Towards Generalizing Defect Prediction Models

by

Feng Zhang

A thesis submitted to the
School of Computing
in conformity with the requirements for
the degree of Doctor of Philosophy

Queen's University
Kingston, Ontario, Canada
January 2016

Abstract

Software quality is vital to the success of a software project. Fixing defects is the major activity to continuously improve software quality. Given that a real development team usually exhibits limited resources and tight schedules, it is important to prioritize testing activities and optimize development resources. Predicting defective entities (e.g., files or classes) ahead helps achieve such a goal. Defect prediction has attracted considerable attention from both academia and industry in the last decade.

A typical defect prediction model is built upon software metrics and labelled defect data that are collected from the historical data of a software project. A defect prediction model can be applied within the same project (within-project defect prediction) or on other projects (cross-project defect prediction). However, due to the diversity in development processes, a defect prediction model is often not transferable and requires to be rebuilt when the target project changes. As it consumes additional effort to build and maintain a defect prediction model for a particular project, it is of significant interest to generalize a defect prediction model. A generalized defect prediction model relieves the need to rebuild a defect prediction model for each target project. Moreover, it helps reveal a general relationship between software metrics and defect data.

In this thesis, we analyze the feasibility of generalizing defect prediction models. First, we analyze how the distribution of the values of software metrics varies across projects of different context factors (e.g., programming language and system size). We observe that such distributions do vary across projects, but can also be similar across projects of different context factors. Second, we investigate the impact that the pre-processing steps (in particular, transformation and aggregation of software metrics) have on the performance of defect prediction models. We find that the pre-processing steps impact the performance of defect prediction models, and therefore need to be considered towards generalizing defect prediction models. Finally, we propose two approaches for generalizing defect prediction models with supervised (requiring the training data) and unsupervised (without the training data) methods, respectively. Our results show that both approaches are feasible to generalize defect prediction models.

Statement of Originality

I, Feng Zhang, hereby declare that I am the sole author of this thesis. All ideas and inventions from others have been properly attributed. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners. I understand that my thesis may be made electronically available to the public.

Acknowledgments

Eventually, I'm at the final stage of my PhD studies. Looking back to the long journey, I would like to thank my supervisor Dr. Ying Zou who is always very supportive as a tremendous advisor and a great friend. I would like to thank you for all your guidance and encouragement on how to do great research. A special thank you to Dr. Ahmed E. Hassan for your priceless advice on both my research and my career. I would also like to thank Dr. Audris Mockus from whom I have learned much about rigorous research.

I'm very lucky to work with a smart and amazing group at Queen's University, but I would like to particularly thank Dr. Foutse Khomh and Dr. Iman Keivanloo who have closely worked with me and shaped my research. I would also like to thank Dr. Daniel M. German, Dr. Shane McIntosh, Dr. Emad Shihab, and Dr. Meiyappan Nagappan for their fruitful advice on my work.

I'm grateful for my supervisory and examination committee members, Dr. Patrick Martin, Dr. Hossam Hassanein, and Dr. Mohammad Zulkernine for their insightful feedback and continued critique. Many warm thanks to my examiners Dr. Giuliano Antoniol and Dr. Diane Kelly for taking the time to read and critique my work.

I'm fortunate to work with all of my lab mates and classmates with whom I have a joyful experience during my PhD studies. They are very kind people and great friends, Shaohua Wang, Dr. Bipin Upadhyaya, Shuai Xie, Hao Yuan, Liliane Barbour, Tejinder Dhaliwal, Quan Zheng, Tse-Hsun Chen, Ehsan Salamati, Haoran Niu, Phoena Pang, Yu Zhao, Hanfeng Chen, Mariam El Mezouar, Ehsan Noei, Pradeep Venkatesh, Yonghui Huang, and Reena Muthalaly.

I would like to thank all of my family and dear friends for your continuous support that makes this thesis possible. In particular, the unconditional support from my parents and parents-in-law always encourages me to approach without hesitation. Finally, I'll be eternally grateful to my beloved wife Yunzhu for your devotion and sacrifice, and to my adorable son Dylan for bringing me endless joys during my PhD studies.

Dedication

To my beloved wife Yunzhu, son Dylan, parents and parents-in-law.

Related Publications

Early versions of the work in this thesis have been published as follows.

- **How does Context affect the Distribution of Software Maintainability Metrics?** (Chapter 3). Feng Zhang, Audris Mockus, Ying Zou, Foutse Khomh, and Ahmed E. Hassan. Proceedings of the 29th IEEE International Conference on Software Maintainability (ICSM'13), pp. 350-359. [201]
- **Towards Building a Universal Defect Prediction Model** (Chapter 6). Feng Zhang, Audris Mockus, Iman, Keivanloo, and Ying Zou. Proceedings of the 11th Working Conference on Mining Software Repositories (MSR 2014), pp.182-191. [202]

  – ***This paper has received the distinguished paper award of MSR 2014.***
- **Towards Building a Universal Defect Prediction Model with Rank Transformed Predictors** (Chapter 6). Feng Zhang, Audris Mockus, Iman, Keivanloo, and Ying Zou. Springer Journal of Empirical Software Engineering (EMSE), 1-39, 2015. [203]
- **Defect Prediction Without Training Data Via Spectral Clustering** (Chapter 7). Feng Zhang, Quan Zheng, Ying Zou, and Ahmed E. Hassan. Proceedings of the 38th International Conference on Software Engineering (ICSE'16), 2016, Austin, TX, United States. [204]

Table of Contents

List of Tables

List of Figures

"The journey of a thousand miles begins with one step."

— Lao Tzu (604 BC - 531 BC)

Chapter 1 Introduction

Releasing a software system with high quality is vital to the success of a software project. However, software systems generally contain defects [45, 71, 112, 128, 129, 139]. A defect is an error in software behaviour that causes undesired results. It was estimated that software defects cost U.S. economy $59.5 billion annually [184]. Usually, 50% to 75% of the total software development budget is spent in fixing defects [70]. The cost of fixing defects can be reduced significantly if defects are fixed as early as possible [6, 32, 53, 128, 129, 148, 177].

In an ideal case, developers inspect each file carefully to find out every possible defect and fix them before each release. However, it is impractical to inspect all files of a non-trivial system, because most software organizations experience limited resources and tight release schedules. To this end, it is important to determine what files should be inspected immediately and what files could be examined at a later time. Therefore, defect prediction models are proposed to prioritize quality improvement and defect avoidance efforts.

Numerous approaches have been proposed to build defect prediction models (e.g., [8, 45, 71, 112, 156]), but the industry does not widely adopt defect prediction [143, 159, 187]. The benefit of defect prediction models comes at a cost – additional efforts are

required to build an appropriate defect prediction model. Specifically, it requires collecting sufficient training data and selecting proper technologies for model building. Sometimes, manual analysis is required, such as verifying the correctness of the collected defect data and removing noise from the training data. As a software system evolves, the development environment may change. Thus the model needs to be updated as well, which introduces extra cost. As a summary, possible barriers to adopt defect prediction models include: 1) the cost to collect up-to-date training data (e.g., defect data) [143, 159, 187, 188]; 2) the low generalizability of prediction models [159]; and 3) the lack of automated tooling for the prediction process [31, 159, 187]. In fact, Ostrand and Weyuker [143] report that many companies lack the needed resources and technical expertise to prepare data for building defect prediction models.

Furthermore, there are often insufficient training data to build a model for new or small software systems. To predict defects for such systems, defect prediction models need to be built using the training data from other projects (cross-project defect prediction). We distinguish within-project and cross-project defect prediction as follows:

- **Within-project defect prediction:** models are built and applied on the same project.
- **Cross-project defect prediction:** models are built on some projects and applied on some other projects.

Comparing to within-project defect prediction, cross-project prediction is more likely to experience a significantly different distribution of metric values between the training and target data. This is because the development environment varies across projects. Given the difficulty to transfer existing within-project and cross-project models, it is desirable to generalize a defect prediction model.

In this thesis, we define a generalized defect prediction model by:

A generalized defect prediction model *is a single model that is applicable upon a large set of projects.*

A generalized defect prediction model would relieve the need for refitting project-specific or release-specific models for an individual project. Therefore, a generalized defect prediction model has the potential to boost the adoption of defect prediction models in practice. For instance, a generalized model can be integrated into an Integrated Development Environment (IDE) for instant evaluation of software quality, thus helping software organizations to deliver high-quality software in a timely manner. Furthermore, a generalized model would also help interpret basic relationships between software metrics and defects, potentially mitigating the challenge of lacking theoretical basis for the defect prediction problem. Therefore, it is of significant interest to generalize a defect prediction model.

1.1 Thesis Statement

As presented in Figure 1.1, the typical process to build defect prediction models has four major steps: 1) compute software metrics (e.g., product and process metrics) from code repository; 2) pre-process software metrics before fitting them to a model; 3) collect defect information with code repositories and issue tracking systems; and 4) apply appropriate modelling techniques (e.g., such as random forest) to build a defect prediction model.

It is tedious and time-consuming to maintain a defect prediction model for each individual project. Therefore, defect prediction models are rarely applied in practice. To resolve such problem, we aim to work on generalizing defect prediction models. Our thesis statement is listed as follows.

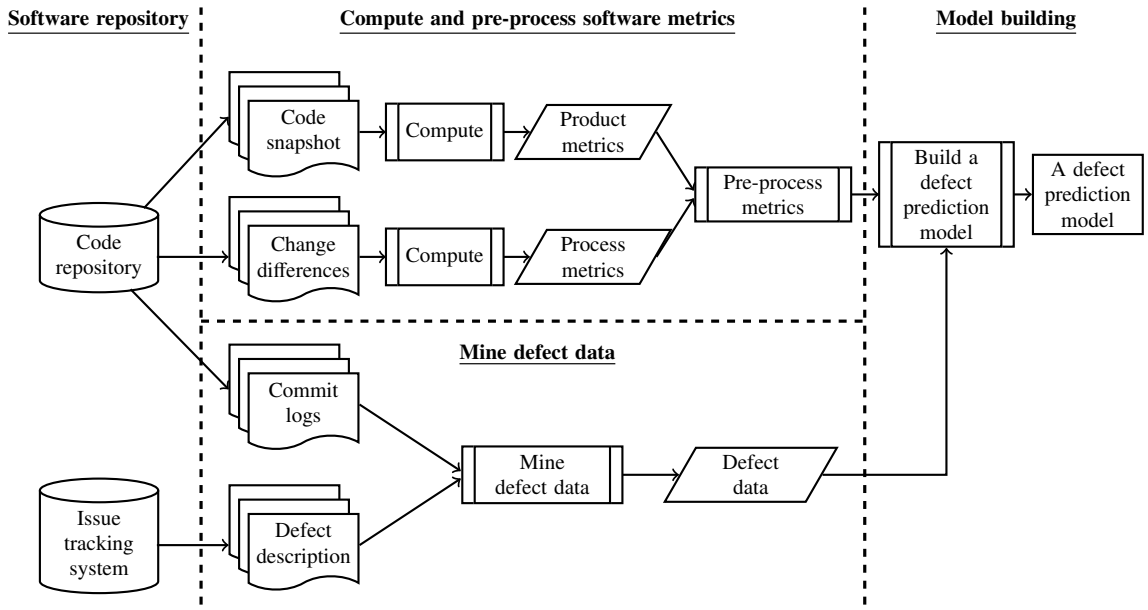


Figure 1.1: An overview of the typical process of building defect prediction models.

Thesis statement: *A key step to generalize defect prediction is to achieve successful prediction across projects, which is greatly challenged by the heterogeneity between the training and target projects. It is essential to understand how the heterogeneity varies across projects. Then we can select appropriate data pre-processing and modelling techniques to mitigate the heterogeneity, in order to build a generalized defect prediction model.*

Predicting defects across projects is a great challenge (e.g., [71, 188, 210]), due to the heterogeneity of both dependant variables (e.g., software metrics) and the independent variable (i.e., defect data) across projects [26, 43, 135]. The heterogeneity is likely to be caused by diverse development environments (e.g., varying user requirements and developer experience) [26, 113].

In this thesis, we first 1) investigate how the distribution of metric values of software entities (e.g., files or classes) varies across projects with varied contexts (e.g., programming

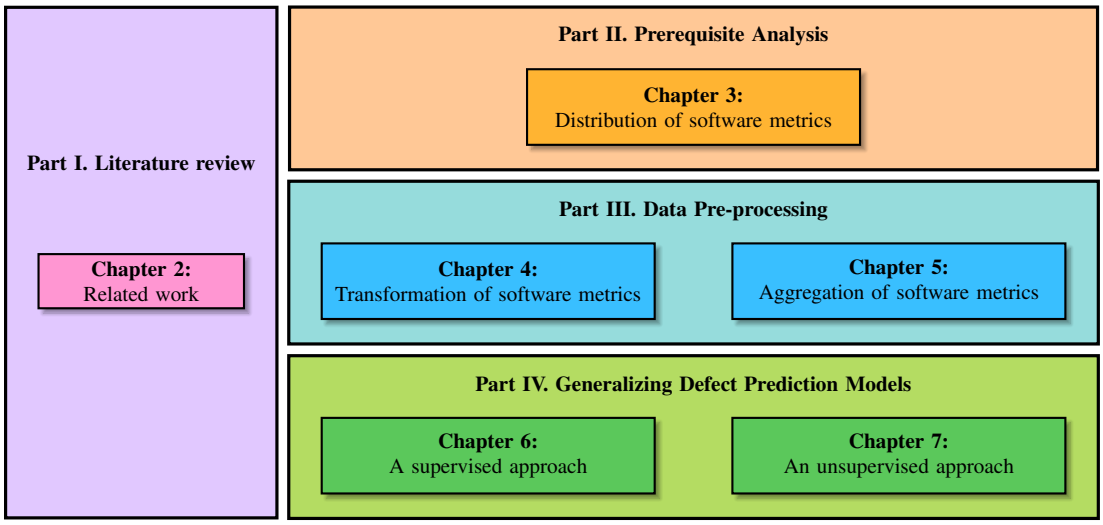


Figure 1.2: An overview of the contents of this thesis.

language and system size). Then 2) we study if the pre-processing steps (i.e., transformation and aggregation of software metrics) impact the predictive power of defect prediction models. Finally, 3) we propose two approaches (i.e., supervised and unsupervised) towards generalizing defect prediction models, and evaluate the feasibility of our proposed approaches. The supervised approach requires the training data, while the unsupervised approach does not require any training data.

1.2 Thesis Overview

We present the overview of this thesis in Figure 1.2. In this thesis, there are four major parts. They are described as follows.

Part I. Literature review

Chapter 2: *Related work.* In this chapter, we review possible challenges towards generalizing defect prediction models and present a survey of research on within-project and cross-project defect prediction models.

Part II. Prerequisite Analysis

Chapter 3: *Distribution of software metrics.* Software metrics are commonly used as predictors in defect prediction models. We investigate if the distributions of commonly used metrics do, in fact, vary with six context factors: application domain, programming language, age, lifespan, the number of changes, and the number of downloads. We also briefly discuss how each context factor may affect the distribution of metric values.

Part III. Data Pre-processing

Chapter 4: *Transformation of software metrics.* Transformation is often applied to pre-process software metrics, therefore can impact the performance of defect prediction models. We conduct an exploratory study to investigate the impact of two commonly used transformation methods (i.e., log and rank transformations), as well as the Box-Cox transformation.

Chapter 5: *Aggregation of software metrics.* Historical defect data is often mined at the file-level, while software metrics are usually calculated at the class- or method-level. To address the disagreement in granularity, the class- and method-level software metrics are aggregated to file-level, often using summation. In this chapter, we investigate how different aggregation schemes impact defect prediction models.

Part IV. Generalizing Defect Prediction Models

Chapter 6: *A supervised approach towards generalizing defect prediction models.* A formidable obstacle to generalize a defect prediction model is the variations in the distribution of predictors among projects of diverse contexts (e.g.,

size and programming language). Hence, we propose to cluster projects based on the similarity of the distribution of predictors, and derive the rank transformations using quantiles of predictors for a cluster.

Chapter 7: *An unsupervised approach towards generalizing defect prediction models.* The heterogeneity between the training and target data is the main challenge in cross-project defect prediction. An unsupervised classifier does not require any training data, therefore the heterogeneity problem is no longer an issue. In this chapter, we propose a new unsupervised connectivity-based classifier that is based on spectral clustering.

1.3 Thesis Contributions

The major contributions of this thesis are summarized as follows.

- Although all studied six context factors impact the distribution of the values of 51% of metrics, the values of software metrics can also experience similar distributions across projects of different context factors (Chapter 3).

- Cross-project prediction models built with the studied three transformations (i.e., log, Box-Cox, and rank transformations) have similar overall performances, but do not always experience wrong predictions at the same time. Hence, we further propose to combine these models and achieve statistically significant improvement in predictive power (Chapter 4).

- Aggregation can significantly impact both the correlation among software metrics and the correlation between software metrics and defect count. For instance, summation significantly inflates the correlation between SLOC and other metrics (not just

Cc). Moreover, using only the summation (i.e., the most commonly applied scheme) often hinders the predictive power of defect prediction models (Chapter 5).

- We propose an approach of context-aware rank transformation to mitigate the large variations in the distribution of software metrics across projects of diverse contexts. The defect prediction model built using the transformed software metrics is generalizable (Chapter 6).

- We propose a connectivity-based unsupervised classifier for defect prediction which can achieve good performance in a cross-project setting. Furthermore, we observe that there exist two (defective and clean) separated communities of software entities based on the connectivity between the metrics of the entities in each community (Chapter 7).

1.4 Thesis Organization

In the next chapter, the background and related work are discussed. The investigation on the distribution of software metrics across projects with varied contexts is presented in Chapter 3. In Chapter 4 and Chapter 5, the impact of two pre-processing methods (i.e., data transformation and data aggregation) are examined, respectively. Based on the findings, an approach towards building a generalized defect prediction model is proposed and described in Chapter 6. Such approach is supervised that requires training data, therefore may cause the heterogeneity between the training and target data. Therefore, an unsupervised approach is further introduced and examined in Chapter 7. Finally, Chapter 8 concludes the work and discusses future directions.

Part I

Literature Review

"If I have seen further, it is by standing on the shoulders of giants."

— Isaac Newton (1642 - 1727)

CHAPTER 2 Related work

Despite the significant achievement in cross-project defect prediction, most existing approaches (e.g., [33, 76, 118, 135, 188]) are hardly to provide a generalized defect prediction model. For instance, the transfer learning based approach [135] always requires to re-process software metrics in both training and target projects once any peer (i.e., the training or the target project) changes. In the following sections, we present details regarding possible challenges towards generalizing defect prediction models.

2.1 Software Metrics Collection

Software metrics are commonly used as predictors in defect prediction models. An intuitive threat to generalize defect prediction models is whether metrics are collected in a consistent manner. Three challenges are described as follows.

1) **Definition and computation.** Metrics computed from the same input are assumed to be consistent despite which tool is used. In fact, this assumption is invalid. Lincke *et al.* [114] investigate values of nine OO metrics that are computed through ten metric computation tools (including both commercial and free tools), and observe large variations

in complex metrics (e.g., Coupling Between Object classes (CBO), and Lack of Cohesion of Methods (LCOM)). The differences even exist after abstraction (e.g., ranking metric values) [114]. Hence, Lincke *et al.* [114] suggest to clearly describe the scope and mapping definitions of programming languages.

2) **Metric granularity.** Metrics are collected at different granularity, such as method level, class level, file level, or project level. The information provided by metrics of various levels is different. Catal and Diri [29] report that the best algorithm to model defect proneness is different for method-level metrics and class-level metrics.

3) **Variation with contexts.** The diversity of developers leads to the variation in software metrics across projects. Systa and Muller [183] explicitly state that their best model should not be considered as a universal model for Java projects due to the variance in functionality or development team.

2.2 Software Metrics Pre-processing

Data pre-processing has been proved to improve the performance of defect prediction models by Menzies *et al.* [123]. The non-normality of software metrics can negatively impact the performance of linear models [37]. It is common to apply data transformation, such as log transformation [93, 123] and rank transformation [93, 202]. In addition, the variation in scales of metrics pose a challenge to generalize defect prediction models. To deal with the varied granularities (e.g., file, class, or method) of software metrics, aggregation (e.g., summation) is usually conducted. Therefore, pre-processing software metrics may be a mandatory step needed to build a successful cross-project defect prediction model.

1) **Software transformation.** Many software metrics (e.g., the number of tokens) follow power law distributions [41, 205]. To build a defect prediction model, researchers often apply the natural log transformation (e.g., [93, 123, 180]) and rank transformation (e.g., [93, 202]) on software metrics. However, Jiang *et al.* [93] compare log and rank transformations, and find that different classifiers prefer different transformations. Cruz and Ochimizu [43] observe that log transformations can improve the performance of cross-project predictions, only if the data of the target project is not as skewed as the data of the training project.

   The state-of-the-art approaches to improve the performance of cross-project defect prediction mainly use two data pre-processing techniques: 1) use data from projects with a similar distribution to the target project (e.g., [124, 188]); or 2) transform predictors in both training and target projects to make them more similar in their distribution (e.g., [118, 135]). For instance, Turhan *et al.* [188] propose to use the nearest neighbour filter and Nam *et al.* [135] propose to transform both training and target projects to the same latent feature space, and build models on the latent feature space. However, the aforementioned approaches use only partial dataset and end up with multiple models (i.e., one model per target project).

2) **Software metrics aggregation.** While the most commonly used granularity of defect prediction is the file-level [75, 135, 202, 210], many software metrics are calculated at the method- or class-levels. The difference in granularity creates the need for aggregation of the finer method- and class-level metrics to file-level. Simple aggregation schemes, such as summation and mean, have been explored in the defect prediction literature [82, 106, 111, 112, 128, 135, 138, 154, 202, 208, 209]. However, Landman *et al.* [109] show that prior findings (e.g., [54, 67]) about the high correlation between

summed Cc and summed lines of code (i.e., SLoc) may have been overstated for Java projects, since the correlation is significantly weaker at the method-level.

Apart from summation and mean, more advanced metric aggregation schemes have been also explored [60, 63, 76, 167, 190, 191], including the Gini index [62], the Atkinson index [9], the Hoover index [83], and the Kolm index [105]. For example, D'Ambros *et al.* [44] compute the entropy of both code and process metrics. Hassan [75] applies Shannon's entropy [169] to aggregate process metrics as a measure of the complexity of the change process. Vasilescu *et al.* [192] find that the correlation between metrics and defect count is impacted by the aggregation scheme that is used. He *et al.* [76] apply multiple aggregation schemes to construct various metrics about a project in order to find appropriate training projects for cross-project defect prediction. Giger *et al.* [60] use the Gini index to measure the inequality in the ownership of files and obtain acceptable performance for defect proneness models. However, it is still unclear on how data aggregation impacts the performance of defect prediction models.

2.3 Software Metrics Thresholds

Deriving appropriate thresholds and ranges of metrics is important to interpret software metrics [110]. For instance, McCabe [119] proposes a widely used complexity metric to measure software maintainability and testability, and further interprets the value of his metric in such a way: sub-functions with the metric value between 3 and 7 are well structured; sub-functions with the metric value beyond 10 are unmaintainable and untestable [119]. Lorenz and Kidd [115] propose thresholds and ranges for many object-oriented metrics, which are interpretable in their context. However, direct application of these thresholds and ranges without taking into account the contexts of systems might be problematic. Erni

and Lewerentz [52] propose to consider mean and standard deviations based on the assumption that the metrics follow normal distributions. Yet many metrics follow power-law or log-normal distributions [80, 116]. Thus many researchers propose to derive thresholds and ranges based on statistical properties of metrics. For example, Benlarbi *et al.* [14] apply a linear regression analysis. Shatnawi [170] use a logistic regression analysis. Yoon *et al.* [200] use a k-means cluster algorithm. Herbold *et al.* [79] use machine learning techniques. Sánchez-González *et al.* [161] compare two techniques: ROC curves and the Bender method. In addition, Bouktif *et al.* [21] propose to update thresholds based on the feedback from developers.

A recent attempt to build benchmarks is by Alves *et al.* [4]. They propose a framework to derive metric thresholds by considering metric distributions and source code scales, and select a set of software systems from a variety of contexts as measurement data. Baggen *et al.* [11] present several applications of Alves *et al.* [4]'s framework. Bakota *et al.* [12] propose a different approach using probabilities other than thresholds or ranges, and focus on aggregating low-level metrics to the maintainability that is described in the ISO/IEC 9126 standard [87].

The aforementioned studies do not consider the potential impact of the contexts of software systems. The contexts can affect the effective values of various metrics [50]. Contexts of software systems are considered in Ferreira *et al.* [56]'s work. They propose to identify thresholds of six object-oriented software metrics using three context factors: application domain, software types (i.e., tool, framework and library) and system size (in terms of the number of classes). However, they directly split all software systems using the context factors without examining whether the context factors affect the distribution of metric values or not, thus result in a high ratio of duplicated thresholds. To reduce duplications and

maximize the samples of measurement software systems, a split is necessary only when a context factor impacts the distribution of metric values.

2.4 Defect Data Collection

The major challenges come from possible noise and varied ratio of defects across projects. If no issue tracking system is present, defect data is mined solely from commit logs with heuristics (e.g., [126, 179]), and noise can be introduced. If an issue tracking system is used, the wrong or missing links between issue reports and commit logs can also introduce noise [10, 196]. The ratio of defects in a project relates to the choice of best cut-off values to determine the defect proneness of an entity. The varied ratios of defects among diverse projects increase the difficulty to generalize defect prediction model, and further investigation is needed.

1) **Defect distribution.** Many studies (e.g., [5, 40, 54, 72, 142]) state that the Pareto principle applies to defects [86]. The Pareto principle is often referred as the 20-80 rule, which means that most defects (80%) exist in few modules (20%). Besides the Pareto distribution, or instance, Janes *et al.* [89] find that defects follow a negative binomial distribution in their subject projects that are telecommunication software systems.

2) **Imbalanced defect data.** As following the Pareto principle, defect data is usually imbalanced. The imbalance problem can result in a low accuracy in predicting the minority class (i.e., defect proneness) [149]. Two common methods are to under-sample the majority class or to oversample the minority class. Pelayo and Dick [149] observe that uniform under-sampling and the combination of under-sampling and oversampling

have the potential to improve the accuracy of defect prediction models. In addition to re-sampling techniques, Wang and Yao [193] investigate threshold moving and Boosting-based ensembles and find that AdaBoost.NC achieves the best performance.

3) **Defect data quality.** The high quality of defect data is vital to the success of a defect prediction model. However, even the widely used NASA data sets contain noise. In particular, Gray *et al.* [68] find duplicated data points in NASA data sets that may result in excessive estimate of performance. Kim *et al.* [102] investigate the impact of noise on defect prediction models, and find that 20% to 35% of both false positive and false negative noise significantly decrease the predictive power. Rahman *et al.* [158] report that simply enlarging samples can mitigate the impact by bias in defect data.

4) **Lack of defect data.** Most defect prediction models are built using supervised classifiers that rely on labelled training projects to infer a function between the independent variables and the dependant variable. However, in a distributed development environment, defect data may be collected at particular locations [166]. With limited defect data, a supervised learning technique may not yield good performance [166]. To this end, Seliya and Khoshgoftaar [166] propose semi-supervised defect prediction based on the Expectation Maximization (EM) algorithm; Li *et al.* [113] propose a sampling-based approach to build defect prediction models. For unsupervised learning, experts are often involved to manually determine the defect proneness of each cluster [166, 206].

2.5 Predictive Techniques

The statistical model outperforms the expert model for defect prediction [186]. Tomaszewski *et al.* [186] conjecture that human encounters seriously limited ability to estimate the complexity of large systems, while the statistical model is not affected by the size of subject

systems. The statistical model can also be used without involving experts who may be unavailable (e.g., when a project is previously developed by another organization).

Supervised modelling techniques can be used, if there exist sufficient defect data to build a prediction model. Otherwise, a semi-supervised modelling technique is necessary. For projects without defect data, either an unsupervised technique or a cross-project prediction model is needed.

2.5.1 Supervised techniques

A technique that works well for one project does not necessary to be good for another project. For instance, Lessmann *et al.* [112] find that the best prediction technique varies with projects. In this subsection, studies regarding supervised techniques are discussed.

1) **Significant difference in predictive power across modelling techniques.** Janes *et al.* [89] build models for predicting defect count, and find that three techniques (i.e., poison regression, negative binomial regression, and zero-inflated negative binomial regression) are suitable to deal with overdispersion and heterogeneity of predictors. Gondra [64] compares two modelling techniques (i.e., support vector machine (SVM) and artificial neural network(ANN)) and find that SVM significantly outperforms ANN for predicting defect proneness. Hall *et al.* [71] report that Naive Bayes and logistic regression perform well in defect prediction, while SVM performs less well. Weyuker *et al.* [195] perform a comparison using 28 to 35 releases of three large industrial projects and find that random forests and the negative binomial regression perform significantly better using two other techniques (i.e., Bayesian additive regression trees, and recursive partitioning). Jia *et al.* [92] compare 10 modelling techniques using three releases of industrial projects, and find that random forest and Bayesian belief network outperform other techniques in terms of predictive accuracy. Moreover, Jia *et al.* [92] conclude that

random forest is more cost-effective, and Bayesian Belief Network has the highest recall of defective modules.

2) **Non-significant difference in predictive power across modelling techniques.** As opposite to the aforementioned studies, some researchers perform more extensive comparisons and do not observe significant difference across different prediction techniques. Arisholm *et al.* [7] perform a case study using Java projects to compare the performance across several modelling techniques (e.g., decision tree, PART, SVM, logistic regression, neural networks). They observe there is no significant difference among their studied modelling technique when predicting defect proneness. However, they find significant differences in cost-effectiveness analysis. In particular, relatively simple techniques (e.g., decision tree) perform as well as more complex techniques (i.e., neural networks). Arisholm *et al.* [7] even deploy prediction models built with decision trees to developers, and receive very positive feedback from developers who manage to identify defects that are not found during unit testing. The success of simple prediction techniques increases the chance to generalize defect prediction models with acceptable performance.

   Lessmann *et al.* [112] compare the performance across 22 modelling techniques in predicting defect proneness for 10 projects. Although the best technique varies with projects, they do not observe significant difference in predictive power (i.e., in terms of AUC value) among the 17 top-ranked techniques. Similar as [7], Lessmann *et al.* [112] also find that simple techniques (e.g., logistic regression) can achieve comparable performance as more complex techniques. Lessmann *et al.* [112] further suggest that different techniques should not be compared only based on predictive power but also additional criteria such as computational efficiency.

Arisholm *et al.* [8] conduct a systematic and comprehensive investigation of techniques to build and evaluate models for predicting defect proneness. Arisholm *et al.* [8] conclude that the choice of prediction techniques has limited impact on the performance in terms of both AUC and cost-effectiveness. Arisholm *et al.* [8] further note that the choice of the best model strongly depends on the criteria used for model evaluation.

3) **Other factors to consider.** The configuration parameter is rarely considered in aforementioned studies. For a particular technique, using its default configurations may not achieve the best performance. For instance, Sarro *et al.* [162] find that the performance in predicting defect proneness using SVM can be improved by tuning the parameters with a genetic algorithm. When several techniques have comparable predictive power, Jiang *et al.* [94] suggest to select the technique that experiences the least variance in performance, since a technique with smaller variance is more stable. The selection of metric sets can also impact the predictive power [45, 107]. For instance, D'Ambros *et al.* [45] find that given a list of candidate metrics, different techniques for metric selection produces very different sets of metrics.

2.5.2 Unsupervised techniques.

Unsupervised defect prediction is to predict defect proneness without requiring access to training projects. As illustrated in Figure 2.1, a typical process to predict defects using an unsupervised classifier has two steps: 1) clustering software entities into k clusters (usually two clusters); and 2) labelling each cluster as a defective or clean cluster. However, there exist a limited number of studies in the literature on unsupervised defect prediction. One reason is that unsupervised classifiers usually underperform supervised ones (e.g., random forest and logistic regression) in terms of the predictive power.

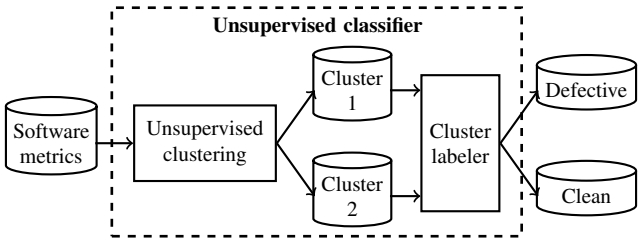


Figure 2.1: A typical process to do defect prediction using an unsupervised classifier.

An initial attempt to use unsupervised defect classifiers is by Zhong *et al.* [206] who apply k-means and neural-gas clustering in defect prediction. Zhong *et al.* [206] observe that a neural-gas classifier outperforms k-means in terms of predictive power, but runs slower. However, their approach requires one to specify the expected number of clusters, and involves experts to determine which cluster contains defective entities (i.e., label the cluster). Catal *et al.* [30] propose to use metric values to determine which cluster has defective entities. Bishnu and Bhattacherjee [18] propose to apply quad trees to initialize the cluster centres of k-means clustering. In addition to k-means clustering based classifiers, Abaei *et al.* [1] propose to use self-organizing maps (SOM) and Yang *et al.* [198] propose to apply the affinity propagation clustering algorithm. Recently, Nam and Kim [133] proposed to cluster software entities using thresholds on selected metrics with very promising performance on seven studied projects.

2.6 Cross-release Defect Prediction

Cross-release defect prediction is a practical case in within-project defect prediction. Cross-release prediction models are often built using the nearest previous releases or all previous releases. Different releases of software projects are typically developed following the same development process (methods, process, and environment) by the developers hired under

similar criteria and requirements [48]. Kastro and Bener [99] conjecture that the introducing of new features causes the major difference across releases in terms of product and process metrics.

1) **Positive results in cross-release defect prediction.** Denaro and Pezzè [48] build many models from Apache Web server 1.3, and use the best models to predict defects for Apache Web server 2.0. Denaro and Pezzè [48] find that high quality multivariate models can successfully predict defect proneness across releases. As different releases of projects can be viewed as homogeneous projects, Denaro and Pezzè [48] conclude that it is feasible to generalize defect prediction models across homogeneous projects. Hence, it is important to identify homogeneous projects towards generalizing defect prediction models. Similarly, Bell *et al.* [13] study 140 releases of seven projects, and build defect prediction models based on data from all previous releases. Bell *et al.* [13] observe that such models achieve very high performance to predict the files likely to have the most number of defects in the following release. For instance, the 20% of files predicted as defective contain 75% to 95% of the total defects. One possible reason may be that defects follow the Pareto distribution across releases. Indeed, Holschuh *et al.* [82] find that defects at both package and class levels follow the Pareto distribution across releases. By studying three releases, Holschuh *et al.* [82] observe that 20% of all packages contain 70% of all defects during the eight-month period after the first release, and 20% of all packages contain 80% of all defects during the eight-month period after the second release. In addition to cross-release prediction, Yadav and Yadav [197] build a fuzzy logic based model to predict defect density across each phase of the development process, and report that the predicted defects are very close to the actual defects on their studied 20 projects.

2) **Negative results in cross-release defect prediction.** Moser *et al.* [128] observe that models built using data of the previous release cannot provide as accurate predictions for the following release as for the same release. In some cases, the performance is even unacceptable, e.g., experiencing a false positive rate of over 30%. Similarly, Shatnawi and Li [171] report a decrease in the predictive power of cross-release defect prediction. Tosun *et al.* [187] show that models trained using the historical data of previous versions experience high false positives. Tosun *et al.* [187] further find that the false positive rate can be reduced if building defect prediction models upon previous releases of all available projects. Kläs *et al.* [104] also suggest to use more projects to build a model, as it can increase the accuracy of defect prediction models.

   The aforementioned contradictory findings in cross-release defect prediction raises the interest to understand the underlying rationale. The possible reason for successful cross-release defect prediction is that there is no significant changes among releases, therefore the distribution of both metric values and defect data are similar across releases. On the other hand, significant changes across releases may reduce the performance of cross-release defect prediction.

2.7 Cross-project Defect Prediction

Cross-project prediction is necessary if there is only limited data available to build a model ([27, 132]). For instance, Turhan *et al.* [188] find that few companies have a repository to store software metrics and defect data of past projects, due to the additional effort for manual collection of data and the maintenance of data. Tosun *et al.* [187] fail to build within-project modelling due to limited resources to collect defect data from an industrial

project, and eventually seek to build cross-company models. Cross-project prediction models are more generalizable than within-project models, but are more challenging.

1) **Challenges in cross-project defect prediction.** Initial attempts on cross-project prediction are not successful (e.g., [71, 188, 210]). For instance, Zimmermann *et al.* [210] perform a case study using 28 datasets from 12 projects. Among all the 622 possible pairs of projects. Zimmermann *et al.* [210] find that only 21 pairs achieve very good performance in cross-project prediction, i.e., all precision, recall and accuracy are greater than 0.75. Turhan *et al.* [188] observe that cross-project prediction can experience high false positive rates. Premraj and Herzig [154] replicate existing studies and confirm the challenges in cross-project defect prediction. Nagappan *et al.* [132] report that predictors learnt from one project can rarely be applied in another project. On the other hand, Rahman *et al.* [157] argue that from cost effectiveness perspective, cross-project defect prediction can achieve similar performance as within-project prediction. Nonetheless, there still exists the challenge in cross-project prediction.

   One possible challenge in cross-project prediction is that metrics in various projects can experience significantly different distributions [26, 43, 135]. Denaro and Pezzè [48] and Nagappan *et al.* [132] report that defect prediction models can achieve good predictive power only across homogeneous projects. Similarly, Briand *et al.* [26] perform a study using different projects developed by the same team but with different design strategies and coding standards, and find that a model built on one project can be accurately applied to rank classes for another project based on their probability of defect proneness. Hall *et al.* [71] perform a systematic review on 36 studies, and conclude that the performance of cross-project prediction can be affected by some context factors (e.g., system size and application domain). Hence, project contexts are suggested by Zimmermann

et al. [210] and Menzies *et al.* [124] in cross-project defect prediction.

2) **Solutions to deal with data heterogeneity.** To solve the issue of data heterogeneity across diverse projects, it is necessary to improve the similarity of data between the training and target projects. There are two major approaches: (a) filtering the training project; and (b) transforming both training and target projects.

(a) Filtering-based approaches. Turhan *et al.* [188] observe that cross-company data cannot be used as is, otherwise the predictive power is low. While the recall can be increased from 75% to 92%, there is an increase in false positive rate, i.e., from 29% to 64%. However, after applying nearest neighbour (NN) filtering to cross-company data, Turhan *et al.* [188] achieve similar but not better performance of cross-company defect prediction models than within-company models. For example, the false positive rate is reduced from 64% to 32% after the NN-filtering. Turhan *et al.* [188] also find that the dataset to build effective defect prediction models can be small (i.e., 100 examples are enough), and can be collected within a few months. Hence, Turhan *et al.* [188] suggest a two-phase approach to deal with the problem of lacking training data. First, apply NN-filtered cross-company data to build cross-company models. Second, switch to within-company models after a few months of data collection. He *et al.* [76] find that the performance of cross-project prediction depends on the distributional characteristics (e.g., mean and standard deviation) of datasets. Hence, He *et al.* [76] propose an approach to select training set based on the characteristics of datasets. The results show that their approach can provide comparable performance as within-project prediction. In 18 out of 34 cases, cross-project prediction meets their criteria for acceptance, i.e., at least 70% recall and 50% precision. In the best cases, cross-project models can even provide better predictions than within-project models. He *et al.* [77] further perform an

experiment using 34 releases of 10 open source projects, and 34 releases of seven proprietary projects, and state that the fundamental challenge in cross-project prediction is to select the most appropriate training data. Hence, He *et al.* [77] propose an approach using the similarities in distributions between training and target projects, and demonstrate that their approach is relatively better than the nearest neighbour based filtering method ([188]) in terms of computation cost and predictive power.

(b) Transformation-based approaches. To deal with the varied distributions across projects, Cruz and Ochimizu [43] apply the simple log transformation, and find that the log transformation is effective if metric values for the target project are not as spread as for the training project. Ma *et al.* [118] report that transfer learning is useful for the cases where the distributions of training and testing sets are different, and explore the transfer learning method to build faster and highly effective prediction model. Ma *et al.* [118] propose an algorithm, namely Transfer Naive Bayes (TNB), to weight the training project based on the statistical characteristics learnt from the target project. Transfer component analysis (TCA) [146] aims to minimize the difference in the distribution of metric values between training and testing data while preserving the original data properties. Nam *et al.* [135] investigate the impact that several scenarios in data preprocessing have on the predictive power, and observe that TCA responds better when data is normalized. Hence, Nam *et al.* [135] extend TCA by normalizing both training and target projects prior to applying TCA. Recently, Chen *et al.* [33] propose to apply double transfer boosting (DTB) model to reshape the entire distribution of cross-company data to fit within-company data. DTB aims to increase the similarity in distributions between training and target projects. Chen *et al.* [33] find that their proposed model outperforms within-company defect prediction model trained with limited data, and

provides comparable performance as within-company defect prediction model trained with sufficient data.

Besides the two aforementioned approaches that deal with training and target data, an alternative is ensemble learning. Khoshgoftaar *et al.* [101] investigate the value of using multiple classifiers against using multiple training projects. Their results show that using a single modelling technique with multiple projects is generally better than using a single technique with a single training project, as well as using multiple techniques with a single training project. Hence, Khoshgoftaar *et al.* [101] suggest to use as many existing similar projects as possible to train a defect prediction model. Canfora *et al.* [27] propose a multi-objective approach based on logistic regression for cross-project defect prediction, and demonstrate that their approach outperforms within-project models from cost-effectiveness perspective. Panichella *et al.* [147] find that different modelling techniques can complement each other, and propose to combine different modelling techniques to improve the cross-project defect prediction.

2.8 Chapter Summary

Cross-project defect prediction may more frequently experience different distributions in metric values and defect data than cross-release defect prediction. It is worth studying the distribution of metric values across projects (Chapter 3). Improving the data pre-processing step (Chapter 4 and Chapter 5) is a promising way to increase the similarity between the distribution of metric values across projects. Existing studies on cross-project and cross-release defect prediction suggest that it may be a mandatory step to transform metric values of both training and target projects (Chapter 6). Moreover, a simpler modelling technique is preferred if multiple modelling techniques yield similar performance (Chapter 7).

