Thesis no: MECS-2014-06

Software defect prediction using machine learning on test and source code metrics

Mattias Liljeson **Alexander Mohlin**

Faculty of Computing
Blekinge Institute of Technology
SE–371 79 Karlskrona, Sweden

This thesis is submitted to the Faculty of Computing at Blekinge Institute of Technology in partial fulfilment of the requirements for the degree of Master of Science in Engineering: Game and Software Engineering and Master of Science in Engineering: Computer Security. The thesis is equivalent to 20 weeks of full-time studies.

Contact Information:
Author(s):
Mattias Liljeson Alexander Mohlin
E-mail:
mdli09@student.bth.se, almo09@student.bth.se

External advisor:
Johan Piculell
Ericsson AB

University advisor:
Michael Unterkalmsteiner
Department of Software Engineering

Faculty of Computing
Blekinge Institute of Technology
SE–371 79 Karlskrona, Sweden

Internet : www.bth.se
Phone : +46 455 38 50 00
Fax : +46 455 38 50 57

Abstract

Context. Software testing is the process of finding faults in software while executing it. The results of the testing are used to find and correct faults. Software defect prediction estimates where faults are likely to occur in source code. The results from the defect prediction can be used to optimize testing and ultimately improve software quality. Machine learning, that concerns computer programs learning from data, is used to build prediction models which then can be used to classify data.
Objectives. In this study we, in collaboration with Ericsson, investigated whether software metrics from source code files combined with metrics from their respective tests predicts faults with better prediction performance compared to using only metrics from the source code files.
Methods. A literature review was conducted to identify inputs for an experiment. The experiment was applied on one repository from Ericsson to identify the best performing set of metrics.
Results. The prediction performance results of three metric sets are presented and compared with each other. Wilcoxon's signed rank tests are performed on four different performance measures for each metric set and each machine learning algorithm to demonstrate significant differences of the results.
Conclusions. We conclude that metrics from tests can be used to predict faults. However, the combination of source code metrics and test metrics do not outperform using only source code metrics. Moreover, we conclude that models built with metrics from the test metric set with minimal information of the source code can in fact predict faults in the source code.

Keywords: Software defect prediction, Software testing, Machine learning

Acknowledgements

We would like to thank our university supervisor, Michael Unterkalmsteiner, for great guidance regarding the conduction of a degree project and scientific methods. We would also like to him for giving feedback on the project and the thesis during its development. We would like to thank our industry supervisors Johan Piculell and Joachim Nilsson for continuous feedback during the experiment implementation, the metric collection process and the results analysis. Lastly, we would like to thank Jonas Zetterquist and Ericsson for making this degree project and ultimately the thesis possible by allowing us to work closely with the software development teams and giving us access to Ericsson's source code repositories.

Contents

Chapter 1

Introduction

In collaboration with Ericsson, we have investigated whether *Software Defect Prediction (SDP)* using test and source code metrics can identify hot-spots, i.e. files in the source code where defects often occurs. Ericsson has provided realistic industry-grade data which made our research and development possible and the results applicable to Ericsson. A discussion of the background follows as well as the formulation of the problem statement.

1.1 Motivation

The Ericsson M-commerce division maintains a platform for mobile commerce, written in the object-oriented programming language Java. Due to the domain, money transfers, the requirements for creating qualitative and secure products are highly important. Since the number of defects in products can be an indicator of software quality, Ericsson M-commerce have strict requirements regarding defects in their products.

Before a product can be shipped, it needs to be very well tested. The problem, however, is that it is not feasible to test every path in a program. As an example, Myers et al. [33, p. 10] constructed a toy program with 11 decision points and 15 branches. To perform exhaustive path testing on that simple program, it would require tests for 10^{14} paths. The number of paths in a realistic program render complete testing impractical due to the exponential growth of paths. Test management at Ericsson is therefore interested in how they can focus their testing on parts that are likely to be defective.They have problems identifying the source code files that are likely to fail during testing, though. If the testers were provided with information about files which might fail, the testing process could be focused on those files.

The lack of knowledge on what to focus the testing on can also lead to faults slipping through to later stages of the development which raises the cost associated with fixing them [4, 50]. Duka and Hribar note that the cost of rework due to faults can be lowered by 30-50% by finding faults earlier [16].

This area of study is important because of the cost saving potential it poses to Ericsson. If fault prone files are detected and testing is focused on the set of fault prone files, the testers can save time by not focusing the testing effort on files that are unlikely to contain faults and instead use that time to thoroughly test the fault prone files. By

identifying files likely to fail and focusing unit testing improvements to these areas, the testing process can be made more effective. The external quality[1] of the product is also raised when defects are found and corrected before the product is shipped.

An implementation of a tool that provides testers with information on which files that are more likely to fail can lead to some negative effects as well. Developers and testers may rely too much on the tool and actual faulty files that are not caught by the tool are not considered. This false sense of security can lead to faults slipping to later stages of the development and, as described before, introduce additional cost when correcting these faults.

Several studies that investigate defect prone modules in the source code have been conducted, even within the domain of telecommunication [1, 2, 25, 54]. This master thesis differs in its approach, compared to previous studies. The novel approach we propose is to use metrics from test(s), that are testing the source code file, in a combined metric set. To the authors' knowledge, this approach have not been previously studied.

1.2 Scope and limitations

This master thesis concerns the investigation and development of a tool for predicting faulty files which includes the following:

- A tool for extracting metrics from a *Source Code Management system (SCM)*
- A preprocessing tool for structuring the metrics into a desired format.
- An analysis, together with explanations, of produced results.
- A conclusion of why the results came out as they did.

The thesis do not, however, include;

- The development of a tool for data analysis, as this is done with the existing Weka environment [56].
- The measurement and evaluation of the prediction performance of the tool in a production environment.
- The production of a front-end tool for graphical representation of the results.
- The observation of the effect the tool has on real users, e.g. whether the tool leads to a sense of false security by its user, or, on the contrary, to mistrust in the tool.

[1] The term *"Quality"* is used in the sense defined by McConnell [29, p. 463]

1.3 Research questions

Following is the list of research questions this master thesis attempts to answer.

RQ1 - What Machine Learning algorithms have been used for SDP?

Identifying which *Machine Learning (ML)* algorithms that have been used in previous studies of SDP is highly important for this master thesis to make sure that it contributes to the research community. [The purpose of this master thesis is to investigate if the prediction performance of metric sets are increased if test metrics are included in the set. By using algorithms used in other studies, [the effects of incorporating test metrics can be directly studied]. The prediction performance of the produced tool can also be made state of the art as lessons from other studies regarding prediction performance can be incorporated.

RQ2 - What types of metrics have been used in prior SDP studies?

It is of major importance to recognize what kind of source code metrics that previously have been studied. This makes sure that the results of the study can be compared to those of other studies, and in turn making sure that the thesis contributes to the research community.

RQ3 - Are there any existing test metrics or test metric suites?

The answer of this question determines if there are any test metrics available which can be incorporated into the tool, or if all test based metrics need to be created for this master thesis. By using existing metrics, the results of the study can be compared to those of other studies and the contribution of this master thesis to the scientific community is thus strengthened.

RQ4 - Which ML algorithm(s) provide the best SDP performance?

As this master thesis investigates a novel metric family, test and source code metrics combined, there are no prior studies identifying suitable algorithms. The answer to this question will therefore contribute to the research area.

RQ5 - To what extent do test metrics impact the performance of SDP?

The novelty proposed by this master thesis is to use test metrics together with source code metrics. Therefore it is of importance to demonstrate to which extent the test metrics really contribute to the SDP performance.

1.4 Thesis outline

This master thesis is structured as follows. Chapter 1 introduces the thesis and its motivation. Chapter 2 discusses the thesis background, related work and how our work fit in and contributes to the computer science community. Chapter 3 discusses the scientific methods chosen for this master thesis. Chapter 4 discusses the experiment implementation, our approach and development methods. Chapter 5 presents the results while Chapter 6 discusses the results and answers the stated research questions. Chapter 7 chapter summarizes the thesis, presents conclusions and the final analysis as well as future work.

Chapter 2

Background and related work

Almost all computer software contain flaws that were introduced during code construction. Over time, some of these defects are discovered, e.g. by quality assurance testing. The later in the development process the defects are discovered and corrected, the more cost is associated with fixing them [4, 50]. It is therefore of major importance to find the flaws as early as possible.

One way of finding flaws or defects is to perform testing. Myers et al. defines software testing as *"the process of executing a program with the intent of finding errors"* [33]. Testing is done at many levels including unit testing, function testing, system testing, regression testing and integration testing. The first level of testing is the unit test level, where the functionality of a specified section of code is tested. As stated earlier, it is important to find the defects as early as possible to reduce costs. By focusing unit tests on the parts that are most fault prone, defects can be found earlier, and thereby reducing costs.

The process of predicting parts that are fault prone in software is called SDP. The idea behind SDP is to use measurements extracted from e.g. the source code and the development process, among others, to find out if these measurements can provide information about defects. Studies have shown that testing related activities consume between 50% to 80% of the total development time [12]. As this is a substantial part of the development process, it is important to focus the testing on the parts where defects are likely to occur due to its cost saving potential. SDP has been studied since the 1970's where simple equations, with measurements from source code as variables, was used as prediction techniques [17]. Since then, the focus for prediction techniques has shifted towards statistical analysis, expert estimations and ML [8]. Of these techniques, ML has proven to be the most successful approach [8, 22].

2.1 ML

ML is a branch of artificial intelligence concerning computer programs learning from data. ML aims at imitating the human learning process with computers, and is basically about observing a phenomenon and generalizing from the observations [24]. ML can be broadly divided into two categories: supervised and unsupervised learning. Supervised learning concerns learning from examples with known outcome for each of

the training samples [56, p. 40], while unsupervised learning tries to learn from data without known outcome. Supervised learning is sometimes called classification, as it classifies instances into two or more classes. It is classification that this master thesis focuses on since information about which files that have already been found faulty, exists in a *Trouble Report (TR)* database.

There has been a large variety of ML algorithms (also known as classifiers) for supervised learning from a variety of algorithm families presented in previous SDP studies. These families includes decision trees, classification rules, neural networks and probabilistic classifiers.

Catal et al. [9] concluded that Naïve Bayes, a probabilistic classifier, performs well on small [1] data sets. Naïve Bayes assumes that the attributes are independent from each other. This is rarely the case, but by using feature selection in SDP this assumption does not affect the results [53]. Feature selection is described in section 2.2.6

Numerous studies [26, 31, 32, 46, 58] have been using J48, which is a Java implementation of the C4.5 decision tree algoritm [43]. These studies shows that the algorithm does not produce any significantly better or worse results compared to other algorithms in the studies, except in [58] where J48 performs among the best algorithms, according to accuracy, precision, recall and f-measure, on publicly available data sets. Arisholm et al. [1] claims that the C4.5 algorithm performs *"very well overall"*, and suggests that more complex algorithms are not required.

Random Forest, which classifies by constructing a randomized decision tree in each iteration of a bagging (bootstrap aggregating) algorithm [56, p. 356], has proven to give high accuracy when working with large[2] data sets [9]. Bagging is an ensemble learning algorithm that uses other classifiers to construct decision models [5]. Random Forest can be seen as a special case of bagging, that uses only decision trees to build prediction models [26].

Several studies have tested the ability of neural networks for discriminating between faulty and nonfaulty modules in source code [1, 27, 58]. The results from these studies have shown that neural networks perform comparable to other prediction techniques. Menzies et al. [31], however, proposed using neural networks for future work, as they thought neural networks were too slow to include in their study.

Another previously used method is JRip, an implementation of RIPPER which is a classification rule learner [3, 48]. Classification rule learners greatest advantage is that they are, in comparison, easier for humans to interpret than corresponding decision trees.

Menzies et al. [31] maked use of OneR, a classification rule algorithm, to test thresholds of single attributes. They conclude that OneR is outperformed by the J48 decision tree most of the times. Shafi et al. [48] used, in addition to OneR, another classification rule algorithm called ZeroR and was outperformed by OneR. ZeroR predicts the value of the majority class [56, p. 459]. Arisholm et al. has in two different stud-

[1] In their case around 20 000 lines of code [9]

[2] In their case up to 460 000 lines of code [9]

ies [1, 2] used the meta-learners Decorate and AdaBoost, together with J48 decision tree. They claim that Decorate outperforms AdaBoost on small data sets and performs comparably well on large data sets. They do not however disclose their definition of *"small"* and *"large"* data sets.

2.2 Software metrics

Software metrics can be described as a quantitative measurement that assigns numbers or symbols to attributes of the measured entity [18]. An attribute is a feature or property of an entity, e.g. length, age and cost. An entity can be the source code of an application or a software development process activity. There are several different families of software metrics. Those that are discussed in this master thesis are presented below.

2.2.1 Static code metrics

Static code metrics are metrics that can be directly extracted from the source code, such as *Lines of code (LOC)* and cyclomatic complexity.

Source Lines Of Code (SLOC) are a family of metrics centred around the count of lines in source files. The SLOC family contains, among others: *Physical LOC (SLOC-P)*, the total line count, *Blank LOC (BLOC)*, empty lines, *Comment LOC (CLOC)*, lines with comments, *Logical LOC (SLOC-L)* and lines with executing statements [41].

The *Cyclomatic Complexity Number (CCN)*, also known as McCabe metric, is a measure of the complexity of a module's decision structure, introduced by Thomas McCabe [28]. CCN is equal to the number of linearly independent paths and is calculated as follows: starting from zero, the CCN is incremented by one whenever the control flow of a method splits, i.e. when `if`, `for`, `while`, `case`, `catch`, `&&`, `||` or `?` is encountered in the source code.

Other static code metrics include compiler instruction counts and data declaration counts.

Object-oriented metrics

A subcategory of static code metrics are object-oriented metrics, since they are also metrics derived from the source code itself. The *Chidamber-Kemerer (CK) Object-Oriented (OO) Metric Suite* [10] used in this master thesis consists of eight different metrics: six from the original CK metric set and two additional metrics, which follows below.

Weighted Methods per Class (WMC) counts the number of methods in a class.

Depth of Inheritance Tree (DIT) counts length of the maximum path from class to root class.

Number of Children (NOC) counts the number of subclasses for one class.

Coupling Between Objects (CBO) counts the number of other classes to which the class is coupled, i.e. the usage of methods declared in other classes.

Response For Class (RFC) is the number of methods in the class, added to the number of remote methods called by methods of the class.

Lack of Cohesion in Methods (LCOM) counts the number of methods in a class that does not share usage of some member variables.

The additional OO metrics that are used in the thesis are:

Afferent couplings (Ca) counts how many other classes that use the class (cf. CBO).

Number of Public Methods (NPM) counts how many methods in the class that is declared as "public".

