., Aune, E., Duffy, J., Bernstein, A., Filkov, V., & Devanbu, P. (2009a). 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*.

Bird, C., Nagappan, N., Gall, H., Murphy, B., & Devanbu, P. (2009b). Putting it all together: Using socio-technical networks to predict failures. In *20th International Symposium on Software Reliability Engineering, IEEE, pp 109–119*.

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.

Bowes, D., Hall, T., & Gray, D. (2013). DConfusion: a technique to allow cross study performance evaluation of fault prediction studies. *Automated Software Engineering*, 1–27. doi:10.1007/s10515-013-0129-8.

Bowes, D., Hall, T., & Petrić, J. (2015). Different classifiers find different defects although with different level of consistency. In *Proceedings of the 11th International Conference on Predictive Models and Data Analytics in Software Engineering, PROMISE '15, pp 3:1–3:10*. doi:10.1145/2810146.2810149.

Briand, L., Melo, W., & Wust, J. (2002). Assessing the applicability of fault-proneness models across object-oriented software projects. *IEEE Transactions on Software Engineering*, *28*(7), 706–720.

Catal, C., & Diri, B. (2009). A systematic review of software fault prediction studies. *Expert Systems with Applications*, *36*(4), 7346–7354.

Chawla, N.V., Japkowicz, N., & Kotcz, A. (2004). Editorial: special issue on learning from imbalanced data sets. *SIGKDD Explorations*, *6*(1), 1–6.

Chen, H., & Yao, X. (2009). Regularized negative correlation learning for neural network ensembles. *IEEE Transactions on Neural Networks*, *20*(12), 1962–1979.

Chen, W., Wang, Y., Cao, G., Chen, G., & Gu, Q. (2014). A random forest model based classification scheme for neonatal amplitude-integrated eeg. *Biomedical engineering online*, *13*(2), 1.

D'Ambros, M., Lanza, M., & Robbes, R. (2009). On the relationship between change coupling and software defects. In *16th working conference on reverse engineering, 2009. WCRE '09., pp 135 –144*.

D'Ambros, M., Lanza, M., & Robbes, R. (2012). Evaluating defect prediction approaches: a benchmark and an extensive comparison. *Empirical Software Engineering*, *17*(4), 531–577. doi:10.1007/s10664-011-9173-9.

Dougherty, J., Kohavi, R., & Sahami, M. (1995). Supervised and unsupervised discretization of continuous features. In *ICML, pp 194–202*.

Elish, K., & Elish, M. (2008). Predicting defect-prone software modules using support vector machines. *Journal of Systems and Software*, *81*(5), 649–660.

Fayyad, U., Piatetsky-Shapiro, G., & Smyth, P. (1996). From data mining to knowledge discovery in databases. *AI magazine*, *17*(3), 37.

Fenton, N., & Neil, M. (1999). A critique of software defect prediction models. *IEEE Transactions on Software Engineering*, *25*(5), 675–689.

Gao, K., Khoshgoftaar, T.M., & Napolitano, A. (2015). Combining feature subset selection and data sampling for coping with highly imbalanced software data. In *Proc. of 27th International Conf. on Software Engineering and Knowledge Engineering, Pittsburgh*.

Gray, D. (2013). *Software defect prediction using static code metrics : Formulating a methodology*. University of Hertfordshire: PhD thesis, Computer Science.

Gray, D., Bowes, D., Davey, N., Sun, Y., & Christianson, B. (2011). The misuse of the NASA metrics data program data sets for automated software defect prediction. In *EASE 2011, IET, Durham, UK*.

Gray, D., Bowes, D., Davey, N., Sun, Y., & Christianson, B. (2012). Reflections on the NASA MDP data sets. *IET Software*, *6*(6), 549–558.

Hall, M.A., & Holmes, G. (2003). Benchmarking attribute selection techniques for discrete class data mining. *IEEE Transactions on Knowledge and Data Engineering*, *15*(6), 1437–1447.

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

Jiang, Y., Lin, J., Cukic, B., & Menzies, T. (2009). Variance analysis in software fault prediction models. In *SSRE 2009, 20th International Symposium on Software Reliability Engineering, IEEE Computer Society, Mysuru, Karnataka, India, 16-19 November 2009, pp 99–108*.

Jureczko, M., & Madeyski, L. (2010). Towards identifying software project clusters with regard to defect prediction. In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering, ACM, New York, NY, USA, PROMISE '10, pp 9:1–9:10*. doi:10.1145/1868328.1868342.

Kamei, Y., & Shihab, E. (2016). Defect prediction: Accomplishments and future challenges. In *2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER), vol 5, pp 33–45*. doi:10.1109/SANER.2016.56.

Khoshgoftaar, T., Yuan, X., Allen, E., Jones, W., & Hudepohl, J. (2002). Uncertain classification of fault-prone software modules. *Empirical Software Engineering*, *7*(4), 297–318.

Khoshgoftaar, T.M., Gao, K., & Seliya, N. (2010). Attribute selection and imbalanced data: Problems in software defect prediction. In *2010 22nd IEEE international conference on tools with artificial intelligence (ICTAI), vol 1, pp 137–144*.

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*.

Kutlubay, O., Turhan, B., & Bener, A. (2007). A two-step model for defect density estimation. In *33rd EUROMICRO Conference on Software Engineering and Advanced Applications, 2007., pp 322–332*.

Laradji, I.H., Alshayeb, M., & Ghouti, L. (2015). Software defect prediction using ensemble learning on selected features. *Information and Software Technology*, *58*, 388–402. doi:10.1016/j.infsof.2014.07.005. http://www.sciencedirect.com/science/article/pii/S0950584914001591.

Lessmann, S., Baesens, B., Mues, C., & Pietsch, S. (2008). Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Transactions on Software Engineering*, *34*(4), 485–496.

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*.

Madeyski, L., & Jureczko, M. (2015). Which process metrics can significantly improve defect prediction models? An empirical study. *Software Quality Journal*, *23*(3), 393–422. doi:10.1007/s11219-014-9241-7.

Malhotra, R. (2015). A systematic review of machine learning techniques for software fault prediction. *Appl Soft Comput*, *27*(C), 504–518. doi:10.1016/j.asoc.2014.11.023.

Mende, T. (2011). On the evaluation of defect prediction models. In *The 15th CREST Open Workshop*.

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., Greenwald, J., & Frank, A. (2007). Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering*, *33*(1), 2–13.

Menzies, T., 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, pp 47–54*.

Menzies, T., Caglayan, B., He, Z., Kocaguneli, E., Krall, J., Peters, F., & Turhan, B. (2012). The promise repository of empirical software engineering data. http://promisedata.googlecode.com.

Minku, L.L., & Yao, X. (2012). Ensembles and locality: Insight on improving software effort estimation. Information and Software Technology.

Minku, L.L., & Yao, X. (2013). Software effort estimation as a multi-objective learning problem. *ACM Transactions on Software Engineering and Methodology*. to appear.

Mısırlı, A.T., Bener, A.B., & Turhan, B. (2011). An industrial case study of classifier ensembles for locating software defects. *Software Quality Journal*, *19*(3), 515–536.

Mizuno, O., & Kikuno, T. (2007). Training on errors experiment to detect fault-prone software modules by spam filter. In *Proceedings of the the 6th 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 '07, pp 405–414*.

Mizuno, O., Ikami, S., Nakaichi, S., & Kikuno, T. (2007). Spam filter based approach for finding fault-prone software modules. In *Mining Software Repositories, 2007. ICSE Workshops MSR '07. Fourth International Workshop on, p 4*.

Myrtveit, I., Stensrud, E., & Shepperd, M. (2005). Reliability and validity in comparative studies of software prediction models. *IEEE Transactions on Software Engineering*, 380–391.

Nagappan, N., Zeller, A., Zimmermann, T., Herzig, K., & Murphy, B. (2010). Change bursts as defect predictors. In *Software Reliability Engineering, 2010 IEEE 21st International Symposium on, pp 309–318*.

Ostrand, T., Weyuker, E., & Bell, R. (2010). Programmer-based fault prediction. In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering, ACM, pp 1–10*.

Panichella, A., Oliveto, R., & De Lucia, A. (2014). Cross-project defect prediction models: L'union fait la force. In *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), pp 164–173*. doi:10.1109/CSMR-WCRE.2014.6747166.

Petrić, J., Bowes, D., Hall, T., Christianson, B., & Baddoo, N. (2016). Building an ensemble for software defect prediction based on diversity selection. In *The 10th International Symposium on Empirical Software Engineering and Measurement, ESEM'16, p 10*.

Petrić, J., Bowes, D., Hall, T., Christianson, B., & Baddoo, N. (2016). The jinx on the NASA software defect data sets. In *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering, ACM, New York, NY, USA, EASE '16, pp 13:1–13:5*. doi:10.1145/2915970.2916007.

Rodriguez, D., Herraiz, I., Harrison, R., Dolado, J., & Riquelme, J.C. (2014). Preliminary comparison of techniques for dealing with imbalance in software defect prediction. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering, ACM, New York, NY, USA, EASE '14, pp 43:1–43:10*. doi:10.1145/2601248.2601294.

Seiffert, C., Khoshgoftaar, T.M., & Hulse, J.V. (2009). Improving software-quality predictions with data sampling and boosting. *IEEE Transactions on Systems, Man, and Cybernetics Part A*, *39*(6), 1283–1294.

Shepperd, M., & Kadoda, G. (2001). Comparing software prediction techniques using simulation. *IEEE Transactions on Software Engineering*, *27*(11), 1014–1022.

Shepperd, M., Song, Q., Sun, Z., & Mair, C. (2013). Data quality: some comments on the NASA software defect datasets. *IEEE Transactions on Software Engineering*, *39*(9), 1208–1215. doi:10.1109/TSE.2013.11.

Shepperd, M., Bowes, D., & Hall, T. (2014). Researcher bias: The use of machine learning in software defect prediction. *IEEE Transactions on Software Engineering*, *40*(6), 603–616. doi:10.1109/TSE.2014.232358.

Shin, Y., Bell, R.M., Ostrand, T.J., & Weyuker, E.J. (2009). Does calling structure information improve the accuracy of fault prediction? In Godfrey, M.W., & Whitehead, J. (Eds.) *Proceedings of the 6th International Working Conference on Mining Software Repositories, IEEE, pp 61–70*.

Shivaji, S., Whitehead, E.J., Akella, R., & Sunghun, K. (2009). Reducing features to improve bug prediction. In *Automated Software Engineering, 2009. ASE '09. 24th IEEE/ACM International Conference on, pp 600–604*.

Soares, C., Brazdil, P.B., & Kuba, P. (2004). A meta-learning method to select the kernel width in support vector regression. *Machine learning*, *54*(3), 195–209.

Sun, Z., Song, Q., & Zhu, X. (2012). Using coding-based ensemble learning to improve software defect prediction. *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, *42*(6), 1806–1817. doi:10.1109/TSMCC.2012.2226152.

Turhan, B., Menzies, T., Bener, A.B., & Di Stefano, J. (2009). On the relative value of cross-company and within-company data for defect prediction. *Empirical Softw Engg*, *14*(5), 540–578. doi:10.1007/s10664-008-9103-7.

Visa, S., & Ralescu, A. (2004). Fuzzy classifiers for imbalanced, complex classes of varying size. In *Information Processing and Management of Uncertainty in Knowledge-Based Systems, pp 393–400*.

Wahono, R. (2015). A systematic literature review of software defect prediction: research trends, datasets, methods and frameworks. Journal of Software Engineering, 1(1). http://journal.ilmukomputer.org/index.php/jse/article/view/47.

Witten, I. (2005). *Frank E*. Data mining: practical machine learning tools and techniques: Morgan Kaufmann.

Wolpert, D.H. (1992). Stacked generalization. *Neural Networks*, *5*(2), 241–259. doi:10.1016/S0893-6080(05)80023-1.

Zhang, H. (2009). An investigation of the relationships between lines of code and defects. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on, pp 274–283*.

Zhou, Y., Xu, B., & Leung, H. (2010). On the ability of complexity metrics to predict fault-prone classes in object-oriented systems. *Journal of Systems and Software*, *83*(4), 660–674.

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*.

David Bowes received the PhD degree in defect prediction and is currently a Senior Lecturer in software engineering at the University of Hertfordshire. He has published mainly in the area of static code metrics and defect prediction. His research interests include the reliability of empirical studies.

Tracy Hall received the PhD degree in the implementation of software metrics from City University, London. She is currently a Professor in software engineering at Brunel University London. Her expertise is in empirical software engineering and her current research activities are focused on software fault prediction. She has published more than 100 international peer-reviewed papers.

Jean Petrić received the MSc degree in Computer Science from the University of Rijeka, Croatia. He has been working towards his PhD at the University of Hertfordshire, and is currently a research fellow at Brunel University London. His research interest is in software defect prediction.

5.4 An Extended Analysis of "Different Classifier Find Different Defects"

Researchers have reported similar performances of software defect prediction models over the last few decades. In our earlier [Bowes et al. 2015, 2017] work we showed that despite the similar predictive performances of four different classifiers, they predict very different subsets of defects. In the previous work we used public and commercial datasets. My subsequent analysis has shown that some of those datasets are of questionable quality. To ensure the validity of findings from our previous work, I repeat the experiments using cleaned data. I find that the previously reported results hold given the cleaned datasets. In other words, different classifiers do find unique subsets of defects.

5.4.1 Background

Many standalone classifiers have been used for predicting defects. Arisholm et al. [2010] systematically studied the impact of classifiers, metrics and performance measures on predictive performance. They found that the choice of classifier has a limited impact on the performance. Lessmann et al. [2008a] performed defect prediction using 22 classifiers and demonstrated that the top performing 17 classifiers do not differ significantly. Malhotra and Raje [2014] selected 18 classifiers which are a subset of the 22 classifiers used by Lessmann et al. [2008a]. Accounting for the issues previously reported in software defect prediction (e.g. the use of inappropriate performance measures and validation of models without separating training and testing datasets), Malhotra and Raje [2014] compared the performance of the classifiers using 6 datasets. The authors established that Naïve Bayes is the best performing classifier with an average AUC of 0.76 across the datasets. According to Wahono [2015] the most commonly used classifiers in software defect prediction are Logistic Regression, Naïve Bayes, K- Nearest Neighbor, Neural Network, Decision Tree, Support Vector Machine and Random Forest.

Ensembles have been increasingly used in software defect prediction. Tosun et al. [2008] combined the Naïve Bayes, neural networks, and Voting Feature interval classifiers to demonstrate they perform considerably better than Naïve Bayes alone. Wahono and Suryana [2013] used bagging (see Section 3.3.1) to tackle the imbalance problem common in software defect prediction datasets. Their results showed that bagging, when combined with particle swarm optimisation for selecting attributes, can considerably improve prediction performances. Wang and Yao [2013] compared resampling techniques, threshold moving, and ensemble algorithms to investigate which has the greatest potential in dealing with

imbalanced software defect prediction datasets. They found that boosting (see Section 3.3.1) is the most effective technique according to *balance*, G-mean, and AUC performance measures. Panichella et al. [2014] demonstrated that different classifiers combined into stacking complement each other and improve predictions over standalone classifiers in the cross-project software defect prediction set-up.

With the exception of Panichella et al. [2014], previous studies presented predictive performances of software defect prediction models using performance measures alone. However, focusing only on performance figures, without examining the individual defects each classifiers detects or does not detect, is limiting. Such an approach makes it difficult to establish whether specific defects are consistently missed by all classifiers, or whether different classifiers detect different subsets of defects. Establishing the set of defects each classifier detects, rather than just looking at the overall performance figures, allows the identification of classifier ensembles most likely to detect the largest range of defects.

Therefore, in our two studies repeated here, we went beyond performance measure numbers and set out to answer whether different classifiers find different defective components. In **Paper 2** we found that 4 different classifiers achieve similar prediction performances but find unique subsets of defects. We extended the analysis reported in **Paper 2** to 6 additional open source and commercial datasets. In the extended work, we used 3 open source datasets available at PROMISE and three commercial datasets for which we collected defect data. We found that the results reported in **Paper 2** hold across the public and commercial datasets. We reported those results in **Paper 3**.

However, the comprehensive set of integrity checks was not carried out on all datasets reported in the two papers. In **Paper 2** we did not use the NASA datasets that satisfy the integrity checks defined in Petrić et al. [2016]. In **Paper 3**, both, the NASA datasets and commercial datasets were not cleaned by all integrity checks defined in Section 4.5. The PROMISE datasets were cleaned according to the rules reported in the paper (see Section 5.3), which does not contain the comprehensive list of integrity checks. In this extended analysis I address the aforementioned shortcomings.

5.4.2 Methodology

The selection of classifiers and datasets in this analysis is inspired by **Paper 2** and **Paper 3**. I use four classifiers: Naïve Bayes, RPart, SVM and Random Forest. I select these four classifiers because they build models based on different mathematical properties. They are also the most commonly used classifiers in defect prediction [Wahono 2015]. Naïve Bayes makes predictions by applying Bayes' theorem. Even though Naïve Bayes makes an assumption that attributes are independent, which is rarely the case with defect data, it

performs well [Hall et al. 2012, Wahono 2015]. RPart builds a decision tree based on the information entropy of the subsets of training data which can be achieved by splitting the data using different independent attributes. SVMs transfer each data instance as a point to an n-dimensional space, where n is the number of attributes. It then finds a hyper-plane that adequately separates defective from non-defective class instances. Random Forest is an ensemble approach which combines multiple decision trees and uses majority-voting to make the final prediction. These classifiers are described in more detail in 3.2.

Tuning is an important step when building prediction models. Therefore, I tune Random Forest by varying the number of trees from 50 to 200 in steps of 50. For SVM using a radial base function I tune γ from 0.25 to 4 and C from 2 to 32. However, not all models perform significantly better when tuned. Default parameters and splitting algorithms for Naïve Bayes and RPart are known to work well, so I do not perform tuning of these two classifiers. I repeat the experiment 100 times. I use the 10-fold cross validation by splitting each dataset into 9 folds used for training and the last fold for evaluating the performance of the model. This was repeated with each fold being held out in turn. I use Matthews Correlation Coefficient (MCC) to estimate the performance of the models. MCC has been reported as a suitable performance measure for software defect prediction as it deals well with highly imbalanced data [Shepperd et al. 2014].

I use 14 open source and 3 commercial datasets in this analysis. I do not use the NASA datasets due to their poor quality. When the NASA datasets are cleaned by the complete list of integrity checks, the majority of the datasets become unusable for defect prediction as shown in Section 4.5. Table 5.1 depicts the proportion of affected data points in the commercial datasets, where only the HA system is affected by only 0.7%.

Table 5.1 Commercial datasets cleaned according to the integrity checks described in Section 4.5

Dataset	# of modules pre-cleaned	# of modules post-cleaned	% loss due cleaning
PA	4996	4996	0.0
KN	4314	4314	0.0
HA	9062	8998	0.7

I select the 14 PROMISE datasets based on the following criteria. First, I obtain all defect datasets from the PROMISE repository and sort them by the number of data instances. I then select a single version for each project with the highest number of data instances. Two datasets are exceptions to this: *jedit* and *xalan*, as the versions 4.3, and 2.7 contain a small number of tuples belonging to either class. As in this experiment I aim to answer whether

Table 5.2 Datasets used in the repeated experiment

Dataset	# of modules pre cleaning	# of modules post cleaning	% loss due cleaning	% defective methods post cleaning
ant 1.7	745	722	3.1	23.0
arc	234	210	10.3	12.4
camel 1.6	965	877	9.1	21.0
ivy 2.0	352	345	2.0	11.6
jedit 4.2	367	363	1.1	13.2
log4j 1.2	205	202	1.5	92.6
lucene 2.4	340	335	1.5	59.1
poi 3.0	442	397	10.2	64.5
redaktor	176	169	4.0	14.8
synapse 1.2	256	244	4.7	35.2
tomcat	858	791	7.8	9.7
velocity 1.6	229	209	8.7	36.4
xalan 2.6	885	724	18.2	44.6
xerces 1.4	588	482	18.0	77.0

classifiers find different defective components, I use a diverse set of datasets to verify the validity of my previous findings.

To mitigate threats due to data integrity, I repeat the experiment reported in Section 5.3 by using the complete list of integrity checks described in 4.3. I do not repeat the experiment by using all open source and commercial datasets. I make the following decisions when selecting datasets. I use the 14 datasets from the PROMISE repository listed in Table 5.2. I do not use the NASA datasets as they are highly problematic (see Chapter 4), nor the commercial datasets as even after the cleaning the HA system gets affected by less than 1%.

The experiments in **Paper 2** and **Paper 3** provide analyses which go beyond the research question I aim to answer in this chapter. Therefore, the experiments reported in the two papers are partially repeated here. I exclude the parts of the experiment which do not contribute to answering the research question set out in Section 5.1.

5.4.3 Results

Table 5.3 shows Matthews Correlation Coefficient (MCC) values for the four classifiers across the 14 PROMISE datasets. To validate whether classifiers achieve similar performances I perform the Friedman's statistical test. With the $p = 0.05$ level of confidence I confirm that there is a significant difference in the classifiers' performances. The statistical test is carried out for the MCC values on the 14 datasets and the four classifiers. I carry out a

further statistical analysis to establish which classifiers outperform others. For this purpose I select a post-hoc Nemenyi's test. Unlike Friedman's test which only verifies whether there is a significant difference between classifiers' performance or not, Nemenyi's test reports these differences between individual classifiers. Both, Friedman's and Nemanyi's tests have previously been used for the same purpose by Lessmann et al. [2008a].

Table 5.3 Performance Measures All Datasets by Learner

Classifier	Average	StDev
SVM	0.287	0.191
RPart	0.375	0.190
NaiveBayes	0.274	0.105
RandomForest	0.419	0.165

The post-hoc analysis shows no significant differences between the SVM, Naïve Bayes, and RPart classifiers. Random Forest outperforms SVM and Naïve Bayes and has typically achieved good performances in previous studies [Bowes et al. 2017, Lessmann et al. 2008a]. However, the overall performance figures depicted in Table 5.3 mask a range of different performances by classifiers when used on individual datasets. Figure 5.1 demonstrates the true positive instances for four PROMISE datasets. These four datasets are selected as they demonstrate the highest diversity of predictions achieved by individual classifiers. Overall, the results of this extended analysis support our previous findings (see Section 5.3) which suggest that different learners find different defective components.