Part II

Prerequisite Analysis

CHAPTER 3 Distribution of Software Metrics

Key Question

? *How does the distribution of software metrics vary across projects with different contexts?*

3.1 Introduction

The history of software metrics predates software engineering. The first reported software metric is the number of lines of code (LOC) which was used in the mid-1960's to assess the productivity of programmers [55]. Since then, a large number of metrics have been proposed [25, 52, 115, 119], and extensively used in software engineering activities, e.g., defect prediction, effort estimation and software benchmarks[1].

Since software systems are developed in different environments, for various purposes, and by teams with diverse organizational cultures, we believe that context factors, such as

[1]A benchmark is generally defined as "a test or set of tests used to compare the performance of alternative tools or techniques" [178]. In this study, we refer to "benchmark" as a set of metric-based evaluations of software maintainability.

application domain, programming language, and the number of downloads, should be taken into account, when using metrics in software engineering activities (e.g., [46]). It can be problematic [50] to apply metric-based benchmarks derived from one context to software systems in a different context, e.g., applying benchmarks derived from small-size software systems to assess the maintainability of large-size software systems. COCOMO II[2] model supports a number of attribute settings (e.g., the complexity of product) to fine tune the estimation of the cost and system size (i.e., source lines of code). However, to the best of our knowledge, no study provides empirical evidence on how contexts affect the aforementioned metric-based models. The context is overlooked in most existing approaches for building metric-based benchmarks [4, 12, 14, 56, 79, 170, 200].

This preliminary study aims to understand if the distributions of metrics do, in fact, vary with contexts. Considering the availability and understandability of context factors and their potential impact on the distribution of metric values, we decide to study seven context factors: application domain, programming language, age, lifespan, system size, the number of changes, and the number of downloads. Since system size strongly correlates to the number of changes, we only examine the number of changes of the two factors.

In this thesis, we select 320 nontrivial (i.e., both size and lifespan are greater than 25% of the population) software systems from SourceForge[3]. These software systems are randomly sampled from nine popular application domains. For each software system, we calculate 39 metrics commonly used to assess software maintainability. To better understand the impact on different aspects of software maintainability, we further classify the 39 metrics into six categories (i.e., complexity, coupling, cohesion, abstraction, encapsulation and documentation) based on Zou and Kontogiannis [212]'s work.

[2]http://sunset.usc.edu/csse/research/COCOMOII
[3]http://www.sourceforge.net

We investigate the following two research questions:

(RQ1) *What context factors impact the distribution of the values of software maintainability metrics?*

All six context factors (i.e., application domain, programming language, age, lifespan, the number of changes, and the number of downloads) affect the distribution of the values of 51% of metrics (i.e., 20 out of 39). The most influential context factor is the programming language, since it impacts the distribution of the values of 90% of metrics (i.e., 35 out of 39).

(RQ2) *What guidelines can we provide to benchmark software maintainability metrics?*

When obtaining thresholds of metric values to create benchmarks, software systems with similar context factors should be grouped together. We suggest to divide all software systems into 13 distinct groups, including 1) five groups along application domain (i.e., G_{build}, G_{games}, G_{frame}, $G_{build;codegen}$, and $G_{comm;network}$); 2) five groups along programming language (i.e., G_{c}, G_{cpp}, $G_{c\#}$, G_{java}, and G_{pascal}); and 3) three groups along the number of changes (i.e., G_{lowNC}, $G_{moderateNC}$, and G_{highNC}).

Chapter organization. Context factors are discussed in Section 3.2. We describe the experimental setup of our study in Section 3.3 and report our case study results in Section 3.4. Threats to validity of our work are discussed in Section 3.5. In Section 3.6, we present the summary of this Chapter.

3.2 Context Factors

Open source software systems are characterized by Capiluppi *et al.* [28] using 12 context factors: age, application domain, programming language, size (in terms of physical occupation), the number of developers, the number of users, modularity level, documentation level, popularity, status, success of project, and vitality. Considering not all software systems provide information for these factors, we decide to investigate five commonly available context factors: application domain, programing language, age, system size (we redefined it as the total lines of code), and the number of downloads (measured using average monthly downloads). Since software metrics are also affected by software evolution [95], we study two additional context factors: lifespan and the number of changes. The selected context factors are described as follows:

1) **Application domain (AD)** describes the type of software systems (e.g., framework and game). In general, software systems designed as *frameworks* may contain more classes than other types of software systems.

2) **Programming language (PL)** describes the nature of programming paradigms. Generally speaking, software systems written in Java may have deeper inheritance tree than C++, as C++ supports both object oriented programming and structural programming.

3) **Age (AG)** is the time duration after creating a software system. As software development techniques evolve fast, older software systems might be more difficult to maintain than newly created software systems.

4) **Lifespan (LS)** describes the time duration of development activities in the life of a software system. Software systems developed over a long period of time might be harder to maintain than software systems developed over a shorter time period, due to

accumulated features.

5) **System size (SS)** is the total lines of code of a software system. Small software systems might be easier to maintain than large ones.

6) **Number of changes (NC)** describes the total number of commits made to a software system. It might be more difficult to maintain heavily-modified software systems than lightly-modified ones.

7) **Number of downloads (ND)** describes the external quality of a software system. It is of interest to find if popular software systems have better maintainability than less popular ones. In this study, the number of downloads is measured using the average monthly downloads which were collected directly from SourceForge.

3.3 Case Study Setup

This section presents the design of our case study.

3.3.1 Data Collection

Corpus. We use the SourceForge data initially collected by Mockus [125]. There are some updates after that work, and the new data collection was finished on February 05, 2010. The dataset contains 154,762 software systems. However, we find 97,931 incomplete software systems which contain fewer than 41 files, and an empty CVS repository has 40 files. There are 56,833 nontrivial software systems in total from SourceForge. FLOSSMole [85] is another data source, from where we download descriptions (i.e., application domain) of SourceForge software systems. Furthermore, we download latest application

domain information[4] and monthly download data[5] of studied software systems directly from SourceForge.

Sampling. Investigating all the 56,833 software systems requires a large amount of computation resources. For example, the snapshots of our selected 320 software systems occupy about 8 GB hard drive, and the computed metrics take more than 15 GB hard drive. The average time for computing metrics of one software system is 6 minutes. The bottleneck is the slow disk I/O, since we intensively access disks (e.g., to dump snapshots of source code, and to store metric values). Applying SSD or RAID storage or using RAM drive might eliminate this bottleneck. Yet our resource is limited to apply such solution at this moment. For a preliminary study, we perform stratified sampling of software systems by application domains to explore how context factors affect the distribution of metric values. Stratified sampling is to divide all software systems into subgroups before sampling. The limitation of stratified sampling is discussed in Section 3.5. Moreover, we plan to stratify by the remaining six factors in future. In this exploratory study, we pick nine popular application domains containing over 1,000 software systems. We conduct simple random sampling to select 100 software systems from each application domain and obtain 900 software systems in total. Yet there are only 824 different software systems, since a software system may be categorized into several application domains.

3.3.2 Factor Extraction

In this subsection, we describe our approach to extract each of the seven factors.

[4]http://sourceforge.net/projects/ProjectName (NOTE: the ProjectName needs to be substituted by the real project name, e.g., a2ixlibrary)

[5]http://sourceforge.net/projects/ProjectName/files/stats/json?start_date=1990-01-01&end_date=2012-12-31 (NOTE: the ProjectName needs to be substituted by the real project name, e.g., gusi)

1) **Application domain (AD).** We extract the application domain of each software system using the data collected in June 2008 by FLOSSMole [85]. We rank all application domains using the number of software systems and pick nine popular application domains: Build Tools, Code Generators, Communications, Frameworks, Games/Entertainment, Internet, Networking, Software Development (excluding Build Tools, Code Generators, and Frameworks), and System Administration. We replace sub-domains, if exist, by their parent application domain.

2) **Programming language (PL).** In this study, we only investigate software systems that are mainly written in C, C++, C#, Java, or Pascal. For each software system, we dump the latest snapshot, and determine the main programing language by counting the total number of files per file type (i.e., *.c, *.cpp, *.cxx, *.cc, *.cs, *.java, and *.pas).

3) **Age (AG).** For each software system, we compute the age using the date of the first CVS commit. In the sampled 824 software systems, the oldest software system[6] was created on November 02, 1996, and the latest software system[7] was created on May 28, 2008.

4) **Lifespan (LS).** For each software system, we compute the lifespan by computing the intervals between the first and the last CVS commits. The quantiles of lifespan in a unit of day in the sampled 824 software systems are: 0 (minimum), 51 (25%), 338 (median), 930 (75%), and 4,038 (maximum).

5) **System size (SS).** For each software system, we count the total lines of code from the latest snapshot. The quantiles of the total lines of code in the sampled 824 software systems are: 0 (minimum), 1,124 (25%), 3,955 (median), 14,945 (75%), and 2,792,334 (maximum).

[6] gusi, http://sourceforge.net/projects/gusi
[7] pic-gcc-library, http://sourceforge.net/projects/pic-gcc-library

6) **Number of changes (NC).** For each software system, we count the total number of commits from the whole history. The quantiles of the number of changes in the sampled 824 software systems are: 12 (minimum), 123 (25%), 413 (median), 1,142 (75%), and 94,853 (maximum).

7) **Number of downloads (ND).** For each software system, we first sum up all the monthly downloads to get the total number of downloads, and search the first and the last month with at least one download to determine the downloading period. We divide the total downloads by the total number of months of the downloading period to obtain the average monthly downloads. The quantiles of the average monthly downloads in the sampled 824 software systems are: 0 (minimum), 0 (25%), 6 (median), 16 (75%), and 661,247 (maximum).

3.3.3 Data Cleanliness

We observe that some of the 824 software systems are incomplete, hence we perform the following steps to further clean the data set.

(F1) Lifespan (LS). The 25% quantile of the lifespan of the 824 software systems is 51 days. We suspect that software systems with lifespans less than the 25% quantile are never finished or are just used as prototypes. After manually checking such software systems, we find that most of them are incomplete with very few commits. We exclude such software systems from our study. The number of subject systems drops from 824 to 618.

(F2) System size (SS). We manually check the software systems with lines of code less than the 25% quantile (i.e., 1,124) of the 824 software systems. We find that most

of such software systems are incomplete (e.g., vcgenv[8]), or mainly written in other languages (e.g., jajax[9] is written mainly in JavaScript). We exclude such software systems from our study. The number of subject systems drops from 618 to 506.

(F3) Programming language (PL). We filter out software systems that are not mainly written in C, C++, C#, Java, or Pascal. The number of subject systems drops from 506 to 478.

(F4) Number of downloads (ND). Some of the remaining 478 software systems have no downloads. It might because that such software systems are still incomplete to be used, or they are absolutely useless software. We exclude software systems without downloads from our study. The number of subject systems drops from 478 to 390.

(F5) Application domain (AD). A software system might be categorized into several application domains. The combinations of multiple application domains are considered as different application domains from single application domains. The 75% quantile of the number of software systems of all single and combined application domains is seven. We exclude combined application domains which have less than seven software systems, and yield six combined application domains. The number of subject systems drops from 390 to 323.

The collection date of the application domain is not the same as the date of collecting source code. To verify whether the application domains of the 323 software systems remain the same as time elapses, we checked the latest application domain information directly downloaded from SourceForge in April 2013. We find that only three

[8]http://sourceforge.net/projects/vcgenv
[9]http://sourceforge.net/projects/jajax

Table 3.1: The Spearman correlations among four context factors: age, lifespan, system size, and the number of changes.

Context factor	Lifespan	System size	Number of changes
Age	0.35	0.06	0.14
Lifespan	-	0.25	0.46
System Size	-	-	0.67

software systems (i.e., cdstatus, g3d-cpp, and satyr[10]) have changed their application domains, indicating that application domains collected in June 2008 are adequate. The application domain of *g3d-cpp* is removed, and the application domains of *cdstatus* and *satyr* are changed to Audio/Video. We exclude the three software systems from our study. The number of subject systems drops from 323 to 320.

The context factors like age, lifespan, system size, and the number of changes seem to be strongly related. Hence, we compute Spearman correlation among these context factors of the 320 software systems. As shown in Table 3.1, system size strongly correlates to the number of changes. Therefore, we choose to examine the number of changes only.

3.3.4 Stratifying Software Systems

When investigating the impact of application domain on the distribution of metric values, we break down the 320 software systems into 15 groups based on 15 studied application domains, as shown in Table 3.2. As there are five programming languages under study, we separate the 320 software systems into five groups. When investigating the impact of the other four context factors on the distribution of metric values, we divide the 320 software systems into three groups, respectively, in the following way: 1) low (below or at the 25% quantile); 2) moderate (above the 25% quantile and below or at the 75% quantile); and 3)

[10]http://sourceforge.net/projects/ProjectName (NOTE: the ProjectName needs to be substituted by the real project name, e.g., cdstatus)

Table 3.2: The number of software systems per group divided by each context factor.

Context factor	Group (The number of systems)			
Application domain (AD)	G_{build} (31), G_{games} (49), $G_{sysadmin}$ (29), $G_{games;internet}$ (7),	$G_{codegen}$ (26), $G_{internet}$ (19), $G_{build;codegen}$ (14), $G_{internet;swdev}$ (9),	G_{comm} (23), $G_{network}$ (16) $G_{comm;internet}$ (13), $G_{swdev;sysadmin}$ (7).	G_{frame} (29), G_{swdev} (41), $G_{comm;network}$ (7),
Programming language (PL)	G_{c} (57) G_{pascal} (14)	G_{cpp} (85)	$G_{c\#}$ (18)	G_{java} (146)
Age (AG)	G_{lowAG} (80)	$G_{moderateAG}$ (160)	G_{highAG} (80)	
Lifespan (LS)	G_{lowLS} (80)	$G_{moderateLS}$ (160)	G_{highLS} (80)	
Number of changes (NC)	G_{lowNC} (80)	$G_{moderateNC}$ (160)	G_{highNC} (80)	
Number of downloads (ND)	G_{lowND} (90)	$G_{moderateND}$ (150)	G_{highND} (80)	

high (above the 75% quantile). The 25% quantile of lifespan (respectively the number of changes and the number of monthly downloads) is: 287 days (respectively 364 and 6). The 75% quantile of lifespan (respectively the number of changes and the number of monthly downloads) is: 1,324 days (respectively 2,195 and 38). The detailed groups are shown in Table 3.2.

3.3.5 Metrics Computation

In this study, we select 39 metrics related to the five quality attributes (i.e., modularity, reusability, analyzability, modifiability, and testability) of software maintainability (as defined in ISO/IEC 25010 [88]). We further group the 39 metrics into six categories (i.e., complexity, coupling, cohesion, abstraction, encapsulation, and documentation) based on Zou and Kontogiannis [212]'s work. These categories can measure different aspects of software maintainability. For example, low complexity indicates high analyzability and modifiability; low coupling improves analyzability and reusability; high cohesion increases modularity and modifiability; high abstraction enhances reusability; high encapsulation implies high modularity; and documentation might contribute to analyzability, modifiability,

Table 3.3: List of metrics that characterize maintainability.

Category	Metric	Level
Complexity	Total Lines of Code (TLOC) Total Number of Files (TNF) Total Number of Classes (TNC) Total Number of Methods (TNM) Total Number of Statements (TNS)	Project
	Class Lines of Code (CLOC) Number of local Methods (NOM) [78] Number of Instance Methods (NIM) [163] Number of Instance Variables (NIV) [163] Weighted Methods per Class (WMC) [34]	Class
	Number of Method Parameters (NMP) McCabe Cyclomatic Complexity (CC) [119] Number of Possible Paths (NPATH) [163] Max Nesting Level (MNL) [163]	Method
Coupling	Coupling Factor (CF) [74]	Project
	Coupling Between Objects (CBO) [34] Information Flow Based Coupling (ICP) [25] Message Passing Coupling (MPC) [78] Response For a Class (RFC) [34]	Class
	Number of Method Invocation (NMI) Number of Input Data (FANIN) [163] Number of Output Data (FANOUT) [163]	Method
Cohesion	Lack of Cohesion in Methods (LCOM) [34] Tight Class Cohesion (TCC) [24] Loose Class Cohesion (LCC) [24] Information Flow Based Cohesion (ICH) [3]	Class
Abstraction	Number of Abstract Classes/Interfaces (NACI) Method Inheritance Factor (MIF) [74]	Project
	Number of Immediate Base Classes (IFANIN) [163] Number of Immediate Subclasses (NOC) [34] Depth of Inheritance Tree (DIT) [34]	Class
Encapsulation	Ratio of Public Attributes (RPA) Ratio of Public Methods (RPM) Ratio of Static Attributes (RSA) Ratio of Static Methods (RSM)	Class
Documentation	Comment of Lines per Class (CLC) Ratio Comments to Codes per Class (RCCC)	Class
	Comment of Lines per Method (CLM) Ratio Comments to Codes per Method (RCCM)	Method

and reusability [212]. Table 3.3 shows the metrics and their categories. Most metrics can be computed by a commercial tool, called Understand [164]. For the remaining metrics, we computed them by ourselves[11] using equations from the work of Aggarwal *et al.* [3].

[11]http://www.feng-zhang.com/replication/contextstudy

3.4 Case Study Results

In this section, we report and discuss the results of our study.

RQ1: ***What context factors impact the distribution of the values of software maintainability metrics?***

Motivations. This question is preliminary to the other question. It determines the number of pairwise tests in the other question. In this question, we determine if each factor impacts the distribution of each metric values, and should be considered in pairwise comparison.

Approach. To address this research question, we examine each factor individually. For each factor, we divide all software systems into non-overlapping groups as described in Section 3.3.3. To examine the overall impact of a factor f on a metric m, we test the following null hypothesis for the grouping based on factor f.

$H0_1$: *there is no difference in the distribution of metric values among all groups divided by factor* f.

To compare the distribution of metric m values among all groups, we apply Kruskal-Wallis test [173] using the 95% confidence level (i.e., p-value< 0.05). The Kruskal-Wallis test assesses whether two or more samples originate from the same distribution. It does not assume a normal distribution since it is a non-parametric statistical test. As we study six context factors and 39 metrics in total, we apply Bonferroni correction which adjusts the threshold p-value by dividing the number of tests (39×6=234 tests). If there is a statistically significant difference (i.e., p-value$< 0.05/234$=2.14e-04), we reject the null hypothesis and report that factor f impacts the distribution of metric m values.

Findings. We present p-value of Kruskal-Wallis test in Table 3.4. For each factor, statistically significant results indicate the impacting factors. In general, 51% of metrics (i.e., 20

Table 3.4: The p-values of Kruskal-Wallis test. (Note: "n.s." denotes non-statistical significance that means p-value is greater than $2.14e$-04, which equals to $0.05/39/6$).

Category	Metric	Application domain (AD)	Programming language (PL)	Age (AG)	Lifespan (LS)	Number of changes (NC)	Number of downloads (ND)
Complexity	TLOC	n.s.	n.s.	n.s.	1.94e-05	< 2.2e-16	n.s.
	TNF	n.s.	1.11e-05	n.s.	5.97e-06	< 2.2e-16	n.s.
	TNC	3.41e-05	< 2.2e-16	n.s.	n.s.	8.05e-12	n.s.
	TNM	1.46e-04	< 2.2e-16	n.s.	n.s.	1.03e-11	n.s.
	TNS	n.s.	n.s.	n.s.	6.26e-06	< 2.2e-16	n.s.
	CLOC	< 2.2e-16	< 2.2e-16	< 2.2e-16	1.37e-14	n.s.	< 2.2e-16
	NOM	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16
	NIM	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16
	NIV	< 2.2e-16	< 2.2e-16	< 2.2e-16	5.27e-10	5.76e-08	5.27e-11
	WMC	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16
	NMP	< 2.2e-16	< 2.2e-16	< 2.2e-16	2.93e-07	< 2.2e-16	< 2.2e-16
	CC	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16
	NPATH	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16
	MNL	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16	2.22e-12	< 2.2e-16
Coupling	CF	n.s.	n.s.	n.s.	9.55e-05	< 2.2e-16	n.s.
	CBO	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16	n.s.	< 2.2e-16
	ICP	< 2.2e-16	< 2.2e-16	8.34e-11	< 2.2e-16	< 2.2e-16	n.s.
	MPC	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16	1.50e-04
	RFC	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16
	NMI	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16
	FANIN	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16
	FANOUT	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16
Cohesion	LCOM	< 2.2e-16	< 2.2e-16	< 2.2e-16	5.36e-10	n.s.	6.34e-15
	TCC	< 2.2e-16	< 2.2e-16	1.11e-14	< 2.2e-16	8.87e-14	< 2.2e-16
	LCC	< 2.2e-16	< 2.2e-16	8.09e-15	< 2.2e-16	5.68e-14	< 2.2e-16
	ICH	< 2.2e-16	< 2.2e-16	n.s.	< 2.2e-16	5.60e-12	n.s.
Abstraction	NACI	6.30e-05	< 2.2e-16	n.s.	n.s.	5.78e-09	n.s.
	MIF	n.s.	< 2.2e-16	n.s.	n.s.	n.s.	n.s.
	IFANIN	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16	4.75e-05	1.69e-04
	NOC	< 2.2e-16	< 2.2e-16	3.50e-08	8.20e-05	n.s.	1.95e-06
	DIT	< 2.2e-16	< 2.2e-16	< 2.2e-16	1.52e-14	n.s.	< 2.2e-16
Encapsulation	RPA	< 2.2e-16	n.s.	n.s.	n.s.	n.s.	n.s.
	RPM	< 2.2e-16	< 2.2e-16	< 2.2e-16	4.13e-06	< 2.2e-16	< 2.2e-16
	RSA	4.45e-16	3.26e-05	n.s.	3.38e-07	< 2.2e-16	4.85e-07
	RSM	< 2.2e-16	1.83e-08	1.41e-05	< 2.2e-16	n.s.	< 2.2e-16
Documentation	CLC	< 2.2e-16	< 2.2e-16	< 2.2e-16	4.80e-15	1.51e-11	< 2.2e-16
	RCCC	< 2.2e-16	< 2.2e-16	< 2.2e-16	n.s.	< 2.2e-16	< 2.2e-16
	CLM	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16
	RCCM	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16

out of 39) are impacted by all six factors. On the other hand, programming language, application domain, and lifespan are three most important factors since they impact over 80% of metrics (i.e., 35, 34, and 33 out of 39, respectively). Moreover, the number of changes,

age, and the number of downloads affect more than 70% of metrics (i.e., 31, 28 and 28 out of 39, respectively). The effect that any of the six factors has on the distribution of metric values can not be ignored.

Overall, we conclude that all six factors impact the distribution of the maintainability metric values. The programming language is the most influential factor, since it affects 90% of metrics (i.e., 35 out of 39). In the next research question, we examine types (of programming language, application domain) and levels (of lifespan, the number of changes, age, the number of downloads) in more detail to determine what factors should be considered when benchmarking software maintainability.

The distribution of the values of software metrics does vary across projects with different context factors. All six context factors impact the distribution of the values of 51% of metrics.

RQ2: ***What guidelines can we provide to benchmark software maintainability metrics?***

Motivations. In **RQ1**, we found that each of the six factors impact the values of at least 70% of metrics. However, considering all six factors when benchmarking software maintainability can result in a number of small groups, and can increase the possibility of duplicated benchmarks.

To effectively build benchmarks, we suggest to follow three steps: a) separate software systems into distinct groups to ensure each group contains only systems that share a similar context; b) apply existing approaches (e.g., [11, 56]) to build benchmarks of each group; c) for a given software system, determine which groups it belongs to and apply corresponding benchmarks to evaluate its maintainability. The results of several benchmarks can be aggregated when evaluating the maintainability of software.

In this research question, we aim to find the factors that impact the distribution of the maintainability metric values. Such factors can affect the derivation of the thresholds/ranges of the corresponding metrics. Moreover, we provide guidelines in splitting software systems into distinct groups for building benchmarks to measure software maintainability.

Approach. To address this research question, we divide all software systems into non-overlapping groups by each factor, respectively (as described in Section 3.3.3). If examining all possible interactions of all six context factors, the number of groups will be 6,075 ($= 15 \times 5 \times 3 \times 3 \times 3 \times 3$). However, the number of our subject systems is 320, then a large number of groups might be empty. Therefore, interactions of all six context factors are not investigated in this study.

To provide guidelines on how to group software systems for benchmarking maintainability metrics, we break down our analysis method into the following three steps:

(S1) Pairwise comparison of the distribution of metric values. For each impacting factor, we examine the impact in detail by comparing every pair of groups separated by the factor. To investigate the effects of factor f on metric m, we test the following null hypothesis for every pair of groups divided by factor f.

$H0_2$: *there is no difference in the distributions of metric values between the two groups of any pairs*.

To examine the difference in the distribution of the metric m values between every two groups, we apply Mann-Whitney U test [173] using the 95% confidence level (i.e., p-value<0.05). The Mann-Whitney U test assesses whether two independent distributions have equally large values. As a non-parametric statistical test, it does not

assume a normal distribution. Because we conduct multiple tests, we apply Bonferroni correction to adjust the threshold p-value based on the findings of **RQ1**. Table 3.5 presents the corrected threshold p-values. If the difference is statistically significant, we reject the null hypothesis $H0_2$ and claim that factor f is important to metric m.

(S2) Quantifying the importance of the difference. For any comparison exhibiting a statistically significant difference, we further compute the corresponding effect size to quantify the importance of the difference. We apply Cliff's δ as effect size [160] to quantify the importance of the difference, since Cliff's δ is reported [160] to be more robust and reliable than Cohen's d [38]. As Cliff's δ estimates non-parametric effect sizes, it makes no assumptions of a particular distribution. Cliff's δ represents the degree of overlap between two sample distributions [160]. It ranges from -1 (if all selected values in the first group are larger than the second group) to +1 (if all selected values in the first group are smaller than the second group). It is zero when two sample distributions are identical [36].

(S3) Interpreting the effect sizes. Effect sizes are used to split software systems. Cliff's δ is mapped to Cohen's standards via the percentage of non-overlap as shown in Table 3.6 [160]. Cohen [39] states that a medium effect size represents a difference likely to be visible to a careful observer, while a large effect is noticeably larger than medium. In this study, we choose the large effect size as the threshold. If the effect size is large, we conclude that the corresponding factor f has a large impact on the distribution of the corresponding metric m. As a result, we suggest that software systems should be split into different groups based on factor f when benchmarking metric m. Otherwise, all software systems can be put in the same group along with factor f.

Table 3.5: List of the threshold p-values after Bonferroni correction. The number of pairwise tests required for each context factor is determined by $C(n,2)$, which denotes the number of 2-combinations from a given set S of n elements. The number of groups by factors AD, PL, AG, LS, NC, and ND are: 15, 5, 3, 3, 3, and 3, respectively. Hence, the number of pairwise tests are: $C(15,2) = 105$, $C(5,2) = 10$, $C(3,2) = 3$, $C(3,2) = 3$, $C(3,2) = 3$, and $C(3,2) = 3$, respectively.

Metric	The number of pairwise tests by each factor						The total number of	The corrected threshold
	AD	PL	AG	LS	NC	ND	pairwise tests	p-value
TLOC	0	0	0	3	3	0	6	$8.33e-03$
TNF	0	10	0	3	3	0	16	$3.13e-03$
TNC	105	10	0	0	3	0	118	$4.24e-04$
TNM	105	10	0	0	3	0	118	$4.24e-04$
TNS	0	0	0	3	3	0	6	$8.33e-03$
CLOC	105	10	3	3	0	3	124	$4.03e-04$
NOM	105	10	3	3	3	3	127	$3.94e-04$
NIM	105	10	3	3	3	3	127	$3.94e-04$
NIV	105	10	3	3	3	3	127	$3.94e-04$
WMC	105	10	3	3	3	3	127	$3.94e-04$
NMP	105	10	3	3	3	3	127	$3.94e-04$
CC	105	10	3	3	3	3	127	$3.94e-04$
NPATH	105	10	3	3	3	3	127	$3.94e-04$
MNL	105	10	3	3	3	3	127	$3.94e-04$
CF	0	0	0	3	3	0	6	$8.33e-03$
CBO	105	10	3	3	0	3	124	$4.03e-04$
ICP	105	10	3	3	3	0	124	$4.03e-04$
MPC	105	10	3	3	3	3	127	$3.94e-04$
RFC	105	10	3	3	3	3	127	$3.94e-04$
NMI	105	10	3	3	3	3	127	$3.94e-04$
FANIN	105	10	3	3	3	3	127	$3.94e-04$
FANOUT	105	10	3	3	3	3	127	$3.94e-04$
LCOM	105	10	3	3	0	3	124	$4.03e-04$
TCC	105	10	3	3	3	3	127	$3.94e-04$
LCC	105	10	3	3	3	3	127	$3.94e-04$
ICH	105	10	0	3	3	0	121	$4.13e-04$
NACI	105	10	0	0	3	0	118	$4.24e-04$
MIF	0	10	0	0	0	0	10	$5.00e-03$
IFANIN	105	10	3	3	3	3	127	$3.94e-04$
NOC	105	10	3	3	0	3	124	$4.03e-04$
DIT	105	10	3	3	0	3	124	$4.03e-04$
RPA	105	0	0	0	0	0	105	$4.76e-04$
RPM	105	10	3	3	3	3	127	$3.94e-04$
RSA	105	10	0	3	3	3	124	$4.03e-04$
RSM	105	10	3	3	0	3	124	$4.03e-04$
CLC	105	10	3	3	3	3	127	$3.94e-04$
RCCC	105	10	3	0	3	3	124	$4.03e-04$
CLM	105	10	3	3	3	3	127	$3.94e-04$
RCCM	105	10	3	3	3	3	127	$3.94e-04$

Table 3.6: Mapping Cliff's δ to Cohen's standards.

Cliff's δ	% of Non-overlap	Cohen's d	Cohen's Standards
0.147	14.7%	0.20	small
0.330	33.0%	0.50	medium
0.474	47.4%	0.80	large

Table 3.7: Cliff's δ and p-value of Mann-Whitney U test of every statistically significant different pairs of groups divided by factors. (investigation of *complexity* metrics).

Metric	**Factor**	**Group1**	**Group2**	**Cliff's δ**
TLOC	NC	G_{lowNC}	G_{highNC}	0.498
TNF	NC	G_{lowNC}	$G_{moderateNC}$	0.573
			G_{highNC}	0.639
		$G_{moderateNC}$	G_{highNC}	0.513
TNC	AD	G_{frame}	$G_{network}$	–0.519
			$G_{comm;network}$	–0.759
	PL	G_{c}	$G_{c\#}$	0.596
			G_{java}	0.667
		G_{pascal}	G_{cpp}	0.560
			$G_{c\#}$	0.729
			G_{java}	0.885
	NC	G_{lowNC}	$G_{moderateNC}$	0.476
			G_{highNC}	0.552
TNM	AD	G_{frame}	$G_{network}$	–0.599
	PL	G_{c}	$G_{c\#}$	0.614
			G_{java}	0.591
		G_{pascal}	G_{cpp}	0.683
			$G_{c\#}$	0.758
			G_{java}	0.832
	NC	G_{lowNC}	G_{highNC}	0.560
TNS	NC	G_{lowNC}	G_{highNC}	0.541

Findings. To better understand the impact of context factors on different aspects of software maintainability, we report our findings along each of the six categories of metrics (i.e., complexity, coupling, cohesion, abstraction, encapsulation and documentation), respectively.

1) Complexity. As shown in Table 3.7, the factor impacting TLOC, TNF, and TNS is the number of changes. The factors impacting TNC and TNM are application domain, programming language, and the number of changes. Overall, the distributions of metric values in

Table 3.8: Cliff's δ and p-value of Mann-Whitney U test of every statistically significant different pairs of groups divided by factors. (investigation of *coupling* metrics).

Metric	Factor	Group1	Group2	Cliff's δ
CBO	AD	$G_{comm;network}$	$G_{build;codegen}$	0.482
	PL	G_{pascal}	$G_{c\#}$	0.486
			G_{java}	0.550
RFC	*AD*	$G_{comm;network}$	$G_{network}$	0.524
			$G_{internet}$	0.578
			$G_{sysadmin}$	0.492
			$G_{codegen}$	0.764
			G_{frame}	0.620
			G_{build}	0.703
			G_{swdev}	0.847
			$G_{games;internet}$	0.643
			$G_{internet;swdev}$	0.647
			$G_{comm;internet}$	0.642
		$G_{build;codegen}$	$G_{comm;network}$	−0.923
			$G_{swdev;sysadmin}$	−0.531
	PL	$G_{c\#}$	G_{java}	0.666
CF	NC	G_{lowNC}	G_{highNC}	−0.554
NMI	PL	G_{java}	$G_{c\#}$	−0.516
			G_{pascal}	−0.516

the *complexity* category are strongly impacted by three context factors: application domain, programming language, and the number of changes.

2) Coupling. As shown in Table 3.8, the factors impacting CBO and RFC are application domain and programming language. The factor impacting CF (respectively NMI) is the number of changes (respectively programming language). Overall, the distributions of metric values in the *coupling* category are strongly impacted by three context factors: application domain, programming language, and the number of changes.

3) Cohesion. Overall, the distributions of metric values in the *cohesion* category are strongly impacted by application domain only. In particular, we only observe large difference (i.e., Cliff's $\delta = 0.552$) in the distribution of LCOM between software systems of groups $G_{network}$ and $G_{comm;network}$.

4) Abstraction. As shown in Table 3.9, the factor impacting NACI and MIF is programming

Table 3.9: Cliff's δ and p-value of Mann-Whitney U test of every statistically significant different pairs of groups divided by factors. (investigation of *abstraction* metrics).

Metric	Factor	Group1	Group2	Cliff's δ
NACI	PL	G_{java}	G_{pascal}	−0.773
IFANIN	AD	$G_{comm;network}$	$G_{network}$	0.794
			$G_{internet}$	0.751
			$G_{sysadmin}$	0.657
			G_{comm}	0.776
			$G_{codegen}$	0.780
			G_{frame}	0.748
			G_{build}	0.738
			G_{swdev}	0.736
			G_{games}	0.598
			$G_{games;internet}$	0.620
			$G_{swdev;sysadmin}$	0.828
			$G_{internet;swdev}$	0.704
			$G_{comm;internet}$	0.745
			$G_{build;codegen}$	0.824
		G_{games}	$G_{swdev;sysadmin}$	0.486
	PL	G_{java}	G_{cpp}	−0.514
			G_{pascal}	−0.708
DIT	AD	$G_{comm;network}$	$G_{network}$	0.820
			$G_{internet}$	0.861
			$G_{sysadmin}$	0.772
			G_{comm}	0.907
			$G_{codegen}$	0.954
			G_{frame}	0.899
			G_{build}	0.870
			G_{swdev}	0.962
			G_{games}	0.839
			$G_{games;internet}$	0.746
			$G_{swdev;sysadmin}$	0.910
			$G_{internet;swdev}$	0.915
			$G_{comm;internet}$	0.910
		$G_{build;codegen}$	$G_{sysadmin}$	−0.549
			$G_{comm;network}$	−0.983
			$G_{games;internet}$	−0.517
MIF	PL	G_{java}	G_{cpp}	−0.777
			$G_{c\#}$	−0.849
			G_{pascal}	−0.666
		G_{cpp}	$G_{c\#}$	0.657

language. The factor impacting DIT is application domain. The factors impacting IFANIN are application domain and programming language. Overall, the distributions of metric values in the *abstraction* category are strongly impacted by application domain and programming language.

Table 3.10: Cliff's δ and p-value of Mann-Whitney U test of every statistically significant different pairs of groups divided by factors. (investigation of *encapsulation* metrics).

Metric	Factor	Group1	Group2	Cliff's δ
RPA	AD	G_{build}	$G_{internet}$	0.483
			$G_{codegen}$	0.558
			G_{frame}	0.619
			G_{swdev}	0.576
			G_{games}	0.682
			$G_{swdev;sysadmin}$	0.543
			$G_{internet;swdev}$	0.550
			$G_{comm;internet}$	0.505
			$G_{build;codegen}$	0.527
RSM	AD	$G_{comm;network}$	$G_{comm;internet}$	0.710

Table 3.11: Cliff's δ and p-value of Mann-Whitney U test of every statistically significant different pairs of groups divided by factors. (investigation of *documentation* metrics).

Metric	Factor	Group1	Group2	Cliff's δ
RCCC	AD	$G_{build;codegen}$	$G_{network}$	−0.513
	PL	G_{java}	G_{pascal}	−0.611

5) Encapsulation. As shown in Table 3.10, the factor impacting RPA and RSM is application domain. Overall, the distributions of metric values in the *encapsulation* category are strongly impacted by application domain only.

6) Documentation. As shown in Table 3.11, the factors impacting RCCC are application domain and programming language. Overall, the distributions of metric values in the *documentation* category are strongly impacted by application domain and programming language.

In general, application domain, programming language, and the number of changes strongly impact the distribution of maintainability metric values.

Table 3.12: Guidelines to partition software systems for building metric based benchmarks.

Metric Category	Factor	Group
Complexity	AD	G_{frame} and others
	PL	G_c, G_{pascal} and others
	NC	G_{lowNC}, $G_{moderateNC}$, and G_{highNC}
Coupling	AD	$G_{comm;network}$, $G_{build;codegen}$, and others
	PL	G_{pascal}, G_{java}, and others
	NC	G_{lowNC}, $G_{moderateNC}$, and G_{highNC}
Cohesion	AD	$G_{comm;network}$, and others
Abstraction	AD	$G_{comm;network}$, G_{games}, $G_{build;codegen}$, and others
	PL	G_{java}, G_{cpp}, and others
Encapsulation	AD	G_{build}, $G_{comm;network}$, and others
Documentation	AD	$G_{build;codegen}$, and others
	PL	G_{java}, and others

3.4.1 Guidelines for Benchmarking Maintainability Metrics.

Based on our findings, we provide guidelines to create benchmarks of maintainability metrics as follows.

(G1) When benchmarking the 39 metrics, we suggest to partition software systems into 13 groups: 1) five groups along application domain (i.e., G_{build}, G_{games}, G_{frame}, $G_{build;codegen}$, and $G_{comm;network}$); 2) five groups along programming language (i.e., G_c, G_{cpp}, $G_{c\#}$, G_{java}, and G_{pascal}); and 3) three groups along the number of changes (i.e., G_{lowNC}, $G_{moderateNC}$, and G_{highNC}).

(G2) When benchmarking metrics from a particular category, we provide detailed suggestions in Table 3.12.

Furthermore, our approach can be applied to other software metrics and other software systems for generating guidelines on building benchmarks of such software metrics.

3.5 Threats to Validity

We now discuss the threats to validity of our study following common guidelines [199].

Threats to conclusion validity concern the relation between the treatment and the outcome. Our conclusion validity threats are mainly due to sampling errors. Since stratified sampling was performed only along application domain, sampled software systems may not well represent along other five factors. Some differences along these factors, thus, may not be detected, and the detected differences are likely to be only a subset of differences. We plan to stratify along other factors.

Threats to internal validity concern our selection of subject systems and analysis methods. We randomly sample 320 software systems from SourceForge, some of the findings might be specific to software systems hosted on SourceForge. Future studies should consider using software systems from other hosts, and even commercial software systems.

Threats to external validity concern the possibility to generalize our results. Some of the findings might not be directly applicable to different software systems. Yet our approach can be applied to find guidelines for benchmarking maintainability of different open source and commercial software systems.

Threats to reliability validity concern the possibility of replicating this study. We attempt to provide all the necessary details to replicate our study. SourceForge is publicly available to obtain the same data. We make our data and R script available[12] as well.

3.6 Chapter Summary

In this work, we perform a large scale empirical study to investigate how the six context factors affect the distribution of maintainability metric values. We apply statistical methods

[12]http://www.feng-zhang.com/replication/contextstudy

(i.e., Kruskal-Wallis test, Mann-Whitney U test and Cliff's δ effect size) to analyze 320 software systems, and provide empirical evidence of the impact of context factors on the distribution of maintainability metric values.

Our results show that all six context factors impact the distribution of the values of 51% of metrics. The most influential factors are application domain, programming language, and the number of changes. Based on our findings, we further provide guidelines on how to group software systems according the six context factors. We expect our findings to help software benchmarking and other software engineering methods using the 39 software maintainability metrics.

Part III

Data Pre-Processing

CHAPTER 4

Transformation of Software Metrics

Key Question

? *Does the transformation of software metrics impact the performance of defect prediction models?*

4.1 Introduction

Earlier studies (e.g., [41, 116, 205]) report that software metrics rarely follow a normal distribution, but a power-law distribution, which threatens the fitness of prediction models to provide an accurate prediction [37]. In the literature of defect prediction, researchers widely apply log and rank transformations to improve the normality of software metrics (e.g., [43, 93, 123, 180, 202]). Log transformation is basically a mathematical operation that replaces the original metric values by their logarithm, thus suits log-normal data (i.e., normally distributed data after log transformation). Rank transformation substitutes the original metric values with their ranks.

Despite the success in improving the normality of software metrics, the aforementioned transformations fail to consistently improve the performance of defect prediction models in a within-project setting [93]. However, to the best of our knowledge, the impact that such transformations have on the performance of defect prediction models has not been thoroughly investigated in a cross-project setting.

Transformations, if obtained from both the training and target projects, have the potential to mitigate the heterogeneity between the training and target projects. For instance, our previous work [202] successfully implements the context-aware rank transformation towards generalizing defect prediction models. In addition, Ma *et al.* [118] propose to transform the training project based on the statistical characteristics learnt from the target project. Nam *et al.* [135] apply a transfer component analysis (TCA) approach to transform both the training and target projects. Although both approaches on average significantly improve the performance of cross-project predictions, it is unclear how to choose an appropriate transformation for a particular pair of training and target projects. Jiang *et al.* [93] show that the benefit of transformations varies with modelling techniques on the same dataset.

Nonetheless, different transformations retain the information of the original data from various perspectives, especially in the cross-project setting. Therefore, in this study, we investigate if different transformations indeed retain distinct characteristics of software metrics that is beneficial to cross-project defect prediction.