Menzies et al. [31] used static code metrics such as SLOC and CCN for defect prediction. They observed that the prediction performance was not affected by the choice of static code metrics. What does matter is how the chosen metrics are used. Therefore, the choice of learning algorithm is more important than the metrics used for learning. They measure performance with *Probability of Detection (PD)* and *Probability of False alarms (PF)*. Using Naïve Bayes with feature selection, they gained a PD of 71% while keeping the PF at 25%. Zhang et al. [59] commented on them using PD/PF performance measures and proposed the use of prediction and recall instead.

Zhang [58] investigated if there exist a relationship between LOC and defects on publicly available datasets. He argued that LOC, together with ordinary classification techniques (ML algorithms), could be a useful indicator of software quality and that the results are interesting since LOC is one of the simplest and cheapest software metric to collect.

Nagappan et al. [37] concludes that McCabe's CCN can be used together with other complexity metrics for defect prediction. They note, however, that no single set of metrics is applicable for all of their projects.

2.2.2 Dependency metrics

The division of the workload of different components[3] may be decided in the design phase of software development. This division of tasks can impact the quality of the software. Dependency mapping between source code files could therefore be a possible predictor of fault prone components. Schröter et al. [47] proposed a novel approach to detect fault prone source files and packages. They simply collected, for each source code file or package, the import statements and compared these imports to failures in

[3] Defined in IEEE Standard Glossary [23]

components. In this master thesis we denote this procedure of collecting imports as dependency mapping.

Schröter et al. [47] concluded that the sets of components used in another component, determines the likelihood of defects. They speculate that this is due to that some *"domains are harder to get right than others"*. Their results point out that collecting imports on package level yields a better prediction performance than on file level. Still the results on file level perform better than random guesses.

2.2.3 Process metrics

Process metrics are metrics that are based on historic changes on source code over time. These metrics can be extracted from the SCM and include, for example, the number of additions and deletions from the the code, the number of distinct committers and the number of modified lines.

Moser et al. [32] compared the efficiency of using change (process) metrics to static code metrics for defect prediction. Their conclusion was that process data contains more discriminatory information about defect distribution than the source code itself, i.e source code metrics. Their explanation to this is that source code metrics are concerned with the human understanding of the code, e.g. many lines of code or complexity are not necessarily good indicators of software defects.

Nagappan et al. [34] used metrics concerning code churn, a measure of changes in source code over a specified period of time. They conclude that the use of relative code churn metrics predict the defect per source file better than other metrics and that code churn can be used to discriminate between faulty and nonfaulty files.

Ostrand et al. [42] used some process metrics explaining if a file was new and if it was changed or not. The authors used these metrics in conjunction with other metrics families and found that 20% of the files, identified by the prediction model as most fault prone, contained on average 83% of the faults.

2.2.4 Test metrics

Test code metrics can be made up of the same set of metrics that has been previously described, but for the source code of the tests. In addition to this, Nagappan [36] introduced a test metric suite called *Software Testing and Reliability Early Warning metric suite (STREW)* for finding software defects. STREW is constituted by nine metrics, in three different families:

Test quantification metrics

SM1 $\frac{Number\ of\ assertions^4\ in\ test}{Source\ LOC}$

SM2 $\frac{Number\ of\ test\ cases}{Source\ LOC}$

SM3 $\frac{Number\ of\ assertions\ in\ test}{Number\ of\ test\ cases}$

SM4 $\frac{Test\ LOC\ /\ Source\ LOC}{Number\ of\ test\ cases\ /\ Number\ of\ source\ classes}$

Complexity and OO metrics

SM5 $\frac{\sum Cyclomatic\ complexity\ in\ test}{\sum Cyclomatic\ complexity\ of\ source}$

SM6 $\frac{\sum CBO\ of\ test}{\sum CBO\ of\ source}$

SM7 $\frac{\sum DIT\ of\ test}{\sum DIT\ of\ source}$

SM8 $\frac{\sum WMC\ of\ test}{\sum WMC\ of\ source}$

Size adjustment metric

SM9 $\frac{Source\ LOC}{Minimum\ source\ LOC}$

Nagappan et al. conclude that STREW provides an estimate of the software's quality in an early stage of the development as well as identifying fault prone modules [35]. Further studies conducted by Nagappan et al. [39] indicates that *STREW for Java (STREW-J)* metric suite can effectively predict software quality.

2.2.5 Combinations of metrics

Previous studies [6, 32], have combined metrics from several metric families in an attempt to achieve better prediction performance.

Caglayan et al. [6] constructed three different sets of metrics. The first set was static code metrics and the second was repository metrics. The third was the two first two combined. The results from this study showed that while they did not got better prediction accuracy, they did get a decrease in the false positive rate when using the combined set.

Moser et al. [32] constructed a static code set, a process metric set and a combined static and process set. They concluded that the combined set performed equally well as the process set, indicating that it is not worth collecting static code metrics.

2.2.6 Feature selection

Feature (or attribute) selection, is a method for handling large metric sets, to identify which metrics contribute to the SDP performance. By using feature selection, redundant, and hence non-independent, attributes are removed from the data set [56, p. 93].

[4] Assertions are code that is used during development that allows a program to check itself during runtime [29, p. 189]

There are two approaches to feature selection: wrappers and filters. Wrappers use the ML algorithm itself to select attributes to evaluate the usefulness of different features. Filters use heuristics based on the characteristics of the data to evaluate the features [20]. The positive effects of using filters over wrappers is that they operate faster [21], and are hence more appropriate for the selection in large feature sets. Another positive effect of using filters is that they can be used together with any ML algorithm. However, in most cases they require the classification problem to be discrete, whereas wrappers can be put into use with any classification problem. As wrappers uses the same algorithm for feature selection and classification, the feature selection process must be done for every algorithm used for prediction.

A filter based feature selection approach called *Correlation based Feature Selection (CFS)*, presented by Hall [20], has been used in several studies [1, 2, 9]. Unlike other filter based feature selection algorithms that evaluate individual features, CFS evaluates subsets of features by taking both the merit of individual features and their intercorrelations into account [21]. The merit of a feature subset is calculated using Pearson's correlation, by dividing how predictive a feature group is with the redundancy between the features in the subset. If a feature is highly correlated with one or more of the other features in the feature set, that feature will not be considered to be included in the reduced feature set. Hall claims that the algorithm is simple, fast to execute and, albeit a filter algorithm, can be extended to support continuous class problems [21].

Shivaji et al. [49] proposes a feature selection technique for predicting bugs in source code. They show that a small part of the whole feature set led to better classification results than if using the whole set. After the feature selection process, they gain a prediction accuracy of up to 96% on a publicly available dataset. They do however, train the classifier on the feature selected set, which can imply that the model is overfitted. Smialowski et al. [51], states that special consideration is required when using supervised feature selection. They state that one has to make sure that the data used to test a prediction model is not used in the feature selection process. If all data is used for feature selection, the predictive performance of the prediction model will be overestimated. Arisholm et al. [1, 2] have a negative effect in prediction performance when using the reduced data set from feature selection, which can imply that the prediction model in [49] is overfitted.

Chapter 3

Method

As this master thesis covers a new family of metrics in SDP, namely test metrics, having a broad set of algorithms from multiple categories were deemed important as the prediction performance of test metrics was unknown. To be able to compare the results to state of the art research and to gain fair results for the test metrics compared to the source code only metrics, a broad set of metrics of metrics which have performed well in prior studies where therefore deemed necessary. With this in mind, a literature review was conducted to gain further knowledge of these areas. The results from the literature review was then used in an experiment. The relationship between these two methods is visualized in Fig. 3.1. This figure also shows how the answers to research questions regarding the literature review acted as input to the experiment.

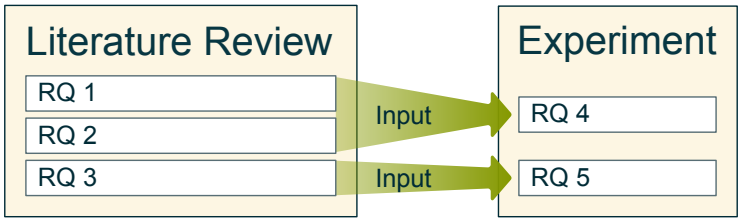

Figure 3.1: The relation between methods and research questions

This chapter is divided into two separate parts to mirror the way this master thesis project was conducted. The first part describes the traditional literature review, which was done to gain the required background about software defect prediction, ML, source code metrics and test metrics. The second part documents the experiment that tested the hypothesis: *"Source code metrics together with test metrics provide better prediction performance than SDP using only source code metrics"*.

3.1 Literature review

To answer the research questions 1 – 3, a literature review was conducted. This was done to ensure a thorough understanding of the area of software defect prediction, i.e. which ML algorithms and which set of metrics that have previously been studied. Another reason for doing a literature review was to determine whether the gap in the

research, that was identified during the planning stage, really exists. This analysis also provided insight into the state of the art research of the area, where the knowledge of domain experts' contributed to the reliability of this master thesis.

3.1.1 Study selection

The material for the literature analysis was collected using a bibliographic search engine that sources data from article databases, journals and conference proceedings. A search strategy as well as a *"Literature Review Protocol"'(LRP)* was developed to distinguish, for the thesis, relevant material.

In the domain of software defect prediction, several systematic literature reviews have been conducted. The method used in this master thesis was to examine the articles mentioned from five *"Systematic Literature Reviews" (SLR)* from 1999 to 2013 [17, 8, 7, 22, 44]. These articles were then examined closer in order to determine if they matched the description outlined in the thesis LRP.

In the LRP it was stipulated that the study selection criteria is based on title and author. For the title, only SDP articles, conference proceedings etc. with some sort of ML were selected for further examination, excluding studies using statistical analysis that were not in the scope of this master thesis. The author or authors of the paper were also examined. If the author had released two or more papers in the domain, and the paper was referenced by other papers, the paper was considered as a candidate for further examination in the literature review.

The other domain related to this master thesis is test metrics. In order to collect articles about test metrics, multiple search strings with multiple spelling possibilities were created iteratively. The selection of these articles followed the approach described above.

3.1.2 Study quality assessment

In order to assess the articles, several culling steps were conducted. Firstly, the abstract of each article was closely examined. If the abstract was related to this master thesis according to the LRP, the article was rated as approved. If not, the article was discarded. The same procedure was repeated for the conclusion. If the articles passed the scrutinization of both abstract and conclusion they underwent a detailed study of the contents. The overall quality of the article was thereafter rated based on: relevance to this master thesis, the type of publication and presented prediction performance.

The average score of the allotted rates was then calculated and the articles were categorized into one out of four categories based on an averaged score. During the score calculation phase, important passages in the articles were highlighted and tagged to point out why the passage was important.

3.1.3 Synthesis of extracted data

In order to be able to synthesize the work, a collaboration tool was used. The tool attach a note section to each article which was used to note a summary of the article, what rating the article got and why it was important for this master thesis. The tool also has the possibility to tag articles, to highlight sections of text and place comments where needed. Based on the averaged score in the study quality assessment phase, the article was categorized. The tags attached to the articles was used to point out where in the master thesis the article should be mentioned. The comments and highlights were used as a basis for the discussions in the thesis.

3.1.4 Threats to validity

The threats to validity in the literature review are mainly concerned with the selection of studies. The selected studies can have both publication biases and agenda driven bias. To mitigate the threat of the articles being publication biased, articles from several SLR's were chosen. Additionally, several different search strings for finding articles concerning the other topic was created iteratively as outlined by Zhang et al. [57]. By conducting the literature review in this way, chances are better that more aspects of the field of study are covered. The threat of agenda driven bias, i.e. that the authors of an article only produces text that suites their purposes, is also mitigated by following the before mentioned procedure.

Another identified threat is that only studies with positive results are published. It could happen that studies with similar settings with negative results have been performed, but their findings might not get published, or, if published, might not get noticed by the research community This threat is difficult to mitigate, but efforts included using many SLR's to broaden the article selection and by using a LRP to minimize the impact of biased mindsets.

3.1.5 Answers to research questions 1 – 3

The results from the literature review was mainly used as input to the ML algorithm selection and metric family selection.

RQ1 - What ML algorithms have been used for SDP?

The main ML algorithms used in prior studies are Bagging, Boosting, Decorate, C4.5, Naive Bayes, Neural networks, Random Forest and RIPPER. These were chosen to be used in the experiment as they have been used in multiple prior studies, are of different categories and have shown results indicating that they can be used successfully in SDP. ZeroR and OneR were selected to be used as baseline to test other algorithms' prediction performance. These algorithms are discussed in Section 2.1. The implementations of these algorithms used in the experiment are presented in Section 4.3.1.

RQ2 - What types of metrics have been used in prior SDP studies?

The main metric families used in prior studies are static code metrics, process metrics, dependency metrics, complexity metrics and object-oriented metrics. These families were selected to be used in the experiment as they have been used in multiple prior studies and have shown results indicating that they can be used successfully in SDP. Object oriented metrics were especially selected as the analysis was done on a object oriented language and as the STREW-J metrics set depended on the CK metric set. The metric families are discussed in Section 2.2.

RQ3 - Are there any existing test metrics or test metric suites?

There have been articles published regarding metrics for unit tests and test in general. These articles are a series of articles written by Nachiappan Nagappan on a set of metrics called STREW. Applied to Java, these metrics are called STREW-J. These are also discussed in Section 2.2.4. These were included in the experiment.

3.2 Experiment

To test the hypothesis *"Source code metrics together with test metrics provide better prediction performance than SDP using only source code metrics"*, a factorial experiment (using feature selection) was conducted. The two factors in the experiment were the ML algorithms and the metric sets.

Factor 1 \ Factor 2	Source code metrics	Test metrics	Source code and test metrics
ML algorithm 1	T_S1	T_U1	T_C1
ML algorithm 2	T_S2	T_U2	T_C2
...	...	...	...
ML algorithm n	T_Sn	T_Un	T_Cn

Table 3.1: Visualization of the experiment design. T denotes *"Treatment"*.

The metric sets comprise metrics from static code analysis, change/process analysis and test metrics among others. These are explained in Section 2.2, while all included metrics families are listed in Section 4.1. The different metric sets are defined as follows. The *Source code metric set (Srcs)* contains only metrics related to the source code itself and their files. The *Test source code metric set (Tsts)* contains metrics related to the test source code, the respective files and metrics only available to tests, e.g. STREW-J metrics. The *Combined metric set (Cmbs)* is the combination of both of these sets, i.e. the union of them, Cmbs $=$ Srcs $\cup$ Tsts. These three sets only contain source code metrics from source files that have tests testing them as the datasets would

otherwise differ. Comparisons between the data sets and their results would not be possible if the data sets would differ.

When constructing the Cmbs and Tsts sets, there is a possibility of contamination of the data. The risk is that if a source file has been reported as faulty, i.e. a TR has been issued, a unit test to test the defected file has been created. Thereby, the mere presence of an unit test can be a clue that a source file is faulty. This problem was mitigated by dividing the collected data set into two separate parts based on a pivot point, a point in time (see Fig. 3.2). Only data older than the pivot point and TRs newer than the pivot point were considered. By doing the separation in this way, the unit tests created to test a faulty file was not considered in the prediction, and only unit tests created before a fault had been reported was included in the metric set.