From Figure 5.1 it is evident that SVMs find the majority of unique defects in each dataset (except *ant*), which other classifiers fail to correctly predict. As SVMs make predictions by finding the most suitable separation plane in high dimensional data, it is possible for them to capture complex patterns in data. However, as performance values suggest, SVMs do not perform the best. This is because SVMs make more incorrect predictions compared to some other classifiers. For instance, SVMs have in total predicted 103 non defective data instances as defective, which is more than Random Forest and RPart classifiers together. Overall, SVMs are a potential candidate to be combined with other classifiers due to their ability to identify a considerable number of unique defects. Apart from SVMs, Naïve Bayes predicts a substantial number of unique defects for the *ant* and *cam* datasets.

Figure 5.2 depicts overall individual predictions made by different learners. SVMs clearly predict the highest number of unique defects, followed by Naïve Bayes, RPart, and Random Forest. Even though SVMs find most defects, they achieve the worst predictive performances. Random Forest, on the other hand, finds the least number of unique defects but achieve the best predictive performance. The reason for this is the number of false positive and false

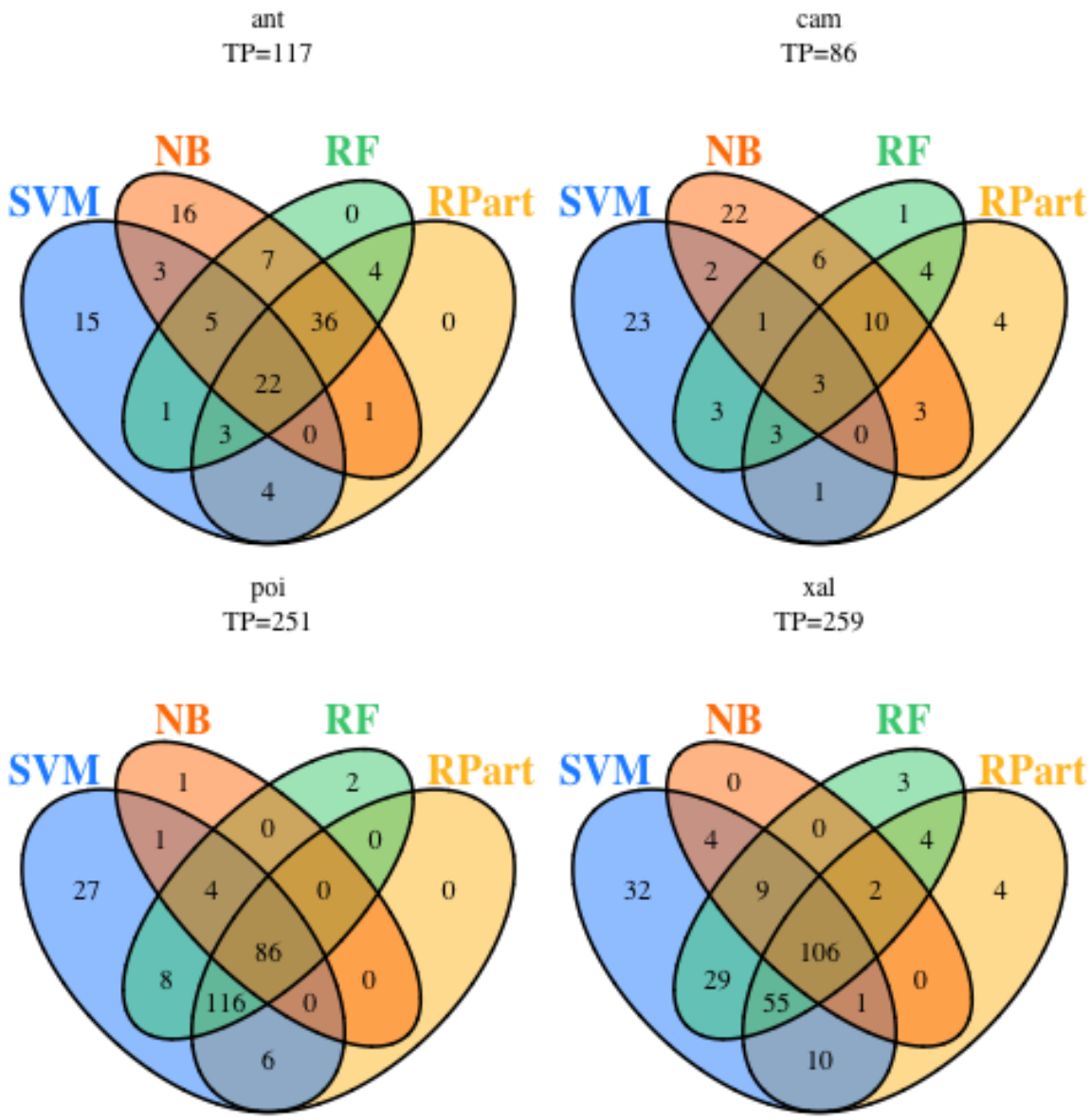


Fig. 5.1 True positive rates for the four top diverse datasets

negative predictions, which is higher for SVMs compared to Random Forest. Therefore, it is likely that a certain combination of classifiers could detect more defects than any of the classifiers individually. These results suggest the use of ensembles, which are described in the Chapter 6. Similar findings are reported in **Paper 2** and **Paper 3**.

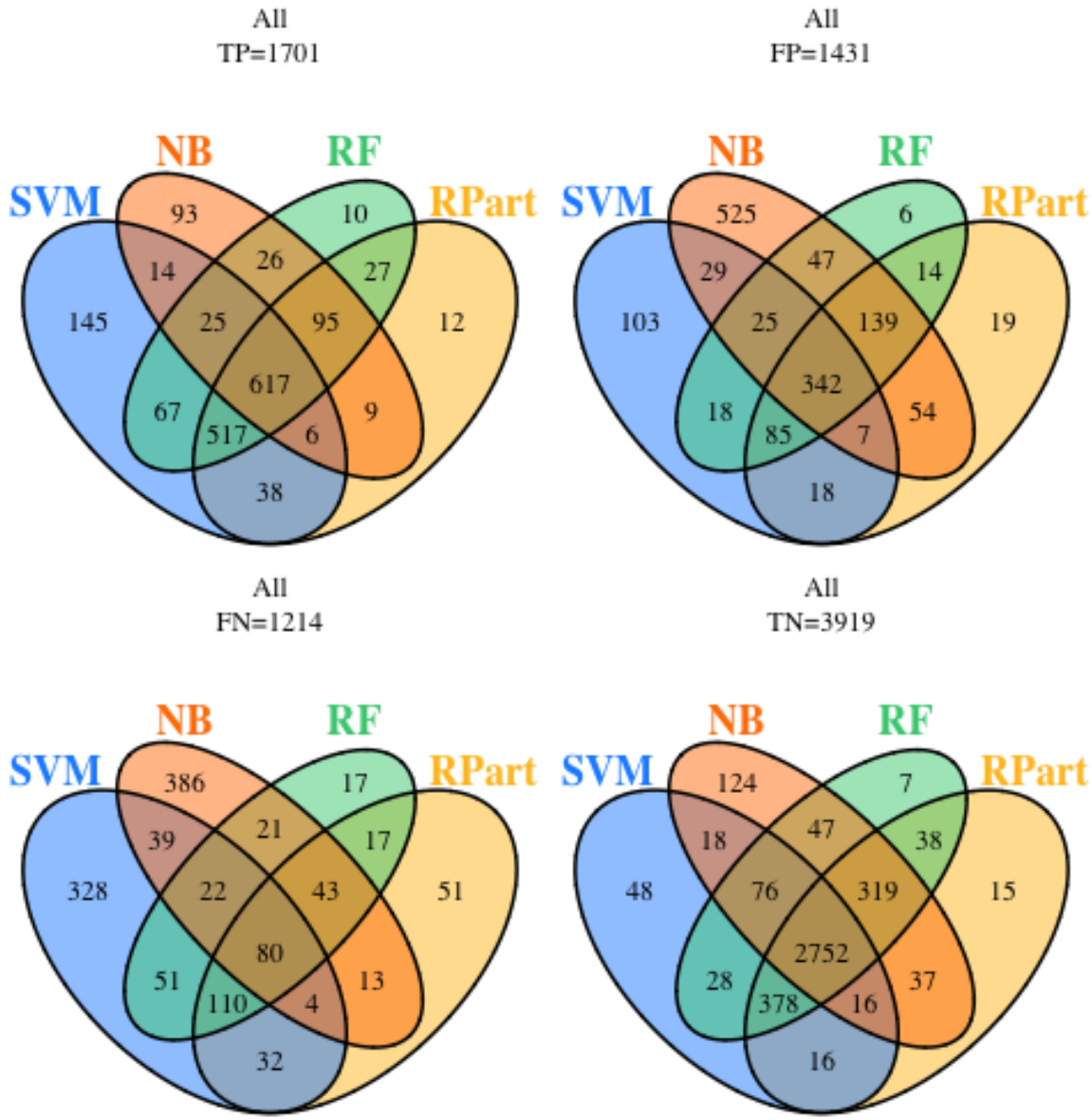


Fig. 5.2 Confusion matrix for all 14 PROMISE datasets

5.4.4 Conclusion

Paper 2 and **Paper 3** report on the ability of classifiers to find different subsets of defects. Both studies use datasets which are of poor quality. For this reason I repeated the experiments to ensure the validity of our previous results. Even when the experiments are repeated using the clean and diverse datasets, I establish that each classifier detects distinct subsets of defects as depicted in Figures 5.1 and 5.2. This result suggests the use of ensembles of machine learners, as standalone classifiers do not comprehensively detect all defects. In addition, the

observation from Figures 5.1 and 5.2 suggest that the majority-voting approach should not be used in software defect prediction. As many true defects are individually predicted by single classifiers, ensembles based on majority-voting approach would misclassify such defects.

5.5 Summary of the Research Question

RQ1. Do models created by different classifiers find different defective components?

Paper 2, **Paper 3** and the extended analysis reported in Section 5.4 suggest that despite the similar predictive performances, models created by different classifiers find unique subsets of defects.

5.6 Summary of My Contributions

Performance measures can make it seem that software defect prediction models are performing similarly. However, even where similar performance figures are obtained, different defective components are identified by different classifiers. This finding suggests that no individual classifier can comprehensively detect all defects different classifiers find. Ensembles of classifiers need to be used, which is investigated in more detail in Chapter 6. The results also suggest that the current approaches for ensembles in software defect prediction need to be reconsidered. In particular, the popular 'majority' voting approach could miss substantial subsets of defects that standalone classifiers correctly predict. Ensembles decision-making strategies that account for the success of individual classifiers should be used.

5.7 Summary of the Contributions to the Papers

The initial version of **Paper 2** was rejected, so I extended the work by addressing all issues raised by the reviewers. I altered the paper in several ways. First, I modified the parts of the paper describing the key findings which are "even though overall performance figures between classifiers are similar, very different subsets of defects are predicted by different classifiers" and "the effect of prediction 'flipping' amongst the four classifiers used in the study". I modified the abstract, introduction, results, discussion and conclusion sections to focus the narrative of the paper in the direction of the two key findings. Second, I revised the background section to account for state-of-the-art related developments in software defect prediction. Third, I compared our work with the work of Panichella et al. [2014]. Finally, I independently repeated all experiments described in the paper before it was submitted for

the second time. David Bowes conducted the initial experiment and analysis, and together with Tracy Hall shaped the paper to its full form. Tracy Hall and myself made the final improvements to the paper. In terms of **Paper 3**, I selected and pre-processed the extended set of datasets. I ran all analyses using the extended set of datasets. I conducted and documented all comparisons between the results of existing and extended sets of datasets. I made all written additions to the paper which are finally improved by Tracy Hall. I independently conducted the extended analysis reported in Section 5.4.

5.8 Threats to Validity

A subsequent threat to validity of the experiment reported in Section 5.4, compared to the already published studies, is the selection of datasets. As it is often the case in software engineering studies, results do not always generalise. However, my selection of datasets has specifically been targeted to avoid the non-generalisability problem. All datasets I selected are from different projects, with very different levels of balance in both classes. Despite these differences, the conclusion of the repeated experiment is unchanged. All datasets were cleaned using state-of-the-art cleaning rules, which adds to the rigour of the conclusions I made.

Chapter 6

Building Ensemble Learners to Improve Prediction Models

Building better-performing prediction models may be possible using the differences in predictions that different models produce. I have confirmed that models created by different classifiers do not make the same decisions. If they had, we would only ever need one classifier. I can therefore investigate if the differences in the decisions can be exploited. In this chapter I investigate if models from different classifier families and different tuning can improve the performance of an ensemble of learners. My investigation shows that ensembles with diverse sets of classifiers can improve software defect prediction. A few diverse classifiers are sufficient to build effective ensemble models. The stacking technique should be preferred over majority-voting as unique subsets of defects are identified by some classifiers and not by others.

6.1 Prelude

In this chapter I investigate the following research questions:

RQ2 How can software defect prediction models be improved by combining classifiers predicting different defective components?

- **RQ2(a)** Can stacking ensembles based on explicit diversity improve prediction performance compared to other software defect prediction models?
- **RQ2(b)** How many classifiers combined into stacking ensembles are needed to provide good software defect prediction models?

RQ2(c) How much diversity and which base classifiers need to be combined to provide good ensemble models?

The overall aim of this chapter is to find whether defect models can be improved by accounting for diverse predictions of different models. I perform an empirical study to answer the research questions. The study is published as a conference paper:

Paper 4. Petrić J, Bowes D, Hall T, Christianson B, Baddoo N. Building an ensemble for software defect prediction based on diversity selection. InProceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement 2016 Sep 8 (p. 46). ACM.

The paper is laid out in Section 6.3. My findings show performance improvement using stacking ensembles compared to other software defect prediction models. Diversity amongst classifiers used for building ensembles is essential to achieving these performance improvements.

This chapter is organised as follows. The next section explains the need for ensembles in software defect prediction, followed by the section incorporating the main study. Summaries of my contributions and contributions to the paper are then discussed.

6.2 The Need for Ensembles to Improve Software Defect Prediction Models

A substantial amount of research has been done to identify best performing classifiers. Mizuno and Kikuno [2007] report that, of the techniques they studied, Orthogonal Sparse Bigrams Markov models (OSB) are best suited to software defect prediction. [Bibi et al. 2006] report that Regression via Classification works well. Khoshgoftaar et al. [2002] report that modules whose defect proneness are predicted as uncertain according to the χ^2 statistical test, can be effectively classified using the TreeDisc technique. Hall et al. [2012]'s systematic literature review shows that Naïve Bayes and Logistic Regression perform well. A large empirical study with the aim to compare 22 classifiers by Lessmann et al. [2008a] demonstrates that no significant difference in performance exists amongst the top 17 performing classifiers. Another empirical analysis by Malhotra and Raje [2014] which compares 18 classifiers confirms that most models achieve similar prediction performances of (with the AUC around 0.7).

As described in more detail in Section 3.3, an ensemble of machine learners is a strategy to combine predictions made by multiple classifiers. This approach typically achieves superior

performances over individual classifiers [Polikar 2006, Rokach 2009]. Some ensemble techniques have been investigated in the domain of software defect prediction. Mısırlı et al. [2011b], Sun et al. [2012], Wang et al. [2011] demonstrate the performance improvement by using ensembles. These works suggest that a collection of heterogeneous classifiers offer the best opportunity to predict defects. However, the approach to model an ensemble has a substantial effect on how well the model will perform in software defect prediction.

Panichella et al. [2014] recognise that different classifiers find different defective components and use an ensemble for cross-project software defect prediction. Di Nucci et al. [2017] suggest a variant of the stacking ensemble called ASCI, where classifiers are selected based on their ability to predict individual software units. Whereas in this work I provide predictions from the first layer of classifiers to a meta-classifier in order to predict defectiveness, Di Nucci et al. [2017] provide the original independent variables. Their goal is to predict the most suitable classifier given the properties of a software unit. As I shall show in this chapter, carefully designed ensemble classifiers can enhance the prediction performance of software defect prediction models.

6.3 Building an Ensemble for Software Defect Prediction Based on Diversity Selection

In this section I present **Paper 4**. This paper is published as a conference paper at the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement. This study uses eight open–source datasets from the PROMISE repository to answer the research questions set out in Section 6.1. The paper can be summarised as follows. **RQ2(a)** The stacking approach and diverse learners show relative increase in the number of identified defects compared to some other commonly used ensemble techniques. **RQ2(b)** Stacking ensembles made of three classifiers are sufficient to achieve improved prediction performances. **RQ2(c)** Classifiers in the stacking ensemble should be diverse and from different classifier families.

Building an Ensemble for Software Defect Prediction Based on Diversity Selection

Jean Petrić
Science and Technology Research Institute
University of Hertfordshire
Hatfield, Hertfordshire
AL10 9AB, UK
j.petric@herts.ac.uk

David Bowes
Science and Technology Research Institute
University of Hertfordshire
Hatfield, Hertfordshire
AL10 9AB, UK
d.h.bowes@herts.ac.uk

Tracy Hall
Department of Computer Science
Brunel University London
Uxbridge, Middlesex
UB8 3PH, UK
tracy.hall@brunel.ac.uk

Bruce Christianson
Science and Technology Research Institute
University of Hertfordshire
Hatfield, Hertfordshire
AL10 9AB, UK
b.christianson@herts.ac.uk

Nathan Baddoo
Science and Technology Research Institute
University of Hertfordshire
Hatfield, Hertfordshire
AL10 9AB, UK
n.baddoo@herts.ac.uk

ABSTRACT

Background: Ensemble techniques have gained attention in various scientific fields. Defect prediction researchers have investigated many state-of-the-art ensemble models and concluded that in many cases these outperform standard single classifier techniques. Almost all previous work using ensemble techniques in defect prediction rely on the majority voting scheme for combining prediction outputs, and on the implicit diversity among single classifiers. **Aim**: Investigate whether defect prediction can be improved using an explicit diversity technique with stacking ensemble, given the fact that different classifiers identify different sets of defects. **Method**: We used classifiers from four different families and the weighted accuracy diversity (WAD) technique to exploit diversity amongst classifiers. To combine individual predictions, we used the stacking ensemble technique. We used state-of-the-art knowledge in software defect prediction to build our ensemble models, and tested their prediction abilities against 8 publicly available data sets. **Conclusion**: The results show performance improvement using stacking ensembles compared to other defect prediction models. Diversity amongst classifiers used for building ensembles is essential to achieving these performance improvements.

Keywords

Software defect prediction, software faults, ensembles of learning machines, stacking, diversity

ESEM '16, September 08-09, 2016, Ciudad Real, Spain

 ISBN 978-1-4503-4427-2/16/09…$15.00
DOI: http://dx.doi.org/10.1145/2961111.2962610

1. INTRODUCTION

A software defect can cause programs to misbehave, leading to negative effects for the software industry. Defects that are fixed pre-release can potentially save companies from high repair costs and a bad reputation. The software industry spends billions of pounds annually finding and fixing defects. Defect prediction assists practitioners to promptly identify parts of software likely to contain defects, and act accordingly before the system is delivered to users. Prediction modelling has been used in several hundred studies conducted in software defect prediction in the last few decades. Some of the most recent work in software defect prediction literature has been covered in several meta-analysis and systematic literature reviews [2, 7, 12, 30].

As a matter of usual practice, researchers have used dozens of publicly available defect data sets, and tested different modelling techniques on those data sets. The standard approach to defect prediction is to use historical data containing quantitative measures about software modules. The historical data is fed into machine learners that produce prediction models. These prediction models can then be used to determine which software instances contain defects, by providing them new instances for which the defectiveness status is unknown to a model. Menzies et al. hypothesized that the current standard approaches used in defect prediction have reached their limits, and that new approaches are needed to make better predictions [14]. In this work, we focus on using ensemble techniques to make improved predictions.

Existing defect prediction studies generally do not consider whether different models find different sets of defects. Lessmann et al. did a study using 22 different machine learners, and concluded that the top 18 classifiers perform similarly [11]. The result that the majority of classifiers perform similarly suggests that it does not matter which classifiers are chosen to build prediction models. Similar average results that various classifiers produce could potentially hide different sets of defects that they identify. However, not all defects are the same. Diverse mathematical properties underlying different classifiers may have an effect on which

defects are found by one classifier, and not by others. Recently, Panichella et al. have confirmed that in cross-project defect prediction different techniques could be combined to find a wider range of defects than by just using single classifiers [17]. In within-project defect prediction, we previously confirmed that different machine learners detect very different subsets of defects [1]. Therefore, we now aim to identify whether different machine learners, when combined, can enhance the performance of prediction models.

To achieve our aim, we performed an experiment to establish whether ensemble techniques based on diversity and stacking can improve defect prediction. Ensembles of machine learners combine multiple classifiers and join their prediction outputs into a final solution. It is widely accepted by the machine learning community that ensemble models should contain diverse classifiers and that their outputs should be combined in a way that will amplify the correct decisions of single classifiers. Some ensemble techniques (e.g. Bagging) implicitly achieve diversity amongst classifiers by randomising a data set in each iteration of the algorithm. Since base classifiers are trained on different training sets, each classifier should make different predictions. In this case, the diversity amongst different classifier families is not accounted for. However, it is likely that ensemble modelling can be improved in the context of defect prediction. We use an explicit diversity scheme, which targets only the most diverse predictors from several different families to build our ensembles, since different classifiers discover different subsets of defects [1, 17]. Schemes like majority voting may not be suitable for defect prediction as unique subsets of defects are identified by some classifiers and not by others, making majority voting a non-optimal approach for combining ensemble outputs. Therefore, we use the stacking technique. Stacking performs a classification task with the prediction results previously made by individual classifiers. The output produced by this classification task gives the final prediction. Particularly, we want to address the following research questions:

RQ1 Can stacking ensembles based on explicit diversity improve prediction performance compared to other defect prediction models?

RQ2 How many classifiers combined into stacking ensembles provide good defect prediction models?

RQ3 How much diversity and which base classifiers are usually combined in stacking ensemble models?

In this work we make several contributions. First, we show that ensemble models based on classifiers from different families can improve defect prediction. We further explore how much diversity affects defect prediction models and what are the most popular classifiers chosen for building the stacking ensembles. We also show that only a few, but diverse classifiers, are sufficient to build effective ensemble models. This knowledge could help other researchers and practitioners to build improved defect prediction models.

Our paper is structured as follows. In the next section, we give an overview of software defect prediction and ensemble techniques. In the third section we detail our methodology followed by the results and discussion in the fourth section. We present the threats to validity of our experiment in the fifth section, give conclusiouns in the sixth section, and introduce ideas for future work in the seventh section.