We perform experiments using three publicly available data sets, i.e., AEEEM [44], ReLink [196], and PROMISE [96]. First, we examine if transformations have the same ability to improve the normality of software metrics. Besides log and rank transformations,

we study the Box-Cox transformation [22] that represents a family of power transformations (e.g., log transformation) but has not been investigated in studies on defect prediction. Second, we study if different transformations cause distinct predictions on the same file in the cross-project setting.

Furthermore, we propose an approach to integrate predictions by multiple models, with each model built using a single transformation. In particular, the weight of each model is determined by its accuracy in predicting defective instances. Finally, we investigate if applying multiple transformations can improve the performance of cross-project defect prediction models. We study the following four research questions:

(RQ1) *Are log, Box-Cox, and rank transformations equally effective in increasing the normality of software metrics?*

It is not a surprise that all three transformations have similar ability to improve the normality of software metrics. The transformed metrics approximately follow a normal distribution.

(RQ2) *Do different transformations result in distinct predictions in cross-project defect prediction models?*

In general, there is no difference in the performance (i.e., precision, recall, false positive rate, balance, F-measure and AUC values) among the three prediction models built with each of the three transformations. However, the results of McNemar's test indicate that the three prediction models judge differently about the defect-proneness of each file. In other words, a defective file can be captured by some models, and overlooked by other models.

(RQ3) *Can our approach improve the performance of cross-project defect prediction models?*

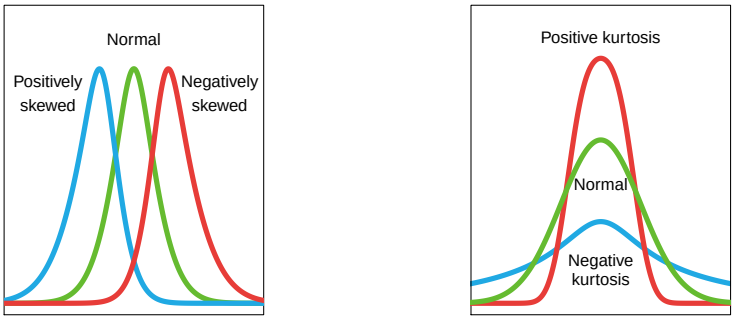
Our approach integrates the judgement received from the three independent prediction models that are built from each transformation, in order to make a final decision on the defect-proneness of a file. The results show that our approach can provide statistically significantly better performance. For instance, the average F-measures for AEEEM, ReLink, and PROMISE datasets are increased by 49%, 47%, and 22%, respectively. The average AUC values are improved by 2.9%, 5.8%, and 3.7%, respectively.

(RQ4) *Does our approach work well for other classifiers?*

We examine the generalizability of our approach using six other classifiers (e.g., Naive Bayes, decision tree, and random forest), since different classifiers are reported to prefer different transformations [93]. We find that our approach can achieve statistically significant improvement in general.

As a summary, varied transformations retain distinct characteristics of metrics. By utilizing the three transformation methods, our approach successfully improves the performance of cross-project defect prediction models with little overhead that is introduced by adding merely mathematical operations.

Chapter organization. Section 4.2 presents background on the three studied transformation methods. The experimental setup is presented in Section 4.3. Our motivation study is described in Section 4.4. Our approach and evaluation are presented in Section 4.5 and Section 4.6, respectively. The threats to validity of our work are discussed in Section 4.7. We summarize the chapter in Section 4.8.



(a) Skewness (b) Kurtosis

Figure 4.1: The illustration of skewness and kurtosis in a distribution.

4.2 Background on Transformation Methods

In this section, we describe two common measurements of data normality, and present the details of the three studied transformation methods.

4.2.1 Normality Measurements

Skewness and kurtosis are two widely applied measurements of data normality. We compute these two measurements to measure the normality of software metrics, using the *R* functions *skewness* and *kurtosis* in the *R*[1] package *e1071*[2]. Skewness and kurtosis are described as follows.

a) Skewness measures the degree of symmetry in the probability distribution of the values of a software metric. The value of skewness can be positive that indicates a long tail to the right, or negative that indicates a long tail to the left. Positively and negatively skewed distributions are illustrated in Figure 4.1 (a). The ideal value of skewness ranges from -0.80 to 0.80 [141].

[1]https://www.r-project.org
[2]https://cran.r-project.org/web/packages/e1071

b) Kurtosis measures the "peakness" (e.g., the width of the peak) in the probability distribution of the values of a software metric. The value of kurtosis can be positive that indicates a more acute peak, or negative that indicates a lower and wider peak. Positive and negative kurtosis are illustrated in Figure 4.1 (b). The ideal value of kurtosis is zero.

4.2.2 Log Transformation

Log transformation is a mathematical operation that computes the logarithm (mostly the natural logarithm) of software metrics to replace the original values. Log transformation is widely used in building software defect prediction models (e.g., [123, 180]).

Log transformation can only transform numerical values that are greater than zero, due to the definition of the function "ln(x)". To deal with zero values, a constant is often added, such as "$ln(x + 1)$". An alternative solution is to replace all values under 0.000001 by 0.000001. In this study, we apply the following commonly used equation: $Log(x) = ln(x + 1)$, where x is the value of a software metric.

4.2.3 Rank Transformation

Rank transformation replaces the original values by their ranks. Rank transformation is recommended to deal with heavy-tailed distributions (i.e., have high kurtosis) [17, 100]. In the literature of defect prediction, Jiang *et al.* [93] observe that rank transformation can improve the performance of some classifiers (e.g., Naive Bayes). Moreover, rank transformation has been successfully applied to mitigate the heterogeneity of software metrics across projects in the cross-project setting [202].

In this study, we convert the original values of each metric into ten ranks, using every $10th$ quantile of the corresponding metric, as defined in Equation (4.1).

$$Rank(x) = \begin{cases} 1 & \text{if } x \in [0, Q_1] \\ k & \text{if } x \in (Q_{k-1}, Q_k],\ k \in \{2, \ldots, 9\} \\ 10 & \text{if } x \in (Q_9, +\infty) \end{cases} \tag{4.1}$$

*where Q_k is the k*10% quantile of the corresponding metric in* ***the union of the training and target projects****.*

4.2.4 Box-Cox Transformation

The Box-Cox transformation represents a family of power transformations, as defined in Equation (4.2). To the best of our knowledge, the Box-Cox transformation has not been explored in the defect prediction literature.

$$BoxCox(x, \lambda) = \begin{cases} \frac{x^\lambda - 1}{\lambda} & \text{if } \lambda \neq 0 \\ ln(x) & \text{if } \lambda = 0 \end{cases} \tag{4.2}$$

where x is the value of a metric, and λ is the configuration parameter of the Box-Cox transformation.

The parameter λ determines the concrete format of the Box-Cox transformation. For example, "$\lambda = 1.0$" means no transformation, "$\lambda = 0.5$" equals to the square root transformation, "$\lambda = 0.0$" represents the log transformation, and "$\lambda = -1.0$" indicates the inverse transformation. As such, the Box-Cox transformation is often used to transform variables that follow a power law distribution. The Box-Cox transformation is suggested to improve the variance homogeneity, increase the precision of estimation, and simplify models [168].

The parameter λ can be estimated from a sample of data points. In the context of cross-project prediction, λ is estimated from **both the training and target projects**. The details to apply the Box-Cox transformation in our study are presented as follows.

1) **Anchoring metric values to 1.0.** As suggested by Guo [69], we anchor the minimum value of a metric in a distribution at exactly 1.0 before applying the Box-Cox transformation. This treatment can increase the accuracy of the Box-Cox transformation [69]. We use the equation $\tilde{x} = x - min(x) + 1$, where x is the value of a software metric.

2) **Estimating the parameter λ.** The parameter λ is estimated for each metric independently, since different metrics rarely follow the same distribution. To ensure the same transformation is applied on both the training and target projects, as aforementioned, we estimate the parameter λ using the values of the corresponding metric from both sets.

   We estimate the parameter λ in an iterative process. First, we select a set of candidate λ values that range from -1.0 to 1.0. Second, we iterate the λ values from -1.0 towards 1.0 with a step of 0.1. At each iteration, we compute the skewness of transformed values. We select the λ value that leads to the minimum skewness of transformed values. The iterative process can be described using the following equation:

$$\widehat{\lambda} = \arg\min_{\lambda \in L} |skewness(BoxCox(\tilde{x}, \lambda))| \tag{4.3}$$
$$\tilde{x} \in X$$

   where L is a set of candidate λ values from -1.0 to 1.0 with a step by 0.1, and X is a vector of anchored metric values.

3) **Normalizing transformed values.** Normalization creates equal scales of software metrics, and is useful for classification algorithms [73, 135]. In this study, we choose the min-max method [73], since it can normalize values exactly into the range of $[0, 1]$. Based on the benefit of anchoring the minimum value to 1.0 [69], we slightly modify this method using the following Equation (4.4).

Table 4.1: List of selected projects in each dataset.

Data set	Projects	# of files	# of LOC	# of buggy files (%)
AEEEM	(A1) Eclipse JDT Core	997	224K	206 (21%)
	(A2) Equinox	324	40K	129 (40%)
	(A3) Apache Lucene	691	73K	64 (9.3%)
	(A4) Mylyn	1862	156K	245 (13%)
	(A5) Eclipse PDE UI	1497	147K	209 (14%)
ReLink	(R1) Apache HTTP Server	194	89K	98 (51%)
	(R2) OpenIntents Safe	56	8K	22 (39%)
	(R3) Zxing	399	27K	118 (30%)
PROMISE	(P1) Camel v1.6	965	113K	188 (19%)
	(P2) POI v3	442	129K	281 (64%)
	(P3) Velocity v1.6	229	57K	78 (34%)
	(P4) Xalan v2.7	909	429K	898 (99%)
	(P5) Xerces v1.4	588	141K	437 (74%)

$$Normalize(\widehat{x}) = \frac{\widehat{x} - \min_{\widehat{x} \in U}(\widehat{x})}{\max_{\widehat{x} \in U}(\widehat{x}) - \min_{\widehat{x} \in U}(\widehat{x})} + 1 \tag{4.4}$$

where $\widehat{x}$ is the transformed value by Equation (4.2) using $\tilde{x}$ and $\widehat{\lambda}$, and U is a set of $\widehat{x}$ from the union of the training and target projects.

4.3 Experimental Setup

In this section, we first describe our corpus. Then, we present classifiers to build cross-project defect prediction models, and performance measures used in this study.

4.3.1 Corpus

In this study, we choose three publicly available data sets: AEEEM [44], ReLink [196], and PROMISE [96]. These three data sets have been widely used for cross-project defect prediction. Table 4.1 presents the summary of the three data sets, and Table 4.2 shows the metrics used in our study for each data set. The diversity of the three data sets can help verify the generalizability of our approach.

Table 4.2: List of metrics used in this study for each data set. (NOTE: The name of each metric is presented as it is in each data set.)

Dataset	Metrics
AEEEM	cbo, fanOut, numberOfAttributes, numberOfLinesOfCode, numberOfAttributesInherited, numberOfPrivateAttributes numberOfPrivateMethods, rfc, wmc, linesAddedUntil, avgLinesAddedUntil, numberOfBugsFoundUntil
ReLink	AvgCyclomatic, AvgLineBlank, CountLineBlank, CountLineCode, CountLineCodeDecl, CountLineComment CountStmtDecl, MaxCyclomatic, MaxCyclomaticModified, SumCyclomatic, SumCyclomaticStrict, SumEssential
PROMISE	wmc, dit, cbo, rfc, lcom, ce, lcom3, loc, moa, mfa, amc, max_cc

The AEEEM data set was made by D'Ambros *et al.* [44], and has been used in earlier studies on cross-project defect prediction (e.g., [135]). In the AEEEM data set, there are two large projects (i.e., Mylyn and PDE) that contain the most number of files among the three data sets. As shown in Table 4.1, the ratio of defective files in AEEEM data set is relatively lower than the other two data sets. In particular, three projects (i.e., Lucene, Mylyn, and PDE) have the lowest ratio of defective files.

The ReLink data set was collected by Wu *et al.* [196], and the defect information in this data set was manually verified by Wu *et al.* [196]. Projects in the ReLink data set are relatively small. For instance, two projects (i.e., Apache HTTP Server and OpenIntents Safe) have the least number of files. As presented in Table 4.1, the ReLink data set has moderate ratio of defective files (i.e., 30% to 51%).

PROMISE data set is another widely used data set. The five selected projects from PROMISE data set were prepared by Jureczko and Madeyski [96]. Table 4.1 shows that projects in the PROMISE data set have moderate number of files (i.e., from 229 to 965), but have the most diverse ratio of defective files (i.e., 19% to 99%).

4.3.2 Classifiers for Defect Prediction

To build defect prediction models, there exist many classifiers. Different classifiers have their own advantages. For instance, logistic regression is easy to interpret and is widely

used [135]. Naive Bayes is robust for defect prediction using data with observable noise [102]. In this study, we choose to apply logistic regression to build cross-project defect prediction models.

Different transformations have varied impacts on the performance of different classifiers [93, 123, 180]. To examine the generalizability of our approach, we further perform sensitive analysis using six other classifiers, i.e., Naive Bayes, Bayes Net, IBk, J48, random forest, and random tree.

4.3.3 Performance Measures

In this study, we compute six commonly used measures (i.e., precision, recall, false positive rate, balance, F-measure, and AUC value) to evaluate the performance of cross-project prediction models. The first five measures can be calculated from the confusion matrix (see Table 4.3) that consists of the following four numbers:

1) True positive (TP) that counts the number of defective instances successfully predicted as defective instances;

2) True negative (TN) that calculates the number of non-defective instances correctly predicted as non-defective instances;

3) False positive (FP) that is the number of non-defective instances incorrectly predicted as defective instances;

4) False negative (FN) that measures the number of defective instances wrongly predicted as non-defective instances.

The details to compute these measures are described as follows:

Table 4.3: Confusion matrix in defect prediction studies.

Actual \ Predicted	defective	non-defective
defective	true positive (TP)	false negative (FN)
non-defective	false positive (FP)	true negative (TN)

Precision (prec) measures the ratio of correctly predicted defective instances. It is defined as: $prec = \frac{TP}{TP+FP}$.

Recall (pd) evaluates the proportion of defective instances that are predicted as defective instances. It is defined as: $pd = \frac{TP}{TP+FN}$.

False Positive Rate (fpr) captures the proportion of non-defective instances that are predicted as defective instances. It is defined as: $fpr = \frac{FP}{FP+TN}$.

Balance is proposed by Menzies *et al.* [123] to balance recall and false positive rate. It is defined as: $balance = 1 - \frac{\sqrt{(0-fpr)^2+(1-pd)^2}}{\sqrt{2}}$.

F-measure is the harmonic mean of precision and recall. It is defined as: $F\text{-}measure = \frac{2\times pd\times prec}{pd+prec}$.

The five aforementioned measures depend on the cut-off value, which is used to compute the four numbers TP, TN, FP, and FN. On the other hand, Area Under Curve (AUC) is the area under the receiver operating characteristics (ROC) curve. The ROC curve is created as a plot of the true positive rate over the false positive rate by varying the cut-off value, thus the AUC value is independent of the cut-off value. Rahman *et al.* [157] recommend to use AUC value to evaluate cross-project defect prediction models.

4.4 Motivation Study

In this section, we aim to find if the three transformation methods have different performances in the context of defect prediction. The investigation is performed from two perspectives:

1) if they can equally improve the normality of software metrics.

2) if cross-project defect prediction models built using each of the transformation methods have similar performance.

We formulate two research questions based on each perspective, respectively. We now present the findings of our research questions, along with our motivation and approach.

RQ1. Are log, Box-Cox, and rank transformations equally effective in increasing the normality of software metrics?

Motivation. Data normality impacts the performance of a prediction model [108]. In earlier studies, log and rank transformations have been applied in defect prediction [e.g. 93, 123, 202]. However, their capability in improving the normality of software metric values has not been explicitly explored. In addition, the Box-Cox transformation introduced in Section 4.2.4 has not been used in the defect prediction studies.

To thoroughly examine the impact that transformations have on defect prediction models, we are interested to examine if the three transformation methods indeed have different performance in improving the normality of software metric values.

Approach. To address this question, software metrics need to be transformed using each of the three transformation methods. As different software metrics exhibit various distributions, we transform the values of each metric independently.

In each project, we apply log transformation on software metric values to get log transformed values. When applying the Box-Cox transformation, we first apply the steps described in Section 4.2.4 to estimate the parameter λ using values of a single metric from the same project, and then apply the Box-Cox transformation. To apply rank transformation, we compute every $10th$ quantile of the distribution of values of a single metric from the same project, and obtain rank transformed values using Equation (4.1). On the transformed metric values, the skewness and kurtosis are computed to evaluate the normality.

To investigate if transformation improves the normality of software metric values, we test the following null hypothesis for each transformation methods:

$H0_{11}$: *there is no difference in the normality of the transformed metric values and the original metric values.*

We conduct paired Wilcoxon rank sum test [173], with the 95% confidence level (i.e., p-value<0.05). The Wilcoxon rank sum test is a non-parametric statistical test to assess whether two independent distributions are equal. Non-parametric statistical methods make no assumptions about the distribution of assessed variables. If there is a statistical significance, we reject the hypothesis and conclude that the examined transformation significantly changes the normality of software metric values.

Furthermore, we compare the capability of the three transformations in improving the normality of software metric values. We apply paired Wilcoxon rank sum test to evaluate the following null hypothesis, with the 95% confidence level (i.e., p-value<0.05).

$H0_{12}$: *there is no difference in the normality of metric values that are processed by transformations A and B.*

If there is a statistical significance, we reject the hypothesis and conclude that the corresponding two transformations have different capability in improving data normality.

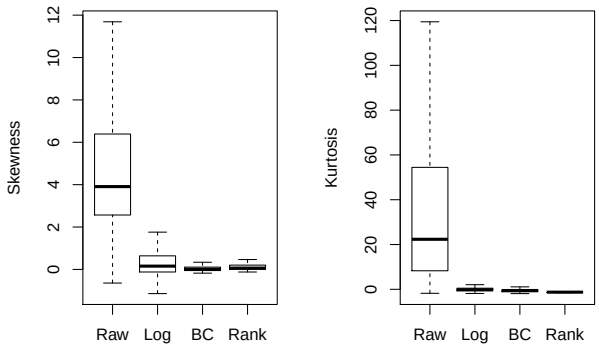


Figure 4.2: Skewness of metric values using different transformations on each subject project.

Findings. The three transformations have similar power to improve the normality of software metrics. Figure 4.2 presents the skewness and kurtosis values of the transformed metric values from all projects. The ideal skewness value is between -0.80 and 0.80, and the perfect kurtosis value is zero (see Section 4.2.1). However, the median skewness and kurtosis of the original metric values are 4 and 20, respectively. It indicates that the original metric values are highly skewed. Using any of the three transformations can make the skewness and kurtosis values become closer to zero (i.e., nearly normally distributed).

The results of Wilcoxon rank sum tests in the skewness and kurtosis between the transformed values and the original values show statistically significant difference, respectively. Hence, we reject hypothesis $H0_{11}$ for all three transformations. When comparing the skewness and kurtosis of the transformed values between any two transformations, we do not find significant difference (i.e., p-values are always greater than 0.05). Therefore, we cannot reject hypothesis $H0_{12}$, and conclude that the three transformations have no significant difference in terms of improving the normality of software metric values.

Regarding the Box-Cox transformation, the estimated parameter λ varies across projects. We present the boxplot of the estimated λ values for each project in Figure 4.3.

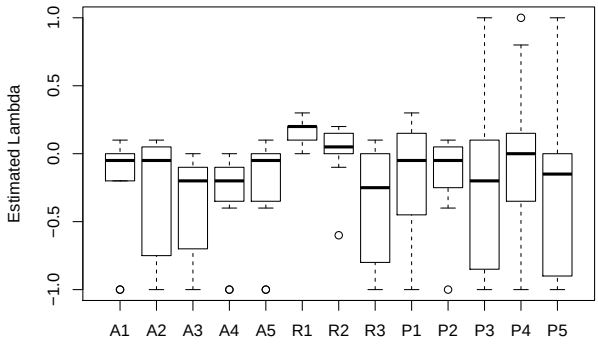


Figure 4.3: Boxplot of estimated λ values for metrics in each project. (The full name of projects are presented in Table 4.1.)

Different projects have various λ values. Therefore, estimating λ values from both the training and target projects can maximize the normality of metrics values in both projects. We observe that few of the estimated λ values are zero (λ=0 indicates a log transformation). This observation suggests that the Box-Cox transformation is not close to log transformation when dealing with software metrics. In addition, the median value of the estimated λ shows that most of the estimated λ values are negative.

Log, Box-Cox, and rank transformations have a similar capability to significantly improve the normality of software metrics values.

RQ2. Do different transformations result in distinct predictions in cross-project defect prediction models?

Motivation. In **RQ1**, we find that the three transformations are equally effective in improving the normality of software metric values. However, it is unclear if cross-project defect prediction models are impacted by applying different transformations.

We are interested to find if the same performance of cross-project predictions could be achieved, when applying each transformation method. In particular, we want to compare the overall performance (e.g., F-measure and AUC value) of cross-project defect prediction models built using the three transformations. Moreover, we want to examine if the three transformations result in distinct predictions in the cross-project setting.

Approach. To address this question, we build cross-project prediction models using all possible pairs of the training and target projects in each dataset. In AEEEM, ReLink, and PROMISE datasets, the total number of possible pairs are 20, 6, and 20, respectively.

We built cross-project defect prediction models using metric values transformed by each of the three methods. To apply the same Box-Cox transformation on the training and target projects, we estimate λ values for each metric using both projects (see Section 4.2.4). Similarly, for rank transformation, we calculate every $10th$ quantiles of the values of each metric using both projects.

We apply logistic regression to build cross-project prediction models using transformed values of the training project, and apply the models on the target project. We compute precision, recall, false positive rate, balance, F-measure, and AUC value to measure the overall performance of these models.

To compare the overall performance of the models built using the three transformations, we test the following null hypothesis. We apply paired Wilcoxon rank sum test with the 95% confidence level (i.e., p-value<0.05).

$H0_{21}$: *there is no difference between the performance of models built with transformations A and B.*

To evaluate if different transformations result in distinct predictions, we compare the prediction errors among models built using the three transformations. To this end, we test

Table 4.4: Contingency matrix to perform McNemar's test.

M_1 \ M_2	Correct prediction	Wrong prediction
Correct prediction	N_{cc}	N_{cw}
Wrong prediction	N_{wc}	N_{ww}

the following null hypothesis using McNemar's test with the 95% confidence level (i.e., p-value<0.05).

$H0_{22}$: *there is no difference between the error rate of models built with transformations A and B.*

McNemar's test is commonly used to compare prediction errors of two prediction models [90]. As a nonparametric test, and it makes no assumptions on the distribution of a subject variable. McNemar's test is applicable only if two models are applied on the same dataset with separated training and target sets. In this study, our models are built on the same training project, and applied on the same target project that is different from the training project. Therefore, McNemar's test is applicable to our study.

To perform McNemar's test, we need to compute a contingency matrix (see Table 4.4) based on the predictions produced by two models (i.e., M_1 and M_2). In the contingency matrix, N_{cc} is the number of instances that both models achieve correct predictions; N_{cw} is the number of instances that model M_1 makes a correct prediction, but model M_2 has a wrong prediction; N_{wc} is the number of instances that model M_1 makes a wrong prediction, but model M_2 produces a correct prediction; and N_{ww} is the number of instances that both models result in wrong predictions.

The null hypothesis of McNemar's test is that both models M_1 and M_2 have the same error rates. We apply the *R* function *mcnemar.test* from the *R* package *stats* to perform McNemar's test.

Table 4.5: Average performance measures of cross-project defect prediction models built using the three transformations (* denotes statistical significance).

Measures	AEEEM			ReLink			PROMISE		
	Log	BC	Rank	Log	BC	Rank	Log	BC	Rank
prec	0.561	0.534	0.410*	0.573	0.550	0.596	0.647	0.653	0.635
pd	0.251	0.243	0.246	0.344	0.386	0.362	0.598	0.614	0.577*
fpr	0.104	0.082	0.111	0.161	0.182	0.173	0.475	0.423	0.462
balance	0.432	0.447	0.433	0.516	0.530	0.507	0.424	0.507*	0.418
F-measure	0.227	0.256	0.209	0.405	0.406	0.406	0.484	0.544	0.461
AUC	0.699	0.720	0.714	0.639	0.640	0.691*	0.671	0.686	0.659

Findings. Applying the three transformation methods yields similar performances of cross-project prediction models. Table 4.5 presents the average performance measures of models built using the three transformations for each data set. The results of Wilcoxon rank sum test show that overall there is no difference among the three transformation methods (except four cases, as shown in Table 4.5). Hence, we can not reject the null hypothesis $H0_{21}$ for most cases. We conclude that the performance of cross-project prediction models built using the three transformations are similar. This finding is consistent to our previous work [202] that rank and log transformations have similar power for cross-project predictions, as well as the work of Jiang *et al.* [93].

The predicted defective files are not consistent among the results of multiple defect prediction models built using different transformation methods. Although having similar overall performances (e.g., F-measure and AUC value), the three models do not necessarily have similar prediction errors. More specifically, some models may make wrong predictions for a file, but other models make correct predictions on the same file. To the best of our knowledge, this phenomenon is overlooked in existing studies. Therefore, it is necessary to apply multiple transformation methods in cross-project defect prediction models, in order to improve the predictive power.

The detailed results of McNemar's tests are presented in Table 4.6. We observe that in

Table 4.6: The p-values of McNemar's test.

Dataset	Tbe training project to the target project	Log v.s. BC	Log v.s. Rank	BC v.s. Rank
AEEEM	Eclipse to Equinox	0.03*	1.00	0.04*
	Eclipse to Lucene	1.00	0.29	0.50
	Eclipse to Mylyn	0.18	0.54	1.00
	Eclipse to PDE	5.96e-03*	7.29e-03*	0.31
	Equinox to Eclipse	4.39e-35*	2.06e-11*	1.37e-22*
	Equinox to Lucene	4.79e-05*	0.07	2.46e-07*
	Equinox to Mylyn	5.42e-17*	1.97e-10*	0.62
	Equinox to PDE	2.41e-15*	3.58e-06*	0.03*
	Lucene to Eclipse	1.02e-06*	2.81e-03*	6.98e-11*
	Lucene to Equinox	0.22	0.45	0.08
	Lucene to Mylyn	3.40e-07*	0.13	9.42e-04*
	Lucene to PDE	1.00	0.78	0.89
	Mylyn to Eclipse	0.42	8.83e-03*	0.02*
	Mylyn to Equinox	0.01*	0.10	2.04e-04*
	Mylyn to Lucene	1.23e-03*	0.02*	0.16
	Mylyn to PDE	0.02*	9.19e-09*	2.64e-05*
	PDE to Eclipse	4.55e-02*	2.17e-04*	0.07
	PDE to Equinox	0.68	0.02*	0.03*
	PDE to Lucene	1.00	0.37	0.62
	PDE to Mylyn	0.28	0.82	0.18
	# of significance (%)	**12 (60%)**	**10 (50%)**	**10 (50%)**
ReLink	Apache to Safe	0.38	0.18	0.02*
	Apache to Zxing	0.19	0.46	0.05
	Safe to Apache	0.77	0.18	0.23
	Safe to Zxing	0.48	0.09	0.25
	Zxing to Apache	0.69	0.01*	8.42e-03*
	Zxing to Safe	0.33	0.45	0.14
	# of significance (%)	**0 (0%)**	**1 (17%)**	**2 (33%)**
PROMISE	Camel to POI	1.09e-09*	0.24	3.19e-10*
	Camel to Velocity	0.71	0.02*	0.56
	Camel to Xalan	9.53e-21*	1.94e-14*	0.03*
	Camel to Xerces	2.62e-40*	0.11	7.83e-38*
	POI to Camel	2.36e-14*	5.81e-04*	2.74e-07*
	POI to Velocity	0.23	0.65	0.06
	POI to Xalan	8.64e-51*	6.80e-06*	8.52e-61*
	POI to PDE	6.22e-11*	0.03*	7.49e-14*
	Velocity to Camel	1.02e-16*	0.60	4.05e-14*
	Velocity to POI	9.01e-04*	1.00	2.12e-03*
	Velocity to Xalan	2.52e-72*	0.17	3.11e-77*
	Velocity to Xerces	1.12e-10*	0.08	7.30e-08*
	Xalan to Camel	1.19e-06*	0.13	1.81e-08*
	Xalan to POI	1.00	1.00	1.00
	Xalan to Velocity	1.46e-05*	1.00	1.46e-05*
	Xalan to Xerces	1.15e-24*	0.68	4.01e-25*
	Xerces to Camel	0.10	0.18	0.77
	Xerces to POI	2.57e-03*	0.42	4.55e-02*
	Xerces to Velocity	0.48	1.00	0.62
	Xerces to Xalan	1.26e-04*	1.77e-07*	0.03*
	# of significance (%)	**15 (75%)**	**6 (30%)**	**15 (75%)**

the AEEEM data set, the prediction error of using log transformation is significantly different from using the Box-Cox and rank transformations in 60% and 50% of cross-project predictions, respectively. The prediction errors of using the Box-Cox and rank transformations are significantly different in 50% of cross-project predictions. Similar findings are observed in PROMISE data set. The exception is the ReLink data set. We conjecture that the non-significance may be caused by too few data points.

In summary, we can reject the null hypothesis $H0_{22}$. We conclude that the prediction errors of the models built using the three transformations are statistically significantly different. The three models built using each of the three transformation methods do not consistently make wrong predictions on the same file. Therefore, each transformation method captures different aspects of the metric values.

Cross-project prediction models built using the three transformations have similar overall performance, but their prediction errors are statistically significantly different. The defect prediction models built using each of the three transformation methods do not consistently make wrong predictions on the same file.

4.5 Our Approach

Transformations may alter the nature of software metrics. Applying various transformations on software metrics captures different perspectives of software metrics. In this section, we describe our approach to integrate a set of predictions made by models built with multiple transformations. For a pair of training and target projects, we build multiple defect prediction models. Each model is built using one of the three transformation methods. Figure 4.4 illustrates the overview of our approach. The details are described as follows.

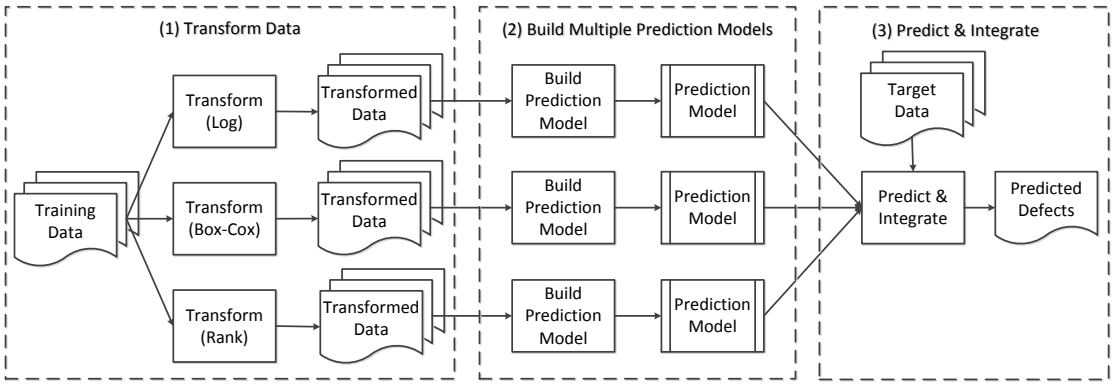


Figure 4.4: Overview of our approach to integrate models built upon differently transformed data.

Let $M=\{M_1,\ldots,M_n\}$ represents a set of prediction models built using n transformations. A file f in a target project is represented as X, a vector of all software metrics. We use $P_{B,i}(X)$ to denote the predicted probability of defect proneness on file f by model M_i, and $P_{C,i}(X)$ to denote the predicted probability of file f as a clean file. Thus, $P_{B,i}(X)+P_{C,i}(X) = 1$. We consider a file is defective, if $P_{B,i}(X)$ is greater than 0.5 [210]. We use $P_B(X)$ to denote the final probability of defect proneness on file f using all n models. We determine $P_B(X)$ in the following two ways: 1) weighting the probability $P_{B,i}(X)$ produced by models that consider file f as defective; or 2) weighting the probability $P_{C,i}(X)$ produced by all models, if no model considers file f as defective. $P_B(X)$ is defined in Equation (4.5).

$$P_B(X) = \begin{cases} min(1, \frac{\sum_{M_i \in M} w_i \times s_i(X) \times P_{B,i}(X)}{N_B(X)}) & \text{if } N_B(X) > 0 \\ \\ max(0, 1 - \frac{\sum_{M_i \in M} w_i \times P_{C,i}(X)}{n}) & \text{otherwise} \end{cases} \tag{4.5}$$

where w_i is the weight assigned to model M_i; $s_i(X)$ is the selector for model M_i that determines whether the probability predicted by model M_i is used to compute the final

probability of defect proneness or not; and $N_B(X)$ is the number of selected models to normalize the summed probability of defect proneness. The min and max limit $P_B(X)$ in the range $[0, 1]$.

A weight is assigned to each model, since the accuracy (i.e., the proportion of correct predictions) of different models varies. We consider that models with higher accuracy should be encouraged, and models with lower accuracy should be penalized. Hence, we use the accuracy a_i of a model to obtain the weight w_i for each model M_i. We set $w_i = 0$, if $a_i = 0$. For a model with non-zero accuracy (i.e., $a_i > 0$), we define its weight w_i as $w_i = \frac{a_i}{minAcc}$, where $minAcc$ is the minimum non-zero accuracy among n models.

A selector $s_i(X)$ for each model M_i is defined to capture every possible defective file. We consider that a file is defective, if it is predicted as defective by one or more models. As such, the selector $s_i(X)$ is defined in Equation (4.6). For each file, as shown in Equation (4.5), the predicted probability of model M_i is used only if the file is predicted as defective by model M_i (i.e., $P_{B,i}(X) > 0.5$).

$$s_i(X) = \begin{cases} 1 & \text{if } P_{B,i}(X) > 0.5 \\ 0 & \text{otherwise} \end{cases} \tag{4.6}$$

The number of selected models $N_B(X)$ for file f is defined in Equation (4.7). As applied in Equation (4.5), $N_B(X)$ is used to normalize the predicted probability of a file that is predicted as defective by at least one model.

$$N_B(X) = \sum_{i=1}^{n} s_i(X) \tag{4.7}$$

4.6 Evaluation of Our Approach

In this section, we evaluate the effectiveness of using our approach in cross-project prediction.

RQ3. Can our approach improve the performance of cross-project defect prediction models?

Motivation. When building defect prediction models, usually only one transformation method is applied. The findings of **RQ2** suggest that cross-project defect prediction models built using the three transformation methods can make different predictions on the same file. To improve the predictive power of cross-project defect prediction models, we propose an approach to integrate multiple transformations (see Section 4.5). In this question, we aim to investigate if our approach can achieve better performance of cross-project predictions, comparing to models built using only one transformation method.

Approach. As aforementioned, log transformation is the most widely used transformation in the literature of defect prediction. Hence, we take models built using log transformation as our baseline models.

Similar to **RQ2**, we build cross-project defect prediction models using all possible pairs of the training and target projects in each dataset. We perform the three transformations on each training project, and build three logistic regression models, respectively. We apply the three models on the target project to obtain predictions on each file of the target project. Therefore, we use our approach to integrate the predictions of the three models.

To evaluate the performance of these models, we calculate precision, recall, false positive rate, balance, F-measure, and AUC value. To investigate if our approach can improve the performance of cross-project prediction, we test the following null hypothesis:

Table 4.7: Average performance measures of cross-project defect prediction models obtained using log transformation and our approach (* denotes statistical significance; **bold** font is used if the corresponding model is better)

Measures	AEEEM dataset		ReLink dataset		PROMISE dataset	
	Log	Ours	Log	Ours	Log	Ours
prec	**0.561***	0.444	**0.573**	0.551	**0.647**	0.641
pd	0.251	**0.384***	0.344	**0.657***	0.598	**0.727***
fpr	**0.104***	0.159	**0.161***	0.350	**0.475***	0.554
balance	0.432	**0.516***	0.516	**0.645***	0.424	**0.478**
F-measure	0.227	**0.339***	0.405	**0.597***	0.484	**0.592***
AUC	0.699	**0.719***	0.639	**0.676***	0.671	**0.696***

$H0_{31}$: *there is no difference between the performance of models built with log transformation (baseline) and our approach.*

We apply paired Wilcoxon rank sum test with the 95% confidence level (i.e., p-value<0.05). If there is a statistical significance, we reject the hypothesis and conclude that our approach has statistically significant improvement in the performance of cross-project predictions.

Findings. Our approach statistically significantly improves the performance of cross-project defect prediction, in terms of recall, balance, F-measure, and AUC value. Table 4.7 presents the average value of the six measures for all possible cross-project predictions. In particular, the average F-measure of the baseline models (i.e., models built using log transformation) in AEEEM dataset is 0.227. Our approach achieves the average F-measure as 0.339 (i.e., 49% improvement). In ReLink and PROMISE datasets, we have similar observations that the average F-measures are improved from 0.405 to 0.597 (i.e., 47% improvement), and 0.484 to 0.592 (i.e., 22% improvement), respectively. A recent work by Nam *et al.* [135] has a similar concept as our approach, i.e., using transformations (namely TCA+) to improve the performance of cross-project defect prediction. In particular, Nam *et al.* [135] propose a set of rules to automatically select the most appropriate normalization method (e.g., min-max and z-score) for each pair of projects. The TCA+

approach successfully improves the average F-measure[3] by 28% in AEEEM dataset, and 24% in ReLink dataset. However, the improvement is lower than our approach from the F-measure perspective. One possible reason is that our approach utilizes multiple models rather than selecting only one model.

The p-values of Wilcoxon test on F-measures between our approach and the baseline models in AEEEM, ReLink, and PROMISE datasets are $2.22e$-04, 0.03 and $6.93e$-03, respectively. Hence, we reject the null hypothesis $H0_{31}$ for each dataset, and conclude that our approach statistically significantly outperforms the baseline models.

The false positive rate is increased by our approach, but it is controllable. We observe that our approach increases the false positive rate, if using the default cut-off on the probability of defective files, i.e., 0.5 [210]. For instance, the false positive rate in AEEEM dataset is increased from 0.104 to 0.159, but it is still acceptable, i.e., less than 0.3 [128]. For ReLink dataset, the average false positive rate is acceptable (i.e., 0.161) in the baseline models but is excessive (i.e., 0.350) using our approach. We conjecture that a particular cut-off may reduce the excessive false positive rate, since the performance is statistically significantly improved in terms of AUC value (i.e., from 0.639 to 0.676). Indeed, when choosing 0.56 as the cut-off for ReLink dataset, the false positive rate of our approach becomes acceptable, and has no statistically significant difference than the baseline models (i.e., from 0.122 to 0.282, and p-value is 0.07). Moreover, our previous work [203] describes two concrete and practical solutions to reduce the false positive rate by automatically determining the cut-off. In PROMISE dataset, the average false positive rate is excessive in both approaches. One possible reason is that projects in this dataset have diverse ratios of defective files (i.e., from 19% to 99%). For instance, the model built from a project with a high ratio of defective files (i.e., the project with the majority of

[3]As only F-measures are reported in their paper, we present their improvement solely using F-measures.

its files are actually defective files) tends to predict files as defective. Such a model can experience high false positive rate, if applied to a project with low ratio of defective files (i.e., the project with the majority of its files are actually clean files).

In summary, we conclude that our approach is effective for cross-project predictions based on the improvement in AUC values that are independent of cut-off values. As our approach only requires to perform three transformations that can be done with very low cost, it is worth experimenting with our approach in defect prediction studies.

Our approach can signiciantly improve the performance of cross-project predictinos, in terms of recall, balance, F-measure, and AUC value.

RQ4. Does our approach work well for other classifiers?

Motivation. We have demonstrated the effectiveness of our approach using a single classifier (i.e., logistic regression). However, there are many other classifiers (e.g., Naive Bayes and random forest) that are also frequently used to build defect prediction models (e.g., [93, 102, 123, 180]). To understand the generalizability of our approach, it is necessary to study if our approach can achieve similar improvement using other classifiers other than using logistic regression.

Approach. We follow the same approach as in **RQ3**, but using different classifiers to build cross-project prediction models. As described in Section 4.3.2, we study six classifiers, i.e., Naive Bayes, Bayes Net, IBk (k-nearest neighbours), J48 (decision tree), random forest, and random tree.

To investigate if our approach can improve the performance of cross-project prediction, we test the following null hypothesis for each classifier. We apply paired Wilcoxon rank

Table 4.8: Average F-measures and AUC values of cross-project defect predictions obtained using log transformations and our approach (* denotes statistical significance; **bold** font is used if the corresponding model is better).

(a) F-measure

Classifier	AEEEM data set			ReLink data set			PROMISE data set		
	Log	Ours	% Inc.	Log	Ours	% Inc.	Log	Ours	% Inc.
Logistic	0.227	0.339*	49%	0.405	0.597*	47%	0.484	0.592*	22%
NaiveBayes	0.409	0.415	1.5%	0.556	0.604	8.6%	0.570	0.595*	4.4%
BayesNet	0.409	0.419	2.4%	0.537	0.614	14%	0.551	0.578*	4.9%
IBk	0.293	0.334*	14%	0.495	0.566	14%	0.505	0.581*	15%
J48	0.281	0.356*	27%	0.391	0.489*	25%	0.499	0.566*	13%
RandomForest	0.282	0.363*	29%	0.398	0.572*	44%	0.456	0.571*	25%
RandomTree	0.295	0.342*	16%	0.490	0.601*	23%	0.503	0.621*	23%

(b) AUC value

Classifier	AEEEM data set			ReLink data set			PROMISE data set		
	Log	Ours	% Inc.	Log	Ours	% Inc.	Log	Ours	% Inc.
Logistic	0.699	0.719*	2.9%	0.639	0.676*	5.8%	0.671	0.696*	3.7%
NaiveBayes	0.710	0.718*	1.1%	0.683	0.698	2.2%	0.685	0.699*	2.0%
BayesNet	0.715	0.723*	1.1%	0.678	0.698	2.9%	0.660	0.678*	2.7%
IBk	0.576	0.593	3.0%	0.606	0.611	0.8%	0.535	0.542	1.3%
J48	0.593	0.645*	8.8%	0.563	0.580	3.0%	0.581	0.571	-1.7%
RandomForest	0.668	0.691*	3.4%	0.677	0.694	2.5%	0.609	0.638*	4.8%
RandomTree	0.579	0.600*	3.6%	0.600	0.627	4.5%	0.535	0.569*	6.4%

sum test with the 95% confidence level (i.e., p-value<0.05).