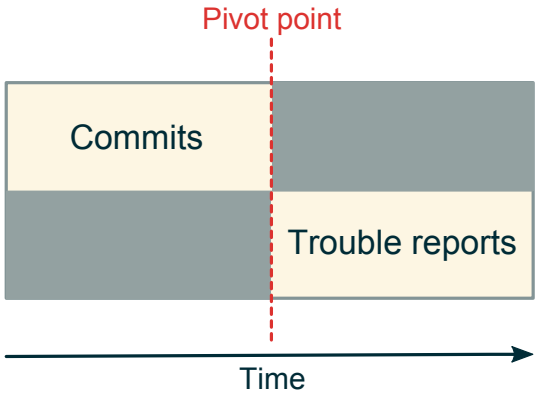

Figure 3.2: Separation of TRs and metric extraction

The prediction model process and evaluation was done three times with a different set of metrics for each of the n ML algorithms, for a total of $n \times 3$ treatments. In the first iteration, only source code metrics were used. In the second iteration, only test metrics were used and in the third iteration, both source code metrics and test metrics were used. The results from these treatments were then compared to test the hypothesis and to answer research questions RQ4 and RQ5.

3.2.1 Threats to validity

The threats to the validity are mainly the reproducibility and generalizability of the experiment since the source data is confidential and only comes from one repository of one product. This is not only a negative aspect as the data is from a real industry product and not a "toy" project. The results should therefore be applicable to similar projects with similar settings. To mitigate the problem with the lack of reproducibility and generalizability of the experiment, relevant information about the product is disclosed. The same experiment can therefore be conducted on similar projects and the results can be compared.

3.2.2 Experiment outline

The experiment design consisted of several independent variables and a single dependent variable. The independent variables were the ML algorithms and the metrics, identified in the literature review. The treatments were the combinations of these. The dependent variable was the occurrence of corrections to the file. The experiment, its variables, objects and hypotheses are further described below.

Dependent variable

The dependent variable was the occurrence of corrections to a tested file due to TRs. The source code file was asserted as faulty if there had been a correction in that particular file. If no correction had been done to a specific file, it was considered as fault free and hence not faulty. This approach will however result in that a file that has had a fault in its lifetime, is always considered as faulty. The mitigation to this was to introduce a time window and a point in time, a pivot point. The file was only considered faulty if the TR was newer than the pivot point. Metrics from the file were extracted from before the time of the pivot point. Furthermore, only files changed in the specified time window were used.

Independent variables

The independent variables were the different ML algorithms: AdaBoost (Boosting), Bagging, Decorate, J48 (C4.5), JRip, Multilayer Perceptron[1], Naïve Bayes, OneRule, Random Forest and ZeroR. In addition to the ML algorithms, the different metric sets, Srcs, Tsts and Cmbs, were also independent variables.

Treatments

One treatment was the combination of the levels of the two independent variables. In this case the metric set and the ML algorithm. Each treatment in the experiment, in Table 3.1 denoted as "T", produces a separate result. In total, we considered 30 treatments in the experiment.

Objects

The objects in the experiment was the files that the metrics was extracted from. From the model built by the treatments, a file was classified as either faulty or not faulty.

Hypothesis

The hypothesis tested in this experiment was: *"Source code metrics together with test metrics provide better prediction performance than SDP using only source code met-*

[1] A neural network implementation

rics". The corresponding null hypothesis used in the experiment is stated as: *"Source code metrics together with test metrics provide equal prediction performance as SDP using only source code metrics"*.

Chapter 4

Experiment implementation

As described in the method section, an experiment was performed to test the discriminatory power of different metric sets with different ML algorithms. In order to collect these metrics, a tool for scraping them, together with a tool for building and evaluating the prediction models was developed. This chapter describes these tools.

Before collecting metrics and faults, a repository on which to conduct the experiment must be selected. The main factors when selecting a repository for this experiment was the size in terms of number of files and test coverage[1]. Larger size was preferred over smaller in order to make the results more accurate and generalizable. The repository had to have a high test coverage since the central theme in the hypothesis was the inclusion of test metrics. Choosing a repository with high test coverage assures that many files are tested and hence minimizes test bias. The selected repository had about 200 KLOC spread over about 2000 files and a test coverage of around 70%.

4.1 Data collection

To be able to collect file level metrics from the SCM Git[2], several Groovy[3] scripts were developed. These tools were then included in a Gradle[4] script as tasks, that allows the scripts to be run sequentially in an automatic manner. The data collection process is outlined in Fig. 4.1. When possible, external tools used in other studies were used when extracting metrics as that would give fair results and make the results comparable to other studies. Where there where no tools available or where the available tools were not applicable due to technical restraints, the metric extraction tools were developed for the experiment.

[1] The source code's degree of testing by tests

[2] `http://git-scm.com/`

[3] `http://groovy.codehaus.org/`

[4] `http://www.gradle.org/`

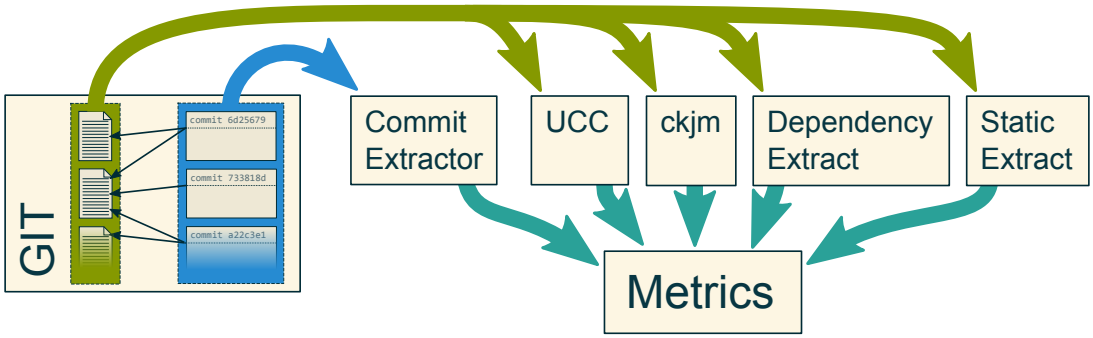

Figure 4.1: Visualization of the extraction process

4.1.1 Identify faulty files

In order to train a learner, the source code files need to be classified as either defective or non-defective. For this analysis, all commits to the SCM were investigated. If a TR identification number is included in the commit, it is an indication that a fault has been corrected and hence, the file that is affected by the commit has been corrected. That file is thereafter classified as defective, while files that are not affected by any TRs are classified as non-defective.

4.1.2 Collection of process metrics

The extraction of different process metrics starts with the analysis of all commits to the repository. Each commit contains information about the files that are affected by that particular commit. Process metrics are generally easy to extract from the SCM, as the SCM contains information about which developer changed what file, when the file was changed and how it was changed during a specified amount of time. This specified amount of time could be the time between two commits that affected the file. The metrics that were extracted from this information included authors and line changes, e.g. added/inserted lines and lines that have been deleted. The author metrics are generated by extracting all authors from a given repository. Each author is added as a separate metric to all files with the value of 0. Each file is then analyzed and the metrics corresponding to the authors that have contributed to a file are set to 1. The line change metrics, additions, deletions and churn, are calculated by collecting all commits related to the given file. Information regarding each line change metric are then extracted from the commits. Statistical measures are then extracted from the gathered information for each line change metric. These measures are minimum value, maximum value, mean, median, standard deviation and sum. These are then added to each file as metrics. A complete listing of extracted metrics is found in Table 4.1 under the header *"Process"* and *"Author"*.

4.1.3 Collection of static code metrics

There are several tools freely available for collecting static code metrics, but they each has a rather limited field of application. Therefore, in order to collect several static code metrics, several different tools have been used. To collect LOC and other related metrics, *Unified Code Count (UCC)*[5] was used. UCC collect metrics such as SLOC-P, BLOC, CLOC and SLOC-L along with compiler instruction counts and data declaration counts. UCC and its features are documented by Nguyen et al. in [41]. The CCN was extracted using a tool that was developed for this master thesis project.

The CK OO metric suite was calculated using an existing tool called *ckjm*[6] [52]. The metrics that are calculated by ckjm constitutes of eight different metrics, six from the original CK metric set and two additional metrics, which are listed in section 2.2.1.

4.1.4 Collection of test and dependency metrics

The dependency metrics discussed in Section 2.2.2 were collected through a dependency mapping script created for this master thesis.

In this script, all files are opened and all import statements gathered. These import statements are then analyzed to see whether the imported files are a part of the repository. If that is the case, they are tested to find out whether they are test related or source code dependencies. Each file in the repository is then scanned to see whether they include a certain dependency. If they do, the dependency is added as a metric with a value of 1. If they do not, the dependency is added as a metric with a value of 0. I.e. each file that has been imported by another file is added as a metric to the file importing it. All of the dependency metrics were collected for both source code and test source code. In the case of this study with a repository of 200K source LOC, the amount of dependency metrics was around 20 000, in other words the number of unique import statements.

The eight first metrics from STREW-J listed in subsection 2.2.4 were applied to the Tsts and Cmbs only as these are test specific metrics. They were collected by synthesizing results from the existing metrics from which they are created. The collection and synthesis of the other test related metrics are elobarated in Section 4.2.

4.2 Metric sets

The extraction of metrics described above results in a single file containing all extracted metrics per file. Another script is then responsible for the creation of three distinct metric sets, Srcs, Tsts and Cmbs, as described in Section 3.2. This script is also responsible for calculating the test specific metrics described earlier. All informa-

[5] `http://sunset.usc.edu/research/CODECOUNT/`

[6] `http://www.spinellis.gr/sw/ckjm/`

Author	Process	Dependencies	ckjm (OO)	Static code Analysis	Universal Code Counter (UCC)	STREW-J
Author 1	Additions (add)	External dep 1	CA	BLOC	BLOC	SM1
Author ...	Deletions (del)	External dep ...	CBO	BLOC-LOC ratio	Compiler directives	SM2
Author n	Commit date	External dep n	DIT	CCN	Data declarations	SM3
	Codechurn (add-del)		LCOM	CCN-LOC ratio	Embedded comments	SM4
		Internal dep 1	NOC	CLOC	Executable statements	SM5
	For each of the above:	Internal dep ...	NPM	CLOC-BLOC ratio	SLOC-L	SM6
	Min	Internal dep n	RFC	CLOC-LOC ratio	SLOC-P	SM7
	Mean		WMC	LOC	LOC	SM8
	Median	Test dep 1		Nr of asserts	CLOC	
	Max	Test dep ...		Nr of class defintions		
	Standard deviation	Test dep n		Nr of interface definitions		
	Sum			Nr of test definitions		

Table 4.1: A listing of all the metrics used to build the metric sets

tion needed for calculating these metrics is available in the produced metric file. This process is visualized in Fig. 4.2.

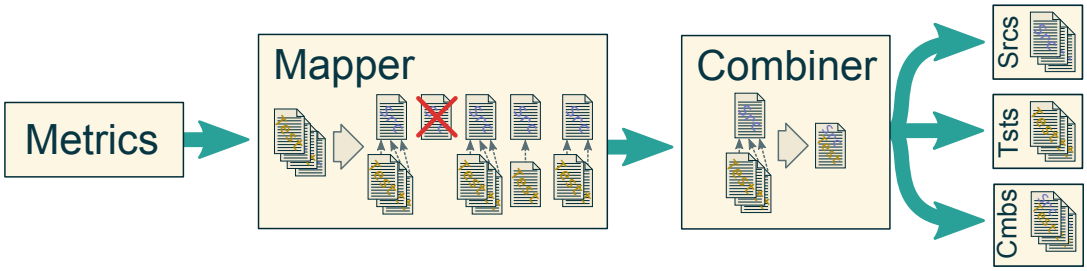

Figure 4.2: Visualization of the mapping and combining process

The mapping between test source code and source code described in Section 3.2 is done at this stage. This is done by analyzing the dependency metrics of each file to map test source code files to source code files. Source code files with no corresponding test source files are culled so that all three metric sets contain the same files. This is done so that results from the different sets can be compared as they consist of the same files, albeit with different metrics. The mapping is then used when synthesizing test metrics. All metrics for the mapped tests are gathered per source file and statistical measures are then calculated from the gathered test file metrics. These measures are: minimum value, maximum value, mean, median, standard deviation and sum. These measures are then added as metrics to the Cmbs and Tsts. For the Cmbs, the metrics from the source file are also added. The last step is calculating STREW-J for the Cmbs and Tsts. The dependent metrics from which STREW-J are calculated are fetched and the calculated STREW-J metrics are added to the sets. As the study only concerns the identification of faulty source code files, data of faulty test source code files are removed from the sets. A listing of all the metrics used to build the metric sets is presented in Table 4.1.

4.3 Model building

To build the prediction models, file level data from one big repository consisting of 200 000 LOC Java source code were used. There exist a large number of smaller repositories that could have been included in this master thesis project. However, it has been suggested that, because of that different processes largely depends on the operating environment, generalizing results between systems and environments might not be viable [2, 38].

4.3.1 ML algorithms

As the experiment was conducted to test the discriminatory power of different metric sets, the ML algorithms included were used with their standard settings in the WEKA environment [56]. The algorithms used to build the prediction models in this master thesis are listed in below:

Naïve Bayes - a simple probabilistic classifier.

J48 - an implementation of the C4.5 decision tree algorithm.

Decorate J48 - Decorate is a meta-learner that builds a diverse ensemble of classifiers using other learning algorithms [30]. In this case, J48 is used.

Random Forest - that classifies by constructing a randomized decision tree in each iteration of a bagging (bootstrap aggregating) algorithm [56, p. 356].

OneR - an implementation of 1-rule (1R), which is a simple classification rule algorithm.

JRip - an implementation of RIPPER, a classification rule generating algorithm that optimizes the rule set. [11].

ZeroR - is a very simple method for generating rules. It simply predicts on the majority class of the test data [56, p. 459].

Multilayer perceptron - a neural network that learns trough backpropagation to set the appropriate weights of the connections [56, p. 235].

AdaBoost - AdaBoost is a meta-learner that, in theory, combines tweaked "weak" learners into a single "strong" learner. In this case, Wekas' standard setting DecisionStump, is used.

Bagging - Bagging is also, like Decorate, an ensemble learning algorithm. In this case, Wekas' standard setting REPTree, is used.