2. BACKGROUND

Software defect prediction uses independent and dependent variables, and mathematical models to predict error-prone locations in software. Independent variables are quantitative measures of software, generally depicting the size and complexity of its components. Size, complexity, and CK object-oriented are the most common metrics used in studies of defect prediction [12]. Fenton and Neil showed that size is in a complex relationship with defects, resulting in studies that range from size giving good prediction to very poor prediction results [3]. Shepperd criticised complexity measures as being a proxy for several other metrics used in defect prediction [23]. Much effort has been put to engineering new metrics for defect prediction. Recently, Shippey presented a new metric based on the Java abstract syntax tree and showed usefulness in predicting a specific subsets of defects [26]. Zimmermann and Nagappan used graph theory, showing that software modules with a greater degree of centrality tend to be more defective [37]. Similarly, Petrić and Galinac used graph theory to show that some graph structures are more related to defects than others [19]. However, there is no definite agreement about which metric is superior for defect prediction. Most defect prediction studies tend to use a combination of available metrics.

Independent variables are usually combined with dependent variables in the form of a data set. Each data set contains a set of software units (modules), where each module is described with its metrics (independent variables) and the corresponding defectiveness (dependent variable). The dependent variable can be a number depicting the density of defects contained in a module, or a flag stating whether a module is defective or not. The systematic collection of such data is a complex and time consuming task. To tackle this problem, the PROMISE repository has been established containing a collection of publicly available defect data. Although very popular, some PROMISE data sets have been shown to be of low quality [5, 18]. However, for the sake of comparability with other studies, many researchers have used the original versions of the data sets available at PROMISE [7].

Researchers mostly use machine learning classifiers in the context of defect prediction [30]. Classifiers are first trained by using historical defect data, and then exploited to make predictions on the new data, previously not seen by a model [28, 33]. Lessmann et al.'s comprehensive study, which was performed using 22 different classifiers, showed no significant difference among the top 18 classifiers [11]. Lessmann et al. have presented average figures for the prediction performances of their classifiers. We showed that these performance figures hide the sets of defects identified by using different prediction models [1]. Specifically, we demonstrated that different classifiers are capable of finding different defects, where some subsets of defects are unique to a specific classifier. Panichella et al.'s study reported similar conclusions in the context of cross-project defect prediction [17].

Following our findings, and those of Panichella et al., we explore the use of ensembles of machine learners. The idea of ensemble models is to combine multiple single classifiers with the aim of improving the predictive performance of single classifiers [15, 21]. In the last few years, ensembles of machine learners have been occasionally used in software defect prediction. One of the fundamental motivations for using ensembles is the performance bottleneck [14] when us-

ing single classifiers. Wang et al. conducted a comparison study between popular ensemble classifiers and some single classifiers [31]. They showed that in many cases ensemble classifiers outperform other classifiers, including the single Naïve Bayes algorithm. Kultur et al. presented an ensemble of neural networks with associative memory that achieve more accurate and stable results compared to neural networks themselves [8]. More recent work also demonstrates the efficacy of ensemble learners against more conventional methods such as Support Vector Machines [10]. Particularly, Laradji et al. demonstrated that ensemble classifiers made up of carefully devised learners and using a few useful features can achieve improved results over other conventional models. The reasons for using ensembles in predictive modelling are covered in detail by Polikar [20].

Two things should be carefully considered when building ensemble models. First, ensembles should be built from diverse classifiers. Ensembles should include classifiers that make different incorrect predictions (because classifiers that make the same prediction errors do not add any information). Second, combining the outputs from all classifiers should be done in a way that encourages the correct decisions are amplified and ignores incorrect decisions. Since different classifiers find different defects, techniques commonly used in defect prediction for combining classifier outputs, such as majority voting, should be reconsidered. Current ensemble models in software defect prediction are not specifically designed to combine prediction outputs in such a way that will amplify correct predictions. If several prediction models have uniquely identified different sets of defects, then majority voting will not be a suitable technique to increase prediction performance. On the contrary, some of the defects uniquely identified by single classifiers will now be misclassified, downgrading the overall performance of the ensemble models. Combining the decisions of individual classifiers can be achieved using other techniques rather than majority voting. In this study, we use the stacking approach first introduced by Wolpert [34]. Stacking uses a two layer approach, where the first layer is constituted of individual classifiers, all trained on the same training data. The second layer, also called the meta layer, uses the output predictions of individual classifiers from the first layer as an input. This input is fed into the second layer classifier which then makes the final predictions. Therefore, the stacking approach seeks patterns in predictions made by the first layer, rather then ignoring classifiers that have minority "votes". Consequently, if a specific subset of defects is detectable only by one classifier, stacking will still have an opportunity to correctly classify such instances. The majority-voting approach would certainty misclassify such instances, since all but one of the classifiers would predict non-defective.

Many techniques for measuring and achieving higher diversity have been proposed by Kuncheva and Whitaker [9]. Some of these measures are Correlation diversity, Q-statistics, Disagreement and Double Fault Measures, Entropy Measures, Kohavi-Wolpert Variance, Weighted Accuracy and Diversity (WAD), etc. Kuncheva and Whitaker argue that there is no diversity measure that consistently correlates with higher accuracy. They recommend the use of Q-statistics because of its simplicity and intuitive meaning. WAD is another relatively simple diversity measure proposed by Zeng et al. [35], who showed that using WAD in combination with a bagging approach can boost prediction performance. We use WAD in this work as a diversity measure in defect prediction.

3. METHODOLOGY

3.1 Data sets

We used several public data sets from PROMISE[1], the repository commonly used by researchers in software defect prediction. We selected 8 data sets shown in Table 1, which come from different domains, to ensure diversity among possible defects that can appear in each project. We performed data cleaning to remove software modules that comply with the following rules:

- $LOC = 0$
- $AnyNumericalMetric < 0$
- $CC_{avg} > CC_{max}$
- $NOC > LOC$
- $NPM > WMC$

We remove instances where lines of code (LOC) is 0, or any numerical metric is negative as suggested by Shepperd et al. [25]. We additionally remove instances where the average cyclomatic complexity (CC_{avg}) exceeds the maximal cyclomatic complexity (CC_{max}), number of comments (NOC) is greater than number of lines of code and number of public methods for a class (NPM) is greater than weighted methods per class (WMC).

3.2 Base classifiers

We experimented with four different classifiers, namely Naïve Bayes , C4.5 decision tree, K-nearest neighbour, and sequential minimal optimisation. These four were chosen since classifiers from different "families" were previously successful in finding different defects [1, 17]. Naïve Bayes belongs to a family of linear classification techniques, where the prediction of a model is made according to conditional probabilities. It requires all variables to be categorical and assumes full independence among them. C4.5 is a tree-based learning algorithm that produces a structured classification model on which all predictions are based. The C4.5 algorithm uses information theory to make an optimal decision about node splitting on the attribute which best separates the data. K-nearest neighbour makes predictions based on the class values of closest neighbours. It uses a distance algorithm to find K nearest neighbours of the instance being predicted, and then assigns to that instance the highest occurring class of its neighbours. Sequential minimal optimisation is used for training Support Vector Machines in a more nearly optimal way than its predecessor algorithms. Support Vector Machine algorithms are designed for solving non-linear problems by mapping data points into a higher-dimensional space and separating the classes with a linear hyper-plane. The hyper-plane is usually chosen to maximise the distance between classes.

From each base classifier we have built several different models, changing the classifier parameters to address two important ideas. First, parameter tuning has been shown

[1]http://openscience.us/repo/

Table 1: PROMISE data sets used in our experiment

Id	Data set	Language	# of modules before cleaning	# of modules after cleaning	% loss due to cleaning	% defective methods after cleaning	Description
1	ant-1.5	Java	293	292	0.3	11.0	A Java library and a command tool commonly known for building Java programs.
2	ant-1.6	Java	351	350	0.3	26.3	Same as ant-1.5
3	ant-1.7	Java	745	742	0.4	22.4	Same as ant-1.5
4	jedit-4.1	Java	312	312	0.0	25.3	JEdit is a text editor mostly used as a programming tool.
5	jedit-4.2	Java	367	367	0.0	13.1	Same as jedit-4.1
6	tomcat	Java	858	852	0.7	9.0	Apache tomcat is a web-server for running Java programs.
7	xalan-2.5	Java	803	797	0.7	48.6	Xalan is a library for transforming XML documents into HTML, text or other XML document types.
8	xalan-2.6	Java	885	880	0.6	46.7	Same as xalan-2.5

to have an important role when building prediction models [6, 27]. Support Vector Machine classifiers are known to perform badly if not tuned. Decision trees might produce over-optimistic models if the number of possible branches is unlimited. K-nearest neighbour has problems with unbalanced data since the probability of predicting the majority class is higher. Naïve Bayes can use different kernel estimators to convert continuous into nominal data. Second, different parameters enrich diversity among classifiers, possibly making mistakes in different instances, which is a valuable characteristic when building ensembles [20]. When more classifiers make mistakes on different instances, a strategic combination of these classifiers may reduce the total error. In total, 15 base classifiers were used, each coming from one of the four families and using different model parameters.

We used two different parameters for a Naïve Bayes (NB) learner. One parameter uses a kernel density estimator rather than normal distribution for continuous attributes. The second parameter uses supervised discretisation for processing continuous attributes. The purpose of both parameters is to best split continuous attributes into nominal ones, since the Naïve Bayes classifier works only with categorical values. Two different parameters are usually tuned in K-nearest neighbour (kNN) learners, the k value and the nearest neighbour searching algorithm. The first parameter, k, denotes how many nearest neighbours the algorithm should take into consideration. The k parameter should be an odd number higher than 0. We chose three different values: 3, 5 and 7. Further increase of the parameter k may have a negative impact on our learners since defect prediction data is generally imbalanced. We left the default value for the nearest neighbour searching algorithm, which is the Euclidian distance. For the Support Vector Machines algorithm we have used sequential minimal optimisation (SMO). SMO uses less complex methods than its predecessors for training support vector machines, providing for the creation of faster models. For SMO we varied the complexity parameter C, assigning four different values: 1, 10, 25 and 50. The complexity parameter C controls the margin that separates the defective instances from the non-defective ones. If the C parameter is very small, the SMO algorithm will try to maximise the margin between two classes. On the other hand, higher C values will force the SMO algorithm to find margins that make the least amount of mistakes on the training data. Consequently, increasing the C values can lead to over-fitting, since the margin is tightly adjusted to the training data. Other parameters for SMO were left at their default values. The last family of classifiers is decision tree. The Weka implementation of decision tree is called J48. We varied the confidence factor of the J48 classifier using five different values: 0.25, 0.20, 0.15, 0.10, 0.05. Lowering the confidence factor, the J48 classifier will result in more pruning.

3.3 Diversity

Diversity is one of the key components when building ensemble models. Having multiple classifiers that make mistakes on the same instances does not add any additional information that we did not have with only a single classifier. Therefore, when building ensemble models, diversity should ensure that we include only classifiers that make mistakes on different instances. The common and simplest way of calculating diversity measures is between each pair of classifiers. An overall diversity is then calculated by averaging these pair-wise values. Several measures are frequently used to measure the diversity between pairs of classifiers. Correlation diversity measures the diversity between each pair of classifiers by obtaining the correlation between two classifier outputs. The Q-Statistic gives positive values when two classifiers make correct predictions, negative values for incorrect predictions and 0 for the maximal diversity between classifiers. Weighted accuracy and diversity (WAD), introduced by Zeng et al. [35], belongs to the family of diversity measures that compare two classifiers at a time. The WAD measure works on a similar principle to the F-measure by finding a weighted harmonic mean between accuracy and diversity:

$$WAD_{\alpha,\beta}(Acc, Div) = \frac{Acc \cdot Div}{\beta \cdot Acc + \alpha \cdot Div} \tag{1}$$

when $\alpha + \beta = 1$. The α and β parameters represent weights that control the importance of accuracy and diversity, where $\alpha > \beta$ gives more focus on accuracy, and $\alpha < \beta$ focuses more on diversity. When $\alpha > \beta$, the WAD measure combines multiple classifiers that are more accurate rather than diverse, and vice versa. In our experiment the accuracy in the WAD equation was replaced with precision, since accuracy is not a suitable measure for imbalanced data sets commonly found in SDP [4].

Diversity was computed among all pairs of classifiers as:

$$Div = \frac{2}{m(m-1)} \sum_{i=1}^{m-1} \sum_{j=i+1}^{m} div_{i,j} \quad (2)$$

where m denotes the number of base classifiers in an ensemble, and i, j indexes of each base classifier. Diversity between each pair of classifiers $div_{i,j}$ was calculated using the following equation:

$$div_{i,j} = \frac{N^{10} + N^{01}}{N^{00} + N^{11} + N^{10} + N^{01}} \quad (3)$$

where N^{11} represents the correct prediction of both classifiers, and N^{10} the situation where classifier i makes the correct prediction whilst classifier j incorrect. N^{00} denotes incorrect prediction of both classifiers, and finally N^{01} depicts correct prediction of the classifier j, but incorrect prediction of the classifier i.

3.4 Experimental setup

We used Song et al.'s and Gray's software defect prediction frameworks to build our predictors [6, 28]. The framework is divided into two parts, a scheme evaluation stage and a defect prediction stage. The scheme evaluation stage evaluates the performance of different classifiers to find the best prediction models among all classifiers. At this stage only training data is used, which is further split into the training' and validation sets. The test set is left out from the evaluation stage and used in the next, defect prediction stage. The defect prediction stage consists of the final prediction model that uses the test set for evaluating the model performance. Each experiment is run using 10 times 10 fold stratified cross-validation. Repeating an experiment 10 times, as well as using cross-validation, reduces the amount of variance in the evaluation of prediction models. The stratified technique guarantees the same distribution of the minority and the majority class as in the original data for each fold, preventing folds constituted only from the majority class. To ensure that we use only relevant attributes, we performed correlation-based feature selection on all training sets. A subset of attributes for each fold was recorded, and applied on the test set at the defect prediction stage.

Base learners can be combined in many different ways, as well as made up of a lot of different classifiers, however for an optimal model this step should be carried out with care. Although ensembles have still not been extensively used in software defect prediction, until now researchers have usually used some sort of majority voting as a decision making rule for ensembles [16, 22]. However, we showed in our previous work that using majority-like voting mechanisms results in some defects being ignored by such ensembles [1]. Similarly to Panichella et al. [17], we proposed the stacking approach when building ensemble based prediction models for software defect prediction. Still, combing all base learners into stacking may not be computationally and performance-wise optimal. More classifiers in an ensemble will inevitably prolong experiments, but it will not simultaneously guarantee better performance results. To address this issue, we selected only a subset of classifiers in a way that is explanied below.

Our stacking ensembles were built using 5 different measures depicted in Table 2. Each stacking ensemble was produced by combining multiple base classifiers according to their measure stated in Table 2. *Precision* and *MCC* belong to the basic group of measures, directly provided by the Weka API. For instance, when stacking is built using precision, only the most precise classifiers are put into the ensemble. Similarly for the other *Basic* measures. *WAD* and *Div* measures are part of the *Advanced* group of measures. These measures are not provided by the Weka API, rather are derived from Equations 1 and 2, respectively. The last measure is *Base*. *Base* indicates one base classifier that performs best among all the other base classifiers.

Table 2: Measures used for building ensemble models

	Measure	Full name	Type of measure
1	BASE	No measure	Basic
2	PRECISION	Precision	Base
3	MCC	Matthews correlation coeficient	Base
4	DIV	Diversity	Advanced
5	WAD	WAD	Advanced

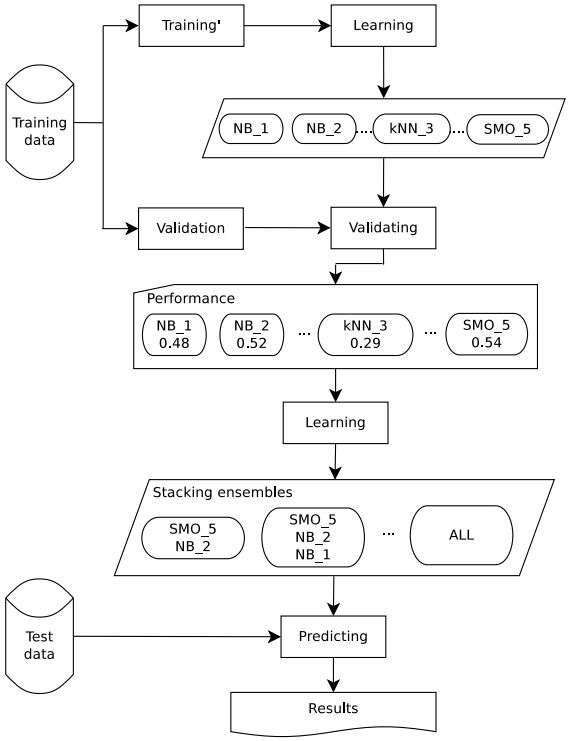


Figure 1: Stacking building

Figure 1 depicts the design used for building our stacking models. All base classifiers were first trained on the training' data sets, and evaluated using the validation set. Performances on the validation set were further sorted from the highest to the lowest value for each performance measure stated in Table 2. Starting from *Precision*, each measure was then taken to form a stacking ensemble. The two most precise base classifiers were taken to form the stacking ensemble. The model was trained on the whole training set, and finally evaluated on the test set. The experiment continued combining the three most precise base classifiers, forming the stacking ensemble, training on the whole training set and evaluating on the test set. After combining all

base classifiers, the experiment carried on the next measure from Table 2, *MCC*. The special *Base* group was evaluated slightly differently. Each base classifier was directly trained on the whole training set, and evaluated on the test set.

3.5 Performance measure

There has been a great debate on measuring the performance of prediction models [4, 13, 36]. When quantifying the performance of classifiers based on a categorical dependent variable, usually some performance measure, derived from confusion matrix, is reported. The confusion matrix is depicted in Table 3. However, some performance measures

Table 3: Confusion matrix

	Predicted defective	Predicted defect free
Observed defective	True Positive (TP)	False Negative (FN)
Observed defect free	False Positive (FP)	True Negative (TN)

are not suitable for use in the defect prediction context. Defect prediction data is commonly imbalanced, which makes some performance measures unusable (e.g. probability of detection [36]). On the other hand, frequently used measures such as precision and recall do not take all four quadrants into consideration, leaving space for making incorrect conclusions. Matthews correlation coefficient (MCC) is an appropriate measure when it comes to imbalanced data sets, and it captures all four quadrants of the confusion matrix [24]. MCC ranges from -1 to 1, where -1 indicates perfect disagreement, whilst 1 indicates perfect agreement between prediction and observation. The MCC value of 0 represents prediction no better than random. Given that MCC captures all quadrants of the confusion matrix, we believe that this measure is a trustworthy indicator of the prediction performance.

4. RESULTS AND DISCUSSION

We conducted a series of experiments to build the final ensemble models. Since we used the WAD measure, proper tuning of α and β parameters depicted in Equation 1 was required. Therefore, we ran a set of prediction models on all data sets changing both parameters. The parameters were changed in the range from 0 to 1 in steps of 0.05. The y-axis on Figure 2 depicts the change in MCC performance for all data sets we used. The x-axis represents the value of α parameter, where β parameter was changed automatically to satisfy $\alpha + \beta = 1$.

Figure 2 shows no compelling difference in MCC amongst all the data sets we used by changing the WAD parameters. The greatest differences were achieved on margins where $\alpha = 0$ and $\alpha = 1$. However, marginal values have already been covered using *Precision* and *Diversity* measures. Use of *Precision* can be compared to $\alpha = 1$ since in this case diversity is ignored. Similarly, when $\alpha = 0$ *Precision* is completely ignored and focus is on the diversity among base classifiers. Considering both *Precision* and *Diversity* as important factors when building ensembles, we set both parameters to 0.5. Setting $\alpha = \beta = 0.5$ ensures that both concepts are equally represented.

Table 4 shows the average performance values of single and ensemble classifiers. The columns contain different prediction techniques, whilst rows depict average MCC and

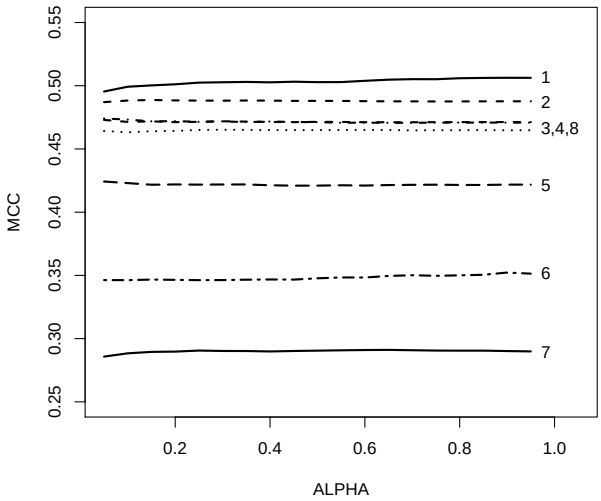


Figure 2: WAD sensitivity for all eight data sets

Table 4: Average MCC performance across all data sets

Data set		SC	BA	BO	PR	MCC	DIV	WAD
ant-1.5	Avg.	0.255	0.338	0.303	0.461	0.485	0.507	0.503
	Dev.	0.276	0.285	0.287	0.254	0.246	0.24	0.242
ant-1.6	Avg.	0.408	0.479	0.412	0.475	0.49	0.488	0.488
	Dev.	0.174	0.145	0.168	0.154	0.148	0.145	0.145
ant-1.7	Avg.	0.395	0.456	0.389	0.448	0.474	0.465	0.465
	Dev.	0.134	0.124	0.128	0.131	0.131	0.123	0.123
jedit-4.1	Avg.	0.427	0.433	0.403	0.459	0.466	0.471	0.471
	Dev.	0.191	0.187	0.181	0.185	0.178	0.182	0.182
jedit-4.2	Avg.	0.334	0.338	0.342	0.346	0.426	0.421	0.421
	Dev.	0.243	0.219	0.217	0.232	0.191	0.193	0.193
tomcat	Avg.	0.171	0.247	0.23	0.3	0.349	0.351	0.348
	Dev.	0.192	0.203	0.175	0.177	0.156	0.153	0.158
xalan-2.5	Avg.	0.257	0.355	0.35	0.274	0.293	0.29	0.29
	Dev.	0.115	0.108	0.1	0.109	0.107	0.107	0.106
xalan-2.6	Avg.	0.474	0.535	0.495	0.471	0.471	0.471	0.471
	Dev.	0.089	0.08	0.093	0.092	0.092	0.092	0.092
	Avg.	0.34	0.398	0.366	0.404	0.432	0.433	0.432
	Dev.	0.177	0.169	0.169	0.167	0.156	0.154	0.155

standard deviation values for each data set. To make our approach more readily comparable to other approaches, we trained two additional ensemble models commonly used in SDP, namely Bagging and Boosting. Both additional models were trained using the same parameters for the base classifiers as described in Section 3.2. *SC* represents the average values of all 15 single classifiers used, whilst *BA* and *BO* are the bagging and boosting approaches added for comparison, respectively. The rest of the table represents the other basic and advanced measures used, *Precision*, *MCC*, *Diversity*, and *WAD*, respectively. Average performance measures show that across all data sets our techniques achieve better results than the other approaches. Particularly, the average figures from Table 4 show that the *DIV* technique is better by 27.2%, *Bagging* by 8.9%, and *Boosting* by 18.5% compared to the *Single classifier* technique. For formal confirmation in favour of the *DIV* technique, we used Wilcoxon signed-rank test to statistically compare the differences [32]. The same form of test was previously used by Sun et al. in the context of defect prediction [29]. The alternative hypothesis tests whether for a given technique (single classifier *OR* bagging *OR* boosting), *DIV* technique performs

better than the other three techniques at significance level $\alpha = 0.05$. The p-values of the *Single Classifier* and *Boosting* techniques were less than 0.05, whilst the p-value for the *Bagging* technique was greater than 0.05. From these results we can conclude that the *DIV* technique is significantly better than the other techniques except *Bagging*. The same conclusions were derived for the WAD and MCC techniques using the same statistical approach.