$H0_{41}$: *there is no difference between the performance of our approach and the models built with log transformation, when using classifier C to build the model.*

Findings. In general, our approach can improve the performance of cross-project defect prediction models. However, the improvement varies with classifiers. Table 4.8 presents the average F-measures and AUC values of models built with log transformation and our approach using each of the six classifiers.

In terms of F-measure, our approach can achieve statistically significant improvement for logistic regression (22% to 49%), IBk (14% to 15%), J48 (13% to 27%), random forest (25% to 44%), and random tree (16% to 23%). However, the improvement is small when using NaiveBayes (1.5% to 8.6%) and BayesNet (2.4% to 14%).

Similar findings can also be observed from the AUC value perspective. In particular, three classifiers can benefit from our approach, i.e., logistic regression (2.9% to 5.8%), random forest (2.5% to 4.8%), and random tree (3.6% to 6.4%). When using any of the three classifiers, the improvement in the AUC value is statistically significant in at least two datasets. There is only one exception (i.e., J48 in PROMISE dataset), where our approach decreases the AUC value by 0.01.

Moreover, we observe that our approach always achieves statistically significant improvement (in terms of both F-measure and the AUC value), when using logistic regression. This might be because data normality improves the performance of linear models [108], and logistic regression model is a special case of generalized linear models.

In summary, our approach generally improves the performance of cross-project defect prediction models, although the improvement varies with classifiers.

Our approach achieves varied improvement for different classifiers, and generally achieves significant improvement for three classifiers (i.e., logistic regression, random forest, and random tree).

4.7 Threats to Validity

We now describe the threats to validity of our study under common guidelines [199].

Threats to conclusion validity concern the relation between the treatment and the outcome. The main threats come from our implementation of the three transformations. For instance, we normalize metric values transformed by the Box-Cox transformation to $[1,2]$. We describe our treatment of the three transformations, so that researchers can replicate our work and yield the same conclusion when applying the same treatments as our study.

Threats to internal validity concern our selection of subject systems and analysis methods. We choose subjects from three publicly available data sets that have been used in many other studies [76, 135, 202]. The selected projects have diversity in size and ratio of defectiveness. The threats to our analysis method come from our choice of logistic regression to study RQ2 and RQ3. Thus, in RQ4, we examine the effectiveness of our approach using six other classifiers.

Threats to external validity concern the possibility to generalize our results. Our approach is based on log, Box-Cox, and rank transformations. All three transformations are applicable to software metrics, since many metrics follow power law distributions [41, 116, 205]. The diversity in size and defect-proneness of our subject projects helps verify the generalizability of our approach. Nevertheless, further validations on other open source projects and even commercial projects are recommended.

Threats to reliability validity concern the possibility of replicating this study. All three data sets used in this study are publicly accessible. We also provide details of our experiments on the internet[4].

4.8 Chapter Summary

In this chapter, we observe that three simple transformation methods (i.e., log, Box-Cox, and rank transformations) have similar power to significantly improve the normality of software metrics. Moreover, cross-project prediction models built with each of the three transformation methods achieve similar performances (i.e., precision, recall, false positive rate, balance, F-measure and AUC value). However, we find that these models do not always make wrong predictions on the same file, since the results of McNemar's tests

[4]http://www.feng-zhang.com/replication/transformation

clearly show that these models experience significantly different error rates.

Therefore, we propose an approach to integrate predictions by the cross-project defect prediction models built using each of the three transformation methods. We perform an experiment using three public data sets, such as AEEEM [44], ReLink [196], and PROMISE [96]. The results show that, comparing to the models built with only one transformation method (i.e., the widely used log transformation), our approach statistically significantly improves recall, F-measure and AUC value in all three data sets. For instance, the average F-measures are improved by 49%, 47%, and 22% in AEEEM, ReLink, and PROMISE datasets, respectively. Furthermore, our approach generally leads to the performance improvement in cross-project defect prediction models, regardless of using any of the classifiers under study (e.g., random forest). Especially when using logistic regression, random forest, or random tree, our approach can achieve significant improvement in the performance of cross-project defect prediction models.

CHAPTER 5

Aggregation of Software Metrics

Key Question

? *Can the use of summation to aggregate software metrics hinder the performance of defect prediction models?*

5.1 Introduction

To build a defect prediction model, historical defect-fixing activity is mined and software metrics, which may have a relationship with defect proneness, are computed. The historical defect-fixing activity is usually mined from a Version Control System (VCS), which records change activity at the file-level. Although it is possible to collect defect data at the method-level, it is time consuming and often impractical [187]. On the other hand, software metrics such as cyclomatic complexity (Cc) [119] are computed at the method- or class-level.

It has been reported that predicting defective files is more effective than predicting defective packages for Java systems [45, 138, 153]. Typically, in order to train file-level defect prediction models, the method- or class-level software metrics are aggregated to

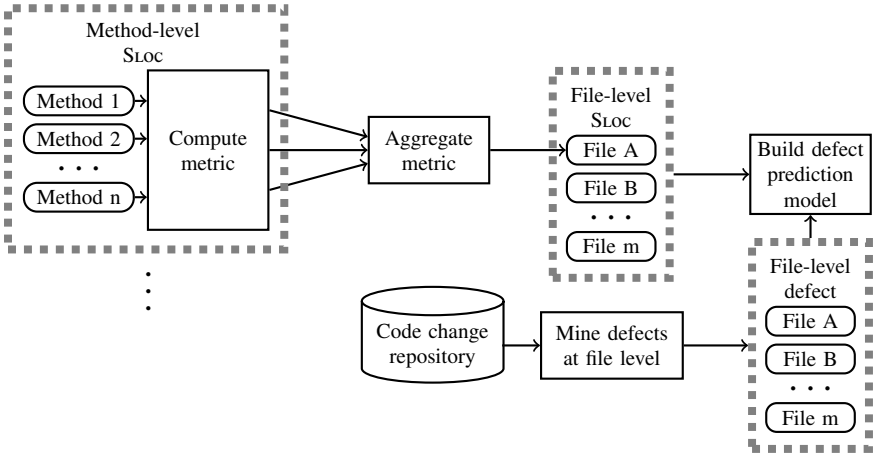


Figure 5.1: A typical process to apply method-level metrics (e.g., Sloc) to build file-level defect prediciton models.

the file-level. Such a process is illustrated in Figure 5.1. Summation is one of the most commonly applied aggregation schemes in the literature [82, 106, 111, 112, 128, 135, 138, 154, 202, 208, 209]. However, Landman *et al.* [109] show that prior findings (e.g., [54, 67]) about the high correlation between summed Cc and summed lines of code (i.e., Sloc) may have been overstated for Java projects, since the correlation is significantly weaker at the method-level. We suspect that the high correlation between many metrics at the file-level may also be caused by the aggregation scheme. Furthermore, the potential loss of information due to the summation aggregation may negatively affect the performance of defect prediction models.

Besides summation, there are a number of other aggregation schemes that estimate the central tendency (e.g., arithmetic mean and median), dispersion (e.g., standard deviation and coefficient of variation), inequality (e.g., Gini index [62], Atkinson index [9], and Hoover index [83]), and entropy (e.g., Shannon's entropy [169], generalized entropy [42], and Theil index [185]) of a metric. However, the impact that aggregation schemes have on defect prediction models remains unexplored.

We, therefore, set out to study the impact that different aggregation schemes have on defect prediction models. We perform a large-scale experiment using data collected from 255 open source projects. First, we examine the impact that different aggregation schemes have on: (a) the correlation among software metrics, and (b) the correlation between software metrics and defect count. Second, we examine the impact that different aggregation schemes have on the performance of four main types of defect prediction models, namely:

(a) Defect proneness model that classifies files as defect-prone or not [123, 202, 210];

(b) Defect rank model that ranks files according to their defect proneness [131, 208];

(c) Defect count model that estimates the number of defects in a given file [144, 209];

(d) Effort-aware model that incorporates fixing effort in the ranking of files according to their defect proneness [98, 121].

To ensure that our conclusions are robust, we conduct a 1,000-repetition bootstrap experiment for each of the studied systems. In total, over 12 million prediction models are constructed during our experiments.

The main observations of our experiments are as follows:

- **Correlation analysis.** We observe that aggregation can significantly impact both the correlation among software metrics and the correlation between software metrics and defect count. Indeed, summation significantly inflates the correlation between Sloc and other metrics (not just Cc). Metrics and defect count share a substantial correlation in 15%-22% of the studied systems if metrics are aggregated using summation, while metrics and defect count only share a substantial correlation in 1%-9% of the studied systems if metrics are aggregated using other schemes.

- **Defect prediction models.** We observe that using only summation (i.e., the most commonly applied aggregation scheme) often does not achieve the optimal performance. For example, when constructing models to predict defect proneness, solely applying the summation scheme achieves the optimal performance in only 11% of the studied projects, whereas applying all of the studied aggregation schemes achieves the optimal performance in 40% of the studied projects. Furthermore, when constructing models for effort-aware defect prediction, the mean or median aggregation schemes yield performance values that are significantly closer to the optimal ones than any of the other studied aggregation schemes. On the other hand, when constructing models to predict defect rank or count, either applying only the summation or applying all of the studied aggregation schemes achieves similar performance, with both achieving closer to optimal performance more often than the other studied aggregation schemes.

Broadly speaking, solely relying on summation tends to underestimate the predictive power of defect prediction models. Given that the cost of computing the additional aggregation schemes is negligible, we strongly recommend that future defect prediction studies use the 11 aggregation schemes that we explore in this thesis, and even experiment with other aggregation schemes.

Chapter Organization. We present the 11 studied aggregation schemes in Section 5.2. Section 5.3 describes the data that we use in our experiments. The approach and results of our case study are presented and discussed in Sections 5.4 and 5.5. We discuss the threats to the validity of our work in Section 5.6. Summary of the chapter is described in Section 5.7.

Table 5.1: List of the 11 studied aggregation schemes. In the formulas, m_i denotes the value of metric m in the ith method in a file that has N methods. Methods in the same file are sorted in the ascending order of the values of metric m.

Category	Aggregation scheme	Formula	
Summation	Summation	$\Sigma_m = \sum_{i=1}^{N} m_i$	(1)
Central Tendency	Arithmetic mean	$\mu_m = \frac{1}{N} \sum_{i=1}^{N} m_i$	(2)
	Median	$M_m = \begin{cases} m_{\frac{n+1}{2}} & \text{if N is odd} \\ \frac{1}{2}(m_{\frac{n}{2}} + m_{\frac{n+2}{2}}) & \text{otherwise.} \end{cases}$	(3)
Dispersion	Standard deviation	$\sigma_m = \sqrt{\frac{1}{N} \sum_{i=1}^{N} (m_i - \mu_m)^2}$	(4)
	Coefficient of variation	$\text{Cov}_m = \frac{\sigma_m}{\mu_m}$	(5)
Inequality Index	Gini index [62]	$\text{Gini}_m = \frac{2}{N\Sigma_m}[\sum_{i=1}^{N}(m_i * i) - (N+1)\Sigma_m]$	(6)
	Hoover index [83]	$\text{Hoover}_m = \frac{1}{2} \sum_{i=1}^{N} \lvert \frac{m_i}{\Sigma_m} - \frac{1}{N} \rvert$	(7)
	Atkinson index [9]	$\text{Atkinson}_m = 1 - \frac{1}{\mu_m}(\frac{1}{N} \sum_{i=1}^{N} \sqrt{m_i})^2$	(8)
Entropy	Shannon's entropy [169]	$E_m = -\frac{1}{N} \sum_{i=1}^{N} [\frac{freq(m_i)}{N} * ln \frac{freq(m_i)}{N}]$	(9)
	Generalized entropy [42]	$\text{GE}_m = -\frac{1}{N\alpha(1-\alpha)} \sum_{i=1}^{N} [(\frac{m_i}{\mu_m})^{\alpha} - 1], \quad \alpha = 0.5$	(10)
	Theil index [185]	$\text{Theil}_m = \frac{1}{N} \sum_{i=1}^{N} [\frac{m_i}{\mu_m} * ln(\frac{m_i}{\mu_m})]$	(11)

5.2 Aggregation Schemes

In this section, we introduce the 11 aggregation schemes that we studied. We also discuss why we exclude some other aggregation schemes from our study. Table 5.1 shows the formulas of the 11 schemes. The details are presented as follows.

1) Summation. An important aspect of a software metric is the accumulative effect, e.g., files with more lines of code are more likely to have defects than files with few lines of code [58]. Similarly, files with many complex methods are more likely to have defects than files with many simple methods [103]. Summation captures the accumulative effect of a software metric. Specifically, we study the *summation* scheme, which is commonly used in defect prediction studies [82, 106, 111, 112, 128, 135, 138, 154, 202, 208, 209].

2) **Central tendency.** In addition to the accumulative effect, the average effect is also important. For example, it is likely easier to maintain a file with smaller methods than a file with larger ones, even if the total file size is equal. Computing the average effect can help to distinguish between files with similar total size, but different method sizes on average. The average effect of a software metric can be captured using central tendency metrics, which measure the central value in a distribution. In this thesis, we study the *arithmetic mean* and *median* measures of central tendency.

3) **Dispersion.** Dispersion measures the spread of values of a particular metric, with respect to some notion of central tendency. For example, dispersion can capture how much the size of methods in the same file differ from the average size. We study the *standard deviation* and the *coefficient of variation* measures of dispersion.

4) **Inequality index.** An inequality index explains the degree of imbalance in a distribution. Inequality indices are often used by economists to measure income inequality in a specific group [191]. In this thesis, we study the Gini [62], Hoover [83], and Atkinson [9] inequality indices. These three indices have previously been analyzed by Vasilescu [191] in the broad context of software engineering.

   Each index captures one aspect of inequality. The Gini index measures the degree of inequality, but cannot identify the unequal part of the distribution. The Atkinson index can indicate which end of the distribution introduces the inequality. The Hoover index represents the proportion of all values that, if redistributed, would counteract the inequality. The three indices range from zero (perfect equality) to one (maximum inequality).

5) **Entropy.** In information theory, entropy represents the information contained in a set of variables. Larger entropy values indicate greater amounts of information. In the extreme

case, files full of duplicated code snippets contain less information than files with only unique code snippets. In this thesis, we study the *Shannon's entropy* [169], *generalized entropy* ($\alpha = 0.5$) [42], and the *Theil index* [185]. Shannon's (and generalized) entropy measure redundancy or diversity in the values of a particular metric. The Theil index, an enhanced variant of the generalized entropy, measures inequality or lack of diversity.

6) **Excluded aggregation schemes.** Distribution shape is another widely used family of aggregation schemes. Skewness and kurtosis are two commonly used measures that capture the shape of a distribution. In the formulas for computing skewness and kurtosis, the denominator is the standard deviation. If the standard deviation is zero, the skewness and kurtosis are both undefined. In our data set, we observe that a large number of methods have exactly the same value of a particular metric, producing zero variance. Hence, we exclude skewness and kurtosis from our analysis, since they are undefined for many files.

   Kolm index [105] is another candidate scheme that measures the absolute inequality of a distribution. However, the computation of Kolm index requires the exponentiation of metric values. Since many of our metrics have values larger than 1,000, the Kolm index becomes uncomputable. Therefore, it is not suitable for our study.

5.3 Experimental Data

In this section, we describe our experimental data, including the characteristics of the dataset, the defect data, and the software metrics that we use.

5.3.1 Dataset Characteristics

In this study, we begin with the dataset that was initially collected by Mockus [125]. The dataset contains 235K open source systems hosted on SourgeForge and GoogleCode. However, there are many systems that have not yet accumulated a sufficient amount historical data to train defect models. Similar to Chapter 3, we apply a series of filters to exclude such systems from our analysis. Specifically, we exclude the systems that:

(F1) Are not primarily written in C, C++, C#, Java, or Pascal, since the tool [164] that we use to compute the software metrics only supports these languages.

(F2) Have a small number of commits (i.e., less than the 25% quantile of the number of commits across all remaining systems), as systems with too few commits have not yet accumulated enough historical data to train a defect model.

(F3) Have a lifespan of less than one year, since most defect prediction studies collect defect data using two consecutive six-month time periods [208].

(F4) Have a limited number of fix-inducing and non-fixing commits (i.e., less than the 75th percentile of the number of fix-inducing and non-fixing commits across all remaining systems, respectively). We do so to ensure that we have enough data to train stable defect models.

(F5) Have less than 100 usable files (i.e., without undefined values of aggregated metrics). This ensures that we have sufficient instances for bootstrap model validation.

Table 5.2 provides an overview of the 255 systems that survive our filtering process.

Table 5.2: The descriptive statistics of our dataset.

Programming language	Number of systems	Number of files	Number of methods	Defect ratio (mean ± sd)
C	34	8,140	167,146	43% ± 26%
C++	85	20,649	479,907	40% ± 27%
C#	15	2,951	666,046	38% ± 23%
Java	121	32,531	527,203	37% ± 27%
All	255	64,271	1,840,302	39% ± 27%

5.3.2 Defect Data

In general, defect data is mined from commit messages. Since these commit messages can be noisy, data mined from commit messages are often corroborated using data mined from Issue Tracking Systems (ITSs, e.g., Bugzilla[1]) [208]. However, we find that only 53% of the studied systems are using ITSs. Hence, to treat every studied system equally, we mine defect data solely based on commit messages. While this approach may introduce bias into our dataset [16, 81, 102], prior work has shown that this bias can be offset by increasing the sample size [158]. There are 255 subject systems in our dataset, which is larger than most defect prediction studies to date [151].

Similar to Chapter 3, we consider that a commit is related to a defect fix if the commit message matches the following regular expression:

$(bug|fix|error|issue|crash|problem|fail|defect|patch)$

To further reduce the impact that noise in commit messages may introduce, we clean up noisy words like "debug" and "prefix" by removing all words that end with "bug" or "fix". A similar strategy was used by Mockus and Votta [126] and is at the core of the popular SZZ algorithm [179]. In addition, similar to prior work [208], we use a six-month time period to collect defect data, i.e., we check for defect-fixing commits that occur in a six-month time period after a software release has occurred. Unfortunately, many systems

[1] http://www.bugzilla.org/

Table 5.3: List of software metrics at method-level.

Metric	Description
SLOC	Source lines of code, excluding comments and blank lines.
CC	McCabe's cyclomatic complexity.
EVG	Essential complexity is a modified version of cyclomatic complexity.
NPATH	The number of possible execution paths in a method.
FANIN	The number of inputs, including parameters, global variables, and method calls.
FANOUT	The number of outputs, such as updating parameters and global variables.

on SourceForge or GoogleCode have not recorded their release dates. Hence, we simply choose the date that is six months prior to the last recorded commit of each system as the split date. Defect data is collected from commit messages in the six-month period after the split date.

5.3.3 Software Metrics

We group software metrics into three categories, i.e., traditional metrics, object-oriented metrics, and process metrics. In the scope of defect prediction, Radjenović *et al.* [156] perform a systematic review and report that traditional metrics are often collected at the method-level, object-oriented metrics are often collected at the class-level, and process metrics are often collected at the file-level. In this thesis, we study traditional metrics, so that we can focus on investigating how the studied aggregation schemes impact defect prediction models.

In this study, we choose six method-level metrics that are known to perform well in defect prediction models [61]. Table 5.3 provides an overview of the studied metrics. *Source Lines Of Code* (SLOC) is a measure of the size of a method. *Cyclomatic complexity* (CC) and *essential complexity* (EVG) are measures of the complexity of a method. The *number of possible paths* (NPATH), the *number of inputs* (FANIN), and the *number of outputs* (FANOUT) are measures of the control flow of a method.

To compute these metrics, we use the *Understand* [164] tool on the release (or split) code snapshot of each studied system. This code snapshot is the historical version of the studied system at the date just before the six-month time period used for collecting the defect data.

5.4 Case Study I – Correlation Analysis

In this section, we study the impact that different aggregations have on the correlation among software metrics and the correlation between software metrics and defect counts.

When choosing software metrics to build a defect prediction model, it is common practice to explore the correlations among software metrics, and the correlations between software metrics and defects [97, 181]. Strongly correlated software metrics may be redundant, and may interfere with one another if used together to train a defect prediction model. Furthermore, a substantial correlation between a software metric and defect count may identify good candidate predictors for defect prediction models.

Aggregation schemes are required to lift software metrics to the file-level. However, aggregation schemes may distort the correlation between SLoc and Cc in Java projects [109]. If two metrics have a much stronger correlation after aggregation, it is unclear if the two metrics are actually strongly correlated, or if the aggregation has distorted one or both of the metrics.

Understanding the impact that aggregation schemes have can prevent the removal of useful metrics. Hence, we want to examine the impact that aggregations have in order to avoid potential loss of information in the model construction step.

5.4.1 Research Questions

To study the impact that aggregations have on the correlation among software metrics and their correlation with the defect count, we formulate the following two research questions:

RQ1.1 Do aggregation schemes alter the correlation between software metrics?

RQ1.2 Do aggregation schemes alter the correlation between software metrics and defect count?

5.4.2 Experimental Design

1) Correlation among metrics. In this study, we use Spearman's ρ [173] to measure correlation. Spearman's ρ measures the similarity between two ranks, instead of the exact values of the two assessed variables. Unlike parametric correlation techniques (e.g., Pearson correlation [173]), Spearman correlation does not require that the input data follow any particular distribution. In the presence of ties, Spearman's ρ is preferred [155] over other nonparametric correlation techniques, such as Kendall's τ [173]. Spearman's ρ ranges from -1 to +1, where -1 and +1 indicate the strongest negative and positive correlations, respectively. A value of zero indicates that the two input variables are entirely independent.

Figure 5.2 presents our approach to examine the impact that aggregations have on correlation among software metrics. To understand the correlation among metrics before aggregation, for each system, we calculate ρ between each pair of metrics at the method-level. Assessing if two metrics are strongly correlated is often applied to determine their redundancy in the scope of defect prediction [57, 165]. Similar to prior work [57, 165, 182], we consider that a pair of metrics are too highly correlated to include in the same model if $|\rho| \geq 0.8$. We report the percentage of metrics that have $|\rho| < 0.8$ across all of the systems.

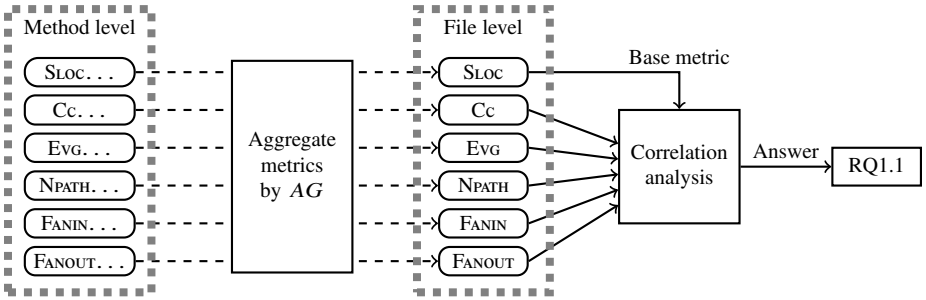


Figure 5.2: Our approach to analyze the impact of aggregations on the correlations between software metrics (RQ1.1).

To study the impact that aggregation schemes have on these correlation values, we use SLOC as our base metric, and for each system, we compute ρ between SLOC and the other metrics at both method- and file-levels. We denote the correlation between SLOC and metric m as $cor.method(\text{SLOC}, m)$ at the method-level, and as $cor.file(\text{SLOC}, AG(m))$ at the file-level after applying an aggregation scheme AG. We test the null hypothesis below for each scheme:

$H0_1$: *There is no difference between the method-level correlation* $cor.method(\text{SLOC}, m)$ *and the file-level correlation* $cor.file(\text{SLOC}, AG(m))$.

To test $H0_1$, we use two-sided Mann-Whitney U tests [173] with $\alpha = 0.05$ (i.e., 95% confidence level). The Mann-Whitney U test checks if the distributions of the two assessed variables have equal values. As a non-parametric statistical method, the Mann-Whitney U test makes no assumptions about the distributions that the input samples are drawn from. If there is a statistically significant difference between the input samples, we can reject $H0_1$ and conclude that the corresponding aggregation scheme yields statistically significantly different correlation values at the method- and file-levels. To control family-wise errors, we apply Bonferroni correction and adjust α by dividing by the number of tests.

We also calculate Cliff's δ [36] to quantify the size of the difference in correlation values at the method- and file-levels. We opt to use Cliff's δ instead of Cohen's d [38] because

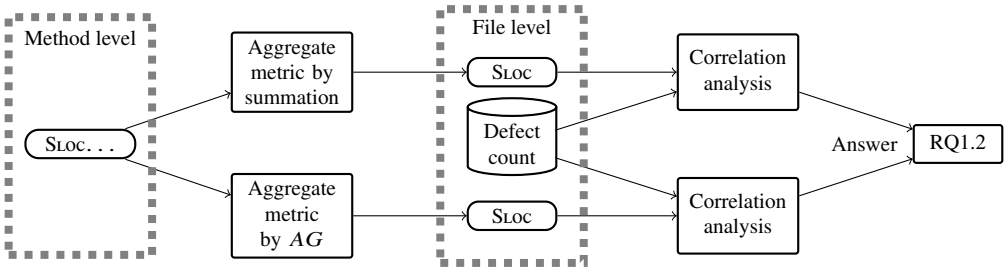


Figure 5.3: Our approach to analyze the impact of aggregations on correlations between software metrics and defect count (RQ1.2).

Cliff's δ is widely considered to be a more robust and reliable effect size measure than Cohen's d [160]. Moreover, Cliff's δ does not make any assumptions about the distributions of the input samples.

Cliff's δ ranges from -1 to +1, where a zero value indicates two identical distributions. A negative value indicates that values in the first sample tend to be smaller than those in the second sample, while a positive value indicates the opposite. To ease interpretation of the effect size results, we use the mapping of Cliff's δ values to Cohen's d significance levels as proposed by prior work [160] (see Table 3.6).

2) Correlation between metrics and defect count. To further understand the impact of aggregations, we investigate the correlation between defect count and metrics aggregated by each of the studied schemes. Figure 5.3 provides an overview of our approach. Software metrics having a substantial correlation with defect count are usually considered to be good candidate predictors for defect prediction models [208]. Similar to prior work [208], we consider that a metric shares a promising correlation with the number of defects if the corresponding $|\rho| \geq 0.4$. Hence, we report the percentage of studied systems that have $|\rho| \geq 0.4$ for the defect count and any given metric after applying any of the studied aggregation scheme.

Table 5.4: The percentage of the studied systems that do not have strong correlations among all six metrics at the method-level.

Scheme	Cc	Npath	Fanin	Fanout	Evg
Sloc	58%	59%	100%	39%	100%
Cc	-	0%	100%	96%	99%
Npath	-	-	100%	96%	99%
Fanin	-	-	-	100%	100%
Fanout	-	-	-	-	100%

5.4.3 Case Study Results

Aggregation can significantly alter the correlation among metrics. Table 5.4 shows that many method-level metrics do not have strong correlation values with one another (i.e., $|\rho| < 0.8$). For example, Fanin is not strongly correlated with the other metrics in any of the studied systems. Moreover, Sloc is not strongly correlated with Cc in 58% of the studied systems.

On the other hand, some method-level metrics also have consistently strong correlation values. For example, Cc is strongly correlated with Npath in all of the studied systems.

Since one would like to preserve the information provided by metrics, keeping the correlation values between metrics low would be ideal. To that end, we find that selecting some aggregation schemes can help to reduce the strong correlation values that we observe at the method-level. For example, although Cc and Npath are strongly correlated in all of the studied systems at the method-level, when they are aggregated by summation to the file-level, they do not share a strong correlation with one another in 96% of the studied systems. This weaker correlation would allow one to safely use both metrics in a defect prediction model.

Different aggregation schemes have various impact on the correlation among metrics. To illustrate the effect of the various aggregation schemes, we compute the *gain ratios*

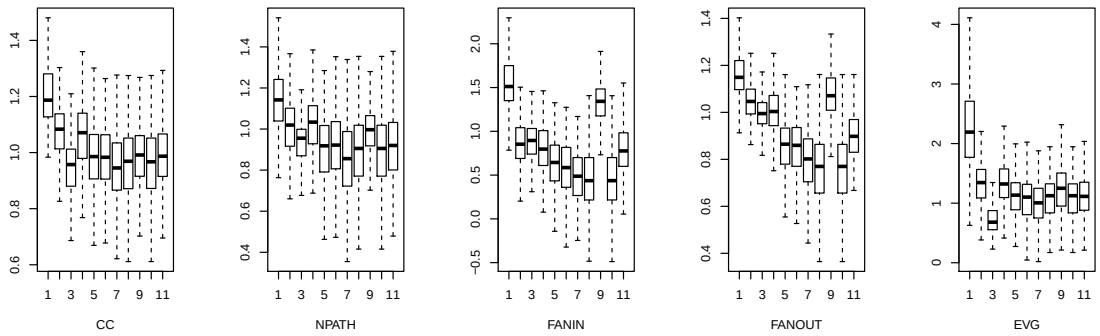


Figure 5.4: Boxplots of the gain ratios in correlations between SLOC and other metrics at file-level. The order of the 11 aggregation schemes are the same as shown in Table 5.1.

of the correlation values of a metric when aggregated to the file-level. Below, we define the gain ratio for a metric m when aggregated using a particular scheme AG:

$$cor.gain.ratio(m, AG) = \frac{cor.file(\text{SLOC}, AG(m))}{cor.method(\text{SLOC}, m)} \tag{5.1}$$

While we find that aggregation schemes do impact correlation values, most aggregation schemes do not have a consistent impact on all of the studied metrics. On the one hand, the gain ratios of Figure 5.4 show that summation tends to increases the correlation between SLOC and all of the other metrics. On the other hand, for the NPATH, FANIN, and FANOUT metrics, Figure 5.4 shows that the median gain ratios are often below 1, indicating that most aggregation schemes decrease the correlation values for these metrics in half of the studied systems.

Table 5.5 presents the results of the Mann-Whitney U tests and Cliff's δ. We find that summation has a consistently large impact (i.e., p-value is below α and the absolute value of Cliff's δ is greater than 0.474) on the correlation between SLOC and the other metrics in software systems developed in C, C++, C#, and Java. This observation is consistent with

Table 5.5: The Cliff's δ of the difference in correlation values between SLOC and other metrics before and after aggregation. (**bold** font indicates a large difference, and n.s. denotes a lack of statistical significance).

Scheme	CC	NPATH	FANIN	FANOUT	EVG
Sum	**-0.881**	**-0.655**	**-0.884**	**-0.907**	**-0.969**
Mean	-0.363	n.s.	0.269	-0.279	-0.386
Median	0.188	0.213	0.206	n.s.	0.239
SD	-0.290	-0.128	0.388	n.s.	-0.401
COV	n.s.	0.345	**0.605**	**0.608**	-0.181
Gini	0.022	0.305	**0.646**	**0.609**	-0.082
Hoover	0.195	**0.505**	**0.737**	**0.729**	n.s.
Atkinson	0.105	0.388	**0.767**	**0.778**	-0.104
Shannon	n.s.	n.s.	**-0.584**	**-0.481**	-0.295
Entropy	0.104	0.388	**0.767**	**0.778**	-0.104
Theil	n.s.	0.370	0.469	0.458	-0.143

Landman *et al.*'s work [109], which found that summation tends to inflate the correlation between SLOC and CC when aggregated from the method- to the file-level in Java projects.

Not all metrics are sensitive to aggregation schemes. Indeed, only the FANIN and FANOUT metrics are significantly sensitive to aggregation schemes. Furthermore, contrary to the CC results, these aggregations tend to weaken their correlation with SLOC.

When compared to the other aggregation schemes, summation has the largest impact on the correlation between the studied metrics and defect count. Table 5.6 shows the percentage of the studied systems that have a substantial correlation (i.e., $|\rho| \geq 0.4$) between defect count and a given metric when aggregated using the studied schemes. File-level metrics that are aggregated by summation share a substantial correlation with defect count in 15% to 22% of the studied systems. The other aggregation schemes show potential to make file-level metrics substantially correlate with defect count, with 1%-9% of the studied systems yielding substantial correlation values. Indeed, in addition to summation, applying other aggregation schemes may create useful new metrics for defect prediction models.

Table 5.6: The percentage of studied systems where the defect count shares a substantial correlation ($|\rho| \geq 0.4$) with the metrics.

Scheme	LOC	CC	NPATH	FANIN	FANOUT	EVG
Sum	20%	22%	15%	16%	20%	16%
Mean	2%	1%	3%	2%	1%	3%
Median	1%	2%	2%	1%	1%	0
SD	4%	2%	5%	4%	1%	4%
COV	3%	3%	7%	1%	1%	4%
Gini	3%	2%	5%	1%	1%	3%
Hoover	1%	2%	5%	1%	1%	3%
Atkinson	1%	2%	6%	1%	1%	4%
Shannon	9%	7%	6%	9%	9%	2%
Entropy	1%	2%	6%	1%	1%	4%
Theil	2%	3%	6%	3%	1%	4%

Aggregation can significantly alter the correlation among metrics and the correlation between metrics and the defect count. Experimenting with aggregation schemes may produce useful new metrics for defect prediction models.

5.5 Case Study II – Defect Prediction Models

In this section, we evaluate the impact of aggregations on four types of defect prediction models, i.e., defect proneness, defect rank, defect count, and effort-aware models. Moreover, we provide comprehensive guidelines regarding the choice of aggregation schemes for future studies.

Our analysis in the prior section shows that aggregation schemes can significantly alter the correlation among metrics and the correlation between defect count and metrics. These results suggest that using additional aggregation schemes may generate new metrics that capture unique characteristics of the studied data, and that may be useful for defect prediction. In this section, we investigate the impact that aggregation schemes have on four types of defect prediction models. While we use the same metrics in each type of defect prediction model, the dependent variable varies as described below:

- **Defect proneness**: A binary variable indicating if a file is defective or not.

- **Defect rank**: A ranked list of files according to the number of defects that they will contain.
- **Defect count**: The exact number of defects in a file.
- **Effort-aware**: A cost-effective list of files ranked in order to locate the most number of defects while inspecting the least number of lines.

5.5.1 Research Questions

To investigate the impact that aggregation schemes have on our four types of defect prediction models, we formulate the following four research questions:

RQ2.1 Do aggregation schemes impact the performance of defect proneness models?

RQ2.2 Do aggregation schemes impact the performance of defect rank models?

RQ2.3 Do aggregation schemes impact the performance of defect count models?

RQ2.4 Do aggregation schemes impact the performance of effort-aware models?

5.5.2 Experimental Design

We now present the design of our experiments, including the evaluation method, the modelling techniques, the performance measures, the model training approach, and the null hypotheses. Figure 5.5 gives an overview of our approach to address RQs 2.1-2.4.

1) Evaluation method. In our experiment, we use the out-of-sample bootstrap model validation technique [51]. The out-of-sample bootstrap is a robust model validation technique that has been shown to provide stable results for unseen data [51]. The process is composed of two steps. First, a bootstrap sample of N rows is selected with replacement from

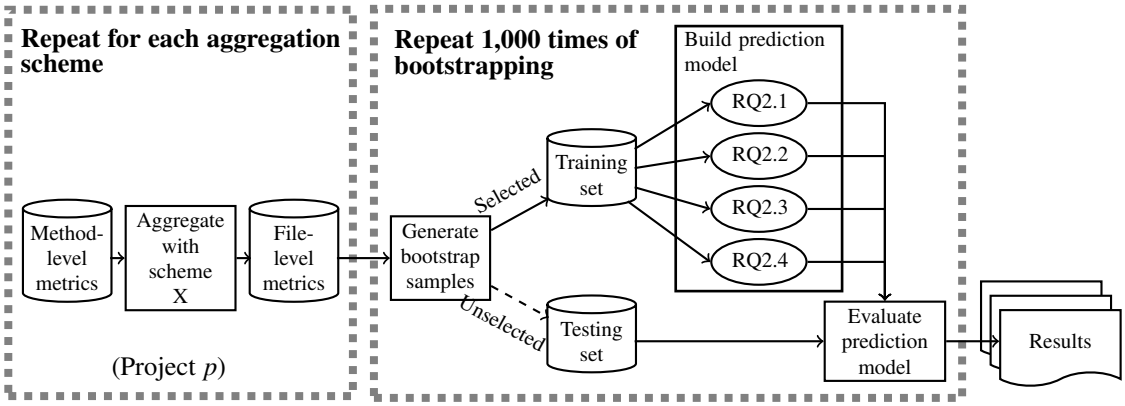


Figure 5.5: Our approach to build and evaluate defect prediction models on each of the studied 255 projects, using file-level metrics aggregated from method-level metrics (RQs 2.1 to 2.4).

an original dataset with N rows. Theoretically, the bootstrap sample will contain 63.2% of the unique rows that appear in the original dataset. Second, a model is trained using the bootstrap sample and tested using the 36.8% of the data from the original dataset that does not appear in the bootstrap sample. The two-step process is repeated several times, drawing a new bootstrap sample with replacement for training a model and testing it on the unselected data. The performance estimate is the average of the performance of each of these bootstrap-trained models. For each studied system, we perform 1,000 bootstrap iterations in order to derive a stable performance estimate.

2) Modelling techniques and performance measures

Defect proneness. We apply the random forest algorithm [23] to train our defect proneness models, since this algorithm tends to achieve better performance than other commonly used algorithms (e.g., regression model) in prior work [66, 98]. Common performance measures for defect proneness models include precision, recall, accuracy, and F-measure. These measures are calculated using a confusion matrix that is obtained using a threshold value. Since these performance measures are sensitive to the selected threshold, we opt to use the

Area Under the receiver operating characteristic Curve (AUC) — a threshold-independent performance measure. AUC is computed as the area under the Receiver Operating Characteristics (ROC) curve, which plots the true positive rate against the false positive rate while varying the internal model threshold. AUC values range between 0 (worst performance) and 1 (best performance). A model with an AUC of 0.5 or less performs no better than random guessing.

Defect rank. We apply linear regression to train our defect rank models. The regression model is applied to all files in the system and the files are ranked according to their estimated defect count. As suggested by prior work [131, 208], we use Spearman's ρ to measure the performance of our defect rank models. We compute ρ between the ranked list produced by the model and the correct ranking that is observed in the historical data. Larger ρ values indicate a more accurate defect rank model.

Defect count. Similar to our defect rank models, we apply linear regression to train our defect count models. We use the Mean Squared Error (MSE) to measure the performance of our linear models, which is defined as follows:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(\hat{Y}_i - Y_i)^2 \tag{5.2}$$

where Y_i and $\hat{Y}_i$ are the actual and predicted value of the ith file, and n is the total number of files. The lower the MSE, the better the performance of the defect count model.

Effort-aware. We also apply linear regression to train our effort-aware models. We use the P_{opt} measure proposed by Mende and Koschke [120] to measure the performance of our effort-aware models. The P_{opt} measure is calculated by drawing two curves (see Figure 5.6) that plot the accumulated lines of analyzed code on the x-axis, and the percentage of addressed bugs on the y-axis. First, the optimal curve is drawn using an ordering of files based on their actual defect densities. Second, the model performance curve is drawn by

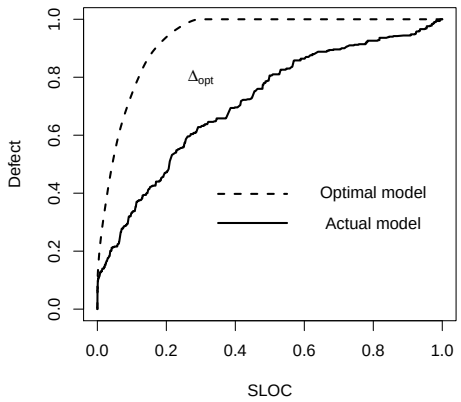


Figure 5.6: Example to illustrate the computation of Δ_{opt}.

Table 5.7: The modelling techniques and performance measures used in this study.

Prediction type	**Modelling technique**	**Performance measure**
Defect proneness	Random forest	AUC
Defect rank	Linear regression	Spearman's ρ
Defect count	Linear regression	MSE
Effort-aware defect count	Linear regression	P_{opt}

ordering files according to their predicted defect density. The area between the two curves is represented as Δ_{opt}, and $P_{opt} = 1 - \Delta_{opt}$. The higher the P_{opt} value, the closer the curve of the predicted model is to the optimal model, i.e., the higher the P_{opt} value, the better.

Table 5.7 summarizes the modelling techniques and performance measures for each type of defect prediction models.

3) Prediction model training

In each bootstrap iteration, we build 48 models — one model for each combination of the four types of defect prediction models and 12 configurations of the studied aggregation schemes. We use one configuration to study each of the 11 aggregation schemes individually, and a 12th configuration to study the combination of all of the aggregation schemes.

In each configuration, we use the six software metrics aggregated by the corresponding

scheme as predictors for our defect prediction models. Thus, for 11 configurations that only involve one aggregation scheme, we use six predictors, and for the configuration that involves all schemes, we use 66 (i.e., 6 method-level predictors $\times$ 11 schemes) predictors.

Since the predictors may be highly correlated with one another, they may introduce multicollinearity, which can threaten the fitness and stability of the models [189]. To address this concern, a common practice is to apply Principal Component Analysis (PCA) to the input set of predictors [48, 55]. Although principal components are difficult to interpret [175], the analysis of the impact of particular metrics is out of the scope of this thesis. Hence, we adopt this technique to simplify the process of building defect prediction models in this study. We order the principal components by the amount of explained variance, and select the first N principle components that can explain at least 95% [48] of variance for inclusion in our defect prediction models. In total, we train over 12 million (i.e., 48 configurations $\times$ 1000 iterations $\times$ 255 systems) models in our defect prediction experiment.