4.3.2 Feature selection

The computational complexity of some of the previously mentioned ML algorithms makes the model building infeasible to use if all of the features in the dataset is used. As an example, Menzies et al. [31] ruled out neural networks as *"they can be very slow"*. Therefore, to be able to compare all of the ML algorithms, as the datasets used in this master thesis project rise up to approximately 20 000 different metrics, feature selection is used to reduce the dataset before building models. As a wrapper uses the ML algorithm itself when creating the subset, a new feature selected subset would have to be created for each classifier. Some of the ML algorithms chosen would also have to be removed, e.g Multilayer Perceptron, as their execution time would make them inappropriate for a production setting. This problem became apparent during the development of the experiment. When choosing a feature selection approach, filtering was therefore chosen. The tool for feature selection in this master thesis uses CFS, presented by Hall [20]. As the search method for finding the metric subset to evaluate with CFS, best first search was chosen. This choice was made because of the claim in [20] that it gave slightly better results over other search methods for CFS. The feature selection process was done on the training set in all of the runs of the evaluation process described in Section 4.3.3.

4.3.3 Model evaluation

To evaluate the prediction models produced, 10 times 10-fold cross validation was performed. By using 10-fold cross validation, the dataset is separated into 10 somewhat equally large parts. Nine parts out of 10 are used as training data and the feature selection process while the 10th part is used for testing exclusively. This process is repeated 10 times so that every instances of the data set is used as testing set one time.

In order to guarantee that the distribution of the samples into training and test set was representative of the distribution of the whole set, stratification was also used in the evaluation process. Stratification means that each fold in the cross validation process gets approximately the same amount of faulty files. If this is not done, a classifier within a fold can perhaps be built with no faulty files occurrences, and therefore skew the result of the whole 10-fold cross validation prediction.

Cross validation can be used in cases where the amount of data is limited, but there are other advantages compared to other evaluation methods, e.g. the holdout method. When using the holdout method, one part of the data set is used for training and the other for testing, typically two thirds for training and the remaining third for testing. This could mean that the samples in the training data is not representative for the whole data set. One way of mitigating this bias is to use cross validation where all the instances have been used exactly one time for testing. Witten et al. [56, p.153] states that a single 10-fold cross validation might not produce a reliable error estimate. They state that the standard way of overcoming this problem is to repeat the 10-fold cross validation 10 times and average the results. This means that, instead of conducting 10

folds, a total of 100 folds is generated and the error estimate is therefore more reliable.

When using feature selection, it is important to remember to take this into account when performing cross validation. For the prediction not to be skewed, each fold of the cross validation must include a feature selection step, in order to keep test data from being present in the training data and hence in the feature selection process. As the feature selection is conducted in every fold of the 10 times 10-fold cross validation, the process is done 100 times. As both the feature selection process and the learning process are computationally intensive, the whole evaluation process consumes a lot of time. In the case of this study, the complete process took about 4 days to complete a data set with about 20000 features and 1000 instances using a mid-range 2013 laptop with 8GB RAM and a 1.8Ghz Intel i5 processor.

All results from every fold and every round are recorded in a result file for further analysis. Hall et al. [22] describes in their systematic literature review different appropriate performance measures. The result file produced by the evaluation process therefore contains these measures, i.e. average precision, average recall and average f-measure. In addition to these performance measures, the average accuracy, the percentage of the correctly classified instances, of the prediction model is also included.

Precision measures the proportion of the identified files, classified as faulty, that actually are faulty. This is a measure of how good the prediction models are at identifying actual faulty files. Recall measures the proportion of faulty files which are correctly identified as faulty. Recall is a measure of how many faulty files that the prediction model is predicted to find. The calculation of accuracy (equation 4.1), precision (equation 4.2) and recall (equation 4.3) makes use of the confusion matrix (fig 4.3) and is done by;

$$Accuracy = \frac{true\ positives\ +\ true\ negatives}{true\ positives + true\ negatives\ +\ false\ positives\ +\ false\ negatives} \tag{4.1}$$

$$Precision = \frac{true\ positives}{false\ positives + true\ positives} \tag{4.2}$$

$$Recall = \frac{true\ positives}{false\ negatives + true\ positives} \tag{4.3}$$

F-measure (equation 4.4) is a measure that combines both recall and precision and is calculated as

$$F - measure = 2 * \frac{precision * recall}{precision + recall} \tag{4.4}$$

By collecting these performance measurements, future predictions on unseen files can be estimated. The whole model building and evaluation process is further described in the pseudo code in Fig. 4.4.

Actual / Predicted	**Positive**	**Negative**
Positive	True Positive (TP)	False Positive (FP)
Negative	False Negative (FN)	True Negative (TN)

Figure 4.3: Confusion matrix

```
classifierType = {
    Naive bayes,
    Random Forest,
    J48,
    ...
}
10.times {
    randomize instance order
    prepare stratified 10-fold cross validation
    fold.each{
        get training set
        get testing set

        create feature selection set from training set
        record selected features

        classifierType.each{
            build classifier
            test classifier against testing set
            record results
        }
    }
}
```

Figure 4.4: Pseudo code describing the building and evaluation of the models

For each of the performance measurements, a matched pair Wilcoxon's signed rank test was conducted. This was done to demonstrate difference between the ML algorithms and the metric sets for each of the performance measurements. As the underlying data can not be guaranteed to meet the parametric assumptions required by a t-test, i.e. it is not guaranteed to be normally distributed, Wilcoxon signed-rank test was used [24]. Wilcoxon's signed rank test has been recommended for studies comparing different algorithms [15]. Due to the large amount of tests being performed, the significance level, α is set to 0.001 This is in line with other studies in the area as well [1, 2].

If there is a statistical significant difference between one algorithm compared to another one, using the same metric set, the "better" one wins. The algorithm that has the most wins for each metric set is considered the best algorithm for that metric set. When the best learning algorithm for each metric set has been determined, the performance of the different metric sets can be assessed.

Chapter 5

Results

This section reports the results from the experiment described in Chapter 3. Runtime performance-wise the collection and preprocessing of the metrics were rather quick. It took about 2 hours and about 8 hours respectively. The model building and evaluation however took about 4 days using a mid-range 2013 laptop with 8GB RAM and a 1.8Ghz Intel i5 processor.

The prediction performance is presented with the four different measures described in Section 4.3.3. First, the accuracy of the different prediction models on different feature selected metric sets is presented and evaluated. Thereafter, the results of f-measure, precision and recall is presented in respective order. This is done to determine the best performing ML algorithm to be used for determining the prediction performance of the different metric sets, that is presented after the ML algorithm evaluations. Tables and graphs of the performance of prediction models are presented and form the basis of the discussions in the subsections. In the tables and figures, the sets are abbreviated as *"src"*, *"tst"* and *"cmb"* which corresponds to the Srcs, Tsts and Cmbs respectively. The original full set of metrics for all source files (abbreviated as *"srcRaw"*) is also included in the graphs which hints of the repository's prediction performance when using ordinary metrics. As the underlying dataset differs from the others, i.e. it is based on a full set of files and not only those that are tested (described in Section 3.2), the results of that set can not be directly compared to the other results. It is included in the graphs as a visual cue of the repository's performance. These sets are explained in detail in Section 3.2.

The tables presented, Tables 5.6 to 5.9, present the outcome from Wilcoxon's signed rank tests [55]. Each combination of ML algorithm and metric set is tested against all other combinations resulting in 900 Wilcoxon's signed rank tests. SrcRaw is not included in these tables as srcRaw uses a different dataset and can not be compared to the other metric sets in this manner. A green cell indicates that the test reveals that the combination of ML algorithm and metric set perform significantly better than the combination tested against. A red cell indicates that the combination performs significantly worse while a white cell indicates a draw. In addition to the coloured cells, the mean value of the performance measurement studied is presented on each row. The best ML algorithm for each of the metric sets is thus the one with the most green cells in the table. The most predictive metric set is likewise the one with the most green

cells in the same table. Total scores for each of the subsets are summarized in the small subtable to the right of the main subtable in each table.

The graphs presented in the sections below, Figs. 5.1 to 5.4, gives visual indication of how the different metric sets perform together with the different ML algorithms. The graphs are box-and-whisker diagrams where the band in the box shows the median, the top and bottom of the box shows third and first quartiles, the whiskers shows minimum and maximum values and circles denote outliers. The box height shows the interquartile range. The results are grouped together with the other results using the same algorithm on the X axis. The keys on the X axis labels the different groups with the algorithm used in that group. The Y axis shows the prediction performance in the given measure in form of percentage (

Summaries with the best performing algorithms are also presented in Tables 5.1 to 5.4. Their mean values and standard deviations are also shown for each prediction performance measure. With this table, the performance of the best performing ML algorithms can quickly be concluded.

A comparison to other studies is also included in Section 5.5 where results from other studies in the domain are compared to those of this study.

Lastly, in Figs. 5.5 to 5.7, the results of the feature selection process along with the families of the selected features are presented, as these are the features of which the models are built on.

5.1 Accuracy

Accuracy is, as described in Section 4.3.3, the percentage of the correctly classified instances. Figure 5.1 shows the accuracy of the 10 different ML algorithms for the three different metric sets defined in 3.2, i.e Srcs, Tsts and Cmbs. The last ML algorithm in Fig. 5.1, ZeroR, is the baseline to test the accuracy of the other algorithms against as described in Section 4.3.1. ZeroR only predicts according to the majority class, on all instances. It reaches just short of 70% accuracy, which suggests that there are about 30% faulty files in the set. This is not at all surprising as the stratification process aims for an evenly distributed dataset. All of the other ML algorithms reach a higher accuracy on all of the provided metric sets. Most of the results are evenly distributed with a few exceptions which indicate that the median is a good indicator of the accuracy performance. There are a few outliers, especially for the Tsts, which indicates a wider spread in the results for that metric set.

Table 5.6 shows which of the ML algorithms that outperform each other for the different metric sets, in terms of accuracy as well as the mean accuracy (described in percentage) of the modeling techniques for the different metric sets. In Table 5.6, one can see that the models typically predict, except for ZeroR, 76% to 82% correct. The best ML algorithm for the different metric sets, based on accuracy, is determined by performing a Wilcoxon's signed rank test as described in Section 4.3.3. As well as which ML algorithm that performs the best, the metric set that yields the best accuracy

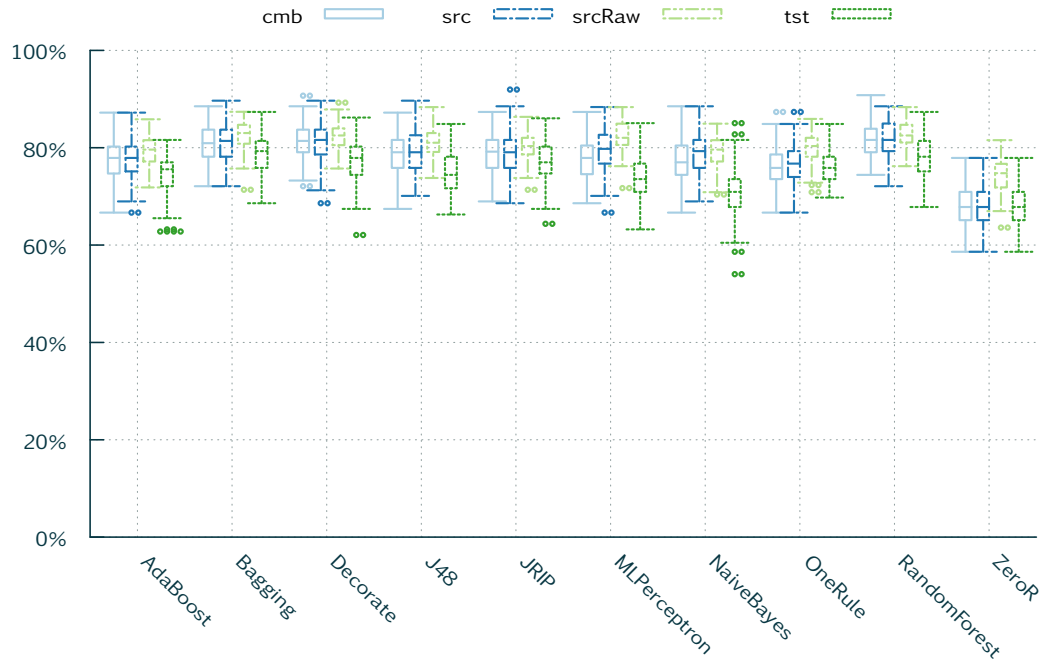

Figure 5.1: Accuracy for all metric sets and algorithms used in the experiment

can also be seen in the table 5.6.

For the accuracy, Random Forest with the Srcs wins and Random Forest with Cmbs come second with one more tie. There is no statistical significant difference between Random Forest for Srcs and Random Forest Cmbs. Overall, the Tsts performs quite badly compared to both the Srcs and Cmbs metric sets. It is not however Random Forest that perform best for the Tsts but Bagging which also performs quite well for the other sets as well. In summary, the Srcs generally scores better than both the Tsts and the Cmbs sets. When comparing the best performing algorithms for each set, shown in Table 5.1, the difference is minimal.

	Cmbs	Srcs	Tsts
Algorithm	Random Forest	Random Forest	Bagging
Mean accuracy	81.7%	81.8%	78.9%
Std. dev. (σ)	3.5%	3.8%	3.7%

Table 5.1: The best algorithms per set rated by accuracy

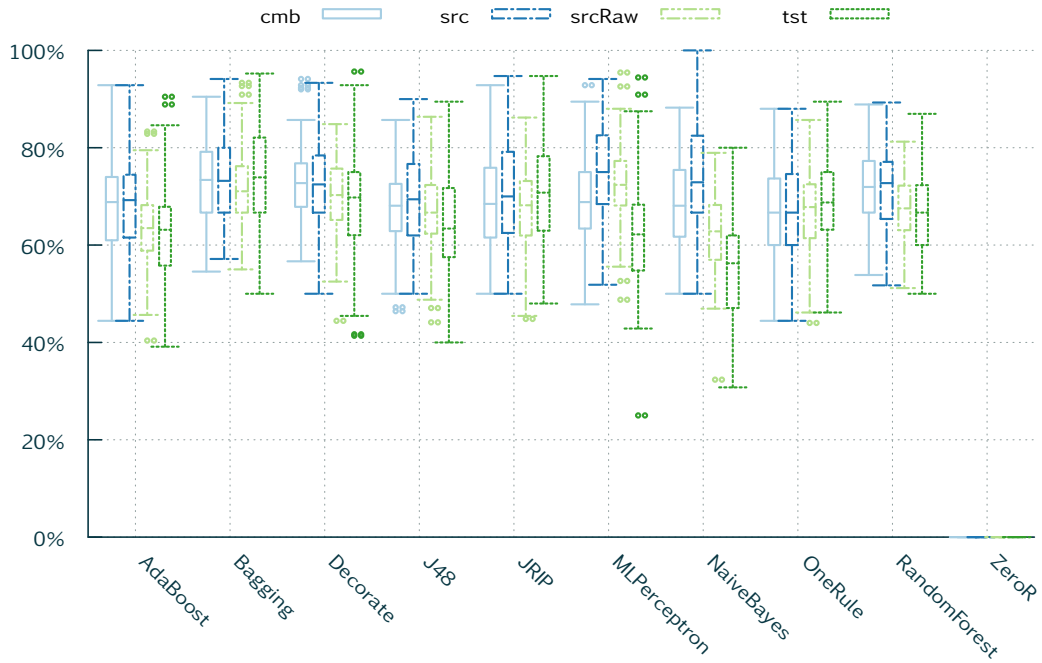

Figure 5.2: Precision for all metric sets and algorithms used in the experiment

5.2 Precision

Another performance evaluation measure, as described in Section 4.3.3 is precision. Precision is the measure of how well the prediction model classifies faulty files that actually are faulty. A table containing the mean values of the ML algorithms over the different metric sets together with a figure of how the algorithms perform against each other for different metric sets in terms of precision, are presented in Table 5.7 and Fig. 5.2. In Fig. 5.2 ZeroR stays at 0% precision as it predict all instances as not faulty and hence does not find any faulty files. As in accuracy, the Srcs performs best generally with the Cmbs coming in second and the Tsts last. For two algorithms, Bagging and JRip, the Tsts is the best performing metric set. The results are for the most part not skewed but the spread however is quite large spanning from around 40% to 90% and 50% to 100% in some cases. This can also be seen in Table 5.2 where the standard deviation is high for all of the three best performing algorithms.