Table 5: Relative increase in true positives of all techniques compared to Bagging

	BA	SC	BO	PR	MCC	DIV	WAD
ant-1.5	1.06	-0.20	-0.07	0.68	0.83	0.94	0.93
ant-1.6	4.95	-0.10	-0.02	0.09	0.20	0.22	0.21
ant-1.7	8.02	-0.11	-0.02	-0.00	0.21	0.24	0.24
jedit-4.1	3.65	-0.03	0.08	0.08	0.18	0.24	0.24
jedit-4.2	1.53	-0.03	0.14	0.24	0.78	0.78	0.78
tomcat	1.40	-0.08	0.31	1.28	2.01	2.02	2.03
xalan-2.5	25.21	-0.15	0.02	-0.20	-0.05	-0.08	-0.08
xalan-2.6	28.99	-0.11	0.02	-0.25	-0.19	-0.18	-0.18
Avg	9.35	-0.10	0.06	0.24	0.50	0.52	0.52

Table 6: Relative increase in false positives of all techniques compared to Bagging

	BA	SC	BO	PR	MCC	DIV	WAD
ant-1.5	1.01	0.19	0.14	0.87	1.07	1.15	1.17
ant-1.6	2.50	0.16	0.38	0.31	0.55	0.62	0.62
ant-1.7	4.41	0.16	0.41	0.09	0.54	0.73	0.72
jedit-4.1	1.95	-0.03	0.44	0.06	0.32	0.46	0.46
jedit-4.2	1.46	-0.06	0.41	0.49	1.35	1.45	1.45
tomcat	1.50	0.54	1.26	3.35	4.82	4.86	4.95
xalan-2.5	12.26	0.00	0.05	-0.13	0.09	0.05	0.05
xalan-2.6	8.35	-0.04	0.27	-0.45	-0.28	-0.25	-0.25
Avg	4.18	0.11	0.42	0.57	1.06	1.14	1.15

Having established the average performances of our models, we further investigated the effect size of all approaches. The effect size serves as a measure of how many defects can each model detect (true positives) for the price of misclassifying certain instances (false positives). To demonstrate effect sizes, we derived the confusion matrix for all runs and across all data sets. To make our comparison fair, we compared *Single Classifier*, *Boosting*, *Precision*, *MCC*, *DIV*, and *WAD* against the *Bagging* technique. The reason for this is that the *Bagging* technique achieved better results than *Single Classifier* and *Boosting*. Also, we want to compare our techniques against others that achieve best results in defect prediction. The first column in Table 5 shows the average number of true positives achieved by *Bagging*. The following columns show the relative improvement in the number of true positives for the techniques used in our study. Clearly, *DIV* and *WAD* techniques achieve better performances relative to *Single Classifiers*, *Bagging* and *Boosting* techniques. More precisely, *DIV* and *WAD* techniques can on average achieve a relative improvement of 0.52 compared to the *Bagging* technique. For instance, in the case of the tomcat data set, the *WAD* technique correctly identifies about two times more defects relative to *Bagging*. Table 6 on the other hand depicts an increase of false positives. Taken together, the tables of relative increase in true and false positives show that there is a trade-off between true positives and false positives. However, increasing the number of true positives in defect prediction is particularly challenging since data sets are known to be imbalanced. Although a higher false positive rate detection is discouraged, finding more real defects for the price of more false positives could be of benefit for some companies. This is especially true for companies where the cost of defects is extremely high.

We additionally compared single classifiers against our best two techniques, *DIV* and *WAD*. The reason is a frequent use of single classifiers in SDP. To make a fair comparison, we extracted only one classifier from each family that on average achieved the best MCC performance. From the Naïve Bayes family, the classifier with a kernel density estimator achieved the best MCC result in average across all data sets. K nearest neighbour with $k = 3$, SMO with $C = 50$, and J48 with $C = 0.10$ achieved the best average performances according to MCC. Single classifiers with the best average MCC performances, along with *DIV* and *WAD* techniques are shown in Table 7. Since *DIV* and *WAD* both achieved the same average MCC performances, they improved over Naïve Bayes by 18.2%, over KNN by 31.4%, over SMO by 22.3%, and finally over J48 by 20.1%. The Wilcoxon significance test, where the alternative hypothesis tests superior performance values of *DIV* and *WAD* over *Single Classifiers* was performed. With the $p = 0.05$ level of confidence we confirmed that the both techniques, *DIV* and *WAD*, are superior to all the other *Single Classifier* techniques.

Table 7: Average best single classifiers against DIV and WAD

	NB	KNN	SMO	J48	DIV	WAD
ant-1.5	0.316	0.169	0.385	0.279	0.507	0.503
ant-1.6	0.482	0.327	0.445	0.397	0.488	0.488
ant-1.7	0.432	0.355	0.407	0.411	0.465	0.465
jedit-4.1	0.432	0.408	0.499	0.400	0.471	0.471
jedit-4.2	0.370	0.398	0.321	0.334	0.421	0.421
tomcat	0.256	0.190	0.029	0.264	0.351	0.348
xalan-2.5	0.169	0.289	0.264	0.314	0.290	0.290
xalan-2.6	0.473	0.498	0.481	0.484	0.471	0.471
Avg	0.366	0.329	0.354	0.360	0.433	0.432

RQ1. Can stacking ensembles based on explicit diversity improve prediction performance compared to other defect prediction models?
From the analysis of MCC performance and relative improvements in effect sizes we can conclude that stacking ensembles can indeed improve prediction performance.

In our final experiment we test how many classifiers, and of which family, used for building stacking ensembles are needed for these ensembles to perform well. It may be the case that combining only a few classifiers, that are most precise and diverse, is sufficient for achieving good performance results. This would have practical benefits reducing the training time of such ensemble classifiers. Since we built stacking ensembles of all possible sizes (combining from 2 to 15 single classifiers used in this study), it is now possible to compare their prediction performances. Furthermore, it is possible to analyse which single classifiers are often picked by the ensemble. In the context of this analysis, we investigate only the WAD technique, since it is based on both aspects that are in the focus of our analysis: diversity and precision. Additionally, the WAD technique performed as

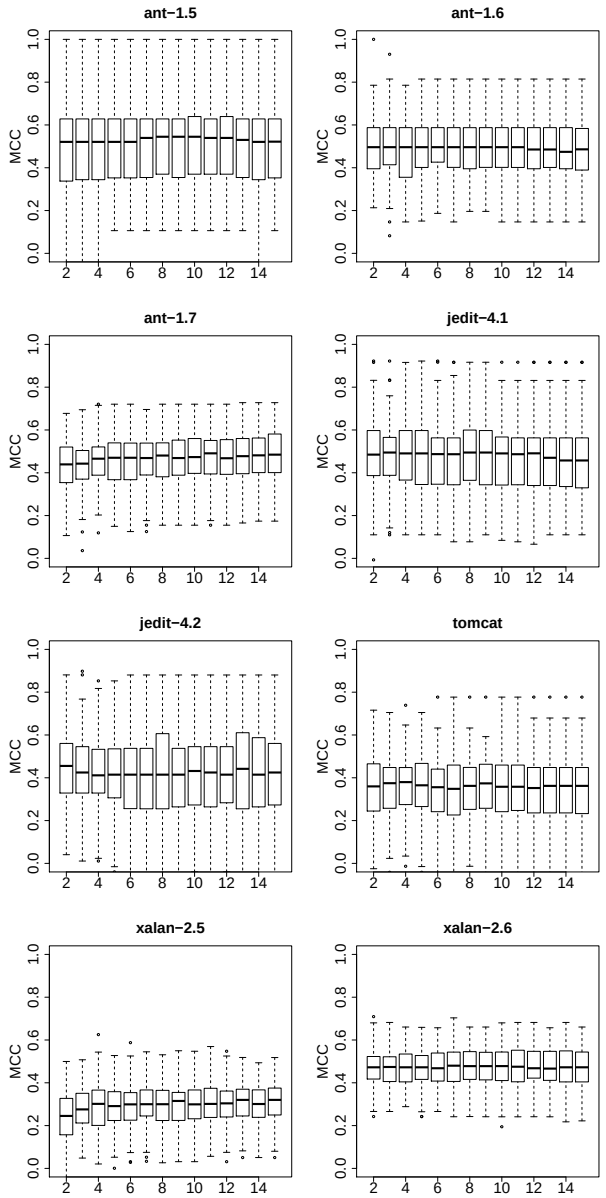


Figure 3: Prediction performance of WAD groups of classifiers. The x-axis represents the size of a group.

well as our other ensemble techniques. Figure 3 presents prediction performance box-plots for all 8 data sets used in our analysis. The x-axis depicts the size of each group using the WAD technique, whilst the y-axis shows the MCC performance. From the figure it is clear that small groups, of just two or three classifiers, combined into the stacking ensemble perform well. Combining only three classifiers into a stacking ensemble gives prediction performance not significantly worse than combining more classifiers, as Figure 3 demonstrates. This statement is valid for all data sets used in our experiment. Achieving such results is important since training only three classifiers, and using an additional classifier in the stacking meta-layer, reduces the time for building a prediction model.

RQ2. How many classifiers combined into stacking ensembles provide good defect prediction models?
Our experiment showed that adding more than three classifiers into the stacking ensembles does not significantly increase the prediction performance.

Table 8: Frequency of the individual classifiers appearing in the stacking ensembles of size 3

#	Data set	Classifier	Frequency (%)
1	ant-1.5	NB default parameters	100
2	ant-1.5	SMO -C 1, else default parameters	90
3	ant-1.5	kNN k=3, else default parameters	34
1	ant-1.6	SMO -C 1, else default parameters	100
2	ant-1.6	kNN k=3, else default parameters	85
3	ant-1.6	NB -D	77
1	ant-1.7	NB -D	100
2	ant-1.7	SMO -C 1, else default parameters	100
3	ant-1.7	kNN k=3, else default parameters	99
1	jedit-4.1	NB -D	100
2	jedit-4.1	SMO -C 1, else default parameters	95
3	jedit-4.1	kNN k=3, else default parameters	90
1	jedit-4.2	NB -D	100
2	jedit-4.2	SMO -C 1, else default parameters	89
3	jedit-4.2	NB default parameters	59
1	tomcat	NB -D	100
2	tomcat	kNN k=3, else default parameters	45
3	tomcat	NB default parameters	44
1	xalan-2.5	kNN k=3, else default parameters	83
2	xalan-2.5	SMO -C 1, else default parameters	56
3	xalan-2.5	J48 -C 0.25, else default parameters	41
1	xalan-2.6	kNN k=3, else default parameters	97
2	xalan-2.6	NB -D	95
3	xalan-2.6	NB default parameters	86

Finally, we investigated the base classifiers that form our stacking ensemble in order to find the ones that are frequently chosen by ensembles. For the purpose of this analysis we again used the WAD technique with stacking ensembles of size 3. Table 8 shows the first three classifiers commonly used for building stacking ensembles for all 8 data sets used. The Naïve Bayes classifier has constantly been chosen by the stacking ensemble, with an average frequency of 86% across all data sets. This suggests that Naïve Bayes classifiers perform well across all data sets, and increase diversity in ensembles. Some variants of SMO have also been repeatedly chosen by stacking ensembles, often with different parameter settings. Interestingly, the frequently used decision tree classifier J48 was not dominant for any of the data sets.

RQ3. How much diversity and which base classifiers are usually combined in stacking ensemble models?
Although only a small proportion of classifiers are needed to build a stacking ensemble that performs well, diversity among classifiers seems to have an important role in this. The frequency table (shown in Table 8) suggests that the ensembles of size 3 are usually combined with classifiers from different families (e.g. in 90% of the runs Naïve Bayes , SMO and kNN combine together in ensembles for Jedit-4.1).

5. THREATS TO VALIDITY

We consider several internal and external threats to validity. In our study, we replaced accuracy with precision in the *WAD* equation. Although, the authors of the *WAD* technique have used accuracy as the measure of how correct a classifier is, they used data sets with a relatively balanced level of class instances. However, software defect prediction often deals with highly imbalanced data sets, and therefore the accuracy measure can be misleading. For that reason we decided to use precision as the measure of how correct a classifier is.

We did not perform a full parameter search to find optimal values for each learner. Parameter tuning is an important step when building prediction models, however it is also performance demanding. To minimise the threat of model optimisation, we changed the most important parameters of the classifiers for each family. By changing the basic parameters we minimised the threat of building models with poor generalisation abilities.

We evaluated our models using the Matthews correlation coefficient measure. There are many ways to measure prediction models, and all come with certain strengths and weaknesses. Although some researchers would argue that one measure is better than another, we decided to use MCC since it covers all aspects of the confusion matrix. Taking into consideration true positives, true negatives, false positives and false negatives at the same time, we believe that our reported values do not hide aspects of the predictions.

Our study is also limited to several data sets from the PROMISE repository. It is possible that by using different data sets we would come to different conclusions. However, most of these data sets have been extensively used in many other SDP studies, giving us and other researchers possibilities to compare results. We want to stress that we performed cleaning of the data sets to remove erroneous data points. By using these data sets, and our cleaning steps given in Section 3.1, others are able to replicate our work and confirm or extend our results.

6. CONCLUSION

Ensembles of machine learners composed of classifiers from different families can outperform some traditional ensemble techniques in defect prediction. Using the stacking approach and diverse classifiers, we showed relative increase in number of identified defects compared to the commonly used bagging technique. Such a stacking ensemble does not require many base classifiers, however our results suggest those classifiers have to be diverse and from different classifier families. Particularly, Naïve Bayes and SMO seem to work well in combination. Although the predictive limitation of using single classifiers has been reached, ensembles of machine learners leave space for improvements. This is due to the fact that different classifiers identify different subsets of defects.

Our findings have important implications in the way future prediction models should be built. We suggest use of ensembles of machine learners composed of different classifiers. Majority-voting should not be used to combine predictions of individual classifiers since this approach may miss some subsets of defects found only by some classifiers and not others. The use of the stacking approach could reduce the number of misclassifications compared to majority-voting. Researchers and practitioners can use our findings to build better defect prediction models. They can also use the fact that different classifiers find different defects and develop even better ways of combining individual classifiers.

7. FUTURE WORK

We plan to extend this work with several enhancements. First, we want to extend the experiment by using more classifiers and apply full parameter search to them. Adding new classifiers may help detect new families of defects, or even help the stacking approach to finding "better" patterns from the first layer. Second, we plan to use different diversity measures, such as Q-statistics, and investigate how much use of those measures affects defect prediction. We do not yet know what effect different diversity measures have on the correct classification of defective instances. Last but not least, we should establish why some defects are identified by some classifiers and missed by others. This should help in building better ensemble techniques that can find a variety of defects, and at the same time reduce the number of false positives, hence achieve higher precision and recall at the same time. Such understanding is necessary if we are to build models with superior performances over simple single models. We should further investigate whether some specific defects are found only by our approaches, and missed by other ensemble techniques. Models with that power may break the performance ceiling identified in 2008 [14].

8. ACKNOWLEDGEMENTS

This work was partly funded by a grant from the UK's Engineering and Physical Sciences Research Council under grant number: EP/L011751/1

9. REFERENCES

[1] D. Bowes, T. Hall, and J. Petrić. Different classifiers find different defects although with different level of consistency. In *Proceedings of the 11th International Conference on Predictive Models and Data Analytics in Software Engineering*. ACM, 2015.

[2] C. Catal and B. Diri. A systematic review of software fault prediction studies. *Expert Systems with Applications*, 36(4):7346 – 7354, 2009.

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

[4] D. Gray, D. Bowes, N. Davey, Y. Sun, and B. Christianson. Further thoughts on precision. In *Evaluation Assessment in Software Engineering (EASE 2011)*, pages 129–133, April 2011.

[5] 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. In *Evaluation Assessment in Software Engineering (EASE 2011)*, pages 96–103, April 2011.

[6] D. P. H. Gray. *Software defect prediction using static code metrics: formulating a methodology*. PhD thesis, University of Hertfordshire, 2013.

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

[8] Y. Kultur, B. Turhan, and A. Bener. Ensemble of neural networks with associative memory (enna) for estimating software development costs. *Knowledge-Based Systems*, 22(6):395 – 402, 2009.

[9] L. I. Kuncheva and C. J. Whitaker. Measures of diversity in classifier ensembles and their relationship with the ensemble accuracy. *Machine Learning*, 51(2):181–207.

[10] I. H. Laradji, M. Alshayeb, and L. Ghouti. Software defect prediction using ensemble learning on selected features. *Information and Software Technology*, 58:388 – 402, 2015.

[11] 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 2008.

[12] R. Malhotra. A systematic review of machine learning techniques for software fault prediction. *Applied Soft Computing*, 27:504 – 518, 2015.

[13] 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 Transactions on Software Engineering*, 33(9):637–640, Sept 2007.

[14] T. Menzies, B. Turhan, A. Bener, G. Gay, B. Cukic, and Y. Jiang. Implications of ceiling effects in defect predictors. In *Proceedings of the 4th International Workshop on Predictor Models in Software Engineering*, PROMISE '08, pages 47–54. ACM, 2008.

[15] L. L. Minku and X. Yao. Ensembles and locality: Insight on improving software effort estimation. *Information and Software Technology*, 55(8):1512 – 1528, 2013.

[16] A. Mısırlı, A. Bener, and B. Turhan. An industrial case study of classifier ensembles for locating software defects. *Software Quality Journal*, 19(3):515–536, 2011.

[17] A. Panichella, R. Oliveto, and A. De Lucia. Cross-project defect prediction models: L'union fait la force. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE 2014)*, pages 164–173, Feb 2014.

[18] J. Petrić, D. Bowes, T. Hall, B. Christianson, and N. Baddoo. The jinx on the nasa software defect data sets. In *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*, EASE '16, pages 13:1–13:5, New York, NY, USA, 2016. ACM.

[19] J. Petrić and T. G. Grbac. Software structure evolution and relation to system defectiveness. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, EASE '14, pages 34:1–34:10. ACM, 2014.

[20] R. Polikar. Ensemble based systems in decision making. *Circuits and Systems Magazine, IEEE*, 6(3):21–45, Third 2006.

[21] L. Rokach. Taxonomy for characterizing ensemble methods in classification tasks: A review and annotated bibliography. *Computational Statistics and Data Analysis*, 53(12):4046 – 4072, 2009.

[22] C. Seiffert, T. Khoshgoftaar, and J. Van Hulse. Improving software-quality predictions with data sampling and boosting. *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, 39(6):1283–1294, Nov 2009.

[23] M. Shepperd. A critique of cyclomatic complexity as a software metric. *Software Engineering Journal*, 3(2):30–36, March 1988.

[24] M. Shepperd, D. Bowes, and T. Hall. Researcher bias: The use of machine learning in software defect prediction. *IEEE Transactions on Software Engineering*, 40(6):603–616, June 2014.

[25] M. Shepperd, Q. Song, Z. Sun, and C. Mair. Data quality: Some comments on the nasa software defect datasets. *Software Engineering, IEEE Transactions on*, 39(9):1208–1215, Sept 2013.

[26] T. J. Shippey. *Exploiting Abstract Syntax Trees to Locate Software Defects*. PhD thesis, University of Hertfordshire, 2015.

[27] C. Soares, P. B. Brazdil, and P. Kuba. A meta-learning method to select the kernel width in support vector regression. *Machine learning*, 54(3):195–209, 2004.

[28] Q. Song, Z. Jia, M. Shepperd, S. Ying, and J. Liu. A general software defect-proneness prediction framework. *Software Engineering, IEEE Transactions on*, 37(3):356–370, May 2011.

[29] Z. Sun, Q. Song, and X. Zhu. Using coding-based ensemble learning to improve software defect prediction. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 42(6):1806–1817, Nov 2012.

[30] R. S. Wahono. A systematic literature review of software defect prediction: Research trends, datasets, methods and frameworks. *Journal of Software Engineering*, 1(1):1–16, 2015.

[31] T. Wang, W. Li, H. Shi, and Z. Liu. Software defect prediction based on classifiers ensemble. *Journal of Information & Computational Science*, 8(16):4241–4254, 2011.

[32] F. Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, 1(6):80–83, 1945.

[33] I. H. Witten and E. Frank. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2005.

[34] D. H. Wolpert. Stacked generalization. *Neural Networks*, 5(2):241 – 259, 1992.

[35] X. Zeng, D. F. Wong, and L. S. Chao. Constructing better classifier ensemble based on weighted accuracy and diversity measure. *The Scientific World Journal*, 2014.

[36] H. Zhang and X. Zhang. Comments on "data mining static code attributes to learn defect predictors". *IEEE Transactions on Software Engineering*, 33(9):635–637, 2007.

[37] T. Zimmermann and N. Nagappan. Predicting defects using network analysis on dependency graphs. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 531–540. ACM, 2008.

6.4 Summary of the Research Questions

RQ2(a) Can stacking ensembles based on explicit diversity improve prediction performance compared to other software defect prediction models?