4) Null hypotheses

As the four types of prediction models are similar, we formulate two general hypotheses to structure our investigation of the impact that aggregation schemes have on defect prediction models. To enable the comparison, we create an ideal model that achieves the optimal performance. The optimal performance is the best performance of models that are obtained using any of the 12 studied configurations. We test the following two null hypotheses for each studied system:

$H0_{2a}$: *There is no difference in the performance of the optimal model and models that are trained using metrics that are aggregated by scheme AG.*

$H0_{2b}$: *There is no difference in the performance of the optimal model and models that are trained using metrics that are aggregated using all 11 schemes.*

To test our hypotheses, we conduct two-sided and paired Mann-Whitney U tests [173] with $\alpha = 0.05$. As we have 255 systems in total, we apply Bonferroni correction to control family-wise errors, and then adjust α by dividing by the number of tests. We use Cliff's δ to quantify the size of the impact.

5.5.3 Case Study Results

In this section, we present our findings from an overall perspective and a programming language-specific perspective.

1) General findings

The summation scheme (i.e., the most commonly applied aggregation scheme in the literature) can significantly underestimate the predictive power of defect prediction models. It is worthwhile to experiment with different aggregation schemes, since they show potential for improving the performance of defect prediction models. Table 5.8 shows the percentage of the studied systems where a model that is built using the corresponding configuration achieves similar predictive power to the optimal model. We consider that a model achieves the optimal performance if the p-value of Mann-Whitney U test is greater than α, or Cliff's $|\delta| < 0.474$ (i.e., does not exhibit large effect).

Table 5.8 shows that solely using summation achieves the optimal performance when predicting defect proneness in only 11% of projects. When predicting defect rank or performing effort-aware prediction, solely using summation yields the optimal performance in 56% and 31% of projects, respectively. Such findings suggest that the predictive power of

Table 5.8: The percentage of the studied systems on which the model built with the corresponding configuration of aggregations achieves the optimal performance. (The **bold** font highlights the best configuration).

Scheme	Defect proneness	Defect rank	Defect count	Effort-aware
All schemes	**102 (40%)**	**153 (60%)**	248 (97%)	42 (16%)
Sum	28 (11%)	143 (56%)	**253 (99%)**	79 (31%)
Mean	19 (7%)	33 (13%)	222 (87%)	176 (69%)
Median	21 (8%)	28 (11%)	210 (82%)	**180 (71%)**
SD	17 (7%)	37 (15%)	230 (90%)	124 (49%)
COV	24 (9%)	40 (16%)	238 (93%)	58 (23%)
Gini	21 (8%)	31 (12%)	231 (91%)	69 (27%)
Hoover	20 (8%)	28 (11%)	227 (89%)	83 (33%)
Atkinson	21 (8%)	37 (15%)	230 (90%)	106 (42%)
Shannon	36 (14%)	92 (36%)	246 (96%)	51 (20%)
Entropy	25 (10%)	39 (15%)	229 (90%)	103 (40%)
Theil	19 (7%)	42 (16%)	232 (91%)	77 (30%)

defect prediction models can be hindered by solely relying on summation for aggregating metrics.

On the other hand, using all of the studied aggregation schemes is significantly better than solely using summation in models that predict defect proneness. Specifically, using all schemes achieves the optimal performance in 40% of projects. This finding indicates that exploring various aggregation schemes can yield fruitful results when building models to predict defect proneness.

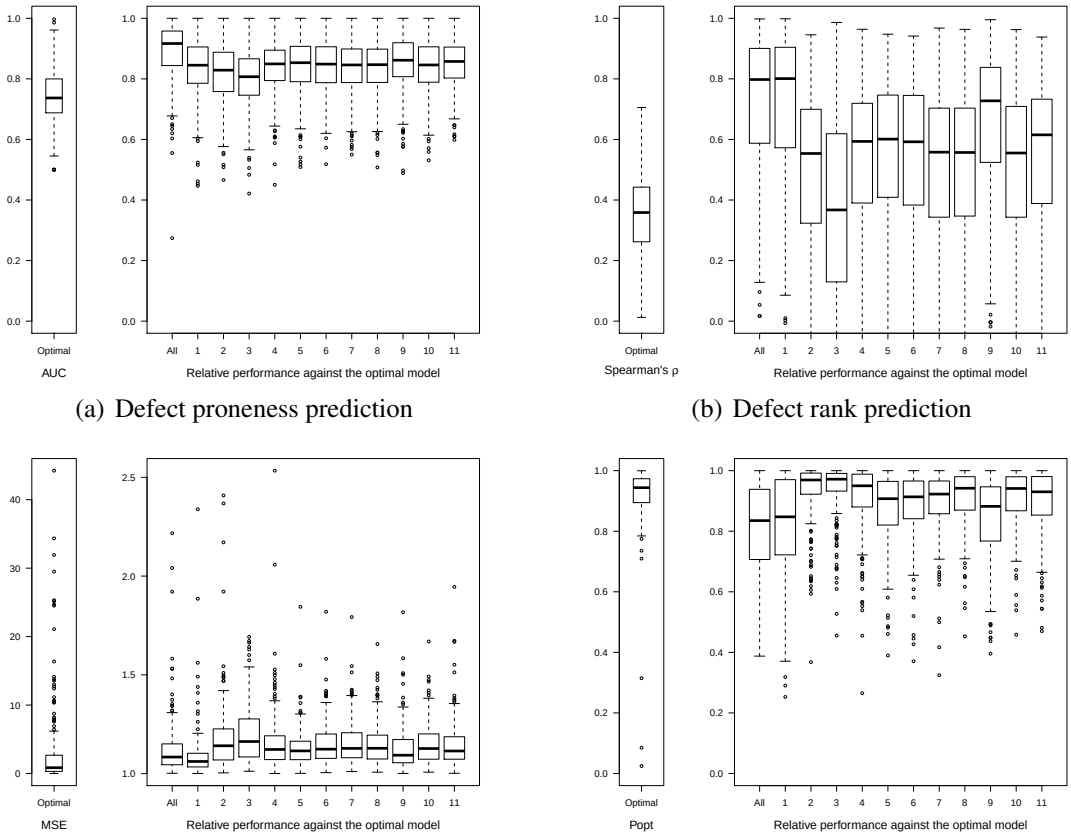
In models that predict defect rank and count, the difference between using all schemes and solely using summation is marginal, and both are closer to the optimal performance than any other aggregation scheme. In models that predict defect rank, using all schemes is slightly better than solely using summation (i.e., 60% vs. 56%). When predicting defect count, solely using summation is slightly better than using all schemes (i.e., 99% vs. 97%).

When fitting effort-aware models, the situation changes, i.e., neither using all schemes nor solely using summation is advisable. The median scheme provides the optimal performance in 71% of projects. Using the mean scheme is a viable alternative, as it achieves the optimal performance in 69% of projects. Both mean and median aggregation schemes are

much better than using any other configuration of aggregators.

We suspect that using either the median or the mean scheme yields the top performance for effort-aware models because these two schemes capture the effort that is required to inspect a method. For example, considering the following toy model to predict defect count: $Y = c * avg\_LOC$, where c is a constant in the fitted model. The effort to inspect a file (i.e., lines of code [98]) is calculated using $E = m * avg\_LOC$, where m is the number of methods in a file. Then the predicted defect density of a file is computed as: $density = \frac{Y}{E} = \frac{c*avg\_LOC}{m*avg\_LOC} = \frac{c}{m}$. Among the files with the same avg_LOC, files with more methods have less defect density, and are therefore ranked below files with fewer methods. This is in agreement with the goal of effort-aware defect prediction, i.e., finding as many defects as possible, while expending the least amount of effort.

Figure 5.7 provides boxplots of the optimal performance of our various model configurations, together with the performance of models built using each configuration relative to the optimal model. Figure 5.7 shows that when using all schemes together, the performances of defect proneness models are generally greater than using a single scheme. Furthermore, when predicting defect rank and count, solely using summation or using all schemes achieve very similar amounts of predictive power, and both configurations are generally better than using any other aggregation scheme. Hence, applying all schemes together is beneficial for defect proneness models, while using summation is likely sufficient for models that predict defect rank and count. Moreover, when building effort-aware models, either using mean or median generally achieves better performance than using any other configuration. Hence, the median or mean schemes are advisable for building effort-aware models.



(a) Defect proneness prediction

(b) Defect rank prediction

(c) Defect count prediction

(d) Effort-aware defect prediction

Figure 5.7: In each sub figure, the left boxplot shows the optimal performance, and the right boxplots present the performance by models built with each aggregation scheme relative to the optimal performance. The order of aggregation schemes: all schemes, summation, mean, median, SD, COV, Gini, Hoover, Atkinson, Shannon's entropy, generalized entropy, and Theil.

2) Programming language-specific findings

The distribution of software metrics tends to vary based on the programming language in which the system is implemented [201]. This varying distribution may interfere with our analysis of aggregation schemes. To investigate the role that a programming language plays, we partition the results of Table 5.8 according to programming languages, and present the results in Table 5.9.

Table 5.9: The percentage of the studied systems per programming language, on which the model built with the corresponding aggregation scheme achieves similar predictive power as the optimal model. (The **bold** font highlights the best performing scheme.)

(a) Defect proneness

Scheme	Programming language			
	C	C++	C#	Java
All schemes	**9 (26%)**	**28 (33%)**	**3 (20%)**	**62 (51%)**
Sum	2 (6%)	11 (13%)	0 (0%)	15 (12%)
Mean	2 (6%)	5 (6%)	**3 (20%)**	9 (7%)
Median	3 (9%)	9 (11%)	0 (0%)	9 (7%)
SD	3 (9%)	5 (6%)	2 (13%)	7 (6%)
COV	5 (15%)	5 (6%)	1 (7%)	13 (11%)
Gini	2 (6%)	9 (11%)	2 (13%)	8 (7%)
Hoover	6 (18%)	4 (5%)	1 (7%)	9 (7%)
Atkinson	2 (6%)	7 (8%)	1 (7%)	11 (9%)
Shannon	4 (12%)	15 (18%)	1 (7%)	16 (13%)
Entropy	4 (12%)	10 (12%)	1 (7%)	10 (8%)
Theil	3 (9%)	4 (5%)	2 (13%)	10 (8%)

(b) Defect rank

Scheme	Programming language			
	C	C++	C#	Java
All schemes	**19 (56%)**	**54 (64%)**	**6 (40%)**	74 (61%)
Sum	16 (47%)	46 (54%)	**6 (40%)**	**75 (62%)**
Mean	5 (15%)	10 (12%)	2 (13%)	16 (13%)
Median	9 (26%)	2 (2%)	1 (7%)	16 (13%)
SD	6 (18%)	10 (12%)	4 (27%)	17 (14%)
COV	6 (18%)	15 (18%)	3 (20%)	16 (13%)
Gini	6 (18%)	10 (12%)	1 (7%)	14 (12%)
Hoover	5 (15%)	7 (8%)	2 (13%)	14 (12%)
Atkinson	4 (12%)	15 (18%)	2 (13%)	16 (13%)
Shannon	13 (38%)	36 (42%)	3 (20%)	40 (33%)
Entropy	4 (12%)	17 (20%)	2 (13%)	16 (13%)
Theil	5 (15%)	15 (18%)	2 (13%)	20 (17%)

(c) Defect count

Scheme	Programming language			
	C	C++	C#	Java
All schemes	**34 (100%)**	**84 (99%)**	13 (87%)	117 (97%)
Sum	**34 (100%)**	**84 (99%)**	**15 (100%)**	**120 (99%)**
Mean	31 (91%)	76 (89%)	14 (93%)	101 (83%)
Median	29 (85%)	71 (84%)	13 (87%)	97 (80%)
SD	32 (94%)	76 (89%)	14 (93%)	108 (89%)
COV	33 (97%)	80 (94%)	14 (93%)	111 (92%)
Gini	32 (94%)	76 (89%)	14 (93%)	109 (90%)
Hoover	32 (94%)	76 (89%)	14 (93%)	105 (87%)
Atkinson	33 (97%)	76 (89%)	14 (93%)	107 (88%)
Shannon	34 (100%)	83 (98%)	14 (93%)	115 (95%)
Entropy	33 (97%)	76 (89%)	14 (93%)	106 (88%)
Theil	33 (97%)	79 (93%)	14 (93%)	106 (88%)

(d) Effort-aware defect count

Scheme	Programming language			
	C	C++	C#	Java
All schemes	8 (24%)	18 (21%)	1 (7%)	15 (12%)
Sum	12 (35%)	28 (33%)	8 (53%)	31 (26%)
Mean	26 (76%)	**62 (73%)**	11 (73%)	77 (64%)
Median	**27 (79%)**	54 (64%)	**12 (80%)**	**87 (72%)**
SD	19 (56%)	51 (60%)	6 (40%)	48 (40%)
COV	12 (35%)	20 (24%)	3 (20%)	23 (19%)
Gini	12 (35%)	26 (31%)	6 (40%)	25 (21%)
Hoover	15 (44%)	30 (35%)	5 (33%)	33 (27%)
Atkinson	17 (50%)	35 (41%)	8 (53%)	46 (38%)
Shannon	10 (29%)	16 (19%)	1 (7%)	24 (20%)
Entropy	17 (50%)	32 (38%)	8 (53%)	46 (38%)
Theil	12 (35%)	30 (35%)	4 (27%)	31 (26%)

Irrespective of the programming language, the impact that aggregation schemes have on defect prediction models remains largely consistent. For instance, using all schemes is generally beneficial to most of the studied systems when predicting defect proneness, no matter what programming language the system is written in. When predicting defect rank, using all schemes achieves results that are the closest to the performance of the optimal model for projects developed in C and C++, while using summation is slightly

better than using all schemes for only one project developed in Java. For projects developed in C# or Java, solely using summation in models that predict defect count is slightly better than using all schemes with a difference of two and three projects, respectively. When building effort-aware models, using the median scheme is beneficial to most of the systems written in C, C#, and Java. For systems written in C++, using the mean scheme achieves results that are slightly closer to the optimal performance than using median. Hence, we conclude that the impact of aggregation schemes is largely consistent across systems developed in any of the four studied programming languages.

Solely using summation rarely leads to the optimal performance in models that predict defect proneness or effort-aware models, where using all schemes and using mean/median are recommended, respectively. Moreover, using all schemes is still beneficial to defect rank models, especially for projects written in C and C++. In models that predict defect count, solely using summation is sufficient. Indeed, applying all schemes is a low-cost option that is worth experimenting with.

5.5.4 Guidelines for Future Defect Prediction Studies

In this section, we discuss the broader implications of our results by providing guidelines for future defect prediction studies:

(G1) Regardless of the programming language, using all studied aggregation schemes is recommended when building models for predicting defect proneness and rank. In particular, defect proneness models that use all aggregation schemes achieve the optimal performance in 40% of the studied systems, while solely using the summation scheme achieves the optimal performance in only 11% of projects. Furthermore, for models that rank files according to their defect density, using all schemes is better than solely using summation for projects developed that are in C and C++.

(G2) Using summation is recommended for defect count models. Solely using summation is better than using all schemes for projects that are developed in C# or Java, and leads to the same predictive power as using all schemes for projects that are developed in C and C++.

(G3) Either the mean or the median aggregation scheme should be used in effort-aware defect prediction models. In particular, the median aggregation scheme should be used for projects developed in C, C#, or Java. The mean aggregation scheme is suggested when building effort-aware defect prediction models for C++ projects. In general, using median achieves the optimal performance for 71% of the studied systems.

5.6 Threats to Validity

In this section, we discuss the threats to the validity of our study with respect to Yin's guidelines for case study research [199].

Threats to conclusion validity are concerned with the relationship between the treatment and the outcome. The threat to our treatments mainly arises from our choice of metrics (i.e., only six method-level metrics, and no class-level metrics). However, the primary goal of our study is not to train the most effective defect prediction models, but instead to measure relative improvements by exploring different aggregation schemes.

Threats to internal validity are concerned with our selection of subject systems and analysis methods. As the majority of systems that are hosted on SourceForge and GoogleCode are immature, we carefully filter out systems that have not accumulated sufficient history to train defect prediction models. To obtain a stable picture of the performance of

our defect prediction models, we perform 1,000 iterations of out-of-sample bootstrap validation. In addition, we apply non-parametric statistical methods (i.e., Mann-Whitney U test and Cliff's δ) to draw our conclusions.

Threats to external validity are concerned with the generalizability of our results. We investigate 11 schemes that can capture five aspects (summation, central tendency, dispersion, inequality index, and entropy) of the distribution of software metrics. Moreover, we study 255 open source systems that are drawn from a broad range of domains. Hence, we believe that our conclusions may apply to other defect prediction contexts. Nonetheless, replication studies could be advised.

Threats to reliability validity are concerned with the possibility of replicating this study. Our subject projects are all open source systems, and the tool for computing software metrics is publicly accessible. Furthermore, we provide details of our experiments in a replication package that we have posted online[2].

5.7 Chapter Summary

Aggregation is an unavoidable step in training defect prediction models at the file-level. This is because defect data is often collected at file-level, but many software metrics are computed at the method- and class-levels. One of the widely used schemes for metric aggregation is summation [111, 112, 128, 135, 138, 154, 202, 208, 209]). However, recent work [109] suggests that summation can inflate the correlation between SLOC and CC in Java projects. Fortunately, there are many other aggregation schemes that capture other dimensions of a low-level software metric (e.g., dispersion, central tendency, inequality, and entropy). Yet, the impact that these additional aggregation schemes have on defect

[2] http://www.feng-zhang.com/replication/aggregation

prediction models remains largely unexplored.

To that end, we perform experiments using 255 open source systems to explore how aggregation schemes impact the performance of defect prediction models. First, we investigate the impact that aggregation schemes have on the correlation among metrics and the correlation between metrics and defect count. We find that aggregation can increase or decrease both types of correlation. Second, we examine the impact that aggregation schemes have on defect proneness, defect rank, defect count, and effort-aware defect prediction models. Broadly speaking, we find that summation tends to underestimate the performance of defect proneness and effort-aware models. Hence, it is worth applying multiple aggregation schemes for defect prediction purposes. For instance, applying all 11 schemes achieves the optimal performance in predicting defect proneness in 40% of the studied projects.

From our results, we provide the following guidelines for future defect prediction studies. When building models for predicting defect proneness and rank, our recommendation is to use all of the available aggregation schemes to generate the initial set of predictors (i.e., aggregated metrics), and then perform PCA or feature selection to remove redundancies. For models that predict defect count, solely using summation is likely sufficient. To generate the initial set of predictors for effort-aware defect prediction models, the median scheme is advised for projects developed in C, C#, or Java, and the mean scheme is suggested for projects written in C++.

Given that the computation cost for these additional aggregation schemes is negligible, we strongly suggest researchers and practitioners experiment with many aggregation schemes when building defect prediction models.

Part IV

Generalizing Defect Prediction Models

Chapter 6 Supervised Approach

Key Question

? *Is it feasible to generalize a defect prediction model that is built using a supervised approach?*

6.1 Introduction

Most approaches apply a supervised classifier to build defect prediction models [45]. A supervised classifier requires the training data to build a model and is applied on the target data. The heterogeneity between the training and target data threatens the performance of cross-project defect prediction. To overcome such a challenge, there are two major approaches: 1) use data from projects with a similar distribution of predictors to the target project as training data (e.g., [124, 188]); or 2) transform predictors in both training and target projects to make them more similar in their distribution (e.g., [118, 135]).

However, the first approach uses partial dataset and results in multiple models for different target projects. The transformation approaches are typically specialized to a particular pair of training and testing datasets. In Chapter 3, we found the distribution of software metrics varies with project contexts (e.g., size and programming language). Therefore, we

combine the three insights in an attempt to build a universal defect prediction model for a large set of projects with diverse contexts.

In Chapter 4, we observed that three transformation methods (i.e., log, rank, and Box-Cox) result in similar performance in cross-project defect prediction. Among these three transformations, rank transformation has the advantage that it makes transformed software metrics have exactly the same scales across projects. Therefore, in this study, we propose a context-aware rank transformation to address the variations in the distribution of predictors before fitting them in the universal defect prediction model. We refer to a single model that is built from the entire set of projects as a universal model. There are six context factors investigated in this study, i.e., programming language, issue tracking, the total lines of code, the total number of files, the total number of commits, and the total number of developers. The context-aware approach stratifies the entire set of projects by context factors, and clusters the projects with a similar distribution of predictors. Inspired by metric-based benchmarks (e.g., [4]), which use quantiles to derive thresholds for ranking software quality, we apply every tenth quantile of predictors on each cluster to specify ranking functions. We use twenty-one code metrics and five process metrics as predictors. After rank transformation, the predictors from different projects will have exactly the same scale. The universal model is then built using the transformed predictors.

We apply our approach on 1,385 open source projects hosted on SourceForge and GoogleCode. We observe that the F-measures and area under curve (AUC) obtained using rank-transformed predictors is comparable to that of logarithmically transformed predictors. The logarithmic transformation uses the logarithmic values of predictors, and is commonly used to build prediction models. After adding the six context factors as predictors, the performance of the universal model built using only code and process metrics can

be further improved. The universal model yields higher AUC than within-project models. Moreover, the universal model achieves up to 48% of the successful predictions of within-project models using loose criteria (i.e., recall is above 0.70, and precision is greater than 0.50) suggested by He *et al.* [76] to determine the success of defect prediction models.

We examine the generalizability of the universal model in two ways. First, we build the universal model using projects hosted on SourceForge and GoogleCode, and apply the universal model on five external projects that are neither hosted on SourceForge nor GoogleCode, including Lucene, Eclipse, Equinox, Mylyn, and Eclipse PDE. The results show that the universal model provides a similar performance (in terms of AUC) as within-project models for the five projects. Second, we compare the performance of the universal model on projects of different context factors. The results indicate that the performance does not change significantly among projects with different context factors. These results suggest that the universal model is context-insensitive and generalizable.

In summary, the major contributions of our study are:

- **Propose an approach of context-aware rank transformation.** The rank transformation method addresses the problem of large variations in the distribution of predictors across projects from diverse contexts. The transformed predictors have exactly the same scales. This enables us to build a universal model for a large set of projects.

- **Improve the performance of the universal model by adding context factors as predictors.** We add the context factors to our universal prediction model, and find that context factors significantly improve the predictive power of the universal defect prediction model (e.g., AUC increases from 0.61 to 0.64 comparing to the combination of code and process metrics).

- **Provide a universal defect prediction model:** The universal model achieves similar

performance as within-project models for five external projects, and does not show significant difference in the performance for projects with different context factors. The universal model is context-insensitive and generalizable. We also provide the estimated coefficients of predictors for the universal model.

Chapter organization. Section 6.2 and Section 6.3 describe our approach and experiment design, respectively. Section 6.4 presents our results and discussions. The threats to validity of our work are discussed in Section 6.5. We summarize the chapter in Section 6.6.

6.2 Approach

We now present the details of our approach for building a universal defect prediction model.

6.2.1 Overview

The poor performance of cross-project prediction may be caused by the significant differences in the distribution of metric values among projects [43, 135]. Therefore, to build a universal model using a large set of projects, it is essential to reduce the difference in distribution of metric values across projects. Our previous work [201] finds that context factors of projects can significantly affect the distribution of metric values. Therefore, we propose a context-aware rank transformation approach to pre-process metric values before fitting them to the universal model. Figure 6.1 shows the following four steps of our approach.

(S1) Partitioning projects. We partition the entire set of projects into non-overlapping groups based on the six context factors (i.e., programming language, issue tracking, the total lines of code, the total number of files, the total number of commits, and the total number of developers). This step aims to reduce the number of pairwise

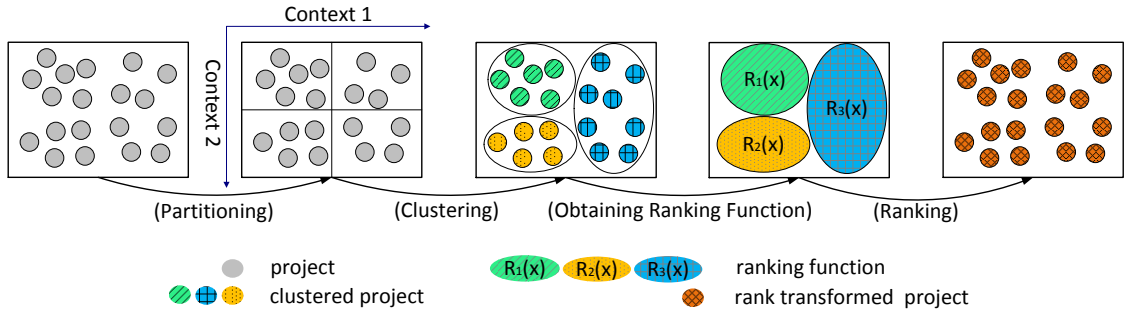


Figure 6.1: Our four-step rank transformation approach: 1) stratify the set of projects along different contexts into non-overlap groups; 2) cluster project groups; 3) derive ranking function for each cluster; and 4) perform rank transformation.

comparisons. We compare the distribution of metric values across groups of projects instead of individual projects.

(S2) Clustering projects. We cluster the project groups with similar distributions of predictor values. This step aims to merge similar groups of projects so that we could include more projects in each cluster for obtaining ranking functions.

(S3) Obtaining ranking functions. We derive a ranking function for each cluster using every $10th$ quantile of predictor values. This transformation removes large variations in the distribution of predictors transforming them to exactly the same scale.

(S4) Ranking software metrics. We apply the ranking functions to convert the raw values of predictors to one of the ten levels. This step aims to remove the difference in the scales of metric values from different projects. The scales of the transformed metric values are exactly the same for all projects.

After the preprocessing steps, we build the universal model based on the transformed predictors. The following subsections describe the context factors used in this study, and the details of each step.

6.2.2 Context Factors

Software projects have diverse context factors. However, it is still unclear what context factors best characterize projects. For instance, Nagappan *et al.* [130] choose seven context factors based on their availability in Ohloh[1], including main programming language, the total lines of code, the number of contributors, the number of churn, the number of commits, project age, and project activity. Along the same lines, our previous work [201] selects seven context factors, i.e., application domain, programming language, age, lifespan, the total lines of code, the number of changes, and the number of downloads. The shared context factors of the aforementioned two studies are programming language, the total lines of code, and the number of commits. These factors are common to all projects with version control systems, and are included in this study. Three factors (i.e., project age, lifespan, and the number of downloads) do not significantly affect the distribution of metric values [201], thus are excluded from this study. The information of application domain is unavailable to our subject projects that are hosted on GoogleCode. Thus we exclude application domain as well. Moreover, we add the number of developers as Nagappan *et al.* [130], and the number of files as size measurement. Finally, we choose the following six context factors.

(C1) ***Programming language (PL)*** describes the nature of programming paradigms. There is a high chance for metric values of different programming languages to experience significantly different distributions. Moreover, it is interesting to investigate the possibility of inter language reuse of prediction models. Due to the limitation of our metric computing tool, we only consider projects mainly written in C, C++, Java, C#, or Pascal. A project is *mainly written* in programming language *pl* if the largest number of source code files are written in *pl*.

[1]https://www.openhub.net (NOTE: 'Ohloh' was changed to 'Open Hub' in 2014.)

(C2) ***Issue tracking (IT)*** describes whether a project uses an issue tracking system or not. The usage of an issue tracking system can reflect the quality of the project management process. It is likely that the distribution of metric values are different between projects with or without usage of issue tracking systems. A project uses an issue tracking system if the issue tracking system is enabled in the website of the project and there is at least one issue recorded.

(C3) ***Total lines of code (TLOC)*** describes the project size in terms of source code. Comment and blank lines are excluded when counting the total lines of code. Moreover, the lines of code of files that are not written in the main language of the project are also excluded. Such exclusion simplifies our approach for transforming metric values, as only one programming language is considered for each project.

(C4) ***Total number of files (TNF)*** describes the project size in terms of files. This context factor measures the project size from a different granularity to the total lines of code. Similar to the total lines of code measurement, we exclude files that are not written in the main language of each project.

(C5) ***Total number of commits (TNC)*** describes the project size in terms of commits. Different from the total lines of code and the total number of files, this context factor captures the project size from the process perspective. The total number of commits can describe how actively the project was developed.

(C6) ***Total number of developers (TND)*** describes the project size in terms of developers. Teams of different sizes (e.g., small or large) may follow different development strategies, therefore the team size can impact the distribution of metric values.

6.2.3 Partitioning Projects

We assume that projects with the same context factors have similar distribution of software metrics, and projects with different contexts might have different distribution of software metrics. Hence, we stratify the entire set of projects based on the aforementioned six context factors.

(C1) ***PL.*** We divide the set of projects into 5 groups based on programming languages: G_c, G_{c++}, G_{java}, $G_{c\#}$, and G_{pascal}.

(C2) ***IT.*** The set of projects is separated into 2 groups based on the usage of an issue tracking system: G_{useIT} and G_{noIT}.

(C3) ***TLOC.*** We compute the TLOC of each project and the quartiles of TLOC. Based on the first, second, and third quartiles, we split the set of projects into 4 groups: $G_{leastTLOC}$, $G_{lessTLOC}$, $G_{moreTLOC}$, and $G_{mostTLOC}$.

(C4) ***TNF.*** We calculate TNF of each project, and the quartiles of TNF. Based on the first, second, and third quartiles, we separate the set of projects into 4 groups: $G_{leastTNF}$, $G_{lessTNF}$, $G_{moreTNF}$, and $G_{mostTNF}$.

(C5) ***TNC.*** We compute the TNC of each project, and the quartiles of TNC. Based on the first, second, and third quartiles, we break the entire set of projects into 4 groups: $G_{leastTNC}$, $G_{lessTNC}$, $G_{moreTNC}$, and $G_{mostTNC}$.

(C6) ***TND.*** We calculate the TND of each project, and the quartiles of TND. Based on the first, second, and third quartiles, we split the whole set of projects into 4 groups: $G_{leastTND}$, $G_{lessTND}$, $G_{moreTND}$, and $G_{mostTND}$.

In summary, we get 5, 2, 4, 4, 4, and 4 non-overlapping groups along each of the six context factors, respectively. In total, we obtain 2560 (i.e., $5 \times 2 \times 4 \times 4 \times 4 \times 4$) non-overlapping groups for the entire set of projects.

6.2.4 Clustering Similar Projects

In the previous step, we obtain non-overlapping groups of projects. However, the size of most groups is small. In some cases the non-overlapping groups of projects do not have significantly different distributions of metrics. In addition, clustering similar projects together helps obtain more representative quantiles of a particular metric. At this step, we cluster the projects with a similar distribution of a metric. We consider two distributions to be similar if neither their difference is statistically significant nor the effect size of their difference is large, as our previous study [201].

For different metrics, the corresponding clusters are not necessarily the same. In other words, we produce a particular set of clusters for each individual metric. We describe a cluster using a vector. The first element shows for what metric the cluster is created, and the remaining elements characterize the cluster from the context factor perspective. For example, the cluster $< m, C{+}{+}, useIT, moreTLOC >$ is created for metric m, contains $C{+}{+}$ projects that use issue tracking systems, and has the TLOC between the second and third quartiles (see Section 6.2.2).

For each metric m, the clusters of projects with a similar distribution of metric m are obtained using Algorithm 6.1. Algorithm 6.1 has two major steps:

1) Comparing the Distribution of Metrics. This step (Line 8 in Algorithm 6.1) merges the groups of projects that do not have significantly different distribution of metric m.

Algorithm 6.1: Clustering Similar Projects

```
   Input: m: the metric m
          N: the number of groups
   Output: clusterOfGroup: the cluster index of projects
   /* Initialize the array clusterOfGroup.                                          */
1  int indexOfCluster = 1;
2  for i = 1 to N do
3  |   clusterOfGroup[i] = indexOfCluster;
4  end
   /* Do the clustering.                                                            */
5  for i = 1 to N − 1 do
6  |   int indexNewCluster = indexOfCluster+1;
7  |   for j = i + 1 to N do
   |   |   /* Compare the distribution of metric values between two groups i and j. */
8  |   |   compareMetricDistribution(m, i, j);
9  |   |   if the difference is statistically significant then
   |   |   |   /* Quantify the importance of the difference.                        */
10 |   |   |   computeCliffsDelta(i, j);
11 |   |   |   if Cliff's δ is large then
   |   |   |   |   /* Put group i and j in different clusters.                     */
12 |   |   |   |   if clusterOfGroup[j] equals to clusterOfGroup[i] then
   |   |   |   |   |   /* Put group j in a new cluster.                           */
13 |   |   |   |   |   clusterOfGroup[j] = indexNewCluster;
   |   |   |   |   |   /* Update the base counter to compute new clusters.       */
14 |   |   |   |   |   indexOfCluster = indexNewCluster;
15 |   |   |   |   end
16 |   |   |   end
17 |   |   end
18 |   end
19 end
```

We apply Mann-Whitney U test [173] to compare the distribution of metric values between every two groups of projects, using the 95% confidence level (i.e., p-value<0.05). The Mann-Whitney U test assesses whether two independent distributions have equally large values. It is a non-parametric statistical test. Therefore it does not assume a normal distribution. As we conduct multiple tests to investigate the distribution of each metric, we apply Bonferroni correction to control family-wise errors. Bonferroni adjusts the threshold p-value by dividing it by the number of tests.

2) **Quantifying the Difference between Distributions.** This step (Lines 10 to 16 in Algorithm 6.1) merges the groups of projects that have significantly different distributions

of metric m, but the difference is not large. We calculate Cliff's δ (Line 10 in Algorithm 6.1) as the effect size [160] to quantify the importance of the difference between the distribution of every two groups of projects. Cliff's δ estimates non-parametric effect sizes. It makes no assumptions of a particular distribution, and is reported [160] to be more robust and reliable than Cohen's d [38]. Cliff's δ represents the degree of overlap between two sample distributions [160]. It ranges from -1 (if all selected values in the first group are larger than the second group) to +1 (if all selected values in the first group are smaller than the second group). It is zero when two sample distributions are identical [36]. Cohen's standards (i.e., small, medium, and large) are commonly used to interpret effect size. Therefore, we map the Cliff's δ to Cohen's standards, using the percentage of non-overlap [160]. The mapping between the Cliff's δ and Cohen's standards is shown in Table 3.6. Cohen [39] states that a medium effect size represents a difference likely to be visible to a careful observer, while a large effect is noticeably greater than medium. In this study, we choose the large effect size as the threshold of the importance of the differences between the distributions (Line 11 in Algorithm 6.1).

6.2.5 Obtaining Ranking Functions

In Section 6.2.4, we create clusters of projects for each metric, independently. For a particular metric, projects within the same cluster exhibit similar distribution of values of the corresponding metric. To remove the variation in the scales of metric values, this step derives ranking functions for each cluster. The ranking function transforms the raw metric values to predefined values (i.e., ranging from one to ten). Therefore, the transformed metrics have exactly the same scale among the projects.

Table 6.1: An example of ranking functions.

Range of Metric Value	[0, 11]	(11, 22]	(22, 33]	...	(99, +∞)
$Rank(m_1, Cl_{12}, value)$	1	2	3	...	10

We use the quantiles of metric values to formulate our ranking functions. This is inspired by metric-based benchmarks (e.g., [4]), which often use the quantiles to derive thresholds of metrics. The thresholds of metrics are used to distinguish files of different quality related to defects.

Let M denote the total number of metrics. For metric m_i (where $i \in \{1, \ldots, M\}$), the corresponding clusters are represented using Cl_{i1}, Cl_{i2}, ..., and Cl_{iN_i}, where N_i is the total number of clusters obtained for metric m_i. We formulate the ranking function for metric m_i in the jth cluster C_{ij} following Equation (6.1).

$$Rank(m_i, Cl_{ij}, value) = \begin{cases} 1 & \text{if } value \in [0, Q_{ij,1}(m_i)] \\ k & \text{if } value \in (Q_{ij,k-1}(m_i), Q_{ij,k}(m_i)] \\ 10 & \text{if } value \in (Q_{ij,9}(m_i), +\infty) \end{cases} \tag{6.1}$$

where the variable value denotes the value of metric m_i to be converted, $Q_{ij,k}(m_i)$ is the $k * 10th$ quantile of metric m_i in cluster Cl_{ij}, $j \in \{1, \ldots, N_i\}$, and $k \in \{2, \ldots, 9\}$.

For example, we assume that every $10th$ quantile for a metric m_1 in cluster Cl_{12} is: 11, 22, 33, 44, 55, 66, 77, 88, and 99, respectively. The ranking function for metric m_1 in cluster Cl_{ij} is shown in Table 6.1. If metric m_1 has a value of 27 in a file of a project that belongs to cluster Cl_{12}, then the metric value in the file will be converted to 3. This is because the value 27 is greater than 22 (i.e., the 20% quantile) and less than 33 (i.e., the 30% quantile).

6.2.6 Building a Universal Defect Prediction Model

1) **Choice of Modelling Techniques.** Lessmann *et al.* [112] and Arisholm *et al.* [8] show that there is no significant difference among different modelling techniques in the performance of defect prediction models. However, Kim *et al.* [102] find that Bayes learners (i.e., Bayes Net and Naive Bayes) perform better when defect data contains noise, even up to 20%-35% of false positive and false negative noise in defect data. Based on their findings, we apply Naive Bayes as the modelling technique in our experiments to evaluate the performance of the universal defect prediction model. When investigating the importance of different metrics in the universal model, we apply a logistic regression model as it is common practice to compare the importance of different metrics [211].

2) **Steps to Build the Universal Defect Prediction Model.** Our universal model is built upon the entire set of projects using rank transformed metric values. The first step is to transform metric values using ranking functions that are obtained from our dataset. In order to locate the ranking function for metric m_i in project p_j, we need to determine which cluster project p_j belongs to. We identify context factors of project p_j, and formulate a vector like $< m_i, C{+}{+}, useIT, moreTLOC, lessTNF, lessTNC, lessTND >$ to present a cluster, where the first item specifies the metric, and the remaining items describe the corresponding context factors that projects in this cluster belong to. The vector of project p_j is then compared to the vectors of all clusters. The exactly matched cluster is the cluster that project p_j belongs to. After the transformation, the values of metrics will have the same scale ranging from one to ten.

   The second and the last step is to build the model. We apply the Naive Bayes algorithm[2]

[2]https://weka.sourceforge.net/doc.dev/weka/classifiers/bayes/NaiveBayes.html

implemented in Weka[3] tool to build a universal defect prediction model upon the entire set of projects.

6.2.7 Measuring the Performance

To evaluate the performance of prediction models, we compute the confusion matrix as shown in Table 4.3. In the confusion matrix, true positive (TP) is the number of defective files that are correctly predicted as defective files; false negative (FN) counts the number of defective files that are incorrectly predicted as clean files; false positive (FP) measures the number of files that are clean but incorrectly predicted as defective; and true negative (TN) represents the number of clean files that are correctly predicted as clean files.

We calculate the following six measures (i.e., precision, recall, false positive rate, F-measure, g-measure, and Matthews correlation coefficient) using the confusion matrix. We also compute the area under curve (AUC) as an additional measure.

- Precision ($prec$) measures the proportion of actual defective entities that are predicted as defective against all predicted defective entities. It is defined as: $prec = \frac{TP}{TP+FP}$.
- Recall (pd) evaluates the proportion of actual defective entities that are predicted as defective against all actual defective entities. It is defined as: $pd = \frac{TP}{TP+FN}$.
- False positive rate (fpr) is the proportion of actual non-defective entities that are predicted as defective against all actual non-defective entities. It is defined as: $fpr = \frac{FP}{FP+TN}$.

[3]http://www.cs.waikato.ac.nz/ml/weka

- F-measure calculates the harmonic mean of precision and recall. It balances precision and recall. It is defined as: $F\text{-}measure = \frac{2 \times pd \times prec}{pd + prec}$.

- g-measure computes the harmonic mean of recall and 1-fpr. The 1-fpr represents *Specificity* (not predicting entities without defects as defective). We report g-measure as Peters *et al.* [151], since Menzies *et al.* [122] show that precision can be unstable when datasets contain a low percentage of defects. It is defined as: $g\text{-}measure = \frac{2 \times pd \times (1 - fpr)}{pd + (1 - fpr)}$.

- Matthews correlation coefficient (MCC) is a balanced measure of true and false positives and negatives. It ranges from -1 to +1, where +1 indicates a perfect prediction, 0 means the prediction is close to random prediction, and -1 represents total disagreement between predicted and actual values. It is defined as: $MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP+FP) \times (TP+FN) \times (TN+FP) \times (TN+FN)}}$.

- Area under curve (AUC) is the area under the receiver operating characteristics (ROC) curve. ROC is independent of the cut-off value that is used to compute the confusion matrix.

Moreover, the confusion matrix can be reconstructed from precision, recall, and d [71], where d represents the proportion of correct predictions (i.e., $d = TP + TN$). We can compute d using precision ($prec$), recall (pd), and false positive rate (fpr) as follows:

$$d = \frac{prec \times fpr}{pd \times (1 - prec) + prec \times fpr} \tag{6.2}$$

6.3 Experiment Setup

6.3.1 Subject Projects

SourgeForge and GoogleCode are two large and popular repositories for open source projects. We use the SourceForge and GoogleCode data initially collected by Mockus [125] with his updates until October 2010. The dataset contains the full history of about 154K projects that are hosted on SourceForge and 81K projects that are hosted on GoogleCode to the date they were collected. The file contents of each revision and commit logs are stored separately and linked together using a universal unique identifier. The storage of file contents of SourceForge and GoogleCode projects distributes in 100 database files. Each database file is about 8 Giga bytes. The storage of commit logs distributes in 13 compressed files that have a total size of about 10 Giga bytes. Although we have 235K projects in total, there are too many trivial projects. Many projects do not have enough history and defect data for evaluation. Hence, we clean the dataset and obtain 1,385 projects for our experiments. Comparing to the 1,398 projects used in our previous work [202], there are 13 projects removed due to an error in data pre-processing. The error is identified during this extension, and has been fixed. The cleaning process is detailed in the following subsection.