Table 5.7 shows the individual scores for the combination of metric sets and ML algorithms. In the case of precision, Bagging performs best in the Cmbs and the Tsts. In the Srcs, both Multilayer Perceptron and NaiveBayes scores better, however. Bagging together with the Tsts outperforms the other algorithms for the Tsts by margin, though. In summary, the Srcs scores significantly better than both the Tsts and the Cmbs in general. When looking at Table 5.2 the difference between the best performing algorithms for the different sets is not that big, though. Surprisingly, the best algorithm for Tsts performs better than the best algorithm for Cmbs.

	Cmbs	Srcs	Tsts
Algorithm	Bagging	Multilayer Perceptron	Bagging
Mean precision	73.1%	74.7%	74.1%
Std. dev. (σ)	8.5%	9.4%	10.1%

Table 5.2: The best algorithms per set rated by precision

5.3 Recall

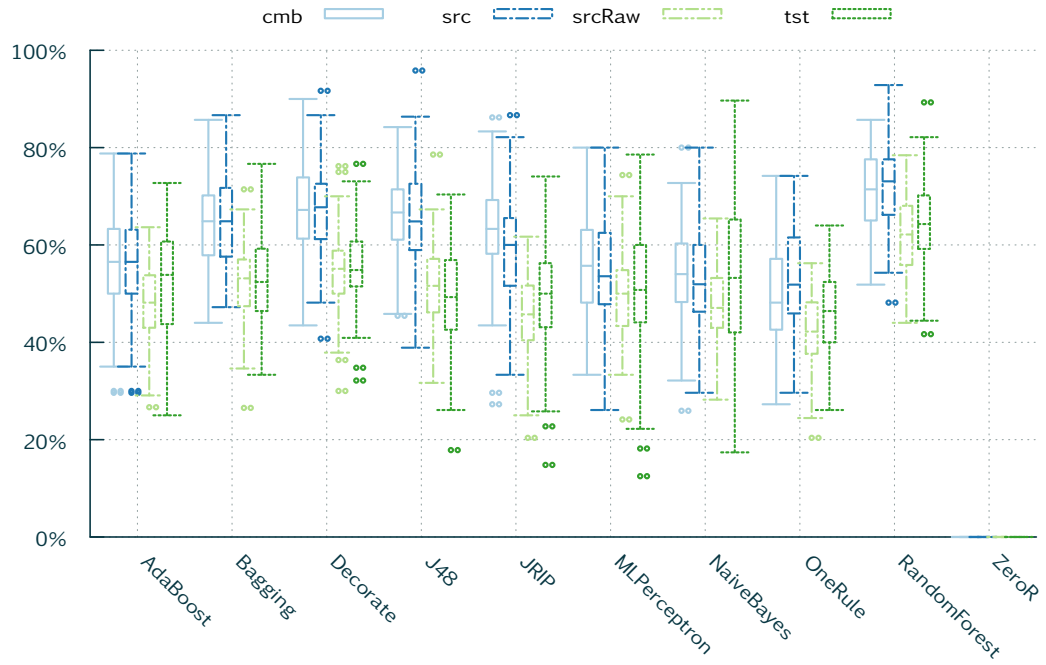

Figure 5.3: Recall for all metric sets and algorithms used in the experiment

A third performance evaluation measure, as described in Section 4.3.3 is recall. Recall measures how many of the faulty files the prediction model finds. Of the same reason presented in Section 5.2 ZeroR stays at 0% recall. Figure 5.3 shows the precision of the ML algorithms over the different metric sets. Table 5.8 contains the mean values of the ML algorithms over the different metric sets together with a figure of how the algorithms perform against each other for different metric sets in terms of recall.

In Fig. 5.3 one can clearly see that the spread is even larger than it is for the precision, over 70% for Naive Bayes using the Tsts, and that the amount of outliers is large as well. The spread is however lower for the Cmbs in most cases when compared to the other sets. As in the other prediction performance measures, the results are not especially skewed. One distinctive feature is that the median is generally higher or about the same for the Cmbs when compared to the Srcs.

The high recall for the Cmbs can also be observed in Table 5.8 where the Cmbs has a higher number of wins and lower number of loses when compared to the Srcs, although with a small margin. In 5.3 one can see that Random Forest with the Srcs set still has the highest recall though with 72.09% with a quite big gap to the other metric sets.

	Cmbs	Srcs	Tsts
Algorithm	Random Forest	Random Forest	Random Forest
Mean recall	70.7%	72.1%	64.6%
Std. dev. (σ)	8.1%	8.5%	8.5%

Table 5.3: The best algorithms per set rated by recall

5.4 F-measure

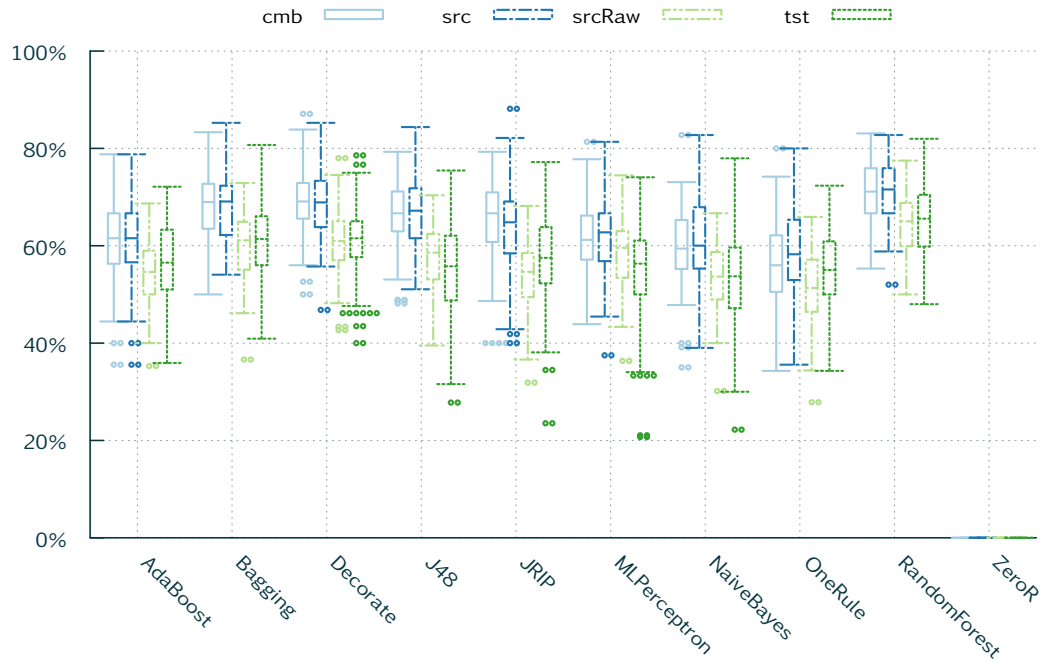

Figure 5.4: F-measure for all metric sets and algorithms used in the experiment

The fourth and last performance evaluation measure, as described in Section 4.3.3, is f-measure. F-measure is a combination of both precision and recall. Of the same reason as presented in previous sections, ZeroR stays at 0% f-measure.

Figure 5.4 shows the f-measure of the ML algorithms over the different metric sets. Table 5.9 contains the mean values of the ML algorithms for the different metric sets

together with a figure of how the algorithms perform against each other for different metric sets in terms of f-measure.

Figure 5.4 shows that the Cmbs and Srcs results generally have medians very close to each other. This can also be seen in 5.9 were the Srcs and Tsts results often results in a tie rather than a win or loss. For some algorithms, the Srcs outperform the Cmbs, though, which can be seen in Table 5.9 where the Srcs has more wins and fewer losses compared to the Cmbs.

	Cmbs	Srcs	Tsts
Algorithm	Random Forest	Random Forest	Random Forest
Mean f-measure	70.9%	71.5%	65.2%
Std. dev. (σ)	6.3%	6.3%	7.1%

Table 5.4: The best algorithms per set rated by f-measure

5.5 Comparison to other studies

To ease the comparison to other studies, the mean results for all sets and all performance measures are compiled in Table 5.5. These values are the ones used in the comparison in this section.

As accuracy is dependent on the balance of the underlying dataset it is hard to compare the accuracy results of this study with others. This measure is therefore not widely discussed in other studies but is a comprehensible measure and is therefore presented in this study.

Hall et al. has in their systematic literature review gathered results from 208 fault prediction studies [22]. Among those, 48 studies deal with fault prediction on the file level. These studies are compared with precision, recall and f-measure. As this study is conducted using the same performance measures as Hall et al. and as they summarize many studies, the results of this study are compared to the ones compiled by Hall et al.

When the precision is compared to other studies, Srcs and Cmbs performs in general above the median, while the Tsts performs slightly below the median. For recall,

	Cmbs	Srcs	Tsts
Accuracy	78.9%	79.4%	75.8%
Precision	69.7%	71.4%	66.1%
Recall	60.8%	60.5%	53.0%
F-measure	64.3%	64.7%	57.8%

Table 5.5: Mean prediction performance of the metric sets

the Cmbs and Srcs performs above the median while the Tsts on the other hand performs below the median. The results for the f-measure are in general slightly better than for other studies where the Cmbs and Srcs performs well above the median and the Tsts performs around the median.

Overall, the algorithms in our study performs around or above the average when compared to other studies [22]. The exceptions being Naïve Bayes with a low recall around lower quartile and Random Forest which performs as the best studies regarding f-measure.

When comparing the study to other studies in the systematic literature review by Hall et al. [22], the results are generally good. When comparing the results to those of studies using combinations of metric families, only the ones including source code text perform better.

Arisholm et al. has performed studies with a telecom company with propitiatory [1, 2]. They also investigated a Java system, used and compared different metric families, different ML algorithms and used feature selection which makes a comparison between the results interesting. As noted earlier, Arisholm et al. has an accuracy of over 93% in their best case which is way above the results of this study [2]. The mean result of this study is around 70% for the Cmbs compared to 10.4% for the best metric set for Arisholm et al [2]. The difference is not as large for the recall where this study has an average of 64% for recall for Cmbs compared to 62% at best for Arisholm et al. [2].

Another study using feature selection was conducted by Shivaji et al. [49] where they studied a series of open source projects. Their results are consistently better than the ones of this study; they have an accuracy of 91%, precision of 96%, recall of 67% and f-measure of 79%.

5.6 Feature selection results

The feature selection process, described in Section 4.3.2, selected a subset of metrics for the three different metric sets, Srcs, Tsts and Cmbs. The feature selection process selected between 25 and 35 features each run. As the feature selection process was done in each fold of the evaluation process, different subsets were picked for different folds. The proportion of the selected metric families is shown in Figs. 5.5 to 5.7.

The feature selection process, described in Section 4.3.2, selected a subset of metrics for the three different metrics sets, Srcs, Tsts and Cmbs. The feature selection process selected between 25 and 35 features each run. As the feature selection process was done in each fold of the evaluation process, different subsets were picked for different folds. The proportion of the selected metric families is shown in Figs. 5.5 to 5.7.

In general, the subsets created by feature selection largely consist of dependency metrics. Author metrics was not selected at all. There are however differences regarding the other metrics families among the metric sets. OO metrics are the second largest part of the Srcs, but the smallest of the Tsts, where STREW-J took its place. The Cmbs

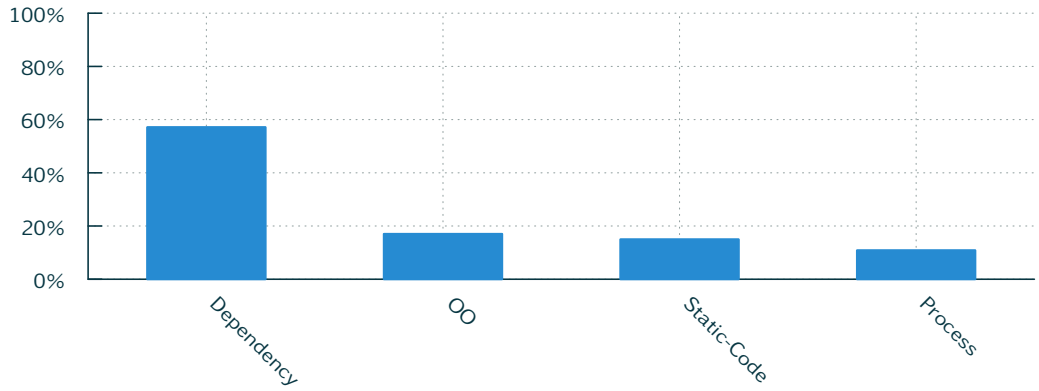

Figure 5.5: Proportion of the selected metrics for the Srcs

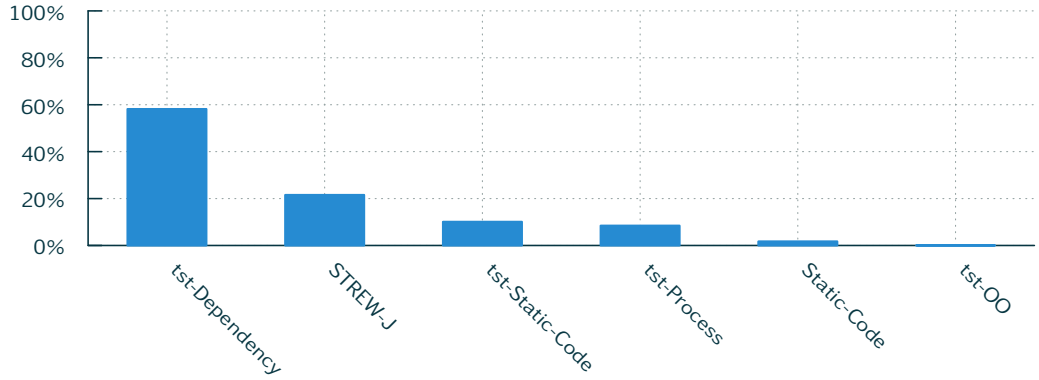

Figure 5.6: Proportion of the selected metrics for the Tsts

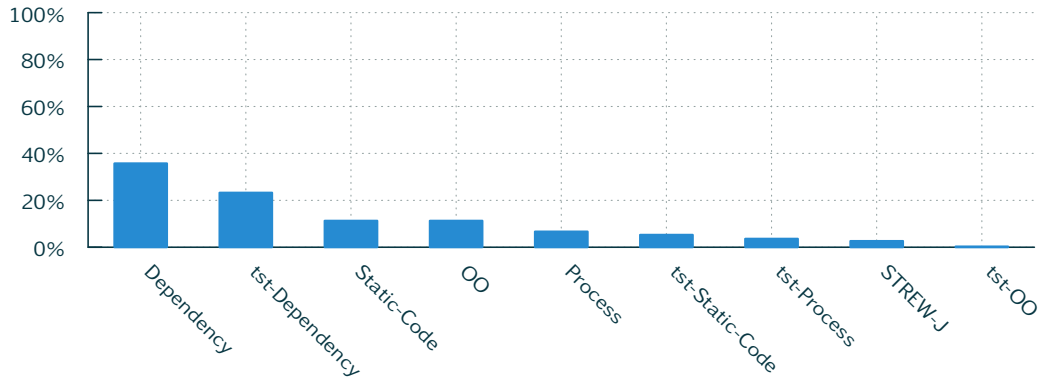

Figure 5.7: Proportion of the selected metrics for the Cmbs

is in general a reflection of the results of the Srcs and Tsts, were the results from the Srcs take precedence, with the exception of dependencies from tests, which are the second largest part.