The models that I have built using stacking ensembles show improved MCC performances and an increase in the relative number of defects they can predict. There is, however, an increase in the false positive rates by stacking ensembles. Taken together, the relative increase in true and false positives show that there is a trade–off between true and false positive predictions. However, increasing the number of true positives in software defect prediction is particularly challenging since data sets are known to be imbalanced. Although a higher false positive rate detection is discouraged, finding more real defects for the price of more false positives could be of benefit for some companies. This is especially true for companies where the cost of defects is extremely high.

RQ2(b) How many classifiers combined into stacking ensembles provide good software defect prediction models?

My findings show that adding more than 3 classifiers into the stacking ensembles does not significantly increase the prediction performance. This is true for all datasets used in the experiment. Achieving such results is important since training only 3 classifiers, and using an additional classifier in the stacking meta-layer, reduces the time for building a prediction model.

RQ2(c) How much diversity and which base classifiers are usually combined in stacking ensemble models?

Classifiers selected in the stacking ensembles are typically coming from different classifier families. For instance, on no occasion is the SMO classifier combined with another SMO classifier with different tuning. The same is true for the kNN classifier. Naïve Bayes is only occasionally combined with another Naïve Bayes classifier with different tuning. This result suggests that diversity is paramount when generating stacking ensembles. Some classifiers are selected more than others. The frequently used decision tree classifier (J48) does not appear to be dominant for any of the datasets used in my study.

6.5 Summary of My Contributions

My results strongly suggest that stacking ensembles should be used in software defect prediction. Stacking ensembles do not require many base learners, however the results suggest those learners have to be diverse and from different classifier families. The Naïve Bayes and SMO learners seem to work well in combination. Majority-voting should not

be used to combine predictions of individual classifiers since this approach will miss some subsets of defects found only by some classifiers and not others. The use of the stacking approach can reduce the number of misclassification compared to majority-voting.

6.6 Summary of the Contributions to the Paper

I devised the methodology of this study and developed an experiment which I independently ran. I collected and analysed all results of the experiment. I wrote the first draft of the paper in full. David Bowes assisted in improving the methodology by checking its correctness and providing useful guidance in building the prediction models. He also suggested a further analysis in terms of the frequency count of the classifiers selected into an ensemble (i.e. RQ3 of the paper). Tracy Hall provided most of the refinements to the paper. Bruce Christianson and Nathan Baddoo provided useful comments to improve the paper.

Chapter 7

Potential Improvements to Ensembles for Software Defect Prediction

In Chapter 5 I established that different classifiers have the ability to find unique subsets of defects. This ability has led to better understanding about how to construct software defect prediction models which outperform state-of-the-art models (see Chapter 6). However, there are many potential avenues to improve these prediction models. A limited number of previous studies have been focused on understanding how well a classifier suits software defect prediction. Most studies, including my own, apply different classification techniques with the aim to achieve better prediction performances. The analysis in this chapter suggests that some classifiers are more conservative when making predictions than others. For example, for particular datasets SVMs predict fewer individual defects compared to other classifiers, but also produce the smallest number of false positives. On the other hand, Naïve Bayes tends to predict a high number of unique defects for the price of producing many false positives. The work in this chapter is an initial attempt to explore why some classifiers are more successful than others in predicting individual defects. This could help future researchers to improve the design of ensembles presented in Chapter 6 by combining the most suitable classifiers.

7.1 Background

Several classifiers have been reported to perform well in software defect prediction. Regression via Classification [Bibi et al. 2008], Orthogonal Sparse Biagrams Markov Models [Mizuno and Kikuno 2007] and TreeDisc techniques [Khoshgoftaar and Hulse 2009] achieve good prediction performances. On the other hand, some classifiers have been reported to

perform less well. In particular, SVMs have been frequently reported to obtain low prediction performances in software defect prediction [Hall et al. 2012]. Decision trees have been shown to struggle with highly imbalanced data [Japkowicz and Stephen 2002]. Random Forest, however, is a good choice for highly imbalanced datasets [Khoshgoftaar et al. 2007]. Overall, no standalone classifier performs best for all defect data.

Previous studies have, however, mostly focused on assessing performance measures only. For example, Lessmann et al. [2008a] and Malhotra [2015] both compared 40 models in terms of their Area Under Curve measure (AUC) alone. An exception to this is Gray et al. [2010] who manually investigated the predictions made by SVM classifiers. Using Principal Component Analysis (PCA) and comparing the confidence of predictions, they found that on average the SVMs make well motivated predictions (i.e. real defects are predicted with greater confidence than false positives). This would not be possible to establish by analysing prediction performances only. The work of Gray et al. [2010] illuminates the importance of deeper manual analysis of individual predictions.

In this work I aim to analyse individual defects predicted by classifiers to find an explanation why a classifier performs well or poorly. Despite the obvious reason that each classifier constructs a different decision boundary, I investigate the scenarios in which some classifiers could under-perform in defect prediction due to their inherent characteristics and the properties of the data. This analysis offers some valuable insights about which classifiers should be chosen to compose an ensemble model. Future research will be needed to establish the usefulness of the identified characteristics in this chapter. For example, the following question could be explored: "Can stacking take advantage of the confidence of the individual predictor?".

I use two exploratory data analyses approaches in this work, namely PCA and density plots. The PCA is a well established technique for dimensionality reduction, where most informative attributes are converted into most significant principal components (or factors) [Wold et al. 1987]. The first few principal components typically explain most of the variance in data. The PCA is useful as the first two or three principal components can easily be visualised. Density plots, on the other hand, visualise the distribution of data over a continuous interval. Density plots are similar to histograms, however they use kernel smoothing which helps in removing noise.

7.2 Methodology

In this experiment I use four classifiers: Naïve Bayes, PART, Support Vector Machines, and Random Forest. I select these four classifiers because they build models based on different

mathematical properties. The four classifiers were used in my previous studies (see Sections 5 and 6) and are amongst the top commonly used classifiers in software defect prediction [Wahono 2015]. The four classifiers belong to different classifier families, which allows for a variety of predictions to be made as demonstrated in Chapter 5. The classifiers are described in more detail in Section 3.2.

In this experiment I tune Random Forest and Support Vector Machine classifiers. Similar to studies in Chapters 5 and 6, I vary the number of trees from 50 to 200 in steps of 50. For SVM using a radial base function I tune γ from 0.25 to 4 and C from 2 to 32. I do not tune Naïve Bayes or PART classifiers since they are known to perform well with the default parameters. I repeat each experiment 100 times, using the 10-fold stratified cross validation. In this study the WEKA tool[1] is used to run the experiments.

I use 14 open source datasets from the PROMISE repository which I clean prior to the analysis. The datasets are listed in Table 4.2. In order to analyse individual predictions made by different classifiers I adjust the datasets in the following way. Prior to running the experiments each data point is given a unique identifier. During predictions I record the fold and run number for each data point, as well as the prediction outcome of a classifier. The results of the experiments are then combined using the R package `reshape2`[2] [Wickham 2017]. As the results are in the long format I first convert them to the wide format. Using the `dcast` function, the predictions from 100 runs get summarised in one line describing how many true positive predictions a classifier has made. For example, if SVMs predict a particular defect correctly in 90 out of the 100 runs, the data point is assigned 0.9. Data points with values equal or over the balance of probability (0.5) are considered as predicted defective.

To analyse the predictions of individual classifiers I create a specific label *category* to distinguish those predictions. Table 7.1 summarises 10 categories I define for the purpose of this analysis. Each true positive prediction identified by a classifier is labelled with one of the categories in Table 7.1. For example, the defect with $rowid = 320$ in the *poi* system gets assigned to $category = RFdef$, as the Random Forest classifier predicts it as defective. Defects correctly predicted by Random Forest and Naïve Bayes are assigned both $RFdef$ and $NBdef$. The categories are used for visualising the results using the PCA and density plots.

On the PCA plots I only present the following five categories: *NBdef, PARTdef, RFdef,* and *SVMdef.* Additionally, I plot the false negative predictions. I do not plot the non-defective data points which were not predicted by the unique classifiers as my primary focus is on the

[1] Weka version 3.9.1
[2] reshape2 version 1.4.3

Category	Alias	Description
NBdef	NB	TP correctly predicted by NB
SVMdef	SVM	TP correctly predicted by SVM
PARTdef	PART	TP correctly predicted by PART
RFdef	RF	TP correctly predicted by RF
NBnondef	-	TN correctly predicted by NB
SVMnondef	-	TN correctly predicted by SVM
PARTnondef	-	TN correctly predicted by PART
RFnondef	-	TN correctly predicted by RF
def	-	Defective
nondef	-	Non defective

Table 7.1 Categories of predictions by different classifiers

defective instances. Another reason is that there are many non-defective data points which makes the distinction between defective and non-defective instances difficult to observe.

Prior to visualising individual predictions I remove the outliers and standardise the data. Outliers distort PCA plots to the level where it is not possible to distinguish individual predictions made by classifiers. I set the minimum threshold for removing outliers to 3σ. The majority of removed outliers are false negative predictions. I standardise the data to ensure that no attribute dominates any of the PCA plots.

However, PCA plots have a limited power to visualise data with no clear patterns. Therefore, in addition to the PCA plots, I develop a visual approach which aims to describe individual predictions using density plots. I break each density plot by an attribute and calculate a density function to be plotted. In other words, I render density functions of individual classifier predictions on the same graph for each attribute separately. This makes it possible to compare the attribute values for which each classifier correctly or incorrectly predicts a defect. My technique allows the selection of an attribute to be rendered on a plot. Below the same plot depicting the categories described in Table 7.1, I render the plot showing the density functions of true defective and non-defective instances (*def* and *nondef* categories depicted in Table 7.1). Both plots are aligned, where the x-axis denotes the value of an attribute, and the y-axis depicts the scaled density value. The two plots enable the comparison of different classifiers predicting different defective components for each individual attribute.

The motivation for using density plots, supplemented by the PCA, is to improve the understanding why some classifiers predict defects other miss. Defects are in many ways different and vary greatly in their characteristics. Some defects are design problems in code, some are because data is being handled incorrectly, some are logic problems. Static code metrics capture various characteristics of the code, which can be used in an attempt to

distinguish those defects. My proposed visual approach is suitable for analysing contributions each attribute has in determining classifiers' decisions. The plots can quickly demonstrate regions of interest, such as which attributes are found to be of more interest for a classifier.

7.3 Results and Discussion

Figure 7.1 shows the first 5 principal components of the PCA analysis for all 14 datasets. Each plot in the figure shows two principal components. The top left plot shows the first principal component (PC1) on the *x*-axis, and the second principal component (PC2) on the *y*-axis. The first two principal components contain most of the variance that can explain the dependent variable (defective/non-defective), whilst the other components contain reduced explanation of the variance. In addition to the defects individually predicted by each classifier, the plots contain false negative predictions. The FNs show areas where none of the classifiers were able to correctly predict defects.

Figure 7.1 demonstrates that some classifiers predict unique defects in different areas of the attribute space. For example, PART and Naïve Bayes tend to correctly predict more instances in the positive direction of *PC2*, whilst the SVMs in the negative direction of *PC2*. Yet, the PCA plots show limited information. As the PCA transforms all attributes into principal components, it becomes unmanageable to visualise differences in individual attribute values between different data points. Therefore, I use density functions to present individual predictions of each classifier broken down by attributes.

To handle outliers my visual approach allows the use of multiple density functions: *nrd*, *ucv*, *bcv*, *SJ-ste*, *SJ-dpi*, and *numeric*. These are the functions for calculating bandwidths [Plummer and Best 2016]. To focus on specific regions in the plots it is possible to modify minimum and maximum values for both axes, which allows zooming in the regions of most interest.

Figure 7.2 is an example of a density plot, where the *wmc* attribute is used for visualising the density functions. I choose the *lucene* dataset for visualisation purposes as it shows a clear separation of several defects predicted by SVMs only. *Lucene* also contains true positive predictions from all classifiers. For comparison, Figure 7.3 shows the principal components for *lucene*. Even though the PCA plot shows the tendency of the SVMs to predict defects at extremes, the density plot gives the exact values of attributes.

The top plot in Figure 7.2 depicts the density functions for each individual classifier correctly predicting defects, whilst the bottom plot contains the density functions of defective and non-defective classes. The top plot clearly indicates that the SVMs classifier is the only one to correctly predict defects where $wmc > 40$. The bottom plot suggests these predictions

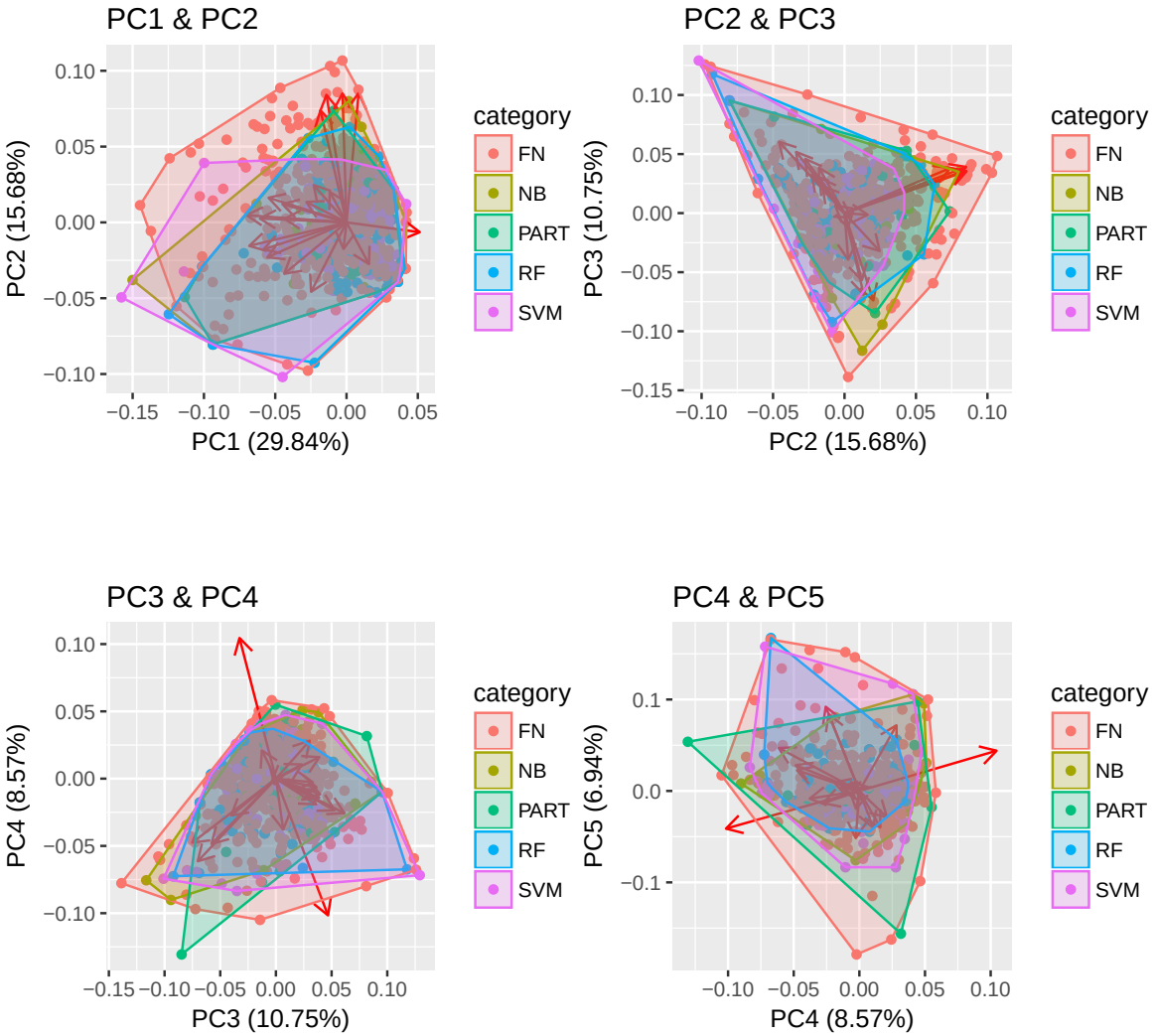


Fig. 7.1 The first five principal components showing the defects predicted by different classifiers for all dataset

are correct, as in that region the data points are indeed defective. From Figure 7.3 it is not obvious that the higher values of *wmc* cause those data points to be defective.

Figure 7.4 shows PCA plots for four different datasets. Each dataset in the plot was selected based on the number of representatives in each category. For most of the datasets there were either not enough data points, or no representatives from different classifier families. The PCA plots demonstrate that in all four datasets Naïve Bayes captures the widest areas of defective data points, compared to the other classifiers. This is likely due to a higher number of false positives across the datasets, relative to the other classifiers, as indicated in

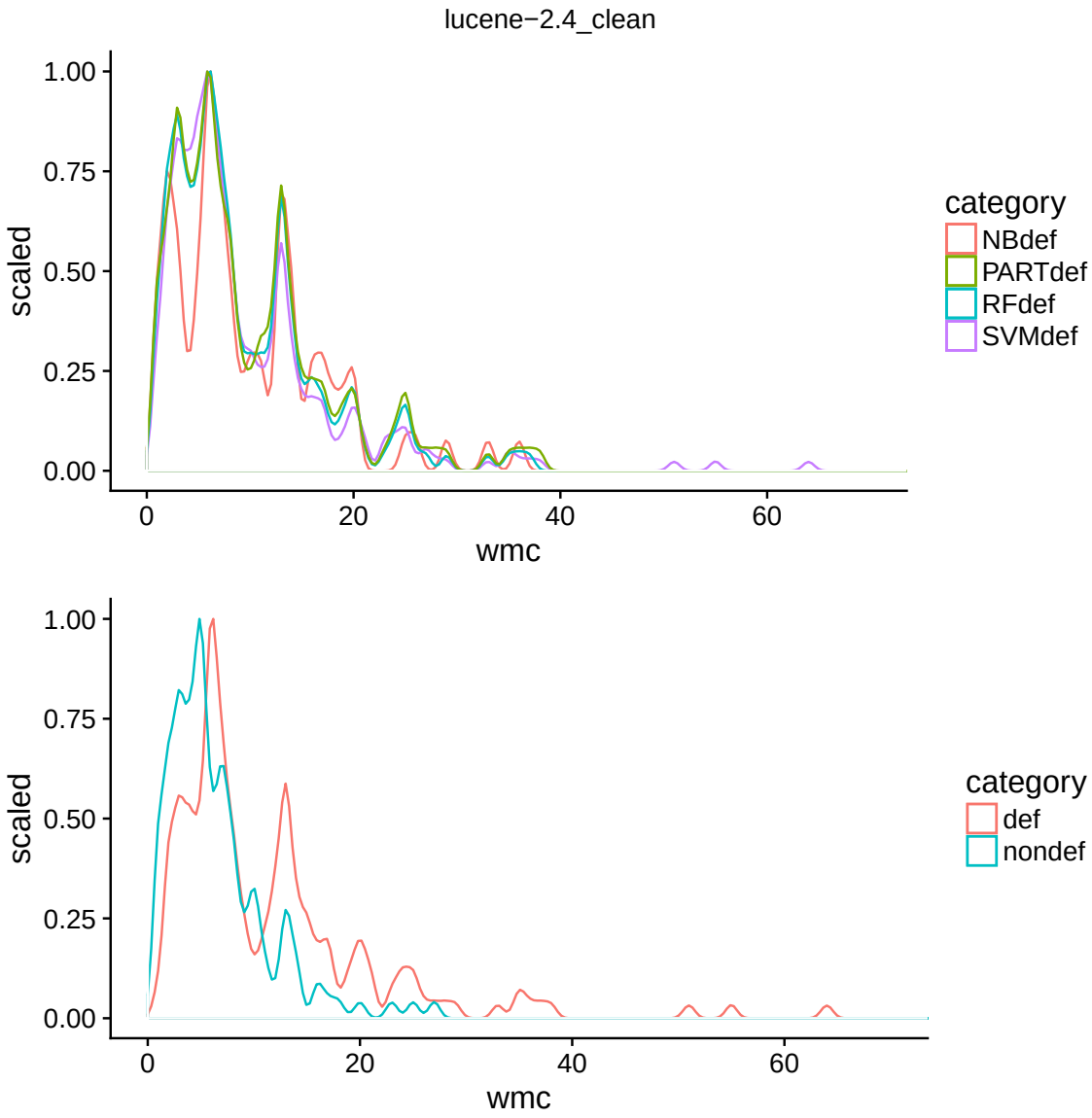


Fig. 7.2 Density functions broken down by the WMC metric showing individual prediction of the four classifiers on lucene

Figure 7.5. For *ant-1.7* and *camel-1.6* Naïve Bayes has the highest number of false positives, and wide areas of true positives. A similar effect is true for Random Forest in *xalan-2.6*, where the widest area of true positives maps to the highest number of false positives. SVMs do not appear in most of the PCA plots at Figure 7.4 because of a low number of true positive predictions as shown in Figure 7.5.