6.3.2 Cleaning the Dataset

(F1) Filtering out projects by programming languages. We use a commercial tool, called *Understand* [164], to compute code metrics. Due to the limitation of the tool, we only investigate projects that are mainly written in C, C++, C#, Java, or Pascal. For each project, we determine its main programming languages by counting the total number of files per file type (i.e., *.c, *.cpp, *.cxx, *.cc, *.cs, *.java, and *.pas).

(F2) Filtering out the projects with a small number of commits. A small number of commits can not provide enough information for computing process metrics and mining defect data. We compute the quantiles of the number of commits of all projects throughout their history. We choose the 25% quantile of the number of commits as the thresholds to filter out projects. In our dataset, we filter out the projects with less than 32 (inclusive) commits throughout their histories.

(F3) Filtering out the projects with lifespan less than one year. Most studies in defect prediction collect defect data from six months' period [208] after the software release, and compute process metrics using the six months' data ahead. However, numerous projects on SourceForge or GoogleCode do not have clear release periods. Therefore, we simply determine the split date for each project by looking 6 months (i.e., 182.5 days) back from its last commit. We collect defect data in the six months' period after the split date, and compute process metrics using the change history in the six months' period before the split date. Thus we filter out the projects with a lifespan less than one year (i.e., 365 days).

(F4) Filtering out the projects with limited defect data. Defect data needs to be mined from enough commit messages. We count the number of fix-inducing and non-fixing commits from a one-year period. We choose the 75% quantile of the number of fix-inducing (respectively non-fixing) commits as the thresholds to filter out the projects with less defect data. For projects hosted on SourceForge, the 75% quantile of the number of fix-inducing and non-fixing commits are: 152 and 1,689, respectively. For projects hosted on GoogleCode, the 75% quantile of the number of fix-inducing and non-fixing commits are: 92 and 985, respectively.

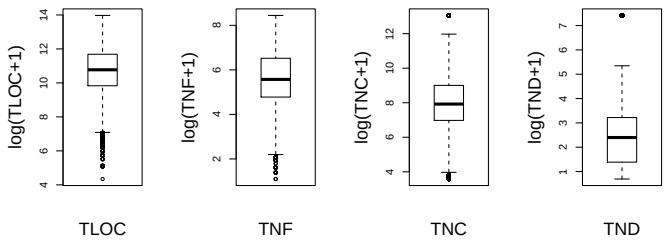


Figure 6.2: Boxplot of four numeric context factors (i.e., TLOC, TNF, TNC, and TND) in our dataset.

(F5) Filtering out the projects without fix-inducing commits. Subject projects in defect prediction studies usually contain defects. For example, the 56 projects used by Peters *et al.* [151] have at least one defect. We consider the projects that have no fix-inducing commits during six months as abnormal projects, therefore we filter out such projects. Moreover, there are 13 projects with few commits during the six-month period of collecting process metrics. We filter out these 13 projects since process metrics are not available for them.

Description of the final experiment dataset. In the cleaned dataset, there are 931 SourceForge projects, and 454 GoogleCode projects. Among them, 713 projects employ CVS as their version control system, 610 projects use Subversion, and 62 projects adopt Mercurial. The number of projects that are mainly written in C, C++, C#, Java, and Pascal are 283, 421, 84, 586, and 11, respectively. There are 810 projects using issue tracking systems, and 575 projects without using any issue tracking system. We show the boxplot of other four context factors in Figure 6.2.

6.3.3 Software Metrics

Software metrics are used as predictors to build a defect prediction model. In this study, we choose 21 code metrics, and 5 process metrics that are often used in defect prediction models. The list of selected metrics is shown in Table 6.2. File and Method level metrics

Table 6.2: List of software metrics. The last column refers to the aggregation scheme ("none" means that aggregation is not performed for file level metrics).

Type	Metric Level	Metric Name	Description	Aggregation
Code Metrics	File	Loc	Lines of Code	none
		Cl	Comment Lines	none
		Nstmt	Number of Statements	none
		Nfunc	Number of Functions	none
		Rcc	Ratio Comments to Code	none
		Mnl	Max Nesting Level	none
	Class	Wmc	Weighted Methods per Class	avg, max, total
		Dit	Depth of Inheritance Tree	avg, max, total
		Rfc	Response For a Class	avg, max, total
		Noc	Number of Immediate Subclasses	avg, max, total
		Cbo	Coupling Between Objects	avg, max, total
		Lcom	Lack of Cohesion in Methods	avg, max, total
		Niv	Number of instance variables	avg, max, total
		Nim	Number of instance methods	avg, max, total
		Nom	Number of Methods	avg, max, total
		Npbm	Number of Public Methods	avg, max, total
		Npm	Number of Protected Methods	avg, max, total
		Nprm	Number of Private Methods	avg, max, total
	Methods	Cc	McCabe Cyclomatic Complexity	avg, max, total
		Fanin	Number of Input Data	avg, max, total
		Fanout	Number of Output Data	avg, max, total
Process Metrics	File	Nrev	Number of revisions	none
		Nfix	Number of revisions a file was involved in bug-fixing	none
		AddedLoc	Lines added	avg, max, total
		DeletedLoc	Lines deleted	avg, max, total
		ModifiedLoc	Lines modified	avg, max, total

are available for all the five studied programming languages. Class level metrics are only available for object-oriented programming languages, and are set to zero in files written in C. As defect prediction is performed at file level in this study, method level and class level metrics are aggregated to file level using three schemes, i.e., average (avg), maximum (max), and summation(total). The code metrics are computed by the *Understand* tool [164]. Process metrics include the number of revisions and bug-fixing revisions (see Section 6.3.4), and lines of added/deleted/modified code. Process metrics are computed by our scripts. For each file, we extract all revisions that are performed during the period for collecting process metrics, and obtain the number of revisions and bug-fixing revisions. The number of added, deleted, and modified lines between each two consecutive revisions

of each file are computed, and then aggregated to file level using the three aforementioned schemes. As mentioned in Section 6.3.2, we look 6 months (i.e., 182.5 days) back from the last commit to obtain the split date. The code metrics are computed using the files from the snapshot of the split date. The process metrics are computed using the change history in the six months' period before the split date.

6.3.4 Defect Data

Defect data are often mined from commit messages, and corrected using defect information stored in an issue tracking system [208]. In our dataset, 42% of subject projects do not use issue tracking systems. For such projects, we mine defect data solely by analyzing the content of commit messages. A similar method for mining defect data is used by Mockus and Votta [126] and in SZZ algorithm [179]. We first remove all words ending with "bug" or "fix" from commit messages, since "bug" and "fix" can be affix of other words (e.g., "debug" and "prefix"). A commit message is tagged as fixing defect, if it matches the following regular expression:

$$(bug|fix|error|issue|crash|problem|fail|defect|patch)$$

Using commit messages to mine defect information may be biased [16, 81, 102]. However, Rahman *et al.* [158] report that increasing the sample size can mitigate the possible bias in defect data. Our dataset contains 1,385 subject projects, and is around 140 to 280 times larger than most studies performed in this field [151]. In addition, the modelling technique (i.e., Naive Bayes) used in this study is shown by Kim *et al.* [102] to have strong noise resistance with up to 20%-35% of false positive and false negative noises in defect data. The defect data is collected in the six months' period after the split date. We show the boxplot of the number of defects and the percentage of defects in our dataset in Figure 6.3.

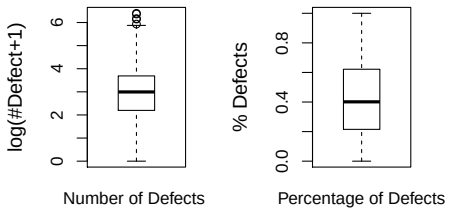


Figure 6.3: Boxplot of the number of defects and the percentage of defects in our dataset.

6.4 Case Study Results

This section first describes the statistics of our project clusters, and then presents the motivation, approach, and findings of the following five research questions.

(RQ1) *Can a context-aware rank transformation provide predictive power comparable to the power of log transformation?*

(RQ2) *What is the performance of the universal defect prediction model?*

(RQ3) *What is the performance of the universal defect prediction model on external projects?*

(RQ4) *Do context factors affect the performance of the universal defect prediction model?*

(RQ5) *What predictors should be included in the universal defect prediction model?*

6.4.1 Project Clusters

In our dataset, there are 1,385 open source projects. The set of projects is stratified into non-overlapped groups along the six context factors: programming language, issue tracking, total lines of code, total number of files, total number of commits, and total number of developers, respectively. In total, we obtain 478 non-empty groups. For each metric, we perform $\binom{478}{2} = \frac{478!}{2!\times 476!} = 114{,}003$ times of Mann-Whitney U tests to compare the difference of the distribution between any pair of groups. To control family-wise errors, we

adjust the threshold p-value using Bonferroni correction to $0.05/114,003 = 4.39e\text{-}07$. Any pair of groups without statistically significant difference in their distribution are merged together. Moreover, the pair of groups without a large difference (measured by Cliff's δ) are also merged together. The maximum number of clusters observed for a metric is 32, which is the number of clusters obtained for the metric total_CBO (i.e., the sum of values of coupling between objects per file).

6.4.2 Research Questions

RQ1: ***Can a context-aware rank transformation provide predictive power comparable to the power of log transformation?***

Motivation. We have proposed a context-aware rank transformation method to eliminate the impact of varied scales of metrics among different projects. The rank transformation converts raw values of all metrics to levels of the same scale. Before fitting the rank transformed metric values to a universal defect prediction model, it is necessary to evaluate the performance of our transformation approach. To achieve this goal, we compare the performance of defect prediction models built using rank transformations to the models built using log transformations. The log transformation uses the logarithm of raw metric values, and has been proved to improve the predictive power in defect prediction approaches [93, 123].

Approach. For each project, we build two within-project defect prediction models using metrics listed in Table 6.2. One uses log transformed metric values, and the other uses rank transformed metric values. We call a model a within-project defect prediction model if both training and testing data are from the same project. To evaluate the performance of predictions, we perform 10-fold cross validation on each project.

To investigate the performance of our rank transformation, we test the following null

hypothesis for each performance measure:

$H0_1$: *there is no difference in the performance of defect prediction models built using log and rank transformations.*

Hypothesis $H0_1$ is two-tailed, since it investigates if rank transformation yields better or worse performance than log transformation. We conduct two-tailed and paired Wilcoxon rank sum test [173] to compare the seven performance measures, using the 95% confidence level (i.e., p-value<0.05). The Wilcoxon rank sum test is a non-parametric statistical test to assess whether two independent distributions have equally large values. Non-parametric statistical methods make no assumptions about the distribution of assessed variables. If there is a statistical significance, we reject the hypothesis and conclude that the performance of the two transformation techniques are different. Moreover, we compare the proportion of the successful predictions. The success of predictions is determined using two criteria: 1) strict criteria (i.e., precision and recall are greater than 0.75), as used by Zimmermann *et al.* [210]; and 2) loose criteria (i.e., precision is greater than 0.5 and recall is greater than 0.7), as applied by He *et al.* [76].

Findings. There are 99 projects that do not contain enough files to perform 10-fold cross validation. Hence, we compare the performance of log and rank transformations on the remaining 1,286 projects. Table 6.3 presents the mean values of the seven performance measures of both log and rank transformations, and the corresponding p-values of Wilcoxon rank sum test. We reject the hypothesis $H0_1$ for most measures (except recall), and conclude that there is significant difference between rank transformation and log transformation in within-project defect prediction in precision, false positive rate, F-measure, g-measure, MCC, and AUC. However, the differences between their average performance measures are negligible (i.e., the absolute value of Cliff's δ is less than 0.147), as shown

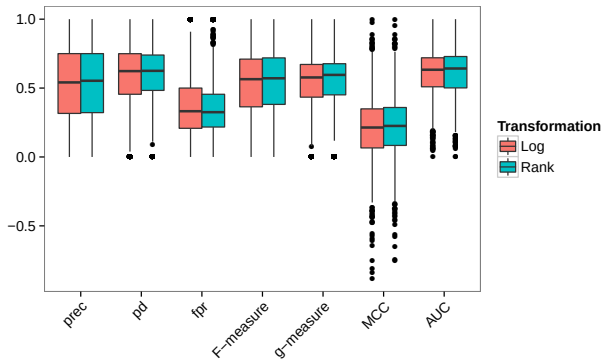


Figure 6.4: Boxplots of performance measures of models built using log and rank transformations.

Table 6.3: The results of Wilcoxon rank sum tests and mean values of the seven performance measures of log transformation and our context-aware rank transformation in the within-project settings. (* denotes statistical significance.)

Measures	**Log transformation**	**Rank transformation**	p**-value**	**Cliff's** δ
prec	0.519	**0.525**	2.42e-04*	-0.091
pd	0.576	**0.580**	0.08	-0.047
fpr	0.369	**0.359**	4.04e-04*	0.113
F-measure	0.527	**0.534**	8.19e-06*	-0.113
g-measure	0.511	**0.521**	1.37e-09*	-0.113
MCC	0.202	**0.214**	7.44e-06*	-0.120
AUC	0.609	**0.615**	8.90e-05*	-0.091

in Table 6.3. To better illustrate the differences, we show the boxplots of performance measures of models built using log and rank transformation in Figure 6.4.

Furthermore, the proportion of successful predictions for both approaches is identical where it is 13% and 27% using the strict and loose criteria for successful prediction, respectively. Therefore, we conclude that rank transformation achieves comparable performance to log transformation. It is reasonable to use the proposed rank transformation method to build universal defect prediction models.

Rank transformation achieves comparable performance to log transformation. The universal model can be built using rank transformed predictors.

RQ2: ***What is the performance of the universal defect prediction model?***

Motivation. The findings of **RQ1** support the feasibility of our proposed rank transformation method for building defect prediction models. However, building an effective universal model is still a challenge. For instance, Menzies *et al.* [124] report the poor performance of a model built on an entire set of diverse projects. This research question aims to investigate the best achievable predictive power of the universal model. First, we evaluate if the predictive power of the universal model can be improved by adding context factors as predictors, together with code metrics and process metrics that are commonly used in prior studies for defect prediction. Second, we study if the universal model can achieve comparable performance as within-project defect prediction models. Accordingly, we split **RQ2** to two sub questions:

RQ2.1: *Can context factors improve the predictive power?*

RQ2.2: *Is the performance of the universal model defect prediction comparable to within-project models?*

Approach. We describe our approaches to address each sub question.

To address RQ2.1, we build the universal model using five combinations of metrics: 1) code metrics; 2) code and process metrics; 3) code metrics and context factors; 4) process metrics and context factors; and 5) code, process metrics, and context factors. All metrics are transformed using the context-aware rank transformation. To evaluate the performance of predictions, we perform 10-fold cross validation on the entire set of projects. To compare the performance of the universal model among different combination of metrics, we test the following null hypothesis for each pair of metric combinations:

$H0_{21}$: *there is no difference in the performance of the universal defect prediction models built using two metric combinations.*

To address RQ2.2, we obtain the performance of within-project and universal models for each project, respectively. The predictive power of within-project models is obtained using 10-fold cross-validation (same as **RQ1**). The performance of universal models on a particular project is evaluated by applying a universal model built upon the remaining set of projects on the project. We compute and compare the proportion of acceptable predictions of both the universal model and the within-project models. To compare the performance of within-project and universal models, we test the following null hypothesis for each performance measure:

$H0_{22}$: *there is no difference in the performance of within-project and universal defect prediction models.*

Hypotheses $H0_{21}$ and $H0_{22}$ are two-tailed, since they investigates if one prediction model yields better or worse performance than the other prediction model. We apply two-tailed and paired Wilcoxon rank sum test (95% confidence level) to examine the hypothesis. If there is significance, we reject the null hypothesis and compute Cliff's δ [36] to measure the difference.

Findings. We report our findings for our two sub questions.

(RQ2.1) Table 6.4 (a) provides the performance measures of the universal model using each combination of metrics. In general, adding context factors increases five performance measures (i.e., precision, F-measure, g-measure, MCC, and AUC value). AUC value is the only measure that is independent of the cut-off value. As the space is limited, Table 6.4 (b) only presents Cliff's δ and p-value of Wilcoxon rank sum tests on comparisons of AUC values. We observe that adding context factors significantly improves performance over

Table 6.4: The results of comparing the universal models built using code metrics (CM), code + process metrics (CPM), code metrics and context (CM-C), process metrics and context (PM-C), and code + process metrics + contexts (CPM-C), respectively.

(a) The seven performance measures (mean ± std.dev).

Measures	CM	CPM	CM-C	PM-C	CPM-C
prec	0.431 ± 0.067	0.437 ± 0.069	0.445 ± 0.061	0.438 ± 0.063	**0.455** ± 0.065
pd	0.551 ± 0.020	0.548 ± 0.015	**0.602** ± 0.048	0.557 ± 0.038	0.591 ± 0.040
fpr	0.404 ± 0.025	**0.392** ± 0.020	0.419 ± 0.070	0.401 ± 0.073	0.396 ± 0.065
F-measure	0.480 ± 0.046	0.484 ± 0.048	0.508 ± 0.043	0.488 ± 0.048	**0.510** ± 0.045
g-measure	0.572 ± 0.016	0.577 ± 0.013	0.587 ± 0.027	0.574 ± 0.029	**0.594** ± 0.022
MCC	0.141 ± 0.032	0.150 ± 0.029	0.175 ± 0.047	0.150 ± 0.060	**0.186** ± 0.045
AUC	0.600 ± 0.020	0.607 ± 0.019	0.636 ± 0.041	0.628 ± 0.046	**0.641** ± 0.038

(b) The Cliff's δ and p-value of Wilcoxon rank sum tests on the comparison of AUC values. (* denotes statistical significance.)

Metric Sets	CPM		CM-C		PM-C		CPM-C	
CM	-0.572	0.07	-0.734	$9.15e$-03*	-0.491	0.16	-0.804	$5.89e$-03*
CPM	-	-	-0.606	0.04*	-0.392	0.23	-0.707	$8.00e$-03*
CM-C	-	-	-	-	0.244	0.49	-0.631	0.02*
PM-C	-	-	-	-	-	-	-0.406	0.13

using only code metrics. The Cliff's δ is -0.734, indicating a large improvement (i.e., the absolute value of Cliff's δ is greater than 0.474). In addition, adding context factors yields significant improvement (Cliff's δ is -0.707) in the performance over using just code and process metrics. Hence, we conclude that the context factors are good predictors for building a universal defect prediction model.

(RQ2.2) The boxplots of performance measures of within-project and universal models are shown in Figure 6.5. Table 6.5 presents the Wilcoxon rank sum test results of performance measures between within-project model and universal models built using rank transformations. We reject the null hypothesis $H0_{22}$ for all measures except precision and g-measure. The results show that the universal model and the within-project model have similar precision, recall, false positive rate, g-measure, and MCC. The differences in these five performance measures are neither significant (i.e., p-value is greater than 0.05) nor

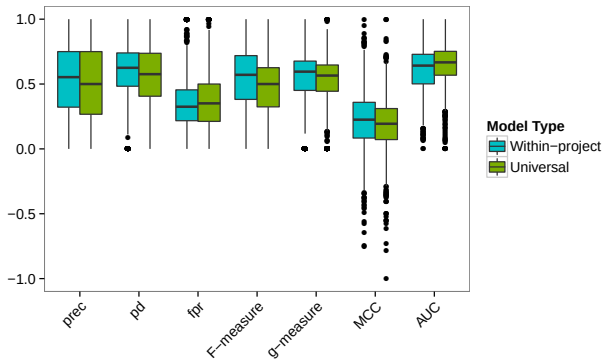


Figure 6.5: Boxplots of performance measures of within-project and universal models.

Table 6.5: The results for Wilcoxon rank sum tests and mean values of the seven performance measures of within-project models and universal models. (* denotes statistical significance.)

Measures	Within-project models	Universal models	p-value	Cliff's δ
prec	**0.525**	0.518	0.47	0.047
pd	**0.580**	0.570	6.60e-03*	0.031
fpr	**0.359**	0.365	0.04*	-0.016
F-measure	**0.534**	0.474	< 2.2e-16*	0.291
g-measure	0.521	**0.534**	0.19	-0.054
MCC	**0.214**	0.184	1.32e-03*	0.113
AUC	0.615	**0.655**	< 2.2e-16*	-0.219

observable (i.e., Cliff's δ is less than 0.147). There is observable small (i.e., Cliff's δ is greater than 0.147, but less than 0.330) difference in F-measure and AUC value. The universal model has lower F-measure but higher AUC value than within-project model. F-measure is computed based on the confusion matrix (see Section 6.2.7) that is obtained using a cut-off value. On the other hand, calculating AUC does not require a cut-off value. The possible cause of lower F-measure but higher AUC value of the universal model is that different cut-off values may be needed for different projects when applying the universal model. Understanding how to choose the best cut-off values might help improve the F-measure of the universal model.

Moreover, the universal models yield similar percentage (i.e., 3.6%) of successful predictions (see **RQ1**) as Zimmermann *et al.* [210] who report a 3.4% success rate. If using

loose criteria, the universal model achieves 13% of successful predictions, much higher than He *et al.* [76] who report 0.32% of successful predictions. The universal model achieves up to 48% (i.e., 13% against 27%) of the successful predictions by within-project model. We conclude that our approach for building a universal model is promising.

The universal model yields similar predictive performance as within-project models.

RQ3: ***What is the performance of the universal defect prediction model on external projects?***

Motivation. In examining **RQ2**, we successfully build a universal model for a large set of projects. The universal model slightly outperforms within-project models in terms of recall and AUC. Although our experiments involve a large number of projects from various contexts, the projects are selected from only two hosts: SourceForge and GoogleCode. It is still unclear if the universal model is generalizable, i.e., whether it works well for external projects that are not managed on the aforementioned two hosts. This research question aims to investigate the capability of applying the universal model to predict defects for external projects that are not hosted on SourceForge or GoogleCode.

Approach. To address the question, we choose to use the publicly available dataset[4] that was collected by D'Ambros *et al.* [44]. The dataset contains four Eclipse projects (i.e., Eclipse JDT Core, Eclipse PDE UI, Equinox Framework, and Mylyn), and one Apache project (i.e., Lucene). We present the descriptive statistics of the five external projects in Table 6.6.

[4]http://bug.inf.usi.ch/download.php

Table 6.6: The descriptive statistics of the five external projects used in this study.

Project	TLOC	TNC	Number of classes	Number of defects	Percentage of defects
Eclipse	224, 055	45, 482	997	206	20.7%
Equinox	39, 534	3, 691	324	129	39.8%
Lucene	73, 184	4, 329	691	64	9.3%
Mylyn	156, 102	20, 451	1,862	245	13.2%
PDE	146, 952	20, 228	1,497	209	14.0%

Table 6.7: The performance measures for within-project model and the universal model.

Measures	Eclipse	Equinox	Lucene	Mylyn	PDE	Type
prec	**0.323**	**0.621**	**0.197**	**0.252**	**0.220**	within-project
	0.222	0.607	0.128	0.168	0.155	universal
pd	0.782	0.775	0.531	0.473	0.732	within-project
	0.937	**0.899**	**0.922**	**0.767**	**0.914**	universal
fpr	**0.427**	**0.313**	**0.222**	**0.213**	**0.422**	within-project
	0.853	0.385	0.643	0.578	0.806	universal
F-measure	**0.457**	0.690	**0.287**	**0.329**	**0.338**	within-project
	0.359	**0.725**	0.224	0.275	0.266	universal
g-measure	**0.661**	0.728	**0.631**	**0.591**	**0.646**	within-project
	0.254	**0.731**	0.515	0.545	0.320	universal
MCC	**0.287**	0.453	**0.207**	**0.204**	**0.216**	within-project
	0.101	**0.512**	0.172	0.131	0.098	universal
AUC	0.764	0.804	0.727	**0.677**	0.700	within-project
	0.766	**0.821**	**0.750**	0.664	**0.704**	universal

We calculate the six context factors of the five aforementioned projects, and apply related ranking functions to convert their raw metric values to one of the ten levels. We predict defects on each project using the universal model which is trained on 1,385 SourceForge and GoogleCode projects. The seven performance measures of within-project models are obtained via 10-folds cross-validation for each project.

Findings. Table 6.7 presents the average values of the seven performance measures of the universal model and within-project models, respectively. Overall, there are clear differences in the performance (all measures except AUC value) of the universal model and within-project models. In particular, the universal model yields lower precision, larger false positive rate, but higher recall than within-project models. However, these performance

measures depend on the cut-off value that is used to determine if an entity is defective or not (see Section 6.2.7). Such performance measures can be significantly changed by altering the cut-off value. The AUC value is independent of the cut-off value and is preferred for cross-project defect prediction [157]. As the universal model achieves similar AUC values to within-project models on the five subject projects, we conclude that the universal model is as effective as within-project defect prediction models for the five subject projects. However, various cut-off values may be needed to yield high precision or low false positive rate, when applying the universal model on different projects. We present further discussions on dealing with high false positive rate as follows.

Discussions on false positive rate. In practice, high false positive is unacceptable, e.g., false positive rate is greater than 0.64 [188]. As shown in Table 6.7, the universal model experiences high false positive rates in three projects (i.e., Eclipse, Lucene, and PDE). In **RQ2**, we observe that the universal model exhibits similar false positive rate to within-project defect prediction models in general. Hence, we conjecture that the high false positive rate in external projects is due to the different percentages of defects in the training set (e.g., the median percentage of defects is 40%) and in the five external projects (e.g., the median percentage of defects is 14%). Nevertheless, it is of significant interest to seek insights on how to determine cut-off values to reduce false positive rate.

1) Effort-aware estimation of the cut-off value. It is time consuming to examine all entities that are predicted as defective. If a development team has limited resources or a tight schedule, it is more realistic to inspect only the top X% of entities that are predicted as defective. To this end, we choose the minimum predicted probability among the top X% of entities as the cut-off value for each project. We recalculate the performance measures, and present the detailed results in Table 6.8. We observe that the median false positive is reduced to

Table 6.8: The performance measures for the universal model on external projects with cut-off values determined by the minimum predicted probability by the universal model among the top 10%, 20%, and 30% of defective entities.

Top %	Measure	Eclipse	Equinox	Lucene	Mylyn	PDE	Median	Average
10%	Cut-off	0.968	0.950	0.948	0.942	0.964	0.950	0.954
	prec	0.722	0.844	0.309	0.404	0.331	0.404	0.522
	pd	0.316	0.209	0.328	0.302	0.196	0.302	0.270
	fpr	0.032	0.026	0.075	0.067	0.064	0.064	0.053
	F-measure	0.439	0.335	0.318	0.346	0.246	0.335	0.337
	g-measure	0.476	0.345	0.484	0.456	0.324	0.456	0.417
	MCC	0.401	0.301	0.246	0.266	0.166	0.266	0.276
	AUC	0.766	0.821	0.750	0.664	0.704	0.750	0.741
20%	Cut-off	0.958	0.912	0.905	0.903	0.946	0.912	0.925
	prec	0.538	0.746	0.234	0.276	0.312	0.312	0.421
	pd	0.510	0.364	0.500	0.412	0.435	0.435	0.444
	fpr	0.114	0.082	0.167	0.164	0.156	0.156	0.137
	F-measure	0.524	0.490	0.318	0.331	0.363	0.363	0.405
	g-measure	0.647	0.522	0.625	0.552	0.574	0.574	0.584
	MCC	0.404	0.349	0.242	0.211	0.244	0.244	0.290
	AUC	0.766	0.821	0.750	0.664	0.704	0.750	0.741
30%	Cut-off	0.944	0.879	0.825	0.844	0.924	0.879	0.883
	prec	0.444	0.680	0.198	0.225	0.274	0.274	0.364
	pd	0.641	0.512	0.641	0.510	0.584	0.584	0.578
	fpr	0.209	0.159	0.265	0.267	0.252	0.252	0.230
	F-measure	0.525	0.584	0.303	0.312	0.373	0.373	0.419
	g-measure	0.708	0.636	0.685	0.602	0.656	0.656	0.657
	MCC	0.383	0.377	0.238	0.180	0.252	0.252	0.286
	AUC	0.766	0.821	0.750	0.664	0.704	0.750	0.741

0.053, if considering only the top 10% of entities as defective. When considering only the top 20% and 30% of entities as defective, the median false positive rate becomes 0.137 and 0.230, respectively. As cut-off values change, the other performance measures are also updated. For instance, the median recall becomes 0.270, 0.444, and 0.578, respectively, if considering only the top 10%, 20%, and 30% of entities as defective. The AUC values remain the same when altering cut-off values. Therefore, we conclude that high false positive rate can be tamed by considering only the top 10%, 20%, or 30% of entities that are predicted as defective by the universal model.

2) Other insights on selecting the cut-off value. Inspired by transfer learning [145], we suppose that appropriate cut-off values may be inferred from target projects. Intuitively, we

Table 6.9: The performance measures for the universal model on external projects with cut-off values obtained by using ratio of defects or minimizing the error rates on the entire set of entities.

	Measure	Eclipse	Equinox	Lucene	Mylyn	PDE	Median	Average
Percentage of defects	Cut-off	0.793	0.602	0.907	0.868	0.860	0.860	0.806
	prec	0.265	0.625	0.235	0.241	0.219	0.241	0.317
	pd	0.879	0.814	0.500	0.482	0.727	0.727	0.680
	fpr	0.635	0.323	0.166	0.230	0.420	0.323	0.355
	F-measure	0.407	0.707	0.320	0.321	0.337	0.337	0.418
	g-measure	0.516	0.739	0.625	0.593	0.645	0.625	0.624
	MCC	0.213	0.481	0.244	0.193	0.214	0.214	0.269
	AUC	0.766	0.821	0.750	0.664	0.704	0.750	0.741
Minimized error rates	Cut-off	0.936	0.519	0.851	0.893	0.923	0.893	0.824
	prec	0.415	0.616	0.210	0.270	0.274	0.274	0.357
	pd	0.699	0.884	0.609	0.449	0.593	0.609	0.647
	fpr	0.257	0.364	0.234	0.184	0.255	0.255	0.259
	F-measure	0.521	0.726	0.312	0.337	0.375	0.375	0.454
	g-measure	0.721	0.740	0.679	0.579	0.661	0.679	0.676
	MCC	0.376	0.514	0.245	0.217	0.256	0.256	0.322
	AUC	0.766	0.821	0.750	0.664	0.704	0.750	0.741

conjecture that the appropriate cut-off values might depend on the percentage of defective entities in target projects. Alternatively, minimizing the total error rates (i.e., FP+FN) may help reduce the false positive rate while maintaining the recall. Therefore, we examine if it is feasible to reduce false positive rate by inferring the cut-off value based on: 1) the percentage of defects in the target project; and 2) the minimized total error rates (i.e., FP+FN). Table 6.9 shows the detailed results of the two methods. In particular, using the percentage of defects in the target project results in median false positive rate as 0.323 along with 0.727 of recall. Minimizing error rates yields median false positive rate as 0.255 along with 0.609 of recall. In both cases, the performances are similar as the work by Turhan *et al.* [188] that successfully reduces high false positive rate to 0.33 along with 0.68 of recall by filtering the training set.

However, it is impractical to obtain the defectiveness of all entities in the target project. Otherwise, a within-project defect prediction model can be constructed. Therefore, it is necessary to investigate how many entities are required to estimate cut-off values. As using

less entities may yield unstable cut-off values, we average the cut-off values determined by the aforementioned two methods as the final cut-off values. We perform an exploratory experiment by randomly sampling 10, 20, and 30 entities from each project. We further repeat the experiment 100 times for each project, and report the average performance measures in Table 6.10. In general, we can observe that increasing the number of randomly sampled entities improves the performance of the universal model in terms of five measures (i.e., precision, false positive rate, F-measure, g-measure, and MCC). When randomly sampling 10 entities, the universal model achieves a median false positive of 0.344 and a median recall of 0.723. Apart from the work by Turhan *et al.* [188] that customizes the prediction model (i.e., filtering the training set) based on the target project, the universal model is not altered for a particular target project. Software organizations do not need to provide their data for customizing prediction models, but tune cut-off values for their goals. The universal model can help address the concern on sharing data or models across companies [150]. Although inspection of defectiveness in a target project is needed, the proportion of required entities is relatively low, such as 1% (10/997) for Eclipse, 3% (10/324) of Equinox, 1% (10/691) for Lucene, 1% (10/1862) for Mylyn, and 1% (10/1497) for PDE.

As a summary, the results show that our universal model can provide comparable performances to within-project defect prediction models for the five subject projects. Considering the five projects might conduct different development strategies than SourceForge or GoogleCode projects, there is a high chance to apply the universal model on more external projects with acceptable predictive power.

The universal model can predict defects for external projects with acceptable false positive rate.

Table 6.10: The performance measures for the universal model on external projects with cut-off values learnt from both the ratio of defects and the minimized error rates, using a subset of randomly sampled entities.

Number of entities	Measure	Eclipse	Equinox	Lucene	Mylyn	PDE	Median	Average
10	prec	0.298	0.618	0.195	0.220	0.221	0.221	0.310
	pd	0.832	0.815	0.644	0.575	0.723	0.723	0.718
	fpr	0.537	0.344	0.298	0.342	0.445	0.344	0.393
	F-measure	0.434	0.696	0.291	0.307	0.332	0.332	0.412
	g-measure	0.573	0.712	0.647	0.585	0.596	0.596	0.623
	MCC	0.249	0.472	0.225	0.174	0.202	0.225	0.264
	AUC	0.766	0.821	0.750	0.664	0.704	0.750	0.741
20	prec	0.308	0.621	0.212	0.235	0.231	0.235	0.321
	pd	0.816	0.824	0.593	0.545	0.687	0.687	0.693
	fpr	0.498	0.338	0.259	0.310	0.402	0.338	0.361
	F-measure	0.442	0.705	0.297	0.310	0.336	0.336	0.418
	g-measure	0.608	0.727	0.629	0.578	0.607	0.608	0.630
	MCC	0.265	0.483	0.228	0.182	0.210	0.228	0.274
	AUC	0.766	0.821	0.750	0.664	0.704	0.750	0.741
30	prec	0.310	0.620	0.226	0.238	0.231	0.238	0.325
	pd	0.814	0.836	0.559	0.529	0.690	0.690	0.686
	fpr	0.488	0.344	0.231	0.290	0.399	0.344	0.350
	F-measure	0.445	0.709	0.299	0.313	0.336	0.336	0.420
	g-measure	0.618	0.730	0.615	0.583	0.614	0.615	0.632
	MCC	0.270	0.488	0.232	0.185	0.212	0.232	0.277
	AUC	0.766	0.821	0.750	0.664	0.704	0.750	0.741

RQ4: ***Do context factors affect the performance of the universal defect prediction model?***

Motivation. In **RQ3**, we verified the capability of the universal model to predict defects for five external projects. The universal model can interpret general relationships between metrics and defect proneness, regardless of the place where projects are hosted. However, the five external projects have limited diversity. For example, they are all written in Java. The generalizability of the universal model is not deeply examined in **RQ3**. This threatens the applicability of the universal model to projects from different contexts [130]. Hence, it is essential to investigate whether the performance of the universal model varies across projects with different context factors.

Approach. To address this question, we compare the performance of the universal model across projects with different context factors (see Section 6.2.2). To train a universal model

Table 6.11: Groups of projects split along different context factors.

Context factor	Groups
Programming language (PL)	G_c, G_{c++}, $G_{c\#}$, G_{java}, and G_{pascal}
Issue tracking (IT)	G_{useIT} and G_{noIT}
Total lines of code (TLOC)	$G_{leastTLOC}$, $G_{lessTLOC}$, $G_{moreTLOC}$, and $G_{mostTLOC}$
Total number of files (TNF)	$G_{leastTNF}$, $G_{lessTNF}$, $G_{moreTNF}$, and $G_{mostTNF}$
Total number of commits (TNC)	$G_{leastTNC}$, $G_{lessTNC}$, $G_{moreTNC}$, and $G_{mostTNC}$
Total number of developers (TND)	$G_{leastTND}$, $G_{lessTND}$, $G_{moreTND}$, and $G_{mostTND}$

with the largest number of projects, we apply leave-one-out cross validation (e.g., [207]). For a particular project, we use all other projects to build a universal model and apply the universal model on the project. We repeat this step for every project and obtain the predictive power of the universal model on each project. As the findings of **RQ2** and **RQ3** suggest that different projects may prefer different cut-off values, we choose to only compare AUC values to avoid the impact of cut-off values on our observations.

We divide the entire set of projects along each context factor, respectively. There are three types of context factors: categorical factor (i.e., programming language), boolean factor (i.e., issue tracking), and numerical factors (e.g., the total lines of code). For categorical factor, we obtain a group per category. We get two groups for boolean factor. For numerical factors, we compute quantiles of the numbers and then derive four groups. All groups are listed in Table 6.11. The details on these groups are described in Section 6.2.3. Please note that these groups are created solely based on context factors, other than the distribution of software metrics.

For each pair of groups on a particular context factor, we test the following null hypothesis:

$H0_4$: *there is no difference in the performance of the universal model between projects of the group-pair.*

Hypothesis $H0_4$ is two-tailed, since it investigates if the universal model yields better

or worse performance in one project group than the other project group of a group-pair. As the size of two groups may be different, we apply two-tailed and unpaired Wilcoxon rank sum test (95% confidence level) to examine the hypothesis.

Findings. For each context factor, we present our findings on the generalizability of the universal model.

1) **Programming language (PL).** The average AUC values for groups G_c, G_{c++}, $G_{c\#}$, G_{java}, and G_{pascal} are 0.64, 0.65, 0.61, 0.64, and 0.64, respectively. There are 5 groups of projects divided by programming languages, and the number of pairwise comparisons is 10. Hence, the threshold p-value is $5.00e$-03 after Bonferroni correction. The p-values of Wilcoxon rank sum test are always greater than $5.00e$-03. We do not find enough evidence to support that there are significant difference across projects with different programming languages. In other word, the universal model yields similar performance for projects written in any of the five studied programming languages. There exist common relationships between software metrics and defect proneness, no matter whether projects are developed using C, C++, C#, Java, or Pascal. Future work is needed to understand such common relationships more deeply.

2) **Issue tracking (IT).** The average AUC values for groups G_{useIT} and G_{noIT} are 0.64 and 0.65, respectively. There is only one pair of groups of projects divided by the usage of issue tracking systems, and therefore the threshold p-value is 0.05. The p-value of Wilcoxon rank sum test is 0.14, indicating that there is no significant difference between projects with or without usage of issue tracking systems.

3) **Total lines of code (TLOC).** The average AUC values for groups $G_{leastTLOC}$, $G_{lessTLOC}$, $G_{moreTLOC}$, and $G_{mostTLOC}$ are 0.63, 0.65, 0.65, and 0.64, respectively. There are 4

groups of projects divided by the total lines of code, and the number of pairwise comparisons is 6. Hence, the threshold p-value is $8.33e$-03 after Bonferroni correction. The p-values of Wilcoxon rank sum test are always greater than $8.33e$-03. The universal model can reveal general relationships between software metrics and defect proneness for small, medium, or large projects.

4) **Total number of files (TNF).** The average AUC values for groups $G_{leastTNF}$, $G_{lessTNF}$, $G_{moreTNF}$, and $G_{mostTNF}$ are 0.62, 0.65, 0.64, and 0.64, respectively. Similarly, the threshold p-value is $8.33e$-03 after Bonferroni correction. The p-values of Wilcoxon rank sum test are always greater than $8.33e$-03. We conclude that no matter how many number of files a project has, the universal model can predict defect proneness without significant difference in its performance.

5) **Total number of commits (TNC).** The average AUC values for groups $G_{leastTNC}$, $G_{lessTNC}$, $G_{moreTNC}$, and $G_{mostTNC}$ are 0.64, 0.65, 0.65, and 0.63, respectively. There are 4 groups and 6 pair-wise comparisons. We correct the threshold p-value to $8.33e$-03. The p-values of Wilcoxon rank sum test are always greater than $8.33e$-03. There is no significant difference in the performance of the universal model across projects with different total number of commits.

6) **Total number of developers (TND).** The average AUC values for groups $G_{leastTND}$, $G_{lessTND}$, $G_{moreTND}$, and $G_{mostTND}$ are 0.63, 0.65, 0.64, and 0.64, respectively. There are 4 groups and 6 pair-wise comparisons. We correct the threshold p-value to $8.33e$-03. The p-values of Wilcoxon rank sum test are always greater than $8.33e$-03. The performance of the universal model does not change significantly across projects with different number of developers.

As a summary, we can not find enough evidence to support the hypothesis that the universal model performs significantly different for projects with different context factors. Hence, we conclude that the universal model is applicable to projects with different context factors.

The universal model yields similar performance when it is applied to projects with different context factors, and therefore it is context-insensitive.

RQ5: ***What predictors should be included in the universal defect prediction model?***

Motivation. The purpose of **RQ3** and **RQ4** is to show that our universal model is applicable to external projects, and is context-insensitive, respectively. Therefore in earlier experiments, we use all metrics together for building Naive Bayes models. However, many of these metrics may be strongly correlated. To build an interpretable model, we need to select a subset of these metrics that are not strongly correlated. In within-project settings, the importance of various metrics has been examined in depth (e.g., [176]). For the universal model, we aim to find an uncorrelated interpretable set of predictors that are associated with the chances that a file will have a fix in the future. We do this to understand the general relationship between predictors and defect proneness for the entire set of projects.

Approach. To make the model more interpretable, we chose to use logistic regression to build the universal model. Our choice was motivated by the ease with which logistic regression coefficients can be interpreted [211]. For instance, the sign of a coefficient presents the direction of the impact, i.e., positive or negative. The magnitude indicates the strength of the impact, i.e., how much the probability of defect proneness is affected by a one-unit change in the corresponding predictor. Further details on the convention from

coefficients to exact probabilities can be found in the book by Hosmer *et al.* [84].

Because predictors may be highly correlated, we first need to select an uncorrelated subset of predictors. We use the following rules to select predictors:

(R1) Select well-known simplest predictors that are uncorrelated. We chose lines of code (Loc) as code metrics and number of revisions (NREV) as process metrics that have been often associated with future fixes.