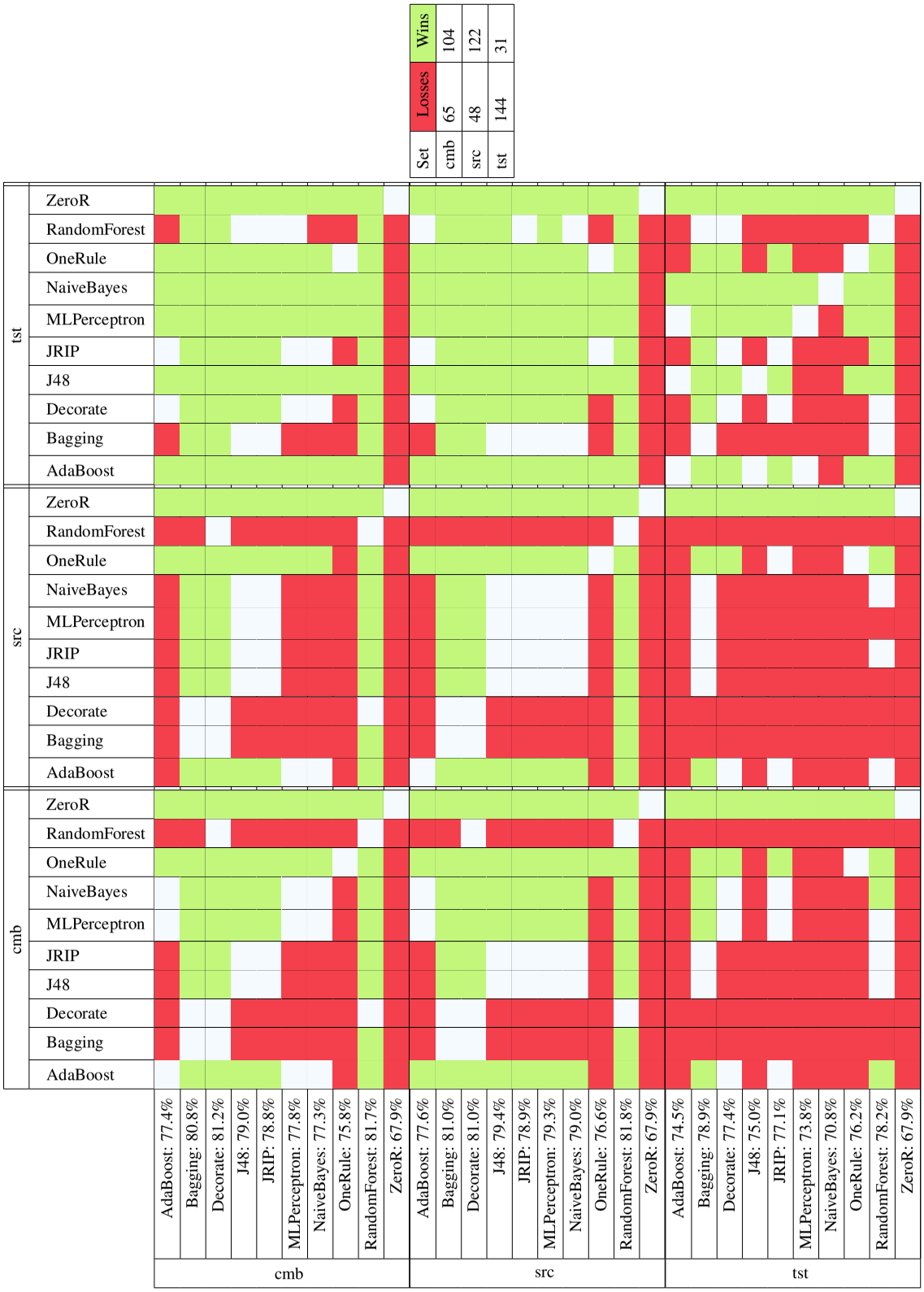

Table 5.6: Table showing results from Wilcoxon's signed rank test on accuracy for ML algorithms and metric sets

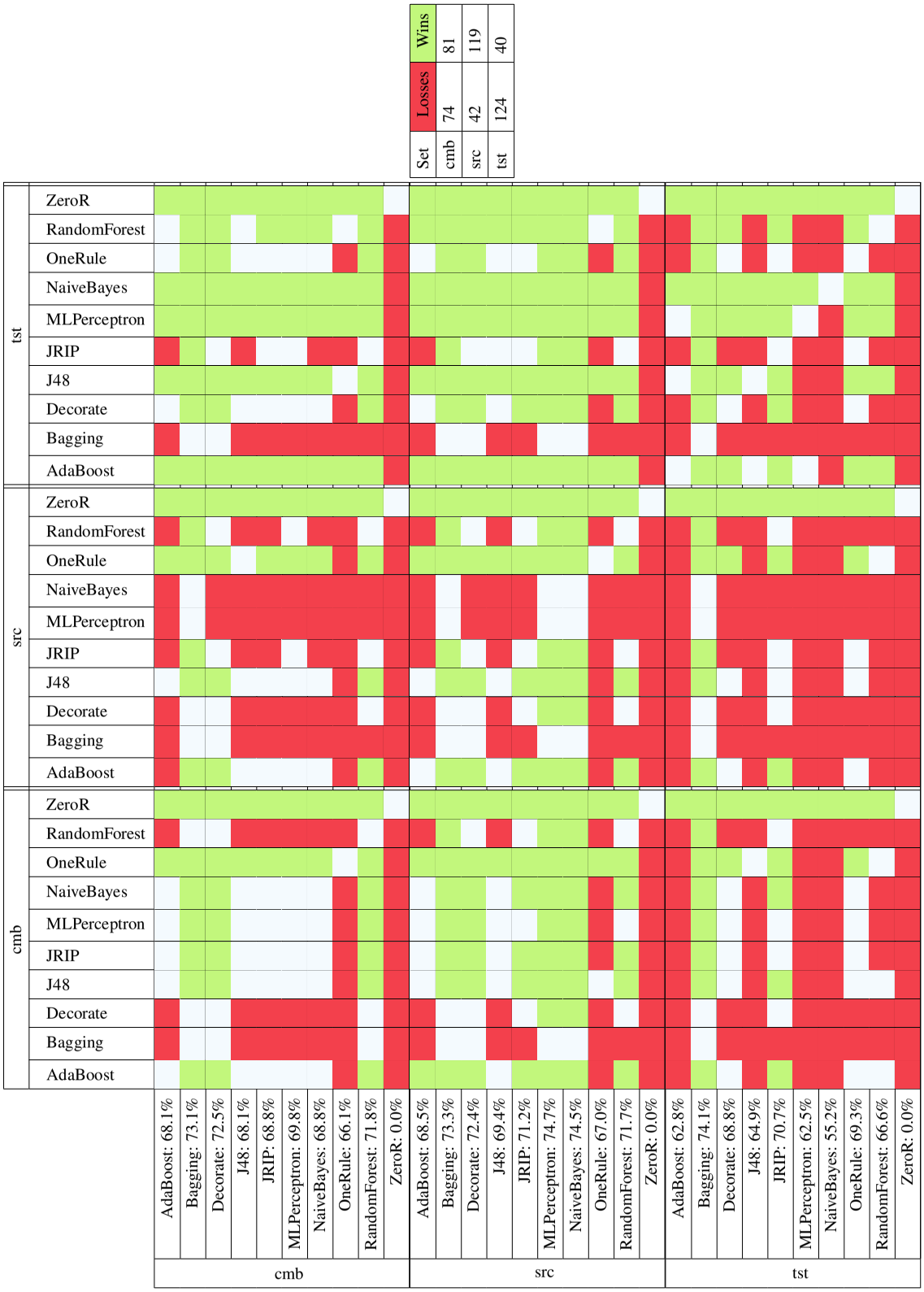

Table 5.7: Table showing results from Wilcoxon's signed rank test on precision for ML algorithms and metric sets

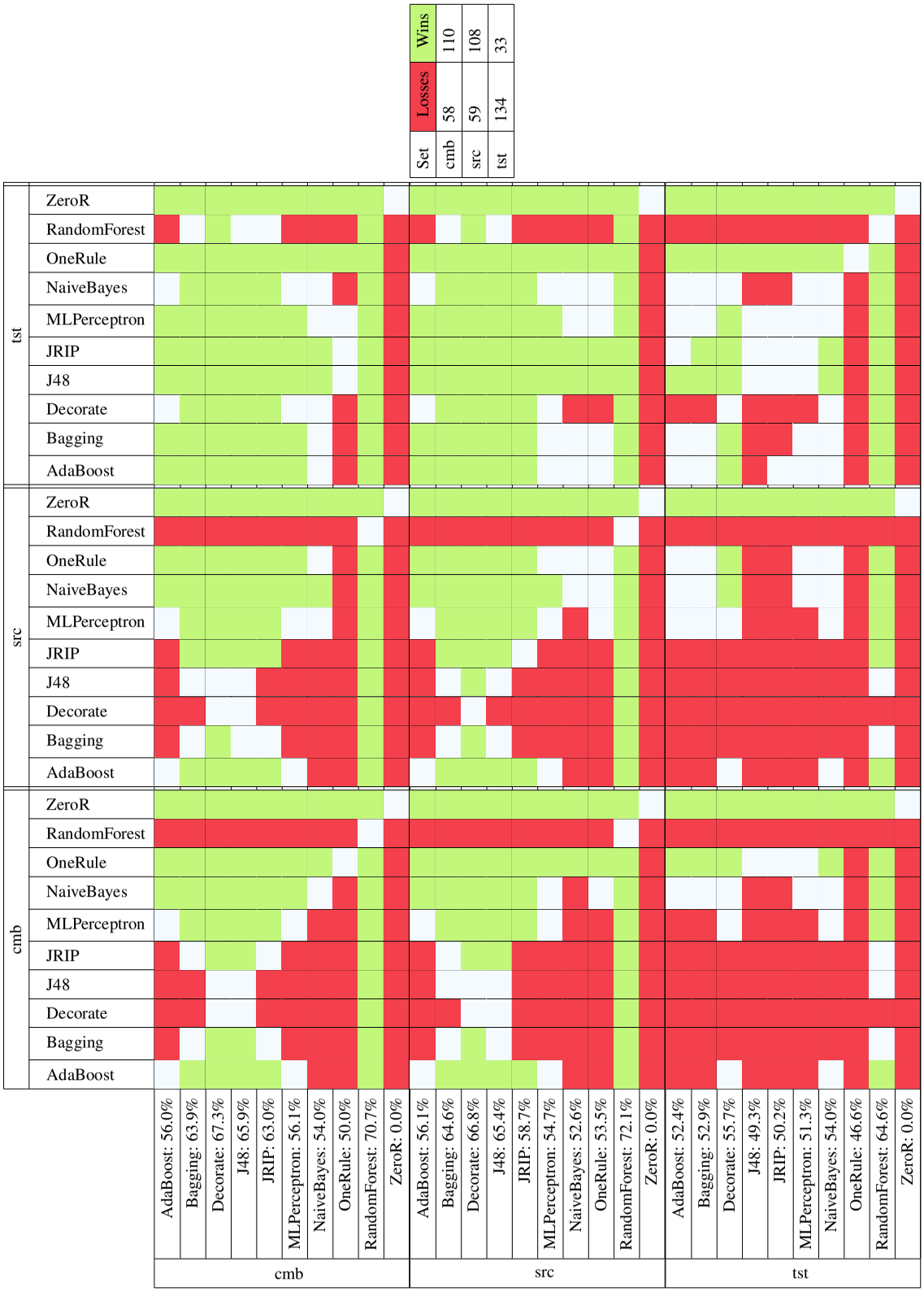

Table 5.8: Table showing results from Wilcoxon's signed rank test on recall for ML algorithms and metric sets