Although Naïve Bayes can stretch its predictions over a wide range of variability in data, it poses a risk of overfitting. PART and Random Forest make predictions in narrower areas of variability, whilst predicting less defects. For *ant*, *camel*, and *synapse* PART and

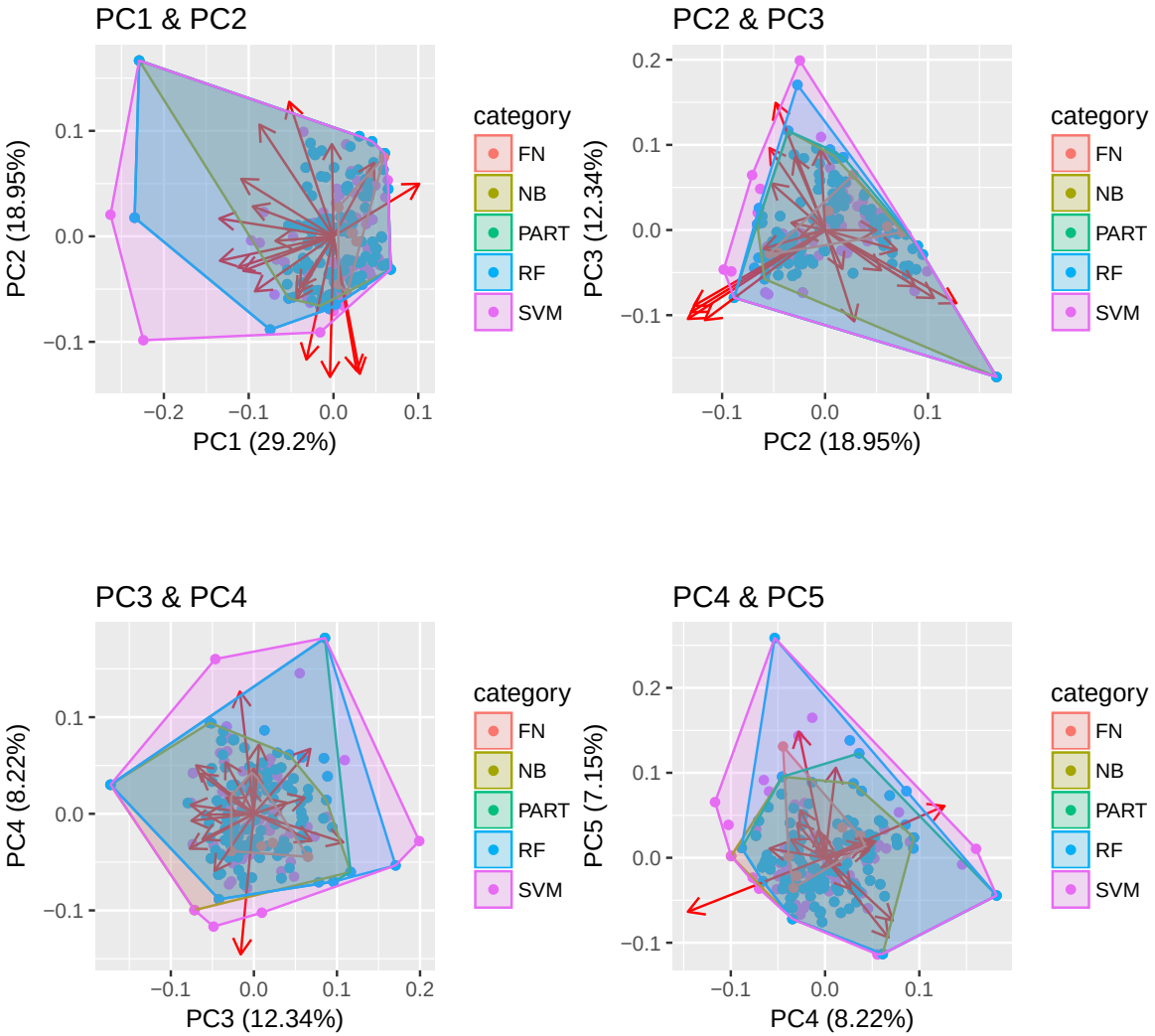


Fig. 7.3 The first five principal components showing the defects predicted by different classifiers for the lucene dataset

Random Forest make predictions in a narrow space of variability compared to the Naïve Bayes classifier.

The SVMs predict most unique defects across the 14 systems I analyse. However, SVMs are often discarded as poor predictors in defect prediction as they achieve low performance values. Table 5.3 shows that even when SVMs are tuned, they generally achieve worse prediction performances compared to other classifiers. Finding most unique defects in systems by simultaneously achieving low performance values on the whole appears contradictory. Therefore, I analyse whether and when SVMs can be useful for software defect prediction.

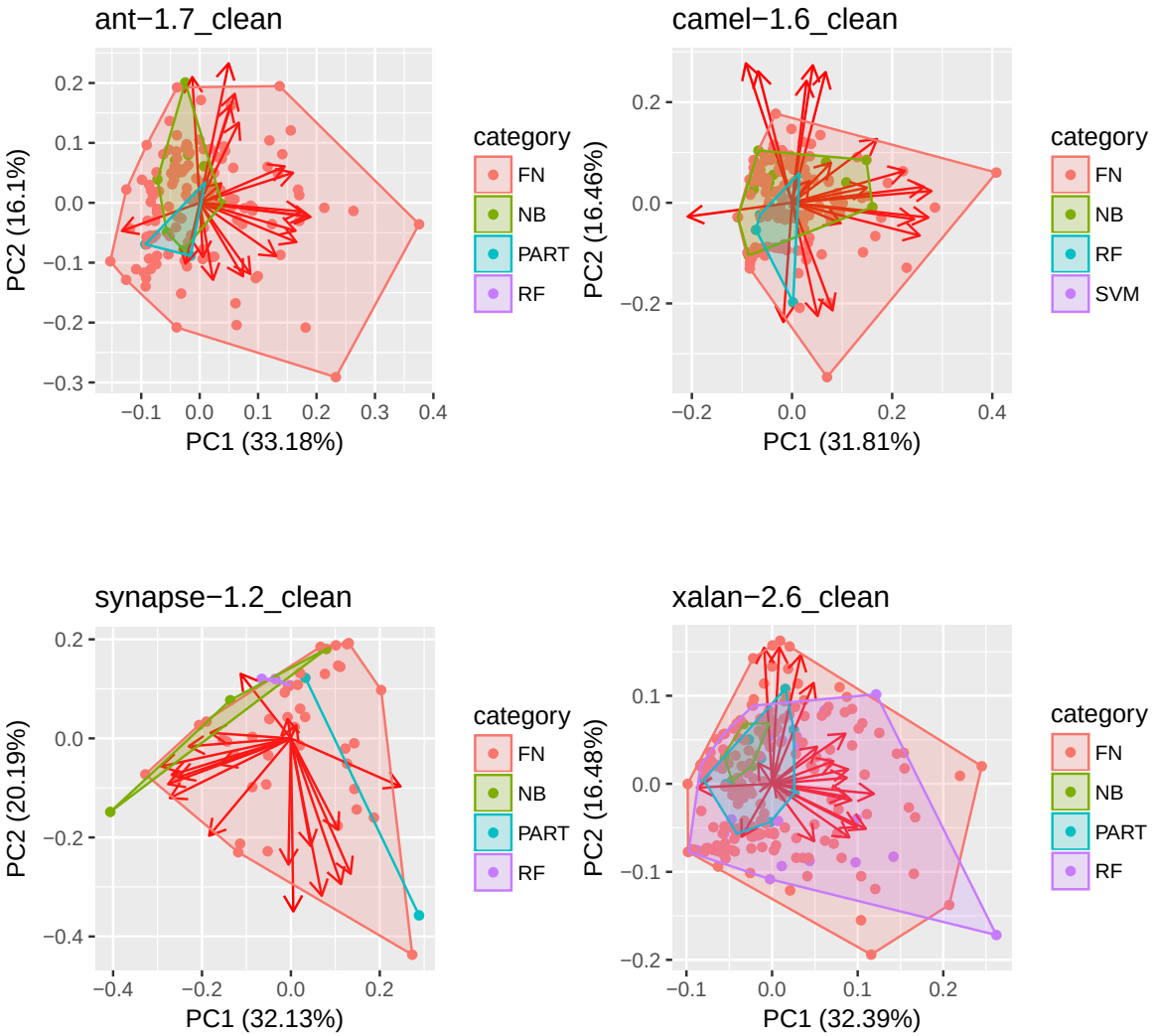


Fig. 7.4 The first two principal components for defects predicted only by a specific classifier indicated in 'category'

SVMs find unique defects in only 5 out of the 14 systems I analyse, where the top 3 systems are *lucene*, *poi* and *xerces*. In all 3 systems there is a higher representation of defective relative to non-defective instances. *Lucene*, *poi* and *xerces* have a better balance of the data than the other systems. Contrary, the SVMs perform poorly for the systems where imbalance is high. For example, the SVMs achieve particularly low prediction performances compared to the other classifiers for *camel*, *ivy*, *jedit*, and *tomcat*. All of these datasets have high imbalance. Figures 7.4 and 7.5 also demonstrate that the SVMs find a low number of true positives across these four systems. Apart from *xalan*, these systems are characterised by

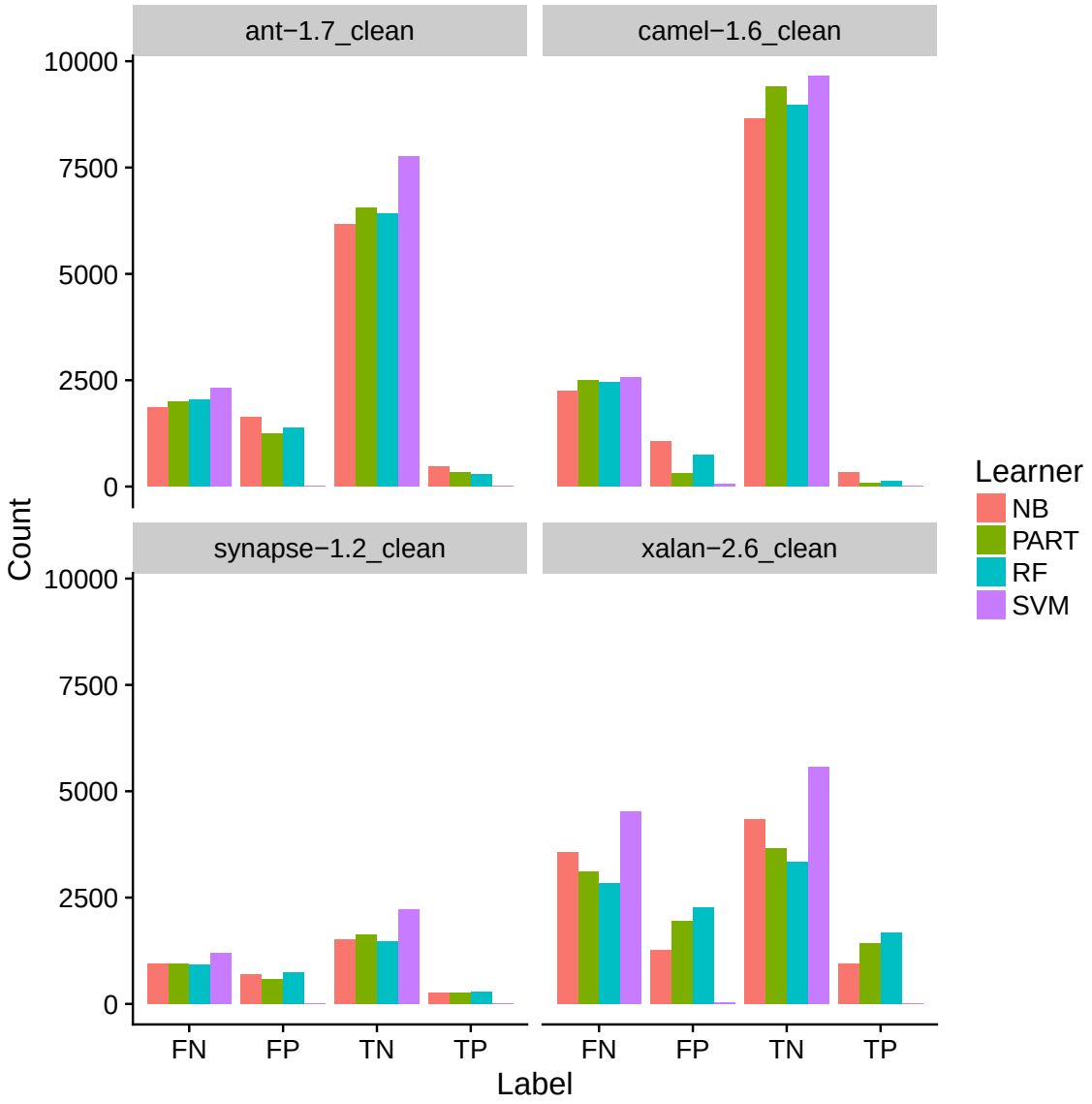


Fig. 7.5 Confusion matrix counts for different classifiers

high imbalance. SVMs struggle to find a hyper-plane which efficiently separates the classes when there are not enough data points representing one class to learn from. However, when data tends to be in balance, SVMs can achieve superior prediction performances in terms of identifying unique defects.

SVMs are particularly successful where there is enough data to train from. For the three systems, *lucene*, *poi* and *xerces*, the SVMs find more unique defects than any other classifier I use in my analysis. Figure 7.2 demonstrates the capability of the SVMs to correctly predict defects no other classifier predicts. The density plots in Figure 7.1 also show that the SVMs mostly make similar predictions to the other classifiers in the regions of the *wmc* attributes

where $wmc < 40$. By analysing all metrics, using the *def* and *nondef* categories for all classifiers, I established that the SVMs consistently make correct unique predictions in specific regions.

There are specific characteristics that distinguish SVMs from other classifiers, making them valuable for defect prediction. As I demonstrated, although the SVMs on average achieved worse performances, they flourish when the data tends to be balanced. Another beneficial characteristic of the SVMs is their ability to introduce diversity in results. Diversity is particularly useful for ensembles of learning machines, where diverse predictions help in making the final decision.

7.4 Conclusion

Understanding why some classifiers predict defects that others miss is necessary for further improvements of ensembles in software defect prediction. A few studies have tried to establish why classifiers predict the defects they predict. However, to build accurate models a better understanding of when classifiers work or do not work is needed. An ensemble design which accounts for characteristics of classifiers that correctly predict defects is likely to further improve prediction performances. My analysis has shown a potential towards improving the ensembles design for software defect prediction.

In this chapter I used two visual techniques, PCA and density plots, to find potential characteristics of classifiers which make them suitable for ensembles. I found that despite the low average prediction performances of SVMs, they predict unique defects which typically reside at extremes of specific attributes. Those defects could potentially be dangerous as they indicate a design not common to the majority of other modules. For this reason, SVMs should be considered when building ensembles for software defect prediction. My results also suggest that SVMs find a small proportion of defects in highly imbalanced datasets, but make a small number of wrong predictions. Therefore, an ensemble design which increases the signal of SVMs outputs could possibly lead to more correct predictions of defects which are located at extremes.

The analysis also demonstrates that Naïve Bayes finds a wide range of unique defects for the price of higher misclassification of non-defective instances (false positives). This could explain why Naïve Bayes classifiers are typically considered good classifiers for software defect prediction. However, misclassified instances cost industries extra effort in time and resources. A solution to this problem is combining Naïve Bayes with conservative classifiers to potentially reduce the number of false positive predictions.

7.5 Threats to Validity

Different tools for prediction modelling are used in this experiment compared to the experiment in Chapter 5. In this experiment I use WEKA, whilst in the previous chapter the R implementation was used. I noticed that this difference can affect the results, as the confusion matrices between WEKA and R do not match. However, the differences in the results are beyond the methodology of my experiment, as they rely on the third-party tools. To mitigate the problem due to the tools used, I only compared high level results with the findings from Chapter 5.

Chapter 8

Conclusions and Future Work

8.1 Reflection on the Research Questions

In this section I reflect on the research questions set out in the Introduction of this dissertation. I then briefly explain the value of data quality in my work.

RQ1: Do models created by different classifiers find different defective components?

I used classifiers from four different families and found that each identifies a unique subset of defects that the others miss. The findings are validated in three separate studies reported in **Paper 2**, **Paper 3**, and **Section 5.4**. In particular, these studies confirm that no single classifier can comprehensively detect all defects in the system. Due to the variations in decision boundaries those classifiers make and the differences in defects, some classifiers are better at predicting particular defects than others. Identifying the defects that different classifiers detect is important as it is well known [Fenton and Neil 1999a] that some defects matter more than others. Identifying defects with critical effects on a system is more important than identifying trivial defects. More research will be needed to establish which type of defects get predicted by different classifiers. In the context of my thesis, the importance of this finding is to explore more efficient ways to improve prediction models, which I do in **RQ2**.

RQ2(a): Can stacking ensembles based on explicit diversity improve prediction performance compared to other software defect prediction models?

In **Chapter 6** and **Paper 4** I demonstrate that the stacking ensemble can achieve significantly superior prediction performances compared to other ensemble or individual models. The ability of the stacking ensemble to correctly classify a higher number of defects compared to

other ensemble and single classifier approaches consistently persist. The results also suggest a consistent relative increase in the number of false predictions. However, the increase of misclassifications is substantially lower compared to the increase in the correct predictions. Such prediction models are useful to industries where any defects posing risk should be identified and fixed. This research question opens a potential for further research to establish the roots of misclassifications and adjust the stacking ensemble models appropriately. Di Nucci et al. [2017] have already shown that by selecting the most appropriate meta-classifier in ensembles improve prediction performances.

RQ2(b) How many classifiers combined into stacking ensembles provide good software defect prediction models?

My analysis of the ensemble's size reported in **Chapter 6** and **Paper 4** have shown that the stacking ensemble of size 3 is sufficient for achieving the highest prediction performance. Adding more classifiers to the ensemble will not significantly improve the performance. An intuitive explanation of this finding is that adding poorer performing and less diverse classifiers to the ensemble cannot improve the overall performance. I analysed the prediction performances of ensemble sizes from 2 to 15.

RQ2(c) How much diversity and which base classifiers are usually combined in stacking ensemble models?

The experiment showed that stacking ensembles combining classifiers from different families, in contrast to optimised classifiers from the same family, are more favoured during the stacking creation. In five out of eight cases, where equal preference was given to precision and diversity (i.e. the WAD technique), the stacking ensembles were constituted of classifiers all belonging to a different family. To reiterate the findings reported in **Paper 4**, the Naïve Bayes classifier has constantly been chosen by the stacking ensemble across all datasets. This suggests that Naïve Bayes classifiers perform well across all data sets, and increase diversity in ensembles. Some variants of SMO have been repeatedly chosen by stacking ensembles, often with different parameter settings. The frequently used decision tree classifier J48 was not dominant for any of the data sets.

I ensured that all experiments ran as part of this work were of high quality before answering the research questions. Therefore, I surveyed the literature to find state-of-the-art data preprocessing rules for software defect prediction (see Chapter 4). Subsequently, I identified new integrity violations in commonly used software defect datasets. I combined newly identified and state-of-the-art rules to create a comprehensive set of data cleaning steps

(reported in Chapter 4). I explicitly considered data quality as an important preprocessing step of the defect prediction methodology as advocated by [Gray 2013] and [Shepperd et al. 2013].

The repeated study from Section 5.4 was necessary to account for data quality issues reported in Chapter 4. In **Paper 2** we used the NASA datasets cleaned by Shepperd et al. [2013], but not cleaned by Petrić et al. [2016]. In **Paper 3**, the NASA and commercial datasets also needed cleaning. The datasets in **Paper 4** were fully cleaned, so no subsequent study was needed. The explicit consideration of data quality in the methodology of this work improves the reliability of the conclusions made in this dissertation.

8.2 Contributions to Knowledge

I make the following contributions to knowledge in this dissertation:

- **Improving the quality of datasets for defect prediction.** In Chapter 4 I introduce novel integrity checks for cleaning software defect datasets. This includes **Paper 1** where we report on two integrity constraints in the highly popular NASA MDP datasets. In **Paper 4** we report three additional integrity constraints specific for the PROMISE datasets. Poor data quality threatens the validity of software defect prediction studies, often leading those studies to incorrect conclusions. Chapter 4 extensively reports on state-of-the-art cleansing methods to improve the reliability of results reported in software defect prediction studies.

- **Models created by classifiers from different families find distinct defective components.** Despite many classifiers achieving similar prediction performances, the findings in Chapter 5 finding suggest that no single classifier can comprehensively detect all defects in the system. Previous software defect prediction studies, including most of those using stacking, have overlooked the implications of distinguishing different defective components. All defects are treated the same. This has led defect researchers to using modelling techniques that may not be suitable for the software defect prediction task. In addition, as some defects matter more than others, it would be beneficial to know what classifiers would be more suitable for predicting critical defects. Future work will be needed to establish the different types of defects certain classifiers are able or not able to predict.

- **Classifiers can be combined to provide improved performances in defect prediction.** This finding is three-fold:

1. The way classifiers are combined into ensembles matters. Most previous work reporting on ensembles in software defect prediction use a form of majority-voting to combine predictions into the final outcome. I show that this approach is suboptimal, as the correct predictions of classifiers, but in minority, will be ignored. I suggest the stacking approach which considers predictions of all classifiers in ensembles.
2. Combining three classifiers in an ensemble is sufficient. Finding out the optimal number of classifiers to combine in ensembles is relevant to practitioners, as precise, but ensembles combining many classifiers, would be slow to build for large systems. However, as my results show, adding more than three classifiers will not significantly improve the performance of stacking ensembles. Therefore, efficient ensembles can be built by using only three classifiers.
3. Diversity in ensembles improves prediction performances. I find that the best performing stacking ensemble combines models from different classifier families. This finding strengthens the observation that a stacking ensemble can more effectively, compared to ensembles based on majority-voting, combine diverse classifiers which complement each other.

- **Visualisation techniques for potential improvements of ensembles for software defect prediction.** In Chapter 7 I introduce visualisation techniques which provide the possibility of analysing individual predictions made by different classifiers. By using these visual techniques, I found that some classifiers make more conservative predictions than others. These findings open new avenues for research in improving ensemble designs that account for the peculiarities of different classifiers.

8.3 Significance of Work

The significance of work in this dissertation is three-fold:

1. The comprehensive set of data cleansing rules improves the reliability of conclusions made in this dissertation. Future researchers can use the comprehensive set of data cleansing rules in their work.
2. I provide practical suggestions on how to improve software defect prediction models by combining multiple classifiers which complement each other. Three classifiers belonging to different classifier families and combined into the stacking ensemble can provide adequate prediction performances.

3. I provide visual techniques which are a novel way of examining individual predictions made by different classifiers. One possible use of these visual techniques for future researchers is to further improve ensemble modelling, which will account for the peculiarities of different classifier techniques.

4. My work has been cited and extended by Di Nucci et al. [2017] who further explored the stacking approach in terms of algorithm recommendation described in Section 3.3.3. Their results on 30 software defect datasets confirmed that a variation of stacking ensembles is effective for software defect prediction.

8.4 Future Work

One possible direction of research the work in this dissertation opens up is refining the ensemble approach to account for the characteristics found in software defect prediction. Some researchers have already investigated possible improvements as a response to the published work included in this dissertation. As reported in Chapter 3, Di Nucci et al. [2017] proposed the ASCI ensemble technique based on the stacking approach. Fully understanding why some defects are predicted by one classifier and not by others could further help to improve the selection of suitable classifiers which are based on the characteristics of data.

A wider spectrum of classifiers needs to be considered in the future. Some classifiers could have a greater potential than others in software defect prediction. As an example, the analysis in Chapter 7 shows frequent swings of the defectiveness label across attributes. This suggests that not all classifiers may be suitable for software defect prediction. For example, classifiers capable of constructing complex decision boundaries (e.g. SVM) could be a better fit for the task compared to simple classifiers (e.g. Logistic Regression). More research is also needed to reduce the amount of variance in predictions. Blending techniques that can reduce variance, such as bagging, could potentially further improve prediction performances.