(R2) We analyzed the correlation among context factors, and found that context factors are strongly associated. Hence, we chose the total number of files (TNF) as a context measure of project size and the total number of developers (TND) as a context measure of project activity. Using the first, second, and third quartiles, we converted the two context measures to four levels, respectively. We treat them as categorical variables (same as programming language) in the model because the odds of future fixes may not increase by the same amount as we go from one level (defined by quartiles) to the next.

(R3) We performed hierarchical clustering for all predictors using distance defined as $1 - \|cor(p_1, p_2)\|^2$, where p_1 and p_2 are two predictors. We used R function *hclust*[5] to get clusters of predictors as shown in Figure 6.6. We then applied R function *cutree*[6] to get eight distinct clusters of predictors.

(R4) For each cluster not containing the aforementioned predictors, we chose the first predictor that used the simplest aggregation (avg).

[5]http://stat.ethz.ch/R-manual/R-patched/library/stats/html/hclust.html
[6]http://stat.ethz.ch/R-manual/R-patched/library/stats/html/cutree.html

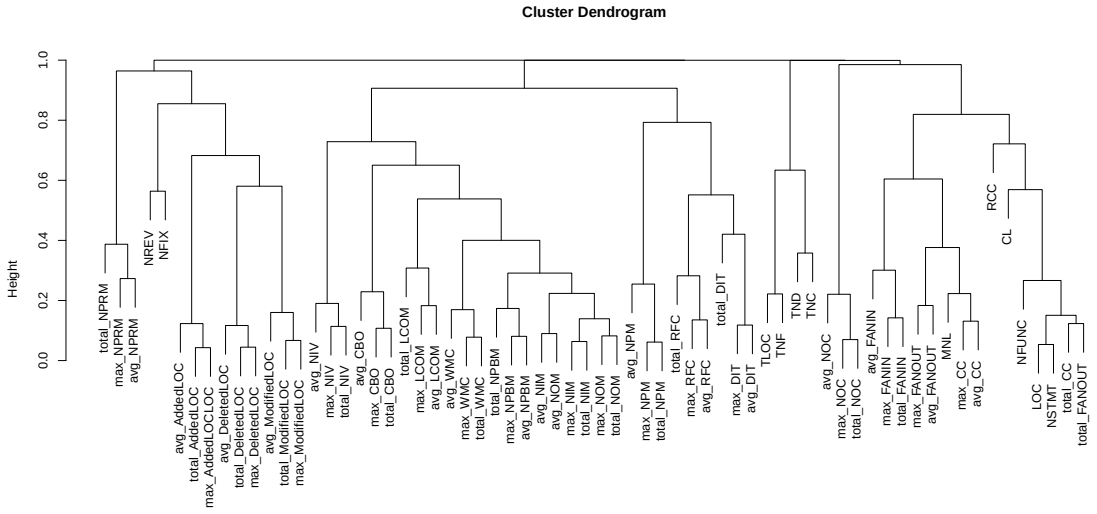


Figure 6.6: Cluster dendrogram of predictors.

Table 6.12: The Pearson correlation among selected code and process metrics. (* denotes statistical significance.)

	avgNoc	avgNiv	avgNpm	avgNprm	avgFanin	Nrev	avgAddedLoc
Loc	-0.08*	0.18*	0.03*	0.15*	0.49*	0.32*	0.27*
avgNoc	-	0.02*	0.00	0.01*	-0.02*	0.03*	0.02*
avgNiv	-	-	0.20*	0.27*	0.07*	0.09*	0.04*
avgNpm	-	-	-	0.05*	-0.02*	0.04*	-0.05*
avgNprm	-	-	-	-	0.05*	0.08*	0.06*
avgFanin	-	-	-	-	-	0.18*	0.12*
Nrev	-	-	-	-	-	-	0.28*

We then used these selected predictors to build the universal model. The *glm*[7] method in R was used to build the logistic regression model. We then inspect the coefficients for each metric to interpret the universal model.

Findings. There are eight code and process metrics selected, i.e., lines of code (Loc), average number of immediate subclasses (avgNoc), average number of instance variables (avgNiv), average number of protected methods (avgNpm), average number of private methods (avgNprm), average number of input data (avgFanin), the number of revisions (Nrev), and average added lines of code (avgAddedloc). The correlation matrix of the selected eight predictors in Table 6.12 does not show any correlations above 0.5.

[7]http://stat.ethz.ch/R-manual/R-patched/library/stats/html/glm.html

Table 6.13: The coefficients of each predictor in the logistic regression model. The numerical intercept is -2.68. For context factors PL, TNF and TND, "C", "$leastTNF$" and "$leastTND$" are folded into the intercept term, respectively. (* denotes statistical significance.)

Type		Metric Name	Coefficients	p-value	Deviance explained	p-value of χ^2-test
		(Intercept)	-2.68	< 2.2e-16*		
Context Factors	TNF	$lessTNF$	-0.24	< 2.2e-16*	4969.5	< 2.2e-16*
		$moreTNF$	-0.44	< 2.2e-16*		
		$mostTNF$	-1.33	< 2.2e-16*		
	TND	$lessTND$	0.05	9.03e-03*	508.9	< 2.2e-16*
		$moreTND$	0.15	1.60e-15*		
		$mostTND$	0.36	< 2.2e-16*		
	PL	C++	-0.28	< 2.2e-16*	549.0	< 2.2e-16*
		C#	0.07	0.01*		
		Java	-0.29	< 2.2e-16*		
		Pascal	-0.91	< 2.2e-16*		
Code Metrics		Loc	0.10	< 2.2e-16*	4555.8	< 2e-16*
		avgNoc	0.06	1.23e-03*	11.4	7.53e-04*
		avgNiv	-0.04	< 2.2e-16*	65.7	5.15e-16*
		avgNpm	0.10	< 2.2e-16*	89.5	< 2.2e-16*
		avgNprm	-0.01	0.29	7.7	5.40e-03*
		avgFanin	0.04	< 2.2e-16*	230.7	< 2.2e-16*
Process Metrics		Nrev	0.10	< 2.2e-16*	1493.7	< 2.2e-16*
		avgAddedLoc	0.01	1.35e-04*	12.0	5.27e-04*

Table 6.13 presents the coefficient for each predictor and the amount of deviance that each predictor explains. The final model explains 7% of deviance of the probability of fixes for the entire set of projects. The null deviance is 177,497 on 136160 degrees of freedom, while residual deviance is 165003 on 136142 degrees of freedom. It is important to note that each predictor should be considered as a representative of all tightly correlated predictors within the cluster. In particular, avgNiv represents a very large number of metrics, including Cbo, Lcom, Wmc, Npbm, Nim, and Nom (see Table 6.2). Also, avgNprm is not significantly different from zero, suggesting that all predictors in the small cluster are not helping model defect proneness. Furthermore, coefficients for avgAddedLoc and avgNoc do not explain as much variance as the remaining predictors and have coefficient values that are barely significantly different from zero. All of these three predictors should be removed

from the final model used for prediction in practice.

In models where each predictor is measured in different units, it is difficult to compare coefficient magnitudes among predictors. In our study, coefficient magnitudes of code and process metrics can be compared, as they have exactly the same units after the rank transformation. In the resulting model, The most important code metric is lines of code (Loc), followed by average number of input data (avgFanin) and average number of private methods (avgNpm). The three code metrics can explain 4876 of deviance for the probability of defect proneness for the entire set of projects. The most important process metrics is the number of revisions (Nrev), which can explain 1493.7 of deviance.

Among context factors, the most important predictor is the total number of files, followed by the programming languages and the total number of developers. The three context factors explain 6027.4 of deviance in total. The R tool treats the alphabetically earliest category of each categorical factor as the reference level, and folds it into the intercept term. The intercept represents the base probability of defect proneness when all categorical factors are at reference levels. In our case, "C" programming language, "leastTNF", and "leastTND" are the reference levels, therefore not shown in Table 6.13. There are differences among languages with "C" code having more fixes than "Java", "C++", and "Pascal" code. Projects with more developers involved (relatively to other projects in the cluster) are also more likely to contain a fix. But projects with more files (relative to other projects in the cluster) are less likely to contain a fix. Future research is needed to fully understand the mechanisms and causes that affect both the predictor values and the chances of future fixes.

The final universal model can be obtained by the following standard equation of logistic

regression models:

$$P = \frac{1}{1 + e^{-(\beta_0 + \beta_1 * m_1 + ... + \beta_k * m_k)}} \tag{6.3}$$

where P is the probability of a file is defective, k is the total number of predictors (in our case k=11, including three context factors and eight metrics), β_0 is the intercept, and β_i is the coefficient of metric m_i ($i = 1, \ldots, k$). For instance, if m_1 is metric Loc, then β_1 is 0.10. This model can be implemented in an integrated development environment (IDE) for instant evaluation of defect proneness, and be used to compare defect proneness across projects.

Our universal model explains 7% of variance of the probability of defect proneness for the entire set of projects. The most important code and process metrics are lines of code (LOC) and the number of revisions (NREV), respectively. The most influential context factor is the total number of files, followed by programming languages and the total number of developers.

6.5 Threats to Validity

We now discuss the threats to validity of our study following common guidelines [199].

Threats to conclusion validity concern the relation between the treatment and the outcome. One conclusion validity threat comes from data cleaning methods. For instance, we remove the projects with negligible fix-inducing or non-fixing commits (both using 75% quantile as the threshold). We plan to investigate the impact of different thresholds in future study. Another threat is due to the extraction of defect data. We mine defect data solely based on commit messages, since 42% of our subject projects do not use issue tracking systems. To deal with this threat, we use a large set of subject projects [158] and apply Naive Bayes as the modelling technique that have strong noise resistance with defect data [158].

Threats to internal validity concern our selection of subject systems and analysis methods. SourceForge and GoogleCode are considered to have a large proportion of not well managed projects. We believe our data cleaning step increases the data quality. The other threats to internal validity is possible biases in the defect data. We plan to include well managed projects (e.g., Linux projects, Eclipse projects, and Apache projects) in future studies.

Threats to external validity concern the possibility to generalize our results. Although we demonstrate the capability of the universal model on predicting defects for four Eclipse projects and one Apache project, it is unclear if the universal model also performs well for commercial projects. Future validation on commercial projects is recommended.

Threats to reliability validity concern the possibility of replicating this study. The subject projects are publicly available from SourceForge and GoogleCode. We attempt to provide all necessary details to replicate our study[8].

6.6 Chapter Summary

In this study, we attempt to build a universal defect prediction model using a large set of projects from various contexts. We first propose a context-aware rank transformation method to pre-process the predictors. This step makes predictors (i.e., software metrics) from the entire set of projects have the same scales. We compare our rank transformation and widely used log transformation, and find that the rank transformation performs as good as log transformation in within-project settings. We then build a universal model using the rank-transformed metrics. For building a universal model, we add different metric sets (i.e., code metrics, process metrics, and context factors) step by step. The studied context

[8]http://www.feng-zhang.com/replication/universalModel

factors include programming languages, presence of issue tracking systems, the total lines of code, the total number of files, the total number of commits, and the total number of developers. The results show that the context factors further increase the predictive power of the universal model besides code and process metrics.

To evaluate the performance of the universal defect prediction model, we compare with within-project models. We find that the universal model has higher AUC values but lower F-measures than within-project models, suggesting that different cut-off values may be needed for different projects. We also study the generalizability of the universal model. First, we apply the universal model that is built using projects from SourceForge and GoogleCode on five external projects from Eclipse and Apache repositories. We observe that the AUC values of the predictions by the universal model are very close to within-project models built from each project. Moreover, we provide several insights on how to select appropriate cut-off values to control false positive rate. For instance, the median false positive rate is reduced to 0.053, if considering only the top 10% of entities as defective.

We further investigate if the universal model performs differently for projects with different contexts, and find there is no statistically significant difference for all context factors. Based on our findings, we conclude that our universal model is context-insensitive and applicable to external projects. Finally, we investigate the importance of different metrics using logistic regression model and present coefficients of each metric in the universal model.

Chapter 7 Unsupervised Approach

Key Question

? *Is it feasible to generalize a defect prediction model that is built using an unsupervised approach?*

7.1 Introduction

A typical solution of cross-project prediction is to apply defect prediction models that are built using data from other training projects using supervised classifiers [187, 202]. The major challenge in cross-project prediction comes from the heterogeneity between the training projects and the target project [43, 135]. Another heterogeneity problem in cross-project prediction, as pointed out recently by Nam and Kim [134], is that different projects may have different sets of metrics all together. To mitigate such challenge, an unsupervised classifier could be used.

Unsupervised classifiers do not require any training data, and are therefore by nature free of the problems that are due to heterogeneity of the training and target projects. To this end, we investigate the feasibility of using unsupervised classifiers in a cross-project

setting. We investigate two types of unsupervised classifiers: a) distance-based classifiers (e.g., k-means clustering) that partition the data based on Euclidean distance; and b) connectivity-based classifiers (e.g., spectral clustering) that partition the data based on the connectivity among all entities. While distance-based unsupervised classifiers have shown disappointing performance for within-project defect prediction (e.g., [59]), connectivity-based unsupervised classifiers have never been explored before in our community.

In this thesis, we propose a new unsupervised connectivity-based classifier that is based on spectral clustering [137, 194]. Spectral clustering has achieved empirical success in many areas. Unlike distance-based classifiers that partition the data based on Euclidean distance, spectral clustering considers the connectivity among all entities and therefore has many advantages [117]. The connectivity among software entities can be determined by their similarity in metric values. Our key intuition for exploring spectral clustering is that defective entities tend to cluster around the same neighbourhoods (i.e., clusters), as observed by Menzies *et al.* [124] and Bettenburg *et al.* [15] in their work on local prediction models.

To evaluate the feasibility of using unsupervised classifiers for cross-project prediction, we perform an experiment using three publicly available datasets (i.e., AEEEM [44], NASA [136], and PROMISE [96]) that include 26 projects in total. Our major findings are presented as follows:

- Unsupervised classifiers underperform supervised classifiers in general. However, spectral clustering, as a connectivity-based unsupervised classifier, can compete with supervised classifiers.

- Our connectivity-based unsupervised spectral classifier achieves an average AUC value of 0.72, and ranks as one of the top classifiers in a cross-project setting.

- We investigate the performance of our connectivity-based spectral classifier in a within-project setting and we find that our connectivity-based unsupervised classifier ranks in the second tier, the same as three commonly used supervised classifiers (i.e., logistic regression, logistic model tree, and naive Bayes). The random forest appears in the first rank.

- A deeper investigation shows that defective entities have significantly stronger connections with other defective entities than with clean entities. Hence, our spectral classifier can successfully separate defective entities from clean entities based on their connectivity.

As a summary, we propose to tackle cross-project predictions from a different perspective, i.e., using unsupervised classifiers. Our connectivity-based unsupervised spectral classifier is relatively simple and fully automated, and can be directly applied on a given project without any training.

Chapter organization. Section 7.2 presents the background on spectral clustering. Section 7.3 describes our connectivity-based unsupervised spectral classifier. Experimental setup and case study results are presented in Sections 7.4 and 7.5, respectively. Section 7.6 examines the defect data in order to better understand the strong performance of our spectral classifier. The threats to validity of our work are discussed in Section 7.7. We summarize the chapter in Section 7.8.

7.2 Background on Unsupervised Classifiers

Unsupervised classifiers make use of clustering methods. Clustering is a common way to explore groups of similar entities. Frequently applied clustering methods include hierarchical clustering and k-means. Hierarchical clustering produces clusters based on the

structure of a similarity or dissimilarity matrix. K-means clustering is used to cluster high-dimensional data that are linearly separable [49].

In recent years, spectral clustering has become one of the most effective techniques for clustering [137, 194]. Unlike distance-based classifiers (e.g., k-means clustering) that divide a data set based on Euclidean distance, spectral clustering partitions a data set based on the connectivity between its entities. Spectral clustering is performed on a graph consisting of nodes and edges. In the context of defect prediction, each node represents a software entity (e.g., file or class). Each edge represents the connection between software entities, and its weight is measured by the similarity of metric values between its two ends.

Similarity definition. A widely used similarity is the dot product between vectors of two nodes [2, 19, 47]. Each node can be represented by a vector of values of multiple metrics of the node. The similarity between two software entities i and j is defined in Equation (7.1).

$$w_{ij} = \mathbf{x_i} \cdot \mathbf{x_j} = \sum_{k=1}^{m} a_{ik} a_{kj} \tag{7.1}$$

where $\mathbf{x_i}$ *and* $\mathbf{x_j}$ *denote the metric values of software entities* i *and* j*, respectively;* a_{ik} *is the value of the* k*th metric on the* i*th software entity, and* m *is the total number of metrics.*

From the geometric perspective, the similarity w_{ij} can be interpreted as the cosine similarity $\mathbf{x_i} \cdot \mathbf{x_j} = |\mathbf{x_i}||\mathbf{x_j}| cos\theta_{ij}$, where $|x_i|$ and $|x_j|$ are the norms, and θ_{ij} is the angle between two vectors. It is the length of the projection of one vector onto the other unit vector.

From a correlation perspective, the similarity w_{ij} is basically the unnormalized Pearson correlation coefficient [20] between nodes i and j. Each element in vector x_i represents a metric value. It is unnormalized, since it makes little sense to normalize the values across metrics belonging to the same software entity. The similarity w_{ij} can be positive, negative or zero. A positive value indicates a positive correlation between two software entities, and

Algorithm 7.1: Spectral clustering based defect prediction

Input: A matrix with rows as software entities and columns as metrics.

Output: A vector of defect proneness of all software entities.

1: Normalize software metrics using z-score.
2: Construct a weighted adjacency matrix W.
3: Calculate the graph Laplacian matrix L_{sym}.
4: Perform the eigendecomposition on L_{sym}.
5: Select the second smallest eigenvector v_1.
6: Perform the bipartition on v_1 using zero.
7: Label each cluster as defective or clean.

a negative value indicates a negative correlation. A value of zero indicates that there is no linear correlation. It is meaningless to study the self-circle of a software entity, therefore we set the self-similarity (i.e., all w_{ii}) to zero.

Spectral clustering steps. A popular algorithm for spectral clustering is to minimize the normalized cut [174]. This algorithm partitions a graph into two subgraphs to gain high similarity within each subgraph while achieving low similarity across the two subgraphs. It has four major steps: 1) computing the graph Laplacian matrix L from a weighted adjacency matrix W that is constructed using software metrics; 2) performing an eigendecomposition on L; and 3) selecting a threshold on the second smallest eigenvector v_1 to obtain the bipartitions of the graph. Appendix A.2 shows definitions of matrices W and L.

7.3 Our Spectral Classifier

In this section, we present details on our proposed spectral classifier. Our classifier is described in Algorithm 7.1. As our approach is relatively simple to implement, we further present its R[1] implementation which consists of 17 lines of code in Appendix A.1. The details are described as follows.

[1]https://www.r-project.org

7.3.1 Preprocessing Software Metrics

Software metrics have varied scales. Hence, software metrics are often normalized before further processing [73, 135, 140]. For instance, Nam *et al.* [135] find that applying z-score to normalize software metrics can significantly improve the predictive power of defect prediction models. The advantage of z-score is that a normalized software metric has a mean value of zero and a variance of one.

Our spectral cluster uses the z-score for the normalization of each metric. We use the original values of m metrics to construct matrix A, as shown in Equation (7.2). We use $\mathbf{y_j}$ to denote a vector of values of the jth metric in a project, i.e., $\mathbf{y_j} = \{a_{1j}, \ldots, a_{nj}\}^T$, where n is the number of entities in the project, and a_{ij} is the value of the jth metric on the ith software entity. We normalize the vector $\mathbf{y_j}$ as $\hat{\mathbf{y}}_\mathbf{j} = \frac{\mathbf{y_j} - \bar{\mathbf{y}}_\mathbf{j}}{s_j}$, where $\bar{\mathbf{y}}_\mathbf{j}$ is the average value of $\mathbf{y_j}$ and s_j is the standard deviation of $\mathbf{y_j}$. This step corresponds to Line 1 in Algorithm 7.1.

$$A = \begin{bmatrix} \mathbf{y_1} & \ldots & \mathbf{y_j} & \ldots & \mathbf{y_m} \end{bmatrix} = \begin{bmatrix} a_{11} & \ldots & a_{1j} & \ldots & a_{1m} \\ \ldots & & \ldots & & \ldots \\ a_{n1} & \ldots & a_{nj} & \ldots & a_{nm} \end{bmatrix} \tag{7.2}$$

7.3.2 Spectral Clustering

As aforementioned, spectral clustering is composed of the following three steps.

(S1) Calculate the graph Laplacian matrix L_{sym}. The symmetric graph Laplacian matrix L_{sym} is derived from the adjacency matrix W that stores the similarity between each pair of software entities. The adjacency matrix W is computed directly from the normalized software metrics (i.e., Line 2 in Algorithm 7.1). In spectral clustering, there is usually an assumption that all values of the similarity are non-negative [127].

Hence, we set all negative w_{ij} to zero.

Then the graph Laplacian matrix L_{sym} is calculated using $L_{sym} = I - D^{-\frac{1}{2}} W D^{-\frac{1}{2}}$ (i.e., Line 3 in Algorithm 7.1), where the matrix I is the unit matrix with size n, the matrix D is a diagonal matrix of row sums of W, and $D^{-\frac{1}{2}} = Diag(d_1^{-\frac{1}{2}}, \ldots, d_n^{-\frac{1}{2}})$, where $d_i^{-\frac{1}{2}} = (\sum_{j=1}^{n} w_{ij})^{-\frac{1}{2}}$.

(S2) Perform the eigendecomposition on the graph Laplacian matrix L_{sym} (i.e., Line 4 in Algorithm 7.1). Eigenvalues will always be ordered increasingly [117, 174]. We follow the normalized cut algorithm by Shi and Malik [174] and use the second smallest eigenvector for clustering (i.e., Line 5 in Algorithm 7.1). We use v_1 to denote the second smallest eigenvector of L_{sym}.

(S3) Separate all entities into two clusters. Shi and Malik [174] propose to apply a particular threshold (e.g., zero or median) on the second smallest eigenvector v_1. If the median is used, then 50% of entities are predicted as defective. Inspecting 50% of entities requires significant effort. Thus we adopt zero as the threshold value of v_1 (i.e., Line 6 in Algorithm 7.1) to create two non-overlapped clusters. We use v_{1i} to denote the ith value of v_1, where $i \in \{1, \ldots, n\}$, and n is the total number of software entities (e.g., files or classes) in the given project. The value v_{1i} corresponds to the ith software entity. All entities with $v_1 > 0$ create a cluster called C_{pos}, and all entities with $v_1 < 0$ create the other cluster called C_{neg}. In the following subsection, we describe how to determine the cluster (i.e., C_{pos} or C_{neg}) containing defective entities.

7.3.3 Labelling Defective Cluster

We use $C_{defective}$ to denote the cluster that contains defective entities only, and use C_{clean} to represent the cluster that contains clean entities only.

To determine whether C_{pos} or C_{neg} is the defective cluster $C_{defective}$, we use the following heuristic: *For most metrics, software entities containing defects generally have larger values than software entities without defects.* This heuristic is based on our field's extensive empirical observations on the relationship between software metrics and defect proneness. For instance, Gaffney [58] finds that larger files have a higher likelihood to experience defects than smaller files. Kitchenham *et al.* [103] report that more complex files are more likely to experience defects than files with lower complexity. Similar findings are also observed in many other studies (e.g., [45, 75, 133]).

With this heuristic in mind, we use the average row sums of the normalized metrics of each cluster to determine which cluster is defective. The row sum is the sum of all metric values of the same entity. We compute the average row sum of all entities within each cluster (i.e., either C_{pos} or C_{neg}). The cluster with larger average row sum is considered as the cluster containing defective entities. We label all entities within this cluster as defective (i.e., $C_{defective}$), and all the remaining entities as clean (i.e., C_{clean}).

7.4 Experiment Setup

In this section, we present the experimental setup to evaluate the performance of our approach.

7.4.1 Corpora

We examined data from three commonly studied datasets: AEEEM [44], NASA [136], and PROMISE [96]. The three datasets are publicly available and have been commonly used in defect prediction studies (e.g., [59, 68, 124, 135]). A brief description on each dataset and our selected metrics are presented as follows.

1) **The AEEEM dataset** was prepared by D'Ambros *et al.* [44] to compare the performance of different sets of metrics. Accordingly, the AEEEM dataset contains the most number of metrics. In particular, it has 61 metrics, including product, process, previous-defect metrics, and entropy-based metrics.

   All projects in the AEEEM dataset have 61 identical software metrics. We use all 61 metrics in our study.

2) **The NASA dataset** was collected by the NASA Metrics Data Program [136]. Shepperd et al. [172] observe that the original NASA dataset contains many repeated and inconsistent data points, and they clean up the NASA dataset. In this study, we use the cleaned NASA dataset that is available in the PROMISE repository.

   In the NASA dataset, projects do not share the same set of metrics. For instance, project KC3 has 39 metrics while project JM1 has 21 metrics. Since supervised classifiers require exact the same sets of metrics, we only select the 20 metrics that commonly exist in all of the 11 studied NASA projects.

3) **The PROMISE dataset** was prepared by Jureczko and Madeyski [96]. It contains open source Java projects and has object-oriented metrics.

   In the PROMISE dataset, projects do not have the same set of metrics. Hence, we only select 20 metrics that exist in all of the 10 studied PROMISE projects.

In general, the selected projects have diverse size (i.e., having 125 to 7,782 entities) and varied ratios of defects (i.e., ranging from 2.1% to 63.6%). The summary of all selected projects is presented in Table 7.1. More details about these metrics can be found on the corresponding website of each dataset.

Table 7.1: An overview of the studied projects.

Data set	Project	Number of entities	Defective	
			(#)	(%)
AEEEM	Eclipse JDT Core	997	206	20.7%
	Equinox	324	129	39.8%
	Apache Lucene	691	64	9.3%
	Mylyn	1,862	245	13.2%
	Eclipse PDE UI	1,497	209	14.0%
NASA	CM1	327	42	12.8%
	JM1	7,782	1,672	21.5%
	KC3	194	36	18.6%
	MC1	1,988	46	2.3%
	MC2	125	44	35.2%
	MW1	253	27	10.7%
	PC1	705	61	8.7%
	PC2	745	16	2.1%
	PC3	1,077	134	12.4%
	PC4	1,287	177	13.8%
	PC5	1,711	471	27.5%
PROMISE	Ant v1.7	745	166	22.3%
	Camel v1.6	965	188	19.5%
	Ivy v1.4	241	16	6.6%
	Jedit v4.0	306	75	24.5%
	Log4j v1.0	135	34	25.2%
	Lucene v2.4	340	203	59.7%
	POI v3.0	442	281	63.6%
	Tomcat v6.0	858	77	9.0%
	Xalan v2.6	885	411	46.4%
	Xerces v1.3	453	69	15.2%
Average		1,036	196	18.9%

7.4.2 Performance Measure

There are many performance measures, such as precision, recall, accuracy, F-measure and the Area Under the receiver operating characteristic Curve (AUC). However, a cut-off value on the predicted probability of defect proneness is required when computing precision, recall, accuracy, and F-measure. The default cut-off is 0.5 which may not be the best cut-off value in practice. On the other hand, the AUC value is independent of a cut-off value and is not impacted by the skewness of defect data. Lessmann *et al.* [112] and Ghotra *et al.* [59] suggest to use the AUC value for better cross-dataset comparability. Hence, we select the AUC measure as our performance measure.

When computing the AUC measure, a curve of the false positive rate is plotted against the true positive rate. Accordingly, the AUC value measures the probability that a randomly chosen defective entity ranks higher than a randomly chosen clean entity. An AUC value of 0.5 implies that a classifier is no better than random guessing. A larger AUC value indicates a better performance. In particular, Gorunescu [65] advises the following guideline to interpret the AUC value: 0.90 to 1.00 as excellent prediction, 0.80 to 0.90 as a good prediction, 0.70 to 0.80 as a fair prediction, 0.60 to 0.70 as a poor prediction, and 0.50 to 0.60 as a failed prediction.

7.4.3 Classifiers for Comparison

To find if our unsupervised spectral classifier is applicable in a cross-project setting, we compare its performance with nine off-the-shelf classifiers. We not only select supervised classifiers, but also choose distance-based unsupervised classifiers.

For supervised classifiers, we select five classifiers that have been commonly applied to build defect prediction models. The five classifiers are random forest (RF), naive Bayes (NB), logistic regression (LR), decision tree (J48), and logistic model tree (LMT).

For distance-based unsupervised classifiers, we choose four classifiers that have been previously used in defect prediction [30, 206]. The four classifiers include k-means clustering (KM), partition around medoids (PAM), fuzzy C-means (FCM), and neural-gas (NG). These classifiers are based on Euclidean distance, therefore employ a different clustering mechanism than spectral clustering.

7.4.4 Scott-Knott Test

To compare the performance across the large number of datasets, we apply the Scott-Knott test [91] using the 95% confidence level (i.e., $\alpha = 0.05$). The Scott-Knott test can overcome the issue of overlapping multiple comparisons that are obtained from other tests, such as the Mann-Whitney U test [173]. The Scott-Knott test has been used in defect prediction studies to compare the performance across different classifiers [59].

The Scott-Knott test recursively ranks the evaluated classifiers through hierarchical clustering analysis. In each iteration, the Scott-Knott test separates the evaluated classifiers into two groups based on the performance measure (i.e., the AUC value). If the two groups have statistically significant difference in the AUC value, the Scott-Knott test executes again within each group. If no statistically distinct groups can be created, the Scott-Knott test terminates [59].

7.5 Case Study Results

We now present our research questions, along with our motivation, approach, and findings.

RQ1. How does the connectivity-based classifier perform in cross-project defect prediction?

Motivation. Unlike supervised classifiers, unsupervised classifiers do not have the problem of heterogeneity between the training projects and the target project. While distance-based classifiers (e.g., k-means clustering) underperform supervised classifiers, connectivity-based unsupervised classifiers have not been explored in our community. Hence, it is of significant interest to investigate if our connectivity-based spectral classifier can provide comparable performance as supervised classifiers in the context of cross-project prediction.

Approach. To address this question, we need to get the performance of all studied classifiers for each project. For each classifier, all entities of the target project are used to obtain its performance.

Supervised classifiers require a training project. All supervised classifiers under study require the exact same set of metrics. As the three studied datasets (i.e., AEEEM, NASA, and PROMISE) have different sets of metrics, we make cross-project defect prediction within the same dataset. For each target project, we select all other projects from the same dataset for training. For instance, if the target project is "Eclipse JDT Core", then each supervised classifier is used to build four models using each of the remaining projects within the AEEEM dataset (i.e., "Equinox", "Apache Lucene", "Mylyn", and "Eclipse PDE UI"), respectively. We compute the average AUC values of these four models to measure the performance of the corresponding classifier on the target project, since it is unknown which model performs the best on the target project prior to the prediction.

Unsupervised classifiers do not require training projects. We directly apply the studied unsupervised classifiers on the target project. When we do clustering, we create k clusters. We set $k = 2$ for clustering, since this setting yields the best performance in defect prediction (e.g., [59]). In the resulting two clusters, one cluster is labelled as defective, and the other cluster is labelled as clean, using the heuristic that is described in Section 7.3.3.

To compare the predictive power among all classifiers, we apply the Scott-Knott test with the 95% confidence level to rank all classifiers across projects within the same dataset. We examine the Scott-Knott ranks per dataset. Furthermore, we perform one large Scott-Knott run where we input all the AUC values for all the classifiers across all datasets.

Findings. Our connectivity-based unsupervised classifier achieves good results in

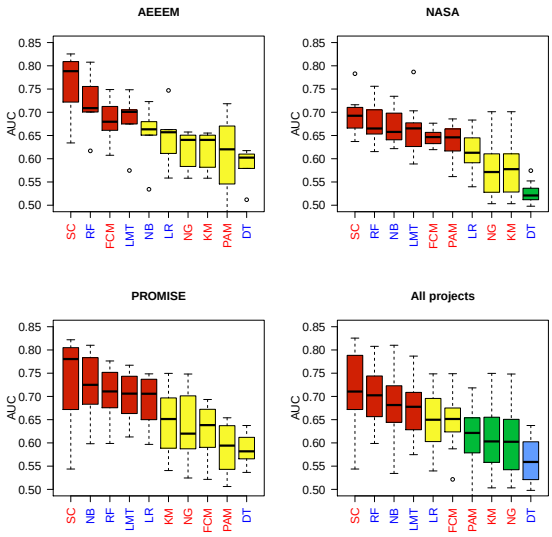


Figure 7.1: The boxplots of AUC values of all supervised (in blue color) and unsupervised classifiers (in red color) under study (for the abbreviations, see Section 7.4.3). Different colors represents different ranks (red > yellow > green > blue).

cross-project defect prediction. In general, our spectral classifier significantly outperforms all other unsupervised classifiers, and it has slightly better performance than the best supervised classifier under study (i.e., random forest).

Our spectral classifier ranks the first in all the three studied datasets. The colors in Figure 7.1 illustrate the ranks of all classifiers. The boxplots show the distribution of the AUC values of each classifier under study. Classifiers with boxplots in the same color are ranked at the same tier. The performances of classifiers in the same tier are not statistically distinct. Among all supervised and unsupervised classifiers, only two supervised classifiers (i.e., random forest and logistic model tree) are in the same ranking tier across all three datasets as our connectivity-based unsupervised spectral classifier.

The exact AUC values of the top four classifiers (i.e., our spectral classifier, random forest, naive Bayes, and logistic model tree) on each project are presented in Table 7.2. In particular, the median AUC values of the top four classifiers across all projects under study are: 0.71, 0.70, 0.68 and 0.68, respectively.

Table 7.2: The AUC values of the top four classifiers in cross-project defect prediction (**Bold** font highlights the best performance).

Dataset	Project	Spectral clustering	Random forest	Naive Bayes	Logistic model tree
AEEEM	Eclipse JDT Core	**0.83**	0.81	0.68	0.75
	Equinox	**0.81**	0.70	0.66	0.71
	Apache Lucene	**0.79**	0.76	0.72	0.70
	Mylyn	**0.63**	0.62	0.53	0.57
	Eclipse PDE UI	**0.72**	0.71	0.65	0.67
NASA	CM1	**0.67**	0.66	0.66	0.62
	JM1	**0.66**	0.62	0.64	0.60
	KC3	0.64	**0.65**	0.62	0.63
	MC1	0.69	**0.71**	0.66	0.67
	MC2	**0.68**	0.62	0.64	0.59
	MW1	**0.70**	0.67	**0.70**	0.67
	PC1	0.71	**0.73**	0.70	0.70
	PC2	0.78	0.76	0.73	**0.79**
	PC3	**0.72**	0.70	0.70	0.68
	PC4	0.65	**0.67**	0.63	**0.67**
	PC5	**0.71**	0.66	0.66	0.63
PROMISE	Ant v1.7	**0.79**	0.75	0.77	0.75
	Camel v1.6	**0.62**	0.60	0.60	0.61
	Ivy v1.4	0.70	**0.71**	0.68	0.70
	Jedit v4.0	**0.79**	0.74	0.75	0.73
	Log4j v1.0	**0.82**	0.76	0.81	0.74
	Lucene v2.4	0.67	0.68	**0.69**	0.66
	POI v3.0	**0.82**	0.71	0.78	0.69
	Tomcat v6.0	**0.80**	0.78	**0.80**	0.77
	Xalan v2.6	0.54	**0.66**	0.60	0.62
	Xerces v1.3	**0.77**	0.69	0.70	0.71
Median		**0.71**	0.70	0.68	0.68

We find that distance-based unsupervised classifiers (e.g., k-means) fail in the competition with supervised classifiers. The poor performance of these distance-based classifiers may explain why unsupervised classifiers are not widely applied in defect prediction.

As a summary, the results clearly show that applying connectivity-based unsupervised classification is a promising way to tackle the heterogeneity problem in cross-project defect prediction that is caused by the training and target data. Our connectivity-based unsupervised classifier is based on spectral clustering. We suspect that the success of spectral clustering is because defective entities are more similar to other defective entities than other clean entities in terms of values of software metrics. Such intuition is supported through recent work by Menzies *et al.* [124] and Bettenburg *et al.* [15] on local defect prediction models.

Our spectral classifier performs the best among all classifiers under study. Connectivity-based unsupervised classification is a promising avenue to tackle the problem of heterogeneous data in cross-project defect prediction.

RQ2. Does the connectivity-based classifier perform well in within-project defect prediction?

Motivation. **RQ1** shows the impressive performance of our connectivity-based unsupervised classifier in a cross-project setting. In comparison to a cross-project setting, the chance of experiencing heterogeneous training and target data is much lower in a within-project setting. As unsupervised classifiers can save significant effort in defect data collection, we are interested to find if our connectivity-based unsupervised classifier can still compete with supervised classifiers in a within-project setting.

Approach. To evaluate the performance of supervised classifiers in a within-project setting, the essential step is to separate all entities of a project into two sets. One set is for training a model and the other one is the target set to apply the model. Both supervised and unsupervised classifiers are applied on the same target set of entities. The only difference is that supervised classifiers require an additional step to build a model using the training set of entities.

To create the training and target sets, we apply the two-fold cross validation (i.e., a 50:50 random split) that has been previously applied in the defect prediction literature [133, 152]. For a 50:50 random split, each classifier is evaluated twice: 1) the first half is used as the training data while the other half is used as the target data; and 2) the second half is used as the training data while the first half is used as the target data. To deal with the

Table 7.3: Ranks within the same project.

Overall ranks	Classifier	Median rank	Average rank	Standard deviation
1	Random forest (RF)	1	1.42	0.64
2	Logistic regression (LR)	2	3.19	2.15
	Spectral classifier (SC)	3	3.35	1.67
	Logistic model tree (LMT)	3	3.42	1.94
	Naive Bayes (NB)	3.5	3.54	1.27
3	Fuzzy C- means (FCM)	6	5.96	1.08
4	Partition around medoids (PAM)	6.5	6.73	1.69
	Neural-gas (NG)	7	6.85	1.67
	Decision tree (DT)	7	6.89	1.56
	k-means (KM)	7.5	7.35	1.55

randomness of sampling, we repeat the random splits 500 times. In total, 1,000 evaluations are performed for each classifier on each project. To get the performance of each classifier on each project, we compute the average AUC value of the total 1,000 evaluations.

To find statistically distinct ranks of all classifiers, we follow the approach of Ghotra *et al.* [59] and perform a double Scott-Knott test. The double Scott-Knott test ensures a robust ranking of all classifiers across projects, regardless of their exact AUC values. The first Scott-Knott test is performed on each individual project to rank all classifiers based on their AUC values for that particular project. The obtained ranks are used in the second run of the Scott-Knott test to yield a global ranking of all classifiers across all studied projects.

Findings. Generally speaking, supervised classifiers perform better than unsupervised classifiers in a within-project setting. In the top five classifiers, there is only one unsupervised classifier, our connectivity-based unsupervised classifier. It implies that the performance of most supervised classifiers can be significantly improved, if the heterogeneity between the training and target projects is mitigated.

The detailed rankings are presented in Table 7.3, including the global ranks of all classifiers across all projects, and the statistics (i.e., median, average, and standard deviation) of the ranks of each classifier obtained in the first Scott-Knott test. In particular, our

Table 7.4: The average AUC values of the top five classifiers in both cross-project (CP) and within-project settings (WP). The column "*diff*" shows the difference between cross-project models and within-project models.

Dataset	Project	RF			LR			SC			LMT			NB		
		CP	**WP**	***diff***	**CP**	**WP**	***diff***	**CP**	**WP**	***diff***	**CP**	**WP**	***diff***	**CP**	**WP**	***diff***
AEEEM	Eclipse JDT Core	0.81	0.87	*0.06*	0.75	0.79	*0.04*	0.83	0.83	*0*	0.75	0.82	*0.07*	0.68	0.74	*0.06*
	Equinox	0.70	0.84	*0.14*	0.61	0.64	*0.03*	0.81	0.80	*-0.01*	0.71	0.79	0.08	0.66	0.72	*0.06*
	Apache Lucene	0.76	0.81	*0.05*	0.66	0.63	*-0.03*	0.79	0.79	*0*	0.70	0.78	0.08	0.72	0.74	*0.02*
	Mylyn	0.62	0.82	*0.20*	0.56	0.79	*0.23*	0.63	0.63	*0*	0.57	0.78	0.21	0.53	0.65	*0.12*
	Eclipse PDE UI	0.71	0.78	*0.07*	0.66	0.73	*0.07*	0.72	0.72	*0*	0.67	0.75	0.08	0.65	0.67	*0.02*
NASA	CM1	0.66	0.68	*0.02*	0.61	0.74	*0.13*	0.67	0.67	*0*	0.62	0.65	0.03	0.66	0.67	*0.01*
	JM1	0.62	0.67	*0.05*	0.55	0.69	*0.14*	0.66	0.66	*0*	0.60	0.68	*0.08*	0.64	0.65	*0.01*
	KC3	0.65	0.71	*0.06*	0.59	0.64	*0.05*	0.64	0.64	*0*	0.63	0.63	*0*	0.62	0.65	*0.03*
	MC1	0.71	0.81	*0.10*	0.64	0.74	*0.10*	0.69	0.69	*0*	0.67	0.58	*-0.09*	0.66	0.68	*0.02*
	MC2	0.62	0.65	*0.03*	0.54	0.66	*0.12*	0.68	0.67	*-0.01*	0.59	0.67	*0.08*	0.64	0.66	*0.02*
	MW1	0.67	0.72	*0.05*	0.59	0.64	*0.05*	0.70	0.70	*0*	0.67	0.63	*-0.04*	0.70	0.71	*0.01*
	PC1	0.73	0.83	*0.10*	0.68	0.82	*0.14*	0.71	0.71	*0*	0.70	0.75	*0.05*	0.70	0.68	*-0.02*
	PC2	0.76	0.74	*-0.02*	0.65	0.66	*0.01*	0.78	0.78	*0*	0.79	0.53	*-0.26*	0.73	0.71	*-0.02*
	PC3	0.70	0.78	*0.08*	0.65	0.81	*0.16*	0.72	0.72	*0*	0.68	0.71	*0.03*	0.70	0.73	*0.03*
	PC4	0.67	0.91	*0.24*	0.63	0.88	*0.25*	0.65	0.65	*0*	0.67	0.88	*0.21*	0.63	0.74	*0.11*
	PC5	0.66	0.76	*0.10*	0.60	0.73	*0.13*	0.71	0.71	*0*	0.63	0.72	*0.09*	0.66	0.68	*0.02*
PROMISE	Ant v1.7	0.75	0.82	*0.07*	0.74	0.80	*0.06*	0.79	0.79	*0*	0.75	0.81	0.06	0.77	0.78	*0.01*
	Camel v1.6	0.60	0.71	*0.11*	0.61	0.73	*0.12*	0.62	0.62	*0*	0.61	0.69	*0.08*	0.60	0.67	*0.07*
	Ivy v1.4	0.71	0.67	*-0.04*	0.69	0.55	*-0.14*	0.70	0.70	*0*	0.70	0.57	*-0.13*	0.68	0.64	*-0.04*
	Jedit v4.0	0.74	0.80	*0.06*	0.72	0.77	*0.05*	0.79	0.78	*-0.01*	0.73	0.78	*0.05*	0.75	0.75	*0*
	Log4j v1.0	0.76	0.80	*0.04*	0.74	0.69	*-0.05*	0.82	0.78	*-0.04*	0.74	0.81	*0.07*	0.81	0.81	*0*
	Lucene v2.4	0.68	0.77	*0.09*	0.65	0.75	*0.10*	0.67	0.66	*-0.01*	0.66	0.75	*0.09*	0.69	0.73	*0.04*
	POI v3.0	0.71	0.88	*0.17*	0.70	0.83	*0.13*	0.82	0.81	*-0.01*	0.69	0.83	*0.14*	0.78	0.82	*0.04*
	Tomcat v6.0	0.78	0.81	*0.03*	0.75	0.82	*0.07*	0.80	0.80	*0*	0.77	0.81	*0.04*	0.80	0.80	*0*
	Xalan v2.6	0.66	0.85	*0.19*	0.60	0.81	*0.21*	0.54	0.54	*0*	0.62	0.81	*0.19*	0.60	0.76	*0.16*
	Xerces v1.3	0.69	0.83	*0.14*	0.72	0.77	*0.05*	0.77	0.77	*0*	0.71	0.74	*0.03*	0.70	0.79	*0.09*
Median		**0.70**	**0.80**	***0.07***	**0.65**	**0.74**	***0.09***	**0.71**	**0.71**	***0***	**0.68**	**0.75**	***0.07***	**0.68**	**0.72**	***0.02***

connectivity-based unsupervised spectral classifier has a median rank of 3, and is ranked in the same tier as three widely used classifiers, i.e., logistic regression, logistic model tree, and naive Bayes.