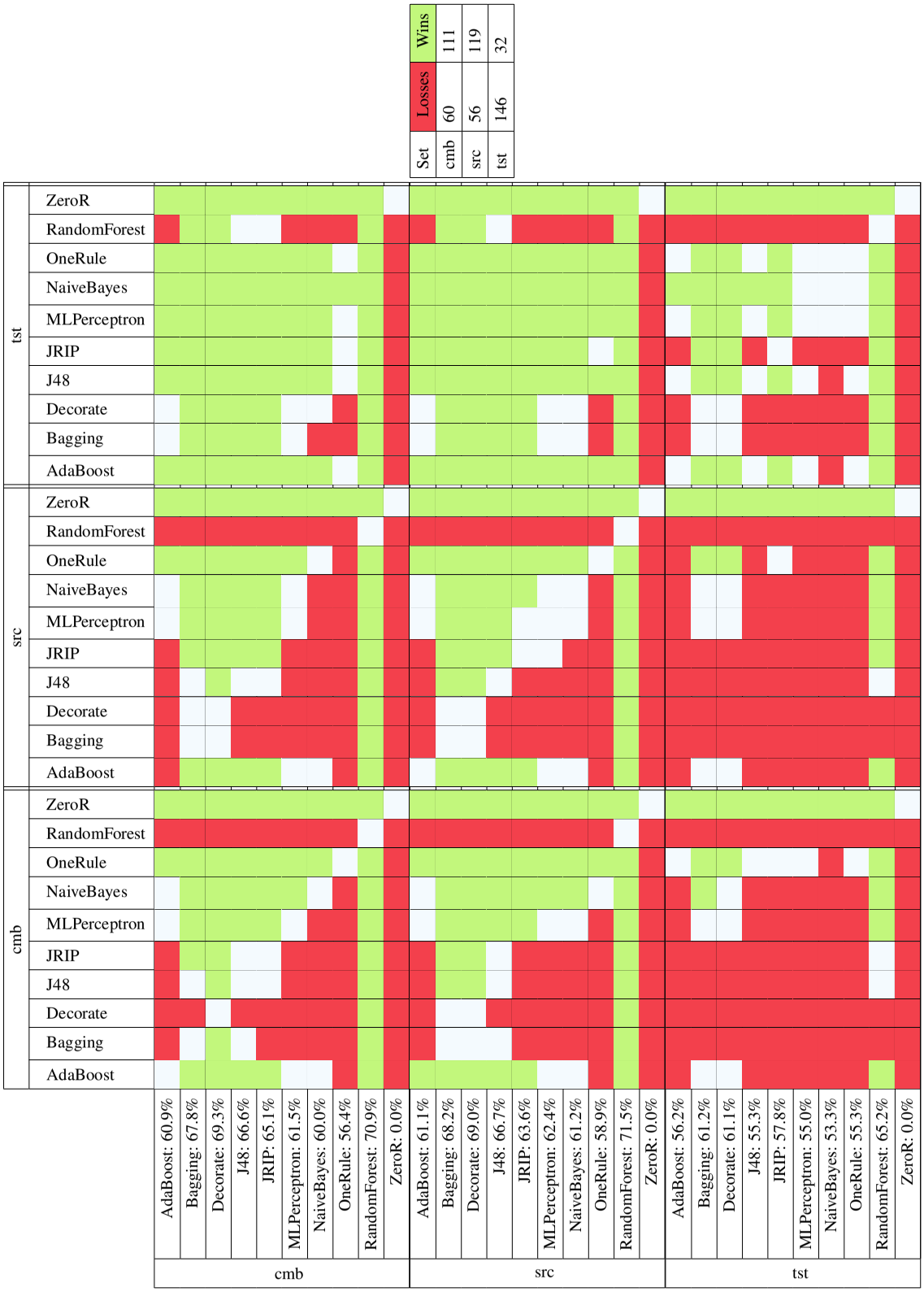

Table 5.9: Table showing results from Wilcoxon's signed rank test on f-measure for ML algorithms and metric sets

Chapter 6

Analysis and discussion

When looking at all algorithms the general result is that Srcs performs best. In some instances, e.g. for recall, Cmbs performs better. At the individual results level, when looking at results from the metric sets and algorithms, the general result is that Random Forest works best throughout all metric sets. However, different metric sets and algorithms may work better in some circumstances. These results are discussed in more detail in the following sections.

6.1 Metric sets results

Generally, the Srcs perform best throughout the performance measures. The only measure where Srcs is not best is in recall where Cmbs in most cases is as good as Scrs and for a few ML algorithms a little better than Srcs. As the recall denotes how good the metric set together with the algorithm is at capturing faulty files, the lower precision for the Cmbs may not come as a surprise. As it is capturing more faulty files, a natural consequence may be that it also catches more unfaulty files and therefore has a lower precision. This is seen in the f-measure, the combination of recall and precision, where the Cmbs and Srcs perform very close to each other as the results from the precision and recall is cancelling each other out. When looking at the f-measure, one can see that the Cmbs has a lower spread, though. The Tsts fares worse overall in all measures compared to Srcs and Cmbs, but not by a very large margin. In general, one may conclude that when choosing between the Cmbs and the Srcs, the preferred prediction performance measure must be taken into account. If recall is of high importance, the Cmbs is the metric set of choice. If precision or accuracy is most important, the Srcs is the metric set to choose.

The difference in prediction performance between Cmbs and Srcs could be due to the feature selection process. Using a feature selected subset, can in some cases result in lower prediction performance [45].

SrcRaw is the metric set that would have been used in case the tests were not included in the experiment and is therefore included as a reference. It generally performs worse compared to the other sets and all of the performance measures are lower than the Srcs, Tsts and Cmbs scores. One interesting aspect of the results is that the Tsts performs better than srcRaw. This clearly shows that models built with metrics from

the tests, that does not have any information about the source file whatsoever, still predicts better than model built with metrics from the source code itself. However, this only holds true for tested source code file for the Tsts and all source code files for the srcRaw set. It must also be noted that if the same underlying dataset is used, source metrics still predicts better. This could be an indicator that tested code is easier to predict. This would explain the big difference between the Srcs and srcRaw which both share the same metric sets but use different data sets.

6.2 ML algorithm results

The best performing algorithm overall is Random Forest which is clearly seen in the top three algorithms tables in 5. Generally, the best performing metric set for the algorithms is the Srcs. There are however some cases where the best set is the Tsts for certain algorithms, e.g. Bagging and JRip in precision. One of the benefits of JRip is the ease of which rules can be extracted and interpreted from a model built with it. The JRip result is therefore especially interesting for Ericsson, as they are interested in models that they can understand and interpret easily.

The Tsts set does not have as clear results as the other metric sets. Bagging comes up as one of the best algorithms for the Tsts multiple times but has especially bad recall. In turn, the precision is high, however. This is also true for the Multilayer Perceptron algorithm for the Srcs for which recall is low and precision is high. In the end Random Forest is still the winner which the f-measure shows for both of Srsc and Tsts.

6.3 Comparison to other studies

The comparison in Section 5.5 showed that the results of this study are comparable to those of other studies. It also showed that the Tsts set is comparable with slightly lower results when compared to other studies. When looking at feature selection studies it is hard to position this study as results differ largely between them. This study does however perform well when compared to Arisholm et al. which also uses a proprietary dataset. Arisholm et al. e.g. had a highly unbalanced data set with only 0.5-2% faulty files indicating a balance of 98%-2% which could explain their low results in other measures than accuracy [2, 22].

6.4 Hypothesis testing results

To test the null hypothesis *"Source code metrics together with test metrics provide equal prediction performance as SDP using only source code metrics"*, a chi-squared (χ^2) test was conducted. The wins, draws and losses for Srcs and Cmbs against each other, for each of the performance measures, were calculated and used as input to the χ^2 test. The significance level was set to $\alpha = 0.05$, indicating a 95% confidence

interval. For accuracy, recall and f-measure, the null hypothesis could not be rejected, while for precision the null hypothesis could in fact be rejected, indicating that there is in fact a difference in prediction performance. By performing a manual inspection of the wins, draw and losses for prediction, it is clear that the Srcs outperforms Cmbs. This means that the hypothesis *"Source code metrics together with test metrics provide better prediction performance than SDP using only source code metrics"*, stated in Section 3.2.2, does not hold.

6.5 Answers to research questions 2 and 3

The research questions associated with the literature study were answered in the method chapter, Section 3.1.5, and they are therefore not discussed further in this section. The two last questions, however, were answered through the experiment and are discussed below.

RQ2 - Which ML algorithm(s) provide the best SDP performance?

As seen in the discussion above, Random Forest generally gives the best predictive performance for all the metric sets, with a few exceptions. Notably precision for all metric sets, and accuracy for Bagging in conjunction with the Tsts. As there is no overall winner, one has to choose a combination of metric set and algorithm that suit the needs. In general, one may conclude that Random Forest is suitable in most cases.

6.6 Answers to research questions 4 and 5

The research questions associated with the literature study were answered in the method chapter, Section 3.1.5, and they are therefore not discussed further in this section. The two last questions, however, were answered through the experiment and are discussed below.

RQ4 – Which ML algorithm(s) provide the best SDP performance?

As seen in the discussion above, Random Forest generally gives the best predictive performance for all the metric sets, with a few exceptions. Notably precision for all metric sets, and accuracy for Bagging in conjunction with the Tsts. As there is no overall winner, one has to choose a combination of metric set and algorithm that suit the needs. In general, one may conclude that Random Forest is suitable in most cases.

RQ5 – To what extent do test metrics impact the performance of SDP?

For most algorithms, recall is as high, or slightly higher for the Cmbs when compared to the Srcs metric set. Regarding the other performance measures, the Cmbs achieve

slightly worse results compared to the Srcs. This results in an f-measure with a comparable median value for the Cmbs when compared to the Srcs, but with a lower variance. One could therefore expect results closer to median value for future predictions. When looking only at Tsts, the performance is generally worse than for the other sets, with a few exceptions. In conclusion, test metrics does not affect the SDP performance much but can give results closer to the estimated results due to its lower variance.

6.7 What the results means to Ericsson

From Ericsson's perspective, the result of JRip in combination with Cmbs shows the most promising results as, in addition to fault predictions, rules can be extracted which can be used throughout the software development process.

The results produced by the tool developed through the experiment implementation can be used in a numerous ways. First off, early estimation of what files that are more fault prone can be predicted with good prediction performance. As described in Chapter 1, finding faults in late stages of the development, or even in the released product is expensive. The results can therefore be used to find faults before they are reported through a TR by doing a code review on files predicted as faulty. If a fault is discovered through code review of files predicted as faulty, the cost of correcting the faults is reduced. Furthermore, the time used to debug a fault reported through a TR to a certain file is eliminated if a fault is discovered through code inspection.

When using a rule based learner, the results from the tool can be used in refactoring[1]. The inspected file's metrics can be analyzed against the rules to see why the file is predicted as faulty. This information can then be used to refactor the file, e.g. split the file into smaller, cohesive units if the rule states that files with high complexity and lack of cohesion are predicted as faulty.

Rules regarding dependencies can also be used in the refactoring process. If a rule states that a dependency causes files to be faulty, this might indicate that the dependency's external interface needs refactoring. Another consideration may be to refactor files using a dependency and remove the dependency where possible. This would limit the number of files using the dependency and its external interface.

The testing might also be improved as the testing effort can focus on files which have been predicted as faulty. Extracted rules predicting on dependencies may also be used where tests of files using the dependency, even if not predicted faulty, can be improved.

Another avenue is software development process improvements. Changes in e.g. team size or configuration management[2] can be evaluated in an early stage by analyzing changes in the predicted amount of faulty files. Again, analyzing changes in the

[1] *"A change made to the internal structure of the software to make it easier to understand and cheaper to modify without changing its oberservable behavior"* [19, p. 53]

[2] *"The practice of identifying project artifacts and handling changes systematically so that a s system can maintain its integrity over time"* [29, p. 664]

resulting rules when using a rule based learner might also give hints of how changes in the development process affect the source code and its tests. The tool is not limited to predicting faults but can also be used to predict on any metric. The characteristics of files with high complexity may, for example, be further examined when using complexity as the dependent variable instead of fault history.

The results of the defect prediction might also be used to prioritize tests and decide which tests to run when all tests can not be run due to execution time constraints. In this case, the tests to the files predicted as faulty, can be prioritized so that they are run. Resulting rules containing dependencies from rule based learners may also be used. Tests of files having dependencies that often causes other files to fail, can be prioritized even if that file is not predicted as faulty.

When using a classifier where metrics from a built model can be extracted, integrity issues can emerge. Author metrics may be extracted and e.g. used for ill-intentioned purposes. If the model is publicized, dependency metrics may be used to gain information about the repository's structure. Due to this, the tool supports turning off certain metric families.

When applying the tool in a production setting the model does not have to be rebuilt 100 times (10 folds, 10 times) as in the case of this study. The 4 days stated in Chapter 5 is therefore not a measure of the performance of such a tool in a production environment, rather 1 hour as only a single model needs to be built. Furthermore, a too, such as this only has to be rerun to rebuild the model after a substantial change in the codebase, e.g. once every release. The runtime of the tool is therefore not a problem in a production setting.

6.8 Lessons learned

The memory model first used in the experiment implementation was memory consuming. The usage of hash maps to record metrics, albeit both simple and flexible, turned out to use 10 times more memory than necessary. They were however simplifying implementation to a large extent as metrics could be easily be added during runtime. Metrics could therefore be created on the fly based on the repository data. This feature was used extensively with the dependency and author metrics. The memory usage became a problem when the dependency and author metrics were used on repositories larger than 20 000 LOC though where the number of metrics were for some repositories as large as 7 000. In these cases, the memory consumption exceeded 8GB which was the amount of installed RAM of the computers used for the experiment. This was due to the number of dependency and author metrics being based on the amount of authors and dependencies within the repository.

The hash maps were replaced with a backend using SoA[3] hosting several frontends implementing Java's `Map` interface. The transition from a hash map based memory lay-

[3] Structure of Arrays. see [13, p.4-21]

out to a SoA one was therefore smooth as the Java `HashMap` could simply be swapped to our own implementation. The lesson one may draw from this is that the memory layout is of importance when using metrics generated per repository as it can grow large quite fast and being able to accommodate for this is important.

The first time results were produced by the experiment, they were extremely good, with an accuracy, precision and recall of above 95% for several ML algorithms and the Cmbs. This result were not in line with other studies and the experiment design were therefore reviewed to make sure that the results were fair. This result turned out to be too good to be true since the feature selection was conducted in a way that made the feature selection process using information from the test set when reducing features. This skewed the results. This problem have been present in former studies using feature selection [45, 51]. This problem was mitigated by performing feature selection on the training set only as proposed by Smialowski et al. [51].

After discussing the problem with an engineer it was revealed that some developers constructs tests after a TR has been reported to stop similar faults in the future. The mere presence of a test could therefore imply that the file has had faults in the past. This problem was mitigated by applying a pivot point as described in Section 3.2. The lesson learned by this is to be critical towards the experiment design if results are not in line with state of the art research. There may also be processes among the developers which are not official policy that may bias the results. It is therefore of importance to talk to developers and engineers so that the results of the experiment are not affected by processes such as the one mentioned above.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

The goal of the thesis was to investigate whether a test, coupled to a source code file, contained enough information to enhance the SDP performance if metrics from both the source file and test file are combined. To investigate this a literature review followed by an experiment was conducted. The literature review helped us identify of what kind of metrics and ML algorithms to use in the experiment, while the experiment ultimately helped us test the hypotheses.

The results of this study show that the use of Cmbs instead of Srcs does not provide better prediction performance. In fact, in the case of precision, Srcs performs significantly better compared to Cmbs. However, the Cmbs produces less fluctuating results in most cases, indicating that the mean value of the evaluated prediction models are closer to the real result, i.e. the prediction on unseen files, when using the Cmbs. This is visualized in the box-whisker plots in Chapter 5, where the Cmbs boxes (indicating 50% of the results) are smaller in most cases, than the corresponding boxes for the Srcs.