The work in this dissertation suggests practical changes on how past studies using ensembles in software defect prediction could be improved based on the findings reported here. For example, the validation and voting techniques [Laradji et al. 2015, Mısırlı et al. 2011b] and bagging techniques [Kim et al. 2011, Wahono and Suryana 2013] used in software defect prediction should be reconsidered. Even though these ensemble techniques often improve performances and reduce bias, this work finds that majority-voting techniques miss defects predicted by individual classifiers. On the other hand, the studies by Panichella et al. [2014] and Di Nucci et al. [2017] are likely to more efficiently exploit ensembles in software defect prediction as their models do not rely on majority-voting approaches. The work in this dissertation further suggests that researchers should explicitly consider data quality,

as erroneous data affects the conclusions made in software defect prediction [Ghotra et al. 2015].

References

Arisholm, E., Briand, L. C., and Johannessen, E. B. (2010). A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. *Journal of Systems and Software*, 83(1):2–17.

Arshad, F. A., Krause, R. J., and Bagchi, S. (2013). Characterizing configuration problems in java ee application servers: An empirical study with glassfish and jboss. In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, pages 198–207.

Bansiya, J. and Davis, C. G. (2002). A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on software engineering*, 28(1):4–17.

Benediktsson, J. A. and Swain, P. H. (1992). Consensus theoretic classification methods. *IEEE Transactions on Systems, Man, and Cybernetics*, 22(4):688–704.

Bennin, K. E., Keung, J., Monden, A., Phannachitta, P., and Mensah, S. (2017). The significant effects of data sampling approaches on software defect prioritization and classification. In *Proceedings of the 11th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '17, pages 364–373, Piscataway, NJ, USA. IEEE Press.

Bezerra, M., Oliveira, A., and Meira, S. (2007). A constructive rbf neural network for estimating the probability of defects in software modules. In *Neural Networks, 2007. IJCNN 2007. International Joint Conference on*, pages 2869 –2874.

Bibi, S., Tsoumakas, G., Stamelos, I., and Vlahavas, I. (2008). Regression via classification applied on software defect estimation. *Expert Systems with Applications*, 34(3):2091 – 2101.

Bibi, S., Tsoumakas, G., Stamelos, I., and Vlahvas, I. (2006). Software defect prediction using regression via classification. In *Computer Systems and Applications. IEEE International Conference on.*

Boetticher, G. D. (2006). Improving credibility of machine learner models in software engineering. *Advanced Machine Learner Applications in Software Engineering (Series on Software Engineering and Knowledge Engineering)*, pages 52–72.

Boser, B. E., Guyon, I. M., and Vapnik, V. N. (1992). A training algorithm for optimal margin classifiers. In *Proceedings of the fifth annual workshop on Computational learning theory*, COLT '92, pages 144–152, New York, NY, USA. ACM.

Boucher, A. and Badri, M. (2018). Software metrics thresholds calculation techniques to predict fault-proneness: An empirical comparison. *Information and Software Technology*, 96:38 – 67.

Bowes, D., Hall, T., Harman, M., Jia, Y., Sarro, F., and Wu, F. (2016). Mutation-aware fault prediction. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 330–341, New York, NY, USA. ACM.

Bowes, D., Hall, T., and Petrić, J. (2015). Different classifiers find different defects although with different level of consistency. In *Proceedings of the 11th International Conference on Predictive Models and Data Analytics in Software Engineering*. ACM.

Bowes, D., Hall, T., and Petrić, J. (2017). Software defect prediction: do different classifiers find the same defects? *Software Quality Journal*.

Bowes, D. H. (2013). *Factors Affecting the Performance of Trainable Models for Software Defect Prediction*. PhD thesis, Computer Science.

Breiman, L. (1996). Bagging predictors. *Machine Learning*, 24(2):123–140.

Briand, L., Melo, W., and Wust, J. (2002). Assessing the applicability of fault-proneness models across object-oriented software projects. *Software Engineering, IEEE Transactions on*, 28(7):706 – 720.

Cahill, J., Hogan, J. M., and Thomas, R. (2013). Predicting fault-prone software modules with rank sum classification. In *2013 22nd Australian Software Engineering Conference*, pages 211–219.

Cano, G., Garcia-Rodriguez, J., Garcia-Garcia, A., Perez-Sanchez, H., Benediktsson, J. A., Thapa, A., and Barr, A. (2017). Automatic selection of molecular descriptors using random forest: Application to drug discovery. *Expert Systems with Applications*, 72:151 – 159.

Catal, C. and Diri, B. (2009). A systematic review of software fault prediction studies. *Expert Systems with Applications*, 36(4):7346–7354.

Ceriani, L. and Verme, P. (2012). The origins of the gini index: extracts from variabilità e mutabilità (1912) by corrado gini. *The Journal of Economic Inequality*, 10(3):421–443.

Chan, J. C.-W. and Paelinckx, D. (2008). Evaluation of random forest and adaboost tree-based ensemble classification and spectral band selection for ecotope mapping using airborne hyperspectral imagery. *Remote Sensing of Environment*, 112(6):2999 – 3011.

Chawla, N., Bowyer, K., Hall, L., and Kegelmeyer, W. (2002). Smote: synthetic minority over-sampling technique. *Journal of Artificial Intelligence Research*, 16(1):321–357.

Chawla, N. V., Japkowicz, N., and Kotcz, A. (2004). Editorial: special issue on learning from imbalanced data sets. *SIGKDD Explorations*, 6(1):1–6.

Chen, L., Fang, B., Shang, Z., and Tang, Y. (2018). Tackling class overlap and imbalance problems in software defect prediction. *Software Quality Journal*, 26(1):97–125.

Chidamber, S. and Kemerer, C. (1994). A metrics suite for object oriented design. *Software Engineering, IEEE Transactions on*, 20(6):476–493.

Cortes, C. and Vapnik, V. (1995). Support-vector networks. *Mach. Learn.*, 20(3):273–297.

D'Ambros, M., Lanza, M., and Robbes, R. (2009). On the relationship between change coupling and software defects. In *Reverse Engineering, 2009. WCRE '09. 16th Working Conference on*, pages 135 –144.

D'Ambros, M., Lanza, M., and Robbes, R. (2010). An extensive comparison of bug prediction approaches. In *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, pages 31–41.

D'Ambros, M., Lanza, M., and Robbes, R. (2012). Evaluating defect prediction approaches: a benchmark and an extensive comparison. *Empirical Software Engineering*, 17(4):531–577.

das Dôres, S. N., Alves, L., Ruiz, D. D., and Barros, R. C. (2016). A meta-learning framework for algorithm recommendation in software fault prediction. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, SAC '16, pages 1486–1491, New York, NY, USA. ACM.

Dasarathy, B. V. and Sheela, B. V. (1979). A composite classifier system design: Concepts and methodology. *Proceedings of the IEEE*, 67(5):708–713.

Davies, S., Roper, M., and Wood, M. (2014). Comparing text-based and dependence-based approaches for determining the origins of bugs. *Journal of Software: Evolution and Process*, 26(1):107–139.

de Menezes, F. S., Liska, G. R., Cirillo, M. A., and Vivanco, M. J. (2017). Data classification with binary response through the boosting algorithm and logistic regression. *Expert Systems with Applications*, 69:62 – 73.

Di Nucci, D., Palomba, F., Oliveto, R., and De Lucia, A. (2017). Dynamic selection of classifiers in bug prediction: An adaptive method. *IEEE Transactions on Emerging Topics in Computational Intelligence*, 1(3):202–212.

Di Nucci, D., Palomba, F., Rosa, G. D., Bavota, G., Oliveto, R., and De Lucia, A. (2018). A developer centered bug prediction model. *IEEE Transactions on Software Engineering*, 44(1):5–24.

Džeroski, S. and Ženko, B. (2004). Is combining classifiers with stacking better than selecting the best one? *Machine Learning*, 54(3):255–273.

Elish, M. O. (2014). A comparative study of fault density prediction in aspect-oriented systems using mlp, rbf, knn, rt, denfis and svr models. *Artif. Intell. Rev.*, 42(4):695–703.

Fanelli, G., Dantone, M., Gall, J., Fossati, A., and Van Gool, L. (2013). Random forests for real time 3d face analysis. *International Journal of Computer Vision*, 101(3):437–458.

Fenton, N. and Bieman, J. (2014). *Software metrics: a rigorous and practical approach*. CRC Press.

Fenton, N. and Neil, M. (1999a). A critique of software defect prediction models. *Software Engineering, IEEE Transactions on*, 25(5):675 –689.

Fenton, N. E. and Neil, M. (1999b). A critique of software defect prediction models. *IEEE Trans. Softw. Eng.*, 25(5):675–689.

Fischer, M., Pinzger, M., and Gall, H. (2003). Populating a release history database from version control and bug tracking systems. In *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.*, pages 23–32.

Fisher, D., Xu, L., and Zard, N. (1992). Ordering effects in clustering. In Sleeman, D. and Edwards, P., editors, *Machine Learning Proceedings 1992*, pages 163 – 168. Morgan Kaufmann, San Francisco (CA).

Freund, Y., Schapire, R. E., et al. (1996). Experiments with a new boosting algorithm. In *Icml*, volume 96, pages 148–156. Bari, Italy.

Fu, W., Menzies, T., and Shen, X. (2016). Tuning for software analytics: Is it really necessary? *Information and Software Technology*, 76:135 – 146.

Gao, K., Khoshgoftaar, T. M., and Napolitano, A. (2015). Combining feature subset selection and data sampling for coping with highly imbalanced software data. In *Proc. of 27th International Conf. on Software Engineering and Knowledge Engineering, Pittsburgh*.

Ghotra, B., McIntosh, S., and Hassan, A. E. (2015). Revisiting the impact of classification techniques on the performance of defect prediction models. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 789–800, Piscataway, NJ, USA. IEEE Press.

Ghotra, B., McIntosh, S., and Hassan, A. E. (2017). A large-scale study of the impact of feature selection techniques on defect classification models. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 146–157.

Gray, D. (2013). *Software Defect Prediction Using Static Code Metrics : Formulating a Methodology*. PhD thesis, Computer Science, University of Hertfordshire.

Gray, D., Bowes, D., Davey, N., Sun, Y., and Christianson, B. (2009). Using the support vector machine as a classification method for software defect prediction with static code metrics. In Palmer-Brown, D., Draganova, C., Pimenidis, E., and Mouratidis, H., editors, *Engineering Applications of Neural Networks*, pages 223–234, Berlin, Heidelberg. Springer Berlin Heidelberg.

Gray, D., Bowes, D., Davey, N., Sun, Y., and Christianson, B. (2011). The misuse of the NASA metrics data program data sets for automated software defect prediction. In *EASE 2011*, Durham, UK. IET.

Gray, D., Bowes, D., Davey, N., Sun, Y., and Christianson, B. (2012). Reflections on the nasa mdp data sets. *Software, IET*, 6(6):549 –558.

Gray, D., Bowes, D., Davey, N., Yi, S., and Christianson, B. (2010). Software defect prediction using static code metrics underestimates defect-proneness. In *Neural Networks (IJCNN), The 2010 International Joint Conference on*, pages 1–7.

Hall, M. A. (1999). *Correlation-based Feature Selection for Machine Learning*. PhD thesis, University of Waikato.

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

Halstead, M. H. (1977). *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., New York, NY, USA.

He, H. and Garcia, E. A. (2009). Learning from imbalanced data. *IEEE Transactions on Knowledge and Data Engineering*, 21(9):1263–1284.

He, P., Li, B., Liu, X., Chen, J., and Ma, Y. (2015). An empirical study on software defect prediction with a simplified metric set. *Information and Software Technology*, 59:170 – 190.

Henderson-Sellers, B. (1995). *Object-oriented metrics: measures of complexity*. Prentice-Hall, Inc.

Hernandez-Gonzalez, J., Rodriguez, D., Inza, I., Harrison, R., and Lozano, J. A. (2018). Learning to classify software defects from crowds: A novel approach. *Applied Soft Computing*, 62:579 – 591.

Howley, T., Madden, M. G., O'Connell, M.-L., and Ryder, A. G. (2006). The effect of principal component analysis on machine learning accuracy with high-dimensional spectral data. *Knowledge-Based Systems*, 19(5):363 – 370. AI 2005 SI.

Hsu, C., Chang, C., and Lin, C. (2003). A practical guide to support vector classification. *Department of Computer Science and Information Engineering, National Taiwai University*.

Huang, J., Lu, J., and Ling, C. X. (2003). Comparing naive bayes, decision trees, and svm with auc and accuracy. In *Third IEEE International Conference on Data Mining*, pages 553–556.

Huang, Y. S. and Suen, C. Y. (1995). A method of combining multiple experts for the recognition of unconstrained handwritten numerals. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 17(1):90–94.

Huanjing, W., Khoshgoftaar, T. M., and Napolitano, A. (2010). A comparative study of ensemble feature selection techniques for software defect prediction. In *Machine Learning and Applications (ICMLA), 2010 Ninth International Conference on*, pages 135–140.

IEEE Std, I. (2009). Ieee standard classification for software anomalies. *IEEE Std*, 1044-2009.

Jacobson, S. H. and Yücesan, E. (2004). Analyzing the performance of generalized hill climbing algorithms. *Journal of Heuristics*, 10(4):387–405.

Japkowicz, N. and Stephen, S. (2002). The class imbalance problem: A systematic study. *Intell. Data Anal*, 6(5):429–449.

Jiang, Y., Cukic, B., and Ma, Y. (2008a). Techniques for evaluating fault prediction models. *Empirical Software Engineering*, 13(5):561–595.

Jiang, Y., Cukic, B., and Menzies, T. (2008b). Can data transformation help in the detection of fault-prone modules? In *Proceedings of the 2008 Workshop on Defects in Large Software Systems*, DEFECTS '08, pages 16–20, New York, NY, USA. ACM.

Jing, X.-Y., Ying, S., Zhang, Z.-W., Wu, S.-S., and Liu, J. (2014). Dictionary learning based software defect prediction. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 414–423, New York, NY, USA. ACM.

Jureczko, M. and Madeyski, L. (2010). Towards identifying software project clusters with regard to defect prediction. In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, PROMISE '10, pages 9:1–9:10, New York, NY, USA. ACM.

Jureczko, M. and Spinellis, D. D. (2010). Using object-oriented design metrics to predict software defects. In *In Models and Methods of System Dependability. Oficyna Wydawnicza Politechniki Wrocławskiej*, pages 69–81.

Kalousis, A. (2002). *Algorithm selection via meta-learning*. PhD thesis, University of Geneva, Genebra.

Kamei, Y. and Shihab, E. (2016). Defect prediction: Accomplishments and future challenges. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 5, pages 33–45.

Kaminsky, K. and Boetticher, G. (2004). Building a genetically engineerable evolvable program (geep) using breadth-based explicit knowledge for predicting software defects. In *Fuzzy Information, 2004. Processing NAFIPS '04. IEEE Annual Meeting of the*, volume 1, pages 10 – 15 Vol.1.

Kanmani, S., Uthariaraj, V. R., Sankaranarayanan, V., and Thambidurai, P. (2004). Object oriented software quality prediction using general regression neural networks. *SIGSOFT Softw. Eng. Notes*, 29:1–6.

Karg, L. M., Grottke, M., and Beckhaus, A. (2011). A systematic literature review of software quality cost research. *Journal of Systems and Software*, 84(3):415 – 427.

Khoshgoftaar, T. M. and Allen, E. B. (2003). Ordering fault-prone software modules. *Software Quality Journal*, 11(1):19–37.

Khoshgoftaar, T. M., Golawala, M., and Hulse, J. V. (2007). An empirical study of learning from imbalanced data using random forest. In *19th IEEE International Conference on Tools with Artificial Intelligence(ICTAI 2007)*, volume 2, pages 310–317.

Khoshgoftaar, T. M. and Hulse, J. V. (2009). Empirical case studies in attribute noise detection. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 39(4):379–388.

Khoshgoftaar, T. M. and Seliya, N. (2004). The necessity of assuring quality in software measurement data. In *Proceedings of the Software Metrics, 10th International Symposium*, METRICS '04, pages 119–130, Washington, DC, USA. IEEE Computer Society.

Khoshgoftaar, T. M., Yuan, X., Allen, E. B., Jones, W. D., and Hudepohl, J. P. (2002). Uncertain classification of fault-prone software modules. *Empirical Software Engineering*, 7(4):297–318.

Kim, M.-J., Kang, D.-K., and Kim, H. B. (2015). Geometric mean based boosting algorithm with over-sampling to resolve data imbalance problem for bankruptcy prediction. *Expert Systems with Applications*, 42(3):1074 – 1082.

Kim, S., Zhang, H., Wu, R., and Gong, L. (2011). Dealing with noise in defect prediction. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 481–490, New York, NY, USA. ACM.

Kim, S., Zimmermann, T., Whitehead Jr., E. J., and Zeller, A. (2007). Predicting faults from cached history. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 489–498, Washington, DC, USA. IEEE Computer Society.

Kirbas, S., Caglayan, B., Hall, T., Counsell, S., Bowes, D., Sen, A., and Bener, A. (2017). The relationship between evolutionary coupling and defects in large industrial software. *Journal of Software: Evolution and Process*, 29(4):e1842–n/a. e1842 smr.1842.

Koru, A. G. and Liu, H. (2005). An investigation of the effect of module size on defect prediction using static measures. *SIGSOFT Softw. Eng. Notes*, 30(4):1–5.

Krstajic, D., Buturovic, L. J., Leahy, D. E., and Thomas, S. (2014). Cross-validation pitfalls when selecting and assessing regression and classification models. *Journal of cheminformatics*, 6(1):10.

Kuncheva, L. and Whitaker, C. (2003). Measures of diversity in classifier ensembles and their relationship with the ensemble accuracy. *Machine Learning*, 51:181–207.

Lanubile, F., Ebert, C., Prikladnicki, R., and Vizcaíno, A. (2010). Collaboration tools for global software engineering. *IEEE Software*, 27(2):52–55.

Laradji, I. H., Alshayeb, M., and Ghouti, L. (2015). Software defect prediction using ensemble learning on selected features. *Information and Software Technology*, 58:388 – 402.

Lessmann, S., Baesens, B., Mues, C., and Pietsch, S. (2008a). Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *Software Engineering, IEEE Transactions on*, 34(4):485–496.

Lessmann, S., Baesens, B., Mues, C., and Pietsch, S. (2008b). Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *Software Engineering, IEEE Transactions on*, 34(4):485 –496.

Li, Z., Jing, X. Y., Zhu, X., and Zhang, H. (2017). Heterogeneous defect prediction through multiple kernel learning and ensemble learning. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 91–102.

Mahmood, Z., Bowes, D., Hall, T., Lane, P. C., and Petrić, J. (2018). Reproducibility and replicability of software defect prediction studies. *Information and Software Technology*.

Mahmood, Z., Bowes, D., Lane, P. C. R., and Hall, T. (2015). What is the impact of imbalance on software defect prediction performance? In *Proceedings of the 11th International Conference on Predictive Models and Data Analytics in Software Engineering*, PROMISE '15, pages 4:1–4:4, New York, NY, USA. ACM.

Maimon, O. and Rokach, L. (2010). *Data Mining and Knowledge Discovery Handbook Second Edition*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Malhotra, R. (2015). A systematic review of machine learning techniques for software fault prediction. *Applied Soft Computing*, 27:504 – 518.

Malhotra, R. and Raje, R. (2014). An empirical comparison of machine learning techniques for software defect prediction. In *Proceedings of the 8th International Conference on Bioinspired Information and Communications Technologies*, BICT '14, pages 320–327, ICST, Brussels, Belgium, Belgium. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

Martin, B. (1995). *Instance-Based learning: Nearest Neighbor With Generalization*. PhD thesis, University of Waikato.

Martin, R. (1994). Oo design quality metrics. *An analysis of dependencies*, 12:151–170.

Mauša, G., Perković, P., Grbac, T. G., and Štajduhar, I. (2014). Techniques for bug–code linking. In *Third Workshop on Software Quality Analysis, Monitoring, Improvement and Applications*.

McCabe, T. (1976). A complexity measure. *Software Engineering, IEEE Transactions on*, SE-2(4):308–320.

Mende, T. (2010). Replication of defect prediction studies: Problems, pitfalls and recommendations. In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, PROMISE '10, pages 5:1–5:10, New York, NY, USA. ACM.

Mende, T. and Koschke, R. (2009). Revisiting the evaluation of defect prediction models. In *Proceedings of the 5th International Conference on Predictor Models in Software Engineering*, PROMISE '09, pages 7:1–7:10, New York, NY, USA. ACM.

Mende, T., Koschke, R., and Leszak, M. (2009). Evaluating defect prediction models for a large evolving software system. In *Software Maintenance and Reengineering, 2009. CSMR '09. 13th European Conference on*, pages 247 –250.

Menzies, T., Dekhtyar, A., Distefano, J., and 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., and Frank, A. (2007b). Data mining static code attributes to learn defect predictors. *Software Engineering, IEEE Transactions on*, 33(1):2 –13.

Menzies, T., Krishna, R., and Pryor, D. (2015). The promise repository of empirical software engineering data.

Mısırlı, A., Bener, A., and Turhan, B. (2011a). An industrial case study of classifier ensembles for locating software defects. *Software Quality Journal*, 19(3):515–536.

Mısırlı, A. T., Bener, A. B., and Turhan, B. (2011b). An industrial case study of classifier ensembles for locating software defects. *Software Quality Journal*, 19(3):515–536.

Mitchell, T. M. et al. (1997). Machine learning. 1997. *Burr Ridge, IL: McGraw Hill*, 45(37):870–877.

Mizuno, O. and Kikuno, T. (2007). Training on errors experiment to detect fault-prone software modules by spam filter. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC-FSE '07, pages 405–414, New York, NY, USA. ACM.

Moser, R., Pedrycz, W., and Succi, G. (2008). A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 181–190, New York, NY, USA. ACM.

Okutan, A. and Yildiz, O. T. (2014). Software defect prediction using bayesian networks. *Empirical Softw. Engg.*, 19(1):154–181.

Ostrand, T. J., Weyuker, E. J., and Bell, R. (2005). Predicting the location and number of faults in large software systems. *Software Engineering, IEEE Transactions on*, 31(4):340 – 355.

Panichella, A., Oliveto, R., and De Lucia, A. (2014). Cross-project defect prediction models: L'union fait la force. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week - IEEE Conference on*, pages 164–173.

Parmezan, A. R. S., Lee, H. D., and Wu, F. C. (2017). Metalearning for choosing feature selection algorithms in data mining: Proposal of a new framework. *Expert Systems with Applications*, 75:1 – 24.