The actual AUC values of the top five classifiers (i.e., random forest, logistic regression, our connectivity-based classifier, logistic model tree, and naive Bayes) on each project are presented in Table 7.4. The AUC values in both cross-project and within-project settings are presented, as well as their difference (i.e., the AUC value in a within-project setting minus the AUC value in a cross-project setting).

Our connectivity-based unsupervised classifier yields almost the same predictive power in both cross-project and within-project settings across all studied projects, as shown in Table 7.4. The size of the target project in a within-project setting is only half of that in a cross-project setting, implying that our connectivity-based unsupervised classifier tends to be robust when the size of the target project changes.

A within-project model can sometimes significantly underperform a cross-project model, although a within-project model generally outperforms a cross-project model. For example, looking at Table 7.4, and for project "Ivy v1.4", the top four supervised classifiers experience a downgraded performance when changing from a cross-project setting to a within-project setting. In particular, the random forest classifier achieves an AUC value of 0.71 in a cross-project setting, but yields a lower AUC value of 0.67 in a within-project setting. We conjecture that the decrease in performance when changing to a within-project setting is caused by the low ratio of defects in the target project. For instance, project "Ivy v1.4" has a ratio of defects of 6.6% with only 16 defective entities. Similar observations are noted in other projects, such as "Apache Lucene" and "PC2".

Supervised classifiers tend to experience a performance decrease, if the ratio of defects becomes lower. To illustrate the relationships between the performance of each classifier and the ratio of defects, we plot regression lines of the performance difference between a within-project setting and a cross-project setting of the top five classifiers over the ratio of defects in Figure 7.2.

In comparison to supervised classifiers, our connectivity-based unsupervised classifier is more robust with varying ratio of defects. One possible reason is that supervised classifiers experience a significant class-imbalance problem on these projects, while our unsupervised classifier has no issue of class-imbalance. We conjecture that, for projects

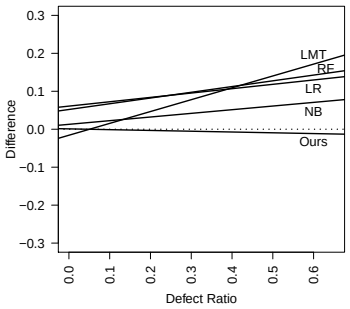


Figure 7.2: The regression lines of the performance difference of the top five classifiers between a within-project setting and a cross-project setting over the ratio of defects of each project. (The dotted line is the horizontal base line.)

with a low ratio of defects, our connectivity-based classifier may be more suitable than the supervised classifiers.

In a within-project setting, our connectivity-based classifier ranks in the second tier with only random forest ranking in the first tier. However, our approach may be more suitable for projects with heavily imbalanced (i.e., very low count of defective entities) defect data.

7.6 Why Does It Work?

In this section, we present an in-depth analysis to understand why our connectivity-based classifier achieves good results in defect prediction.

Our connectivity-based classifier is based on spectral clustering. As aforementioned, spectral clustering separates all entities in a project based on the connections among entities. We conjecture that software entities may reside within two "social network"-like communities based on software metrics: 1) one community is formulated by defective entities; and 2) the other one is established by clean entities.

7.6.1 Essential Definitions

Community definition. We define a community as a set of members (i.e., software entities) that have much stronger connections with each other than with members from other communities. A connection is basically an edge in a graph, as mentioned in Section 7.2. We define an edge between entities i and j using Equation (7.3).

$$e_{ij} = \mathbb{1}(w_{ij}) \tag{7.3}$$

where $\mathbb{1}(w_{ij}) = 1$ *if* $w_{ij} > 0$, *and* $\mathbb{1}(w_{ij}) = 0$ *otherwise.*

As described in Section 7.2, w_{ij} represents the similarity or the correlation between entities i and j. Hence, e_{ij} equals to 1, if there is a positive correlation between entities i and j. We denote the set of all edges as E, then $E = \{e_{ij}\}$.

We construct the community as follows. For each project, we partition the entities into two sets based on their defect proneness. We use V_d to denote the set of actual defective entities, and V_c to denote the set of actual clean entities. A software entity can be either defective or clean. Hence, there is no overlap between V_d and V_c, and the union of V_d and V_c contains all entities within the same project.

Connectivity measurement. We define deg^{dd}, the total degree of all defective entities, using Equation (7.4). We define deg^{dd}, the total degree of all clean entities, using Equation (7.5). Similarly, we define deg^{cd}, the total number of edges between each pair of defective and clean entities, using Equation (7.6).

$$deg^{dd} = \sum_{i \in V_d} \sum_{j \in V_d} e_{ij}, \quad j \neq i \tag{7.4}$$

$$deg^{cc} = \sum_{i \in V_c} \sum_{j \in V_c} e_{ij}, \quad j \neq i \tag{7.5}$$

$$deg^{cd} = \sum_{i \in V_c} \sum_{j \in V_d} e_{ij} \tag{7.6}$$

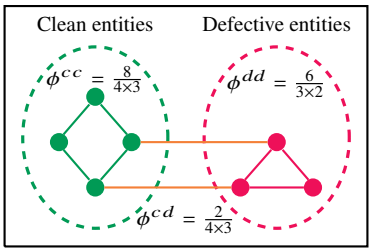


Figure 7.3: Illustrating example of computing the ratio of edges (i.e., ϕ^{dd}, ϕ^{cc}, ϕ^{cd}).

To measure the connectivity among entities within V_d or V_c, or between V_d and V_c, we further define the ratio of edges (i.e., connections) as follows.

$$\phi^{dd} = \frac{deg^{dd}}{|V_d|(|V_d|-1)} \tag{7.7}$$

$$\phi^{cc} = \frac{deg^{cc}}{|V_c|(|V_c|-1)} \tag{7.8}$$

$$\phi^{cd} = \frac{deg^{cd}}{|V_c||V_d|} \tag{7.9}$$

To illustrate the computation, we present an example in Figure 7.3. There are three defective and four clean entities. Each defective entity has connections to all other two defective entities. Hence, $deg^{dd} = 2+2+2 = 6$ and $\phi^{dd} = \frac{6}{3\times2} = 1.000$. Similarly, we can get $deg^{cc} = 2+2+2+2 = 8$ and $\phi^{cc} = \frac{8}{4\times3} = 0.667$, and $deg^{cd} = 2$ and $\phi^{cd} = \frac{2}{4\times3} = 0.167$.

7.6.2 Hypotheses

For each project, we compute the ratios ϕ^{dd}, ϕ^{cc}, and ϕ^{cd} based on the actual defect proneness. To compare the connectivity among entities across all projects under study, we test the following hypotheses:

$H0_1$: there is no difference in the ratios of connections ϕ^{dd} (i.e., among defective entities) and ϕ^{cd}(i.e., between defective and clean entities).

$H0_2$: there is no difference in the ratios of connections ϕ^{cc} (i.e., among clean entities) and

ϕ^{cd} (i.e., between defective and clean entities).

Hypotheses $H0_1$ and $H0_2$ are two sided and paired, since each project has three unique values: ϕ^{dd}, ϕ^{cc}, and ϕ^{cd}. To test the hypotheses, we apply paired Mann-Whitney U test using the 95% confidence level (i.e., $\alpha < 0.05$). We further compute the Cliff's δ [160] as the effect size to quantify the difference. Both the Mann-Whitney U test and the Cliff's δ are non-parametric statistical methods, and do not require a particular distribution of assessed variables. An effect size is large, if Cliff's $|\delta| \geq 0.474$ [160].

7.6.3 Empirical Findings

We observe that in general the connections between defective and clean entities are weaker than the connection among defective entities and the connections among clean entities. Table 7.5 presents the detailed values of our three measures (i.e., ϕ^{cc}, ϕ^{cd}, and ϕ^{dd}) for each project. For instance, in project "Eclipse JDT Core", the ratio of connections among defective entities $\phi^{dd} = 0.564$. The ratio of connections among clean entities $\phi^{cc} = 0.614$. These two ratios are significantly greater than the ratio of connections between clean and defective entities which is $\phi^{cd} = 0.365$.

Defective entities have significantly stronger connections with other defective entities than with clean entities. The p-value of the Mann-Whitney U test is is 4.20e-05, when comparing the ratios ϕ^{dd} and ϕ^{cd} across all projects. The difference is large, as the corresponding Cliff's $|\delta|$ is 0.654>0.474.

Similarly, clean entities have significantly stronger connections with other clean entities than with defective entities (i.e., the p-value of the Mann-Whitney U test is 8.55e-06). The difference is also large, as Cliff's $|\delta|$ is 0.769>0.474.

Table 7.5: The values of ϕ^{cc}, ϕ^{cd}, and ϕ^{dd} for each project. (Bold font highlights the minimum value per row).

Dataset	Project	ϕ^{cc}	ϕ^{cd}	ϕ^{dd}
AEEEM	Eclipse JDT Core	0.614	**0.365**	0.564
	Equinox	0.694	**0.443**	0.470
	Apache Lucene	0.554	**0.374**	0.556
	Mylyn	0.575	**0.442**	0.489
	Eclipse PDE UI	0.576	**0.426**	0.512
NASA	CM1	0.616	**0.497**	0.502
	JM1	0.628	**0.515**	0.519
	KC3	0.585	0.498	**0.477**
	MC1	0.572	**0.437**	0.540
	MC2	0.646	**0.495**	0.496
	MW1	0.551	**0.439**	0.546
	PC1	0.594	**0.470**	0.556
	PC2	0.594	**0.442**	0.602
	PC3	0.586	**0.450**	0.593
	PC4	0.583	**0.489**	0.577
	PC5	0.714	**0.574**	0.588
PROMISE	Ant v1.7	0.522	**0.398**	0.606
	Camel v1.6	0.487	**0.455**	0.481
	Ivy v1.4	0.482	**0.417**	0.508
	Jedit v4.0	0.504	**0.402**	0.536
	Log4j v1.0	0.538	**0.368**	0.535
	Lucene v2.4	0.542	**0.438**	0.459
	POI v3.0	0.605	**0.390**	0.537
	Tomcat v6.0	0.485	**0.380**	0.630
	Xalan v2.6	0.540	0.439	**0.438**
	Xerces v1.3	0.488	**0.394**	0.504
Median		0.576	**0.439**	0.536

As a summary, our observation indicates that either defective or clean entities are similar in terms of metric values, but defective and clean entities are less likely to experience similar metric values. In other words, there roughly exist two communities based on defect proneness. Entities within the same community have stronger connections than cross communities. This may be the reason as to why our connectivity-based unsupervised classifier achieves empirically good results in defect prediction.

There roughly exist two communities of entities: a defective community and a clean community of entities. Within-community connections are significantly stronger than cross-community connections.

7.7 Threats to Validity

We now describe the threats to validity of our study under common guidelines [199].

Threats to conclusion validity concern the relation between the treatment and the outcome. The major threat is that we only compare our approach with off-the-shelf classifiers. Future work should explore state-of-the-art cross project defect classifiers. Unfortunately the implementation of such specialized classifiers are rarely available and often required a considerable amount of setup – making them hard for practitioners to easily adopt. Hence we chose to compare against commonly used and readily available classifiers.

Threats to internal validity concern our selection of subject systems and analysis methods. We select 26 projects that have been heavily used in defect prediction literature. These projects are from different domains, include both open source and industrial projects, and have different sets of metrics. However, evaluating our approach on a large scale of projects is always desirable. Nevertheless our findings highlight the importance of exploring connectivity-based unsupervised classifiers in future defection prediction research. Moreover, the simplicity of our spectral classifier makes exploring it in future studies as a very lightweight and simple step to perform.

Threats to external validity concern the possibility of generalizing our results. Our approach requires software metrics that can be computed in a standard way by publicly available tools. However, only metrics that are collected in the three data sets are applied in our experiments. Replication studies using different sets of metrics would be advisable.

Threats to reliability validity concern the possibility of replicating this study. All the three studied data sets are publicly available. Moreover, the R implementation of our approach is provided in Appendix A.1.

7.8 Chapter Summary

This study brings a new insight to tackle this challenge using connectivity-based unsupervised classifiers. Apart from distance-based unsupervised classifiers (e.g., k-means clustering), the connectivity-based unsupervised classifiers assume that defective entities tend to cluster around the same area.

The experiment results show that our connectivity-based unsupervised classifier achieves impressive performance in a cross-project setting. Specifically, our spectral classifier ranks as one of the top classifiers among five supervised classifiers (e.g., random forest) and five unsupervised classifiers (e.g., k-means). In a within-project setting, our spectral classifier ranks in the second tier the same as three widely used supervised classifiers (e.g., logistic regression, logistic model tree, and naive Bayes) with random forest as the only classifier in the first tier. Our contributions are summarized as follows:

- **Demonstrating that connectivity-based unsupervised classification performs well in a cross-project setting.** Our experiments show that our connectivity-based unsupervised classifier can achieve similar or better performance than several commonly used supervised and unsupervised classifiers. We believe that unsupervised classification holds great promise in defect prediction, especially in a cross-project setting and for highly skewed within-project settings.

- **Demonstrating the existence of two (defective and clean) separated communities of software entities based on the connectivity between the metrics of the entities in each community.** We believe that this observation highlights the importance for the software engineering research community to explore more techniques for unsupervised defect prediction instead of current strong reliance on supervised classifiers.

Part V

Conclusion and Future Work

"The Heavens are in motion ceaselessly; The enlightened minds exert themselves constantly."

— I Ching (Classic of Changes)

Chapter 8 Conclusions and Future Work

A defect prediction model can help prioritize instances (e.g., files or classes) for inspection. It is beneficial to software organizations who often experience limited resources and tight schedules for testing activities. However, the benefit of a defect prediction model comes at a cost – additional effort is required to build an appropriate defect prediction model.

Due to the difficulty in transferring within-project and cross-project models, it is desirable to generalize a defect prediction model to boost the adoption of defect prediction models in practice. In this thesis, we aim to investigate how to generalize a defect prediction model. In the following subsections, we present our major contributions and outline promising directions for future research.

8.1 Contributions and Findings

The overall goal of this thesis is to generalize a defect prediction model. As the major challenge towards generalizing a defect prediction model comes from the heterogeneity across projects, we first investigate the distribution of metric values across projects. We then examine if appropriate data pre-processing steps are helpful to mitigate the heterogeneity. Finally, we propose and evaluate both supervised and unsupervised approaches

for building a generalized defect prediction model. We summarize the major contributions of this thesis as follows.

(1) Analyze how the distribution of metric values varies across projects with varied context factors (Chapter 3). This is a prerequisite analysis to deal with the heterogeneity problem between the training and target data. We find that the distribution of metric values does vary across projects, but projects with different context factors can experience similar distributions of metric values. Such findings inspire us to propose a context-aware rank transformation (see Chapter 6) to pre-process software metrics.

(2) Examine the impact of data pre-processing on the performance of cross-project defect prediction models. In particular, we study two commonly applied pre-processing steps, i.e., transformation and aggregation of software metrics.

a) Transformation of software metrics. Although different transformations do not offer significantly different benefit to cross-project prediction, they do retain non-redundant information of the original metrics (Chapter 4). Therefore, we propose an approach to combine models built with the three transformations, and our approach achieves promising improvements.

b) Aggregation of software metrics. Broadly speaking, aggregating software metrics by the widely used summation ([111, 112, 128, 138, 154, 202, 209]) tends to underestimate the performance of models that predict defect proneness (Chapter 5). The conclusion is drawn from a large-scale empirical study on eleven aggregation schemes (e.g., dispersion, central tendency, inequality, and entropy). Given that the computation cost for these additional aggregation schemes is negligible, we strongly suggest researchers and practitioners experiment with many aggregation schemes when building defect prediction models.

(3) Propose both supervised and unsupervised approaches towards generalizing defect prediction models. Supervised classifiers are widely applied to build defect prediction models. The use of supervised classifiers is challenging because of the heterogeneity between the training and target data. On the other hand, unsupervised classifiers do not require any training data, and therefore have no issue of heterogeneity.

a) The supervised approach. To deal with the heterogeneity between the training and target projects, we propose to transform software metrics by considering the context factors of projects (Chapter 6). Specifically, we propose to group together projects with a similar distribution of metric values, and apply rank transformation to convert metric values into exactly the same scales. Then we build a single model upon the transformed values using a large training set that contains 1,398 projects. We find that such a model is generalizable, since it achieves comparable performance to within-project models and is context-insensitive.

b) The unsupervised approach. We bring a new insight into tackling the heterogeneity problem using connectivity-based unsupervised classifiers (Chapter 7). Our experiments show that our connectivity-based unsupervised classifier can achieve similar or better performance than several commonly used supervised and unsupervised classifiers. Moreover, we demonstrate the existence of two (defective and clean) separated communities of software entities based on the connectivity between the metrics of the entities in each community.

As a summary, it is feasible to generalize defect prediction models.

8.2 Future Research

Although we demonstrate the feasibility of the two proposed approaches towards generalizing defect prediction models, there is plenty of room to further study the generalization of defect prediction models. We highlight several potential avenues for future research.

8.2.1 Combining the Supervised and Unsupervised Approaches

Both the supervised approach (Chapter 6) and the unsupervised approach (Chapter 7) are feasible to generalize defect prediction models, but they are applied separately. From the findings reported in Chapter 4, we learn that models with similar performance do not necessarily make wrong predictions on the same instance (e.g., file or class). Therefore, combining both approaches may have a high chance to create a defect prediction model that is more accurate and still generalizable. Future work can explore if there exists an appropriate way to combine the supervised and unsupervised approaches together.

8.2.2 Integrating a Generalized Defect Prediction Model into the Integrated Development Environment (IDE)

We envision a future that developers can get a list of defective files immediately after each change. The files are ranked based on their probability to experience defects in future. To achieve this goal, a defect prediction model should be integrated into the integrated development environment (IDE). A generalized defect prediction model does not need to be rebuilt for every project, thus making the integration easier. It may be interesting to investigate how defect prediction models would shape the software development process. For instance, does a defect prediction model help improve prioritizing testing resources?

8.2.3 Examining the Impact of Other Pre-processing Methods on the Performance of Defect Prediction Models

The treatment at each step of building a defect prediction model can threaten the generalizability of defect prediction models. In this thesis, we only examine transformation and aggregation. Other pre-processing steps have not yet been explored in the context of generalizing defect prediction models, such as metric reduction. Different software metrics can have varied capability to explain defect proneness across projects. Using the same set of software metrics may distort the performance of the generalized defect prediction model for some projects. Therefore, it is worth examining if metric reduction should be applied on each subset of similar projects separately.

8.2.4 Exploring the Generalization of Other Types of Defect Prediction Models

As mentioned in Chapter 5, there are other types of defect prediction models that predict defect count or rank. Apart from generalizing models that predict defect proneness, the generalization of other types of defect prediction models is also important. Therefore, future work is encouraged to explore approaches to generalize such models.

8.2.5 Evolution of a Generalized Defect Prediction Model

As technology evolves, the development environment may change. For instance, current developers may utilize social networks (e.g., StackOverflow and Twitter) in their development process. Such changes could alter the relationships between software metrics (e.g., the number of commits) and defect proneness. Hence, future researchers should pay attention to the evolution of a generalized defect prediction model. Given the difficulties encountered in the generalization, the evolution can be more challenging.

Bibliography

[1] Golnoush Abaei, Zahra Rezaei, and Ali Selamat. Fault prediction by utilizing self-organizing Map and Threshold. In *2013 IEEE International Conference on Control System, Computing and Engineering*, pages 465–470. IEEE, November 2013.

[2] Charu C. Aggarwal, editor. *Data Classification: Algorithms and Applications*. CRC Press, 2014.

[3] K.K. Aggarwal, Yogesh Singh, Arvinder Kaur, and Ruchika Malhotra. Empirical study of object-oriented metrics. *Journal of Object Technology*, 5(8):149–173, 2006.

[4] T.L. Alves, C. Ypma, and J. Visser. Deriving metric thresholds from benchmark data. In *Proceedings of the 26th IEEE International Conference on Software Maintenance*, pages 1 –10, September 2010.

[5] Carina Andersson and Per Runeson. A Replicated Quantitative Analysis of Fault Distributions in Complex Software Systems. *IEEE Transactions on Software Engineering*, 33(5):273–286, May 2007.

[6] Ömer Faruk Arar and Kürşat Ayan. Software defect prediction using cost-sensitive neural network. *Applied Soft Computing*, 33:263–277, August 2015.

[7] 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.

[8] 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, 2010.

[9] Anthony B Atkinson. On the measurement of inequality. *Journal of Economic Theory*, 2(3):244 – 263, 1970.

[10] K. Ayari, P. Meshkinfam, G. Antoniol, and M. Di Penta. Threats on building models from cvs and bugzilla repositories: The mozilla case study. In *Proceedings of the 2007 Conference of the Center for Advanced Studies on Collaborative Research*, CASCON '07, pages 215–228, Riverton, NJ, USA, 2007. IBM Corp.

[11] Robert Baggen, JoséPedro Correia, Katrin Schill, and Joost Visser. Standardized code quality benchmarking for improving software maintainability. *Software Quality Journal*, 20:287–307, 2012.

[12] T. Bakota, P. Hegedus, P. Kortvelyesi, R. Ferenc, and T. Gyimothy. A probabilistic software quality model. In *Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM'11)*, pages 243–252, 2011.

[13] Robert M. Bell, Elaine J. Weyuker, and Thomas J. Ostrand. Assessing the Impact of Using Fault Prediction in Industry. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 561–565, 2011.

[14] S. Benlarbi, K. El Emam, N. Goel, and S. Rai. Thresholds for object-oriented measures. In *Proceedings of the 11th International Symposium on Software Reliability Engineering*, pages 24 –38, 2000.

[15] Nicolas Bettenburg, Meiyappan Nagappan, and Ahmed E. Hassan. Think locally, act globally: Improving defect and effort prediction models. In *the 9th IEEE Working Conference on Mining Software Repositories*, MSR '12, pages 60–69, June 2012.

[16] Christian Bird, Adrian Bachmann, Eirik Aune, John Duffy, Abraham Bernstein, Vladimir Filkov, and Premkumar Devanbu. Fair and balanced?: bias in bug-fix datasets. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC/FSE '09, pages 121–130, New York, NY, USA, 2009.

[17] Anthony J. Bishara and James B. Hittner. Reducing bias and error in the correlation coefficient due to nonnormality. *Educational and Psychological Measurement*, 2014.

[18] P.S. Bishnu and V. Bhattacherjee. Software fault prediction using quad tree-based k-means clustering algorithm. *IEEE Transactions on Knowledge and Data Engineering*, 24(6):1146–1150, June 2012.

[19] Philippe Blanchard and Dimitri Volchenkov. *Mathematical Analysis of Urban Spatial Networks*. Springer Berlin Heidelberg, Heidelberg, Germany, 2009.

[20] Stephen P Borgatti and Martin G Everett. Models of core/periphery structures. *Social Networks*, 21(4):375 – 395, 2000.

[21] S. Bouktif, G. Antoniol, E. Merlo, and M. Neteler. A feedback based quality assessment to support open source software evolution: the grass case study. In *22nd IEEE*

International Conference on Software Maintenance, pages 155–165, Sept 2006.

[22] G. E. P. Box and D. R. Cox. An analysis of transformations. *Journal of the Royal Statistical Society. Series B (Methodological)*, 26(2):211–252, 1964.

[23] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[24] Lionel C. Briand, John W. Daly, and Jürgen K. Wüst. A unified framework for coupling measurement in object-oriented systems. *IEEE Transaction on Software Engineering*, 25(1):91–121, January 1999.

[25] Lionel C. Briand, Jürgen Wüst, Stefan V. Ikonomovski, and Hakim Lounis. Investigating quality factors in object-oriented designs: an industrial case study. In *the 21st International Conference on Software Engineering*, pages 345–354, 1999.

[26] Lionel C. Briand, Walcelio L. Melo, and Jürgen Wüst. Assessing the applicability of fault-proneness models across object-oriented software projects. *IEEE Transactions on Software Engineering*, 28(7):706–720, July 2002.

[27] Gerardo Canfora, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, Annibale Panichella, and Sebastiano Panichella. Multi-objective Cross-Project Defect Prediction. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 252–261. IEEE, March 2013.

[28] A. Capiluppi, P. Lago, and M. Morisio. Characteristics of open source projects. In *Software Maintenance and Reengineering, 2003. Proceedings. Seventh European Conference on*, pages 317 – 327, March 2003.

[29] 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.

[30] Cagatay Catal, Ugur Sevim, and Banu Diri. Metrics-driven software quality prediction without prior fault data. In Sio-Iong Ao and Len Gelman, editors, *Electronic Engineering and Computing Technology*, volume 60 of *Lecture Notes in Electrical Engineering*, pages 189–199. Springer Netherlands, 2010.

[31] Cagatay Catal, Ugur Sevim, and Banu Diri. Practical development of an Eclipse-based software fault prediction tool using Naive Bayes algorithm. *Expert Systems with Applications*, 38(3):2347–2353, March 2011.

[32] E. Ceylan, F.O. Kutlubay, and Ayşe Basar Bener. Software Defect Identification Using Machine Learning Techniques. In *32nd EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 240–247. IEEE, 2006.

[33] Lin Chen, Bin Fang, Zhaowei Shang, and Yuanyan Tang. Negative samples reduction in cross-company software defects prediction. *Information and Software Technology*, 62:67–77, June 2015.

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

[35] Fan R. K. Chung. *Spectral Graph Theory*. American Mathematical Society, 1997.

[36] Norman Cliff. Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological Bulletin*, 114(3):494–509, November 1993.

[37] J. Cohen, P. Cohen, S.G. West, and L.S. Aiken. *Applied Multiple Regression/Correlation Analysis for the Behavioral Sciences*. Lawrence Erlbaum, Mahwah, NY, USA, 3 edition, 2003.

[38] Jacob Cohen. *Statistical power analysis for the behavioral sciences : Jacob Cohen.* Lawrence Erlbaum, 2 edition, January 1988.

[39] Jacob Cohen. A power primer. *Psychological Bulletin*, 112(1):155–159, 1992.

[40] G. Concas, M. Marchesi, A. Murgia, R. Tonelli, and I. Turnu. On the distribution of bugs in the eclipse system. *IEEE Transactions on Software Engineering*, 37(6): 872–877, November 2011.

[41] Giulio Concas, Michele Marchesi, Sandro Pinna, and Nicola Serra. Power-laws in a large object-oriented software system. *IEEE Transactions on Software Engineering*, 33(10):687–708, October 2007.

[42] Frank A. Cowell. Generalized entropy and the measurement of distributional change. *European Economic Review*, 13(1):147 – 159, 1980.

[43] AE.C. Cruz and K. Ochimizu. Towards logistic regression models for predicting fault-prone code across software projects. In *the 3rd International Symposium on Empirical Software Engineering and Measurement*, pages 460–463, October 2009.

[44] Marco D'Ambros, Michele Lanza, and Romain Robbes. An extensive comparison of bug prediction approaches. In *Proceedings of the 7th IEEE Working Conference on Mining Software Repositories*, pages 31–41. IEEE CS Press, May 2010.

[45] Marco D'Ambros, Michele Lanza, and Romain Robbes. Evaluating defect prediction approaches: a benchmark and an extensive comparison. *Empirical Software Engineering*, 17(4-5):531–577, August 2012.

[46] Lucas Batista Leite De Souza and Marcelo De Almeida Maia. Do software categories impact coupling metrics? In *Proceedings of the 10th Working Conference on*

Mining Software Repositories (MSR'13), pages 217–220. IEEE Press, 2013.

[47] Scott Deerwester, Susan T. Dumais, George W. Furnas, Thomas K. Landauer, and Richard Harshman. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41(6):391–407, 1990.

[48] G. Denaro and M. Pezzè. An empirical evaluation of fault-proneness models. In *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on*, pages 241–251, May 2002.

[49] Inderjit S. Dhillon, Yuqiang Guan, and Brian Kulis. Kernel k-means: Spectral clustering and normalized cuts. In *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 551–556, 2004.

[50] Tore Dybå, Dag I.K. Sjøberg, and Daniela S. Cruzes. What works for whom, where, when, and why?: on the role of context in empirical software engineering. In *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement*, ESEM '12, pages 19–28, 2012.

[51] B. Efron and R.J. Tibshirani. *An Introduction to the Bootstrap*. Chapman & Hall/CRC Monographs on Statistics & Applied Probability. Taylor & Francis, 1994.

[52] K. Erni and C. Lewerentz. Applying design-metrics to object-oriented frameworks. In *the 3rd International Symposium on Software Metrics*, pages 64 –74, March 1996.

[53] M.E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 38(2.3):258–287, 1999.

[54] N.E. Fenton and N. Ohlsson. Quantitative analysis of faults and failures in a complex software system. *IEEE Transactions on Software Engineering*, 26(8):797–814,

2000.

[55] Norman E. Fenton and Martin Neil. Software metrics: roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 357–370, New York, NY, USA, 2000. ACM.

[56] Kecia A. M. Ferreira, Mariza A. S. Bigonha, Roberto S. Bigonha, Luiz F. O. Mendes, and Heitor C. Almeida. Identifying thresholds for object-oriented software metrics. *Journal of Systems and Software*, 85(2):244–257, February 2012.

[57] Takafumi Fukushima, Yasutaka Kamei, Shane McIntosh, Kazuhiro Yamashita, and Naoyasu Ubayashi. An empirical study of just-in-time defect prediction using cross-project models. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 172–181, New York, NY, USA, 2014. ACM.

[58] John E. Gaffney. Estimating the number of faults in code. *IEEE Transactions on Software Engineering*, SE-10(4):459–464, July 1984.

[59] Baljinder Ghotra, Shane McIntosh, and Ahmed E. Hassan. Revisiting the impact of classification techniques on the performance of defect prediction models. In *the 37th IEEE International Conference on Software Engineering*, pages 789–800, 2015.

[60] Emanuel Giger, Martin Pinzger, and Harald Gall. Using the gini coefficient for bug prediction in eclipse. In *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th Annual ERCIM Workshop on Software Evolution*, IWPSE-EVOL '11, pages 51–55, New York, NY, USA, 2011. ACM.

[61] Emanuel Giger, Marco D'Ambros, Martin Pinzger, and Harald C. Gall. Method-level bug prediction. In *Proceedings of the ACM-IEEE International Symposium on*

Empirical Software Engineering and Measurement, pages 171–180, 2012.

[62] Corrado Gini. Measurement of inequality of incomes. *The Economic Journal*, 31 (121):124–126, March 1921.

[63] O. Goloshchapova and M. Lumpe. On the application of inequality indices in comparative software analysis. In *Software Engineering Conference (ASWEC), 2013 22nd Australian*, pages 117–126, June 2013.

[64] Iker Gondra. Applying machine learning to software fault-proneness prediction. *Journal of Systems and Software*, 81(2):186–195, February 2008.

[65] Florin Gorunescu. *Data mining concepts, models and techniques*. Springer, Berlin, 2011.

[66] Georgios Gousios, Martin Pinzger, and Arie van Deursen. An exploratory study of the pull-based software development model. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 345–355, 2014.

[67] T.L. Graves, A.F. Karr, J.S. Marron, and H. Siy. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, 26(7):653–661, July 2000.

[68] D. Gray, D. Bowes, N. Davey, Yi Sun, and B. Christianson. The misuse of the nasa metrics data program data sets for automated software defect prediction. In *15th Annual Conference on Evaluation & Assessment in Software Engineering (EASE 2011)*, pages 96–103. IET, April 2011.

[69] Wei Guo. *A Unified Approach to Data Transformation and Outlier Detection using Penalized Assessment*. PhD thesis, University of Cincinnati, Arts and Sciences:

Mathematical Sciences, 2014.

[70] B. Hailpern and P. Santhanam. Software debugging, testing, and verification. *IBM Systems Journal*, 41(1):4–12, 2002.

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

[72] M. Hamill and K. Goseva-Popstojanova. Common trends in software fault and failure data. *IEEE Transactions on Software Engineering*, 35(4):484–496, July 2009.

[73] Jiawei Han, Micheline Kamber, and Jian Pei. *Data Mining: concepts and techniques*. Morgan Kaufmann, Boston, 3 edition, 2012.

[74] R. Harrison, S.J. Counsell, and R.V. Nithi. An evaluation of the mood set of object-oriented software metrics. *IEEE Transactions on Software Engineering*, 24(6):491 –496, June 1998.

[75] Ahmed E. Hassan. Predicting faults using the complexity of code changes. In *Proceedings of the 31st IEEE International Conference on Software Engineering*, ICSE'09, pages 78 –88, 2009.

[76] Zhimin He, Fengdi Shu, Ye Yang, Mingshu Li, and Qing Wang. An investigation on the feasibility of cross-project defect prediction. *Automated Software Engineering*, 19(2):167–199, June 2012.

[77] Zhimin He, F. Peters, T. Menzies, and Ye Yang. Learning from open-source projects: An empirical study on defect prediction. In *Empirical Software Engineering and Measurement, 2013 ACM / IEEE International Symposium on*, pages 45–54, 2013.

[78] Brian Henderson-Sellers. *Object-Oriented Metrics: Measures of Complexity*. Prentice-Hall Object-Oriented Series. Prentice Hall, 1996.

[79] Steffen Herbold, Jens Grabowski, and Stephan Waack. Calculation and optimization of thresholds for sets of software metrics. *Empirical Software Engineering*, 16(6): 812–841, December 2011.

[80] Israel Herraiz, Daniel M. Germán, and Ahmed E. Hassan. On the distribution of source code file sizes. In *Proceedings of the 6th International Conference on Software and Data Technologies*, volume 2, pages 5–14, 2011.

[81] Kim Herzig, Sascha Just, and Andreas Zeller. It's not a bug, it's a feature: how misclassification impacts bug prediction. In *Proceedings of the 35th International Conference on Software Engineering*, ICSE '13, pages 392–401. IEEE Press, 2013.

[82] Tilman Holschuh, Markus Pauser, Kim Herzig, Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. Predicting defects in SAP Java code: An experience report. In *2009 31st International Conference on Software Engineering - Companion Volume*, pages 172–181. IEEE, 2009.

[83] Jr. Hoover, Edgar M. The measurement of industrial localization. *The Review of Economics and Statistics*, 18(4):pp. 162–171, 1936.

[84] David W. Hosmer, Jr., Stanley Lemeshow, and Rodney X. Sturdivant. *Interpretation of the Fitted Logistic Regression Model*, pages 49–88. John Wiley & Sons, Inc., 2013.

[85] James Howison, Megan Conklin, and Kevin Crowston. Flossmole: A collaborative repository for floss research data and analyses. *International Journal of Information*

Technology and Web Engineering, 1:17–26, July 2006.

[86] Chin-Yu Huang, Chih-Song Kuo, and Shao-Pu Luan. Evaluation and Application of Bounded Generalized Pareto Analysis to Fault Distributions in Open Source Software. *IEEE Transactions on Reliability*, 63(1):309–319, March 2014.

[87] ISO/IEC. *ISO/IEC 9126. Software engineering – Product quality*. ISO/IEC, 2001.

[88] ISO/IEC. *ISO/IEC 25010. System and software quality models*. ISO/IEC, 2010.

[89] A Janes, M Scotto, W Pedrycz, B Russo, M Stefanovic, and G Succi. Identification of defect-prone classes in telecommunication software systems using design metrics. *Information Sciences*, 176(24):3711–3734, December 2006.

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

[91] Enio G. Jelihovschi, José Cláudio Faria, and Ivan Bezerra Allaman. Scottknott: A package for performing the scott-knott clustering algorithm in r. *Trends in Applied and Computational Mathematics*, 15(1):3–17, 2014.

[92] Hao Jia, Fengdi Shu, Ye Yang, and Qing Wang. Predicting Fault-Prone Modules: A Comparative Study. *Software Engineering Approaches For Offshore and Outsourced Development*, 35:45–59, 2009.

[93] Yue Jiang, Bojan Cukic, and Tim Menzies. Can data transformation help in the detection of fault-prone modules? In *Proceedings of the 2008 Workshop on Defects in Large Software Systems*, DEFECTS '08, pages 16–20, 2008.

[94] Yue Jiang, Jie Lin, Bojan Cukic, and Tim Menzies. Variance analysis in software fault prediction models. *IEEE ISSRE09*, pages 99–108, 2009.

[95] Kalpana Johari and Arvinder Kaur. Effect of software evolution on software metrics: an open source case study. *SIGSOFT Softw. Eng. Notes*, 36(5):1–8, September 2011.

[96] Marian Jureczko and Lech Madeyski. Towards identifying software project clusters with regard to defect prediction. In *the 6th International Conference on Predictive Models in Software Engineering*, PROMISE '10, pages 9:1–9:10, 2010.

[97] Marian Jureczko and Diomidis D. Spinellis. Using object-oriented design metrics to predict software defects. *Proceedings of the 5th International Conference on Dependability of Computer Systems*, pages 69–81, 2010.

[98] Y. Kamei, S. Matsumoto, A. Monden, K.-i. Matsumoto, B. Adams, and Ahmed E. Hassan. Revisiting common bug prediction findings using effort-aware models. In *26th IEEE International Conference on Software Maintenance*, pages 1–10, 2010.

[99] Yomi Kastro and Ayşe Basar Bener. A defect prediction method for software versioning. *Software Quality Journal*, 16(4):543–562, May 2008.

[100] Gideon Keren and Charles Lewis. *A Handbook for Data Analysis in the Behavioral Sciences: Statistical Issues*. Lawrence Erlbaum, Hillsdale, NY, USA, 1993.

[101] Taghi M. Khoshgoftaar, Pierre Rebours, and Naeem Seliya. Software quality analysis by combining multiple projects and learners. *Software Quality Journal*, 17(1): 25–49, July 2008.

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

[103] B.A. Kitchenham, L.M. Pickard, and S.J. Linkman. An evaluation of some design

metrics. *Software Engineering Journal*, 5(1):50–58, Jan 1990.

[104] Michael Kläs, Frank Elberzhager, Jürgen Münch, Klaus Hartjes, and Olaf von Graevemeyer. Transparent combination of expert and measurement data for defect prediction. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10*, volume 2, page 119, 2010.

[105] Serge-Christophe Kolm. Unequal inequalities. i. *Journal of Economic Theory*, 12 (3):416 – 442, 1976.

[106] A. Güneş Koru and Hongfang Liu. Building Defect Prediction Models in Practice. *IEEE Software*, 22(6):23–29, November 2005.

[107] Segla Kpodjedo, Filippo Ricca, Philippe Galinier, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. Design evolution metrics for defect prediction in object oriented systems. *Empirical Software Engineering*, 16(1):141–175, February 2011.

[108] Max Kuhn and Kjell Johnson. Data pre-processing. In *Applied Predictive Modeling*, pages 27–59. Springer New York, 2013.

[109] D. Landman, A. Serebrenik, and J. Vinju. Empirical analysis of the relationship between cc and sloc in a large corpus of java methods. In *30th IEEE International Conference on Software Maintenance and Evolution*, pages 221–230, 2014.

[110] Michele Lanza, Radu Marinescu, and Stéphane Ducasse. *Object-Oriented Metrics in Practice*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[111] Taek Lee, Jaechang Nam, DongGyun Han, Sunghun Kim, and Hoh Peter In. Micro interaction metrics for defect prediction. In *Proceedings of the 19th ACM SIGSOFT*

Symposium and the 13th European Conference on Foundations of Software Engineering, ESEC/FSE '11, pages 311–321, New York, NY, USA, 2011. ACM.

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

[113] Ming Li, Hongyu Zhang, Rongxin Wu, and Zhi-Hua Zhou. Sample-based software defect prediction with active and semi-supervised learning. *Automated Software Engineering*, 19(2):201–230, June 2012.

[114] Rüdiger Lincke, Jonas Lundberg