In general, the results of our study are comparable, and in some cases better, compared to other similar SDP studies.

For Ericsson, this means that faults can be predicted on the file level with good prediction performance. Perhaps the most promising result is that of the combination of Cmbs and JRip, which makes the prediction model easy to interpret. Rules extracted from the model can then e.g. be used for identification of areas in need of refactoring. Conclusions drawn from these rules can also be used for e.g. process improvements.

If all of the source code files were used in the metric extraction, as they would have been if test metrics was not included in the prediction, the Srcs models would perform considerably worse. This is obvious when looking at the srcRaw dataset in the figures in Chapter 5. This shows that files with tested code, in this repository, are more likely to be predicted correctly.

One of the most interesting results is the fact that the Tsts predicts so well having minimal information about the source code file. This clearly demonstrates that test and source code are tightly linked which was one of the main reasons that we thought that combining test and source code metrics was a viable approach.

One must however remark that the results of this study come from a single repository and they can therefore not be claimed to be true for all source code. One can however argue that they likely are true for similar repositories.

7.2 Future work

As this master thesis is about determining the performance of the metric sets, only the default settings in the Weka environment were evaluated. As a future work, it would be interesting to investigate how the prediction performance of different ML algorithms could shift the results if the settings were tweaked. Hall et al. concluded that studies with good prediction performance often have optimized their ML algorithm settings [22]. It would be especially interesting to tweak the meta-learners used in this master thesis, i.e. AdaBoost, Bagging and Decorate, and use different modeling techniques together with them as they are dependent on other ML algorithms to make predictions.

Fault Slip Through (FST), introduced by Damm et al. [14], is a measure of the costs associated with not finding faults in the right testing phase. FST is about "to find the right faults in the right phase" [16] of the development process. This is interesting because not all faults can be or, perhaps should not be found with unit tests, but rather with e.g. integration testing, function tests or system tests. FST could be integrated into the prediction models, where only the faults in files that could have been found with unit tests, i.e. errors in the source code, are the files that are considered as faulty. It would also be interesting to investigate how a FST measure, telling the model when it is most cost-effective to find the fault, could affect the predictions. There are however obstacles regarding the integration of FST. For example, it has to be determined in a structured way, exactly in which phase each fault should have been found which can prove to be a difficult task as the bug reporting process has to be expanded. The tester or developer filing the bug report also has to guess where the bug should have been detected which also introduces the human factor.

One way of continuing the work of this master thesis is to include additional tests other than unit test, e.g. system tests and functional tests. The concept of FST could then also be utilized even further.

The term Change Bursts, described by Nagappan et al. [40] as code modules that are frequently altered. By detecting these modules, they gained a precision and recall of 91% and 92%, respectively. Integration of this metric can therefore be recommended as an addition to this master thesis. By using change bursts metric, the predictive power of the models in this study could possibly be improved.

The results from this study could be validated by repeating the study for more repositories from several other departments, conducting the same experiment on them and then comparing the results to those of this study. By doing this, the results presented in this master thesis can be compared to results from repositories in related domains. The results from this comparison can then be used to state whether the re-

sults from this study holds true for other repositories and thereby also if the results hold true in general.

Another way to build on this work would be to conduct a case study to observe the impact that the tool suite has in real settings in production. The real cost saving potential that the suite can pose to Ericsson could then be evaluated. The impact and cost savings of this tool could e.g. be measured with FST.

The tool suite could also benefit from being tested against metrics from publicly available data corpus, such as the Eclipse and NASA datasets. This would possibly be the only fair way to evaluate the prediction performance of the tool against other studies results since the results are highly dependent on the underlying data, and that it is a delicate problem to generalize between two different projects.

List of abbreviations

BLOC	Blank LOC
CA	Afferent couplings
CBO	Coupling Between Objects
CCN	Cyclomatic Complexity Number
CFS	Correlation based Feature Selection
CK	Chidamber-Kemerer
CLOC	Comment LOC
Cmbs	Combined metric set
DIT	Depth of Inheterence Tree
FST	Fault Slip Through
LCOM	Lack of Cohesion in Methods
LOC	Lines Of Code
LRP	Literature Review Protocol
ML	Machine Learning
NOC	Number Of Children
NPM	Number of Public Methods
OO	Object-Oriented
PD	Probability of Detection
PF	Probability of False alarms
RFC	Response For Class
SCM	Source Code Management system
SDP	Software Defect Prediction
SLOC	Source LOC
SLOC-L	Logical SLOC
SLOC-P	Physical SLOC
SLR	Systematic Literature Review
STREW	Software Testing and Reliability Early Warning
STREW-J	STREW for Java
Srcs	Source code metric set
TR	Trouble Report
Tsts	Test source code metric set
UCC	Unified Code Count
WMC	Weighted Methods per Class
srcRaw	The original full set of metrics for all source files

References

[1] Erik Arisholm, Lionel C. Briand, and Magnus Fuglerud. Data Mining Techniques for Building Fault-proneness Models in Telecom Java Software. In *The 18th IEEE International Symposium on Software Reliability (ISSRE '07)*, pages 215–224. IEEE, November 2007.

[2] Erik Arisholm, Lionel C. Briand, and Eivind 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, January 2010.

[3] S Bibi, G Tsoumakas, I Stamelos, and I. Vlahvas. Software Defect Prediction Using Regression via Classification. In *IEEE International Conference on Computer Systems and Applications, 2006.*, pages 330–336. IEEE, 2006.

[4] Barry W. Boehm. *Software Engineering Economics*. Prentice Hall; 1 edition, 1981.

[5] Leo Breiman. Bagging predictors. *Machine learning*, 140:123–140, 1996.

[6] Bora Caglayan, Ayse Bener, and Stefan Koch. Merits of using repository metrics in defect prediction for open source projects. In *2009 ICSE Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development*, pages 31–36. IEEE, May 2009.

[7] Cagatay Catal. Software fault prediction: A literature review and current trends. *Expert Systems with Applications*, 38(4):4626–4636, April 2011.

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

[9] Cagatay Catal and Banu Diri. Investigating the effect of dataset size, metrics sets, and feature selection techniques on software fault prediction problem. *Information Sciences*, 179(8):1040–1058, March 2009.

[10] S.R. Chidamber and C.F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

[11] William W. Cohen. Fast Effective Rule Induction. In *In Proceedings of the Twelfth International Conference on Machine Learning*, pages 115—-123, 1995.

[12] James S. Collofello and Scott N. Woodfield. Evaluating the effectiveness of reliability-assurance techniques. *Journal of Systems and Software*, 9(3):191–195, March 1989.

[13] Intel Corporation. Intel 64 and IA-32 Architectures Optimization Reference Manual. Technical Report March, 2014.

[14] Lars-Ola Damm, Lars Lundberg, and Claes Wohlin. Faults-slip-through—a concept for measuring the efficiency of the test process. *Software Process: Improvement and Practice*, 11(1):47–59, January 2006.

[15] Janez Dem\v{s}ar. Statistical Comparisons of Classifiers over Multiple Data Sets. *Journal ofMachine Learning Research*, 7(12/1/2006):1–30, 2006.

[16] Denis Duka and Lovre Hribar. Fault Slip Through Measurement in Software Development Process. In *ELMAR, 2010 PROCEEDINGS*, number September, pages 177—-182. IEEE, 2010.

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

[18] Norman E. Fenton and Shari Lawrence Pfleeger. Software Metrics: A Rigorous and Practical Approach. January 1998.

[19] Martin Fowler. Refactoring: improving the design of existing code. July 1999.

[20] Mark A. Hall. *Correlation-based feature selection for machine learning*. PhD thesis, The University of Waikato, New Zeeland, 1999.

[21] Mark A. Hall. Correlation-based Feature Selection for Discrete and Numeric Class Machine Learning. 2000.

[22] Tracy Hall, Sarah Beecham, David Bowes, David Gray, and Steve Counsell. A Systematic Literature Review on Fault Prediction Performance in Software Engineering. *IEEE Transactions on Software Engineering*, 38(6):1276–1304, November 2012.

[23] IEEE Computer Society. IEEE Standard Glossary of Software Engineering Terminology. *IEEE Std 610.12-1990*, 121990(Dec):1–84, 1990.

[24] Nathalie Japkowicz and Mohak Shah. *Evaluating Learning Algorithms: A Classification Perspective*. Cambridge University Press, New York, New York, USA, 2011.

[25] TM Khoshgoftaar, X Yuan, and EB Allen. Uncertain classification of fault-prone software modules. *Empirical Software Engineering*, pages 297–318, 2002.

[26] Yan Ma, L Guo, and B Cukic. A statistical framework for the prediction of fault-proneness. *Advances in machine learning application ...*, pages 1–26, 2006.

[27] A Mahaweerawat. Fault prediction in object-oriented software using neural network techniques. *Advanced Virtual and ...*, 49(5):483–492, December 2004.

[28] Thomas J Mccabe. A Complexity Measure. (4):308–320, 1976.

[29] Steve McConnell. *Code Complete: A Practical Handbook of Software Construction, Second Edition*. Microsoft Press, 2004.

[30] Prem Melville and Raymond J. Mooney. Creating diversity in ensembles using artificial data. *Information Fusion*, 6(1):99–111, March 2005.

[31] Tim Menzies, Jeremy Greenwald, and Art Frank. Data Mining Static Code Attributes to Learn Defect Predictors. *IEEE Transactions on Software Engineering*, 33(1):2–13, January 2007.

[32] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 13th international conference on Software engineering - ICSE '08*, page 181, New York, New York, USA, 2008. ACM Press.

[33] Glenford J. Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing*. Wiley, 2011.

[34] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, pages 284–292. IEEe, 2005.

[35] Nachiappan Nagappan. Using In-Process Testing Metrics to Estimate Software Reliability: A Feasibility Study. ... *Software Reliability ...*, pages 1–3, 2004.

[36] Nachiappan Nagappan. *A Software Testing and Reliability Early Warning (Strew) Metric Suite*. PhD thesis, North Carolina State University, 2005.

[37] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. Mining metrics to predict component failures. In *Proceeding of the 28th international conference on Software engineering - ICSE '06*, page 452, New York, New York, USA, 2006. ACM Press.

[38] Nachiappan Nagappan, Laurie Williams, Mladen Vouk, and Jason Osborne. Early estimation of software quality using in-process testing metrics. In *Proceedings of the third workshop on Software quality - 3-WoSQ*, page 1, New York, New York, USA, 2005. ACM Press.

[39] Nachiappan Nagappan, Laurie Williams, Mladen Vouk, and Jason Osborne. Using In-Process Testing Metrics to Estimate Post-Release Field Quality. In *The 18th IEEE International Symposium on Software Reliability (ISSRE '07)*, pages 209–214. IEEE, November 2007.

[40] Nachiappan Nagappan, Andreas Zeller, Thomas Zimmermann, Kim Herzig, and Brendan Murphy. Change Bursts as Defect Predictors. In *2010 IEEE 21st International Symposium on Software Reliability Engineering*, pages 309–318. IEEE, November 2010.

[41] Vu Nguyen, Sophia Deeds-Rubin, Thomas Tan, and Barry Boehm. A SLOC Counting Standard. Technical report, 2007.

[42] Thomas J Ostrand, Elaine J Weyuker, and Robert M Bell. Where the bugs are. *ACM SIGSOFT Software Engineering Notes*, 29(4):86, July 2004.

[43] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. 1993.

[44] Danijel Radjenović, Marjan Heričko, Richard Torkar, and Aleš Živkovič. Software fault prediction metrics: A systematic literature review. *Information and Software Technology*, 55(8):1397–1418, August 2013.

[45] Juha Reunanen. A Pitfall in Determining the Optimal Feature Subset Size. In *Proceedings of the 4th International Workshop on Pattern Recognition in Information Systems*, pages 176–185. SciTePress - Science and and Technology Publications, 2004.

[46] J. Riquelme, J. C., Ruiz, R., Rodrıguez, D., Moreno. Finding defective modules from highly unbalanced datasets. *Actas de los Talleres de las Jornadas de Ingeniería del Software y Bases de Datos*, 2(1):67–74, 2008.

[47] Adrian Schröter, Thomas Zimmermann, and Andreas Zeller. Predicting component failures at design time. In *Proceedings of the 2006 ACM/IEEE international symposium on International symposium on empirical software engineering - ISESE '06*, page 18, New York, New York, USA, 2006. ACM Press.

[48] Sana Shafi, Syed Muhammad Hassan, Afsah Arshaq, Malik Jahan Khan, and Shafay Shamail. Software quality prediction techniques: A comparative analysis. In *2008 4th International Conference on Emerging Technologies*, pages 242–246. IEEE, October 2008.

[49] Shivkumar Shivaji, E. James Whitehead Jr., Ram Akella, and Sunghun Kim. Reducing Features to Improve Bug Prediction. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, number Section II, pages 600–604. IEEE, November 2009.

[50] Forrest Shull, Vic Basili, Barry Boehm, A.W. Brown, Patricia Costa, Mikael Lindvall, Dan Port, Ioana Rus, Roseanne Tesoriero, and Marvin Zelkowitz. What we have learned about fighting defects. In *Proceedings Eighth IEEE Symposium on Software Metrics*, pages 249–258. IEEE Comput. Soc, 2002.

[51] Pawel Smialowski, Dmitrij Frishman, and Stefan Kramer. Pitfalls of supervised feature selection. *Bioinformatics (Oxford, England)*, 26(3):440–3, February 2010.

[52] D. Spinellis. Tool Writing: A Forgotten Art? *IEEE Software*, 22(4):9–11, July 2005.

[53] Burak Turhan and Ayse Bener. Analysis of Naive Bayes' assumptions on software fault data: An empirical study. *Data & Knowledge Engineering*, 68(2):278–290, February 2009.

[54] Burak Turhan, Tim Menzies, Ayşe B. Bener, and Justin Di Stefano. On the relative value of cross-company and within-company data for defect prediction. *Empirical Software Engineering*, 14(5):540–578, January 2009.

[55] Frank Wilcoxon. Individual Comparisons by Ranking Methods. *Biometrics Bulletin*, 1(6):80, December 1945.

[56] Ian H. Witten, Eibe Frank, and Mark A. Hall. *Data Mining: Practical Machine Learning Tools and Techniques, Third Edition (The Morgan Kaufmann Series in Data Management Systems)*. Morgan Kaufmann, 2011.

[57] He Zhang, Muhammad Ali Babar, and Paolo Tell. Identifying relevant studies in software engineering. *Information and Software Technology*, 53(6):625–637, June 2011.

[58] Hongyu Zhang. An investigation of the relationships between lines of code and defects. In *2009 IEEE International Conference on Software Maintenance*, pages 274–283. IEEE, September 2009.

[59] Hongyu Zhang and Xiuzhen Zhang. Comments on "Data Mining Static Code Attributes to Learn Defect Predictors". *IEEE Transactions on Software Engineering*, 33(9):635–637, September 2007.