Pelayo, L. and Dick, S. (2012). Evaluating stratification alternatives to improve software defect prediction. *IEEE Transactions on Reliability*, 61(2):516–525.

Petrić, J., Bowes, D., Hall, T., Christianson, B., and Baddoo, N. (2016). The jinx on the nasa software defect data sets. In *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*, EASE '16, pages 13:1–13:5, New York, NY, USA. ACM.

Petrić, J. and Grbac, T. G. (2014). Software structure evolution and relation to system defectiveness. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, EASE '14, pages 34:1–34:10.

Platt, J. C. (1998). Sequential minimal optimization: A fast algorithm for training support vector machines. Technical report, ADVANCES IN KERNEL METHODS - SUPPORT VECTOR LEARNING.

Plummer, M. and Best, N. (2016). Package 'coda'.

Polikar, R. (2006). Ensemble based systems in decision making. *IEEE Circuits and Systems Magazine*, 6(3):21–45.

Porto, F., Minku, L., Mendes, E., and Simao, A. (2018). A systematic study of cross-project defect prediction with meta-learning. *arXiv preprint arXiv:1802.06025*.

Posnett, D., D'Souza, R., Devanbu, P., and Filkov, V. (2013). Dual ecological measures of focus in software development. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 452–461.

Prusa, J., Khoshgoftaar, T. M., and Dittman, D. J. (2015). Using ensemble learners to improve classifier performance on tweet sentiment data. In *2015 IEEE International Conference on Information Reuse and Integration*, pages 252–257.

Quinlan, J. (1986). Induction of decision trees. *Machine Learning*, 1(1):81–106.

Quinlan, J. (1993). *C4. 5: programs for machine learning*, volume 1. Morgan kaufmann.

Rahman, F. and Devanbu, P. (2013). How, and why, process metrics are better. In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 432–441. IEEE.

Rathore, S. S. and Kumar, S. (2017). Linear and non-linear heterogeneous ensemble methods to predict the number of faults in software systems. *Knowledge-Based Systems*, 119:232 – 256.

Rodriguez, D., Herraiz, I., Harrison, R., Dolado, J., and Riquelme, J. C. (2014). Preliminary comparison of techniques for dealing with imbalance in software defect prediction. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, EASE '14, pages 43:1–43:10, New York, NY, USA. ACM.

Rodríguez, J. J., Díez-Pastor, J. F., Maudes, J., and García-Osorio, C. (2012). Disturbing neighbors ensembles of trees for imbalanced data. In *2012 11th International Conference on Machine Learning and Applications*, volume 2, pages 83–88.

Rokach, L. (2009). Taxonomy for characterizing ensemble methods in classification tasks: A review and annotated bibliography. *Computational Statistics & Data Analysis*, 53(12):4046 – 4072.

Sarro, F., Di Martino, S., Ferrucci, F., and Gravino, C. (2012). A further analysis on the use of genetic algorithm to configure support vector machines for inter-release fault prediction. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 1215–1220, New York, NY, USA. ACM.

Schapire, R. E. (1990). The strength of weak learnability. *Machine Learning*, 5(2):197–227.

Segaran, T. (2007). *Programming collective intelligence: building smart web 2.0 applications*. " O'Reilly Media, Inc.".

Shaikhina, T., Lowe, D., Daga, S., Briggs, D., Higgins, R., and Khovanova, N. (2017). Decision tree and random forest models for outcome prediction in antibody incompatible kidney transplantation. *Biomedical Signal Processing and Control*.

Shepperd, M. (1988). A critique of cyclomatic complexity as a software metric. *Softw. Eng. J.*, 3(2):30–36.

Shepperd, M., Bowes, D., and Hall, T. (2014). Researcher bias: The use of machine learning in software defect prediction. *IEEE Transactions on Software Engineering*, 40(6):603–616.

Shepperd, M. and Schofield, C. (1997). Estimating software project effort using analogies. *IEEE Transactions on Software Engineering*, 23(11):736–743.

Shepperd, M., Song, Q., Sun, Z., and Mair, C. (2013). Data quality: Some comments on the nasa software defect datasets. *Software Engineering, IEEE Transactions on*, 39(9):1208–1215.

Shippey, T. J. (2015). *Exploiting Abstract Syntax Trees to Locate Software Defects*. PhD thesis, University of Hertfordshire.

Sill, J., Takács, G., Mackey, L., and Lin, D. (2009). Feature-weighted linear stacking. *arXiv preprint arXiv:0911.0460*.

Śliwerski, J., Zimmermann, T., and Zeller, A. (2005). When do changes induce fixes? In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 1–5. ACM.

Soares, C., Brazdil, P. B., and Kuba, P. (2004). A meta-learning method to select the kernel width in support vector regression. *Machine learning*, 54(3):195–209.

Song, Q., Jia, Z., Shepperd, M., Ying, S., and Liu, J. (2011). A general software defect-proneness prediction framework. *Software Engineering, IEEE Transactions on*, 37(3):356–370.

Stanić, B. and Afzal, W. (2017). Process metrics are not bad predictors of fault proneness. In *The 2017 IEEE International Workshop on Software Engineering and Knowledge Management SEKM'17, 25 Jul 2017, Prague, Sweden*, pages 493–499.

Sun, Z., Song, Q., and Zhu, X. (2012). Using coding-based ensemble learning to improve software defect prediction. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 42(6):1806–1817.

Surowiecki, J. (2004). The wisdom of crowds: Why the many are smarter than the few and how collective wisdom shapes business. *Economies, Societies and Nations*, 296.

Tang, M.-H., Kao, M.-H., and Chen, M.-H. (1999). An empirical study on object-oriented metrics. In *Proceedings Sixth International Software Metrics Symposium (Cat. No.PR00403)*, pages 242–249.

Tantithamthavorn, C., Hassan, A. E., and Matsumoto, K. (2018). The impact of class rebalancing techniques on the performance and interpretation of defect prediction models. *arXiv preprint arXiv:1801.10269*.

Tantithamthavorn, C., McIntosh, S., Hassan, A. E., and Matsumoto, K. (2016). Automated parameter optimization of classification techniques for defect prediction models. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 321–332.

Tassey, G. (2002). The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology, RTI Project*, 7007(011).

Therneau, T. M., Atkinson, E. J., et al. (1997). An introduction to recursive partitioning using the rpart routines.

Tong, H., Liu, B., and Wang, S. (2017). Software defect prediction using stacked denoising autoencoders and two-stage ensemble learning. *Information and Software Technology*.

Tosun, A. and Bener, A. (2009). Reducing false alarms in software defect prediction by decision threshold optimization. In *Empirical Software Engineering and Measurement, 2009. ESEM 2009. 3rd International Symposium on*, pages 477–480.

Tosun, A., Turhan, B., and Bener, A. (2008). Ensemble of software defect predictors: A case study. In *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '08, pages 318–320, New York, NY, USA. ACM.

Tukey, J. W. (1977). *Exploratory data analysis*, volume 2. Reading, Mass.

Čubranić, D. and Murphy, G. C. (2003). Hipikat: Recommending pertinent software development artifacts. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 408–418, Washington, DC, USA. IEEE Computer Society.

Virtual Machinery (2018). JHawk - the Java metrics tool - List of metrics. http://www.virtualmachinery.com/jhawkmetricslist.htm. [Online; accessed 06-September-2018].

Wahono, R. S. (2015). A systematic literature review of software defect prediction: Research trends, datasets, methods and frameworks. *Journal of Software Engineering*, 1(1):1–16.

Wahono, R. S. and Suryana, N. (2013). Combining particle swarm optimization based feature selection and bagging technique for software defect prediction. *International Journal of Software Engineering and Its Applications*, 7(5):153–166.

Wang, S., Liu, T., and Tan, L. (2016). Automatically learning semantic features for defect prediction. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 297–308, New York, NY, USA. ACM.

Wang, S. and Yao, X. (2013). Using class imbalance learning for software defect prediction. *IEEE Transactions on Reliability*, 62(2):434–443.

Wang, T., Li, W., Shi, H., and Liu, Z. (2011). Software defect prediction based on classifiers ensemble. *Journal of Information & Computational Science*, 8(16):4241–4254.

Weyuker, E., Ostrand, T., and Bell, R. (2008). Comparing negative binomial and recursive partitioning models for fault prediction. In *Proceedings of the 4th international workshop on Predictor models in software engineering*, pages 3–10. ACM.

Wickham, H. (2017). reshape2: Flexibly Reshape Data: A Reboot of the Reshape Package. https://cran.r-project.org/web/packages/reshape2/index.html. [Online; accessed 05-September-2018].

Witten, I. and 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*.

Witten, I. and Frank, E. (2005). *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann.

Wold, S., Esbensen, K., and Geladi, P. (1987). Principal component analysis. *Chemometrics and Intelligent Laboratory Systems*, 2(1):37 – 52. Proceedings of the Multivariate Statistical Workshop for Geologists and Geochemists.

Wolpert, D. H. (1992). Stacked generalization. *Neural Networks*, 5(2):241 – 259.

Woods, K., Kegelmeyer, W. P., and Bowyer, K. (1997). Combination of multiple classifiers using local accuracy estimates. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19(4):405–410.

Xia, X., Lo, D., Wang, X., and Zhou, B. (2014). Automatic defect categorization based on fault triggering conditions. In *2014 19th International Conference on Engineering of Complex Computer Systems*, pages 39–48.

Xia, X., Zhou, X., Lo, D., and Zhao, X. (2013). An empirical study of bugs in software build systems. In *2013 13th International Conference on Quality Software*, pages 200–203.

Xu, Z., Khoshgoftaar, T., and Allen, E. (2000). Prediction of software faults using fuzzy nonlinear regression modeling. In *High Assurance Systems Engineering, 2000, Fifth IEEE International Symposim on. HASE 2000*, pages 281 –290.

Xu, Z., Liu, J., Yang, Z., An, G., and Jia, X. (2016). The impact of feature selection on defect prediction performance: An empirical comparison. In *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, pages 309–320.

Yan, J., Qi, Y., and Rao, Q. (2018). Detecting malware with an ensemble method based on deep neural network. *Security and Communication Networks*, 2018.

Yan, M., Fang, Y., Lo, D., Xia, X., and Zhang, X. (2017). File-level defect prediction: Unsupervised vs. supervised models. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 344–353.

Yang, X., Tang, K., and Yao, X. (2015). A learning-to-rank approach to software defect prediction. *IEEE Transactions on Reliability*, 64(1):234–246.

Yang, Y. and Webb, G. I. (2009). Discretization for naive-bayes learning: managingÂdiscretization bias and variance. *Machine Learning*, 74(1):39–74.

Yen, S.-J. and Lee, Y.-S. (2009). Cluster-based under-sampling approaches for imbalanced data distributions. *Expert Systems with Applications*, 36(3, Part 1):5718 – 5727.

Yu, X., Liu, J., Yang, Z., Jia, X., Ling, Q., and Ye, S. (2017). Learning from imbalanced data for predicting the number of software defects. In *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*, pages 78–89.

Yusifoglu, V. G., Amannejad, Y., and Can, A. B. (2015). Software test-code engineering: A systematic mapping. *Information and Software Technology*, 58:123 – 147.

Zeng, X., Wong, D. F., and Chao, L. S. (2014). Constructing better classifier ensemble based on weighted accuracy and diversity measure. *The Scientific World Journal*, 2014.

Zhang, F., Hassan, A. E., McIntosh, S., and Zou, Y. (2017). The use of summation to aggregate software metrics hinders the performance of defect prediction models. *IEEE Transactions on Software Engineering*, 43(5):476–491.

Zhang, H. (2009). An investigation of the relationships between lines of code and defects. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pages 274–283.

Zhang, H. and Zhang, X. (2007). Comments on "data mining static code attributes to learn defect predictors". *Software Engineering, IEEE Transactions on*, 33(9):635 –637.

Zimmermann, T. and Nagappan, N. (2008). Predicting defects using network analysis on dependency graphs. In *Software Engineering, 2008. ICSE '08. ACM/IEEE 30th International Conference on*, pages 531–540.

Zimmermann, T., Nagappan, N., Gall, H., Giger, E., and 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*, ESEC/FSE '09, pages 91–100, New York, NY, USA. ACM.

Zimmermann, T., Premraj, R., and Zeller, A. (2007a). Predicting defects for eclipse. In *Predictor Models in Software Engineering, 2007. PROMISE'07: ICSE Workshops 2007. International Workshop on*, pages 9–9.

Zimmermann, T., Premraj, R., and Zeller, A. (2007b). Predicting defects for eclipse. In *Predictor Models in Software Engineering, 2007. PROMISE'07: ICSE Workshops 2007. International Workshop on*, page 9.

Appendix A

Appendix

Table A.1 The Summary of the NASA Datasets

Dataset	Language	KLOC	#features	#modules (pre-cleaning)	#modules (post-cleaning)*	%loss due to clean-ing	%faulty modules (pre-cleaning)	%faulty modules (post-cleaning)
CM1	C	20	43	505	296	41.4	9.5	12.8
JM1	C	315	24	10878	29	99.9	27.4	0.0
KC1	C++	43	27	2107	n.a.	0.6	15.4	15.5
KC3	Java	18	43	458	123	73.1	9.4	13.0
KC4	Perl	25	43	125	n.a.	n.a.	48.8	n.a.
MC1	C/C++	63	42	9466	115	98.8	0.7	0.0
MC2	C	6	43	161	1	99.4	32.3	100.0
MW1	C	8	43	403	253	37.2	7.7	10.7
PC1	C	40	43	1107	661	40.3	6.9	7.9
PC2	C	26	43	5589	745	86.7	0.4	2.1
PC3	C	40	43	1563	1043	33.3	10.2	12.2
PC4	C	36	43	1458	228	84.4	12.2	0.0
PC5	C++	164	42	17186	94	99.5	19.0	19.1

*cleaning using the comprehensive set of integrity checks reported in Section 4.5

Table A.2 The Summary of the NASA Dataset Metrics (as documented in Gray [2013])

Metric	Description
McCabe	Cyclomatic Complexity Cyclomatic Density Decision Density Design Density Essential Complexity Essential Density Global Data Density Global Data Complexity Maintenance Severity Module Design Complexity Pathological Complexity Normalised Cyclomatic Complexity
Halstead	Number of Operators Number of Operands Number of Unique Operators Number of Unique Operands Length Volume Level Difficulty Intelligent Content Programming Effort Error Estimate Programming Time
LOC Counts	LOC Total LOC Executable LOC Comments LOC Code and Comments LOC Blank Number of Lines (opening to closing bracket)
Misc.	Node Count Edge Count Branch Count Condition Count

	Decision Count Formal Parameter Count Modified Condition Count Multiple Condition Count Call Pairs Percent Comments
Error	Error Count Error Density

Table A.3 The Summary of the PROMISE Datasets (KLOC from Di Nucci et al. [2017])

Dataset	Language	KLOC	#features	#modules pre cleaning	#modules post cleaning	%loss due to cleaning	%faulty methods pre-cleaning	%faulty methods post cleaning
ant 1.7	Java	208	20	745	722	3.1	22.3	23.0
arc	Java	31	20	234	210	10.3	11.5	12.4
camel 1.6	Java	113	20	965	877	9.1	19.5	21.0
ivy 2.0	Java	87	20	352	345	2.0	11.4	11.6
jedit 4.2	Java	202	20	367	363	1.1	13.1	13.2
log4j 1.2	Java	38	20	205	202	1.5	92.2	92.6
lucene 2.4	Java	102	20	340	335	1.5	59.7	59.1
poi 3.0	Java	129	20	442	397	10.2	63.6	64.5
redaktor	Java	59	20	176	169	4.0	15.3	14.8
synapse 1.2	Java	53	20	256	244	4.7	33.6	35.2
tomcat	Java	300	20	858	791	7.8	9.0	9.7
velocity 1.6	Java	57	20	229	209	8.7	34.1	36.4
xalan 2.6	Java	428	20	885	724	18.2	46.4	44.6
xerces 1.4	Java	4	20	588	482	18.0	74.3	77.0

Table A.4 The Summary of the PROMISE Dataset Metrics (definitions provided in [Jureczko and Spinellis 2010])

Metric	Description	Source
Weighted methods per class (WMC)	The value of the WMC is equal to the number of methods in the class (assuming unity weights for all methods).	C&K
Depth of Inheritance Tree (DIT)	The DIT metric provides for each class a measure of the inheritance levels from the object hierarchy top.	C&K
Number of Children (NOC)	The NOC metric simply measures the number of immediate descendants of the class.	C&K
Coupling between object classes (CBO)	The CBO metric represents the number of classes coupled to a given class (efferent couplings and afferent couplings). This couplings can occur through method calls, field accesses, inheritance, method arguments, return types, and exceptions.	C&K
Response for a Class (RFC)	The RFC metric measures the number of different methods that can be executed when an object of that class receives a message. Ideally, we would want to find for each method of the class, the methods that class will call, and repeat this for each called method, calculating what is called the transitive closure of the method call graph. This process can however be both expensive and quite inaccurate. Ckjm calculates a rough approximation to the response set by simply inspecting method calls within the class method bodies. The value of RFC is the sum of number of methods called within the class method bodies and the number of class methods. This simplification was also used in the [Chidamber and Kemerer 1994]'s description of the metric.	C&K
Lack of cohesion in methods (LCOM)	The LCOM metric counts the sets of methods in a class that are not related through the sharing of some of the class fields. The original definition of this metric (which is the one used in ckjm) considers all pairs of class methods. In some of these pairs both methods access at least one common field of the class, while in other pairs the two methods do not share any common field accesses. The lack of cohesion in methods is then calculated by subtracting from the number of method pairs that do not share a field access the number of method pairs that do.	C&K
Lack of cohesion in methods (LCOM3)	$LCOM3 = \frac{(\frac{1}{a}\sum_{j=1}^{a}\eta(A_j))-m}{1-m}$ m - number of methods in a class; a - number of attributes in a class; $\eta(A)$ - number of methods that access the attribute A.	Henderson-Sellers
Afferent couplings (Ca)	The Ca metric represents the number of classes that depend upon the measured class.	Martin
Efferent couplings (Ce)	The Ca metric represents the number of classes that the measured class is depended upon.	Martin

Number of Public Methods (NPM)	The NPM metric simply counts all the methods in a class that are declared as public. The metric is known also as Class Interface Size (CIS).	QMOOD
Data Access Metric (DAM)	This metric is the ratio of the number of private (protected) attributes to the total number of attributes declared in the class.	QMOOD
Measure of Aggregation (MOA)	This metric measures the extent of the part-whole relationship, realized by using attributes. The metric is a count of the number of class fields whose types are user defined classes.	QMOOD
Measure of Functional Abstraction (MFA)	This metric is the ratio of the number of methods inherited by a class to the total number of methods accessible by the member methods of the class. The constructors and the java.lang.Object (as parent) are ignored.	QMOOD
Cohesion Among Methods of Class (CAM)	This metric computes the relatedness among methods of a class based upon the parameter list of the methods. The metric is computed using the summation of number of different types of method parameters in every method divided by a multiplication of number of different method parameter types in whole class and number of methods.	QMOOD
Inheritance Coupling (IC)	This metric provides the number of parent classes to which a given class is coupled. A class is coupled to its parent class if one of its inherited methods functionally dependent on the new or redefined methods in the class. A class is coupled to its parent class if one of the following conditions is satisfied: • One of its inherited methods uses an attribute that is defined in a new/redefined method. • One of its inherited methods calls a redefined method. • One of its inherited methods is called by a redefined method and uses a parameter that is defined in the redefined method.	Tang
Coupling Between Methods (CBM)	The metric measures the total number of new/redefined methods to which all the inherited methods are coupled. There is a coupling when at least one of the given in the IC metric definition conditions is held.	Tang
Average Method Complexity (AMC)	This metric measures the average method size for each class. Size of a method is equal to the number of Java binary codes in the method.	Tang

McCabe's cyclomatic complexity (CC)	CC is equal to number of different paths in a method (function) plus one. The cyclomatic complexity is defined as: $CC = E - N + P$, where E - the number of edges of the graph; N - the number of nodes of the graph; P - the number of connected components. CC is the only method size metric. The constructed models make the class size predictions. Therefore, the metric had to be converted to a class size metric. Two metrics has been derived: • Max(CC) - the greatest value of CC among methods of the investigated class. • Avg(CC) - the arithmetic mean of the CC value in the investigated class.	Mc
Lines of Code (LOC)	The LOC metric based on Java binary code. It is the sum of number of fields, number of methods and number of instructions in every method of the investigated class.	
C&K [Chidamber and Kemerer 1994] Henderson-Sellers [Henderson-Sellers 1995] Martin (Uncle Bob) [Martin 1994] QMOOD [Bansiya and Davis 2002] Tang [Tang et al. 1999]		

Table A.5 The Summary of the Commercial Datasets

Dataset	Language	KLOC	#features	#modules (pre-cleaning)	#modules (post-cleaning)	%loss due to clean-ing	%faulty modules (pre-cleaning)	%faulty modules (post-cleaning)
PA	Java	21	24	4996	4996	0.0	11.7	11.7
KN	Java	18	24	4314	4314	0.0	7.5	7.5
HA	Java	43	18	9062	8998	0.7	1.3	1.3

Table A.6 The Summary of the Commercial Dataset Metrics (defined in Virtual Machinery [2018])

Metric	Short Description
CAST	Number of class casts in the method
COMP	Cyclomatic Complexity
CREF	Number of different classes referenced in the method
EXCR	Number of exceptions referenced by this method
EXCT	Number of exceptions thrown by this method
HBUG	Estimated Halstead Bugs in the method
HDIF	The Halstead Difficulty of a method is an indicator of method complexity
HEFF	The Halstead Effort for the method is an indicator of the amount of time that it will take a programmer to implement the method
HLTH	The Halstead Length of the method
HVOC	The Halstead Vocabulary of the method
HVOL	The Halstead Volume of a method is an indicator of method size
LMET	Number of calls to local methods i.e. methods that are defined in the class of the method
LOOP	Number of loops in the method
MDN	Maximum Depth of Nesting
MOD	Number of modifiers in the method declaration
NAME	Name of method
NAND	Number of operands in the method
NEXP	Number of Java Expressions in the method
NLOC	Number of Lines of Code in the method
NOA	Number of arguments in method signature
NOC	Number of comments
NOCL	Number of comment Lines
NOPR	Number of operators in the method
NOS	Number of Java statements in the method
TDN	Total Depth of Nesting
VDEC	Number of variables declared in the method
VREF	Number of variable references in the method
XMET	Number of calls to methods that are not defined in the class of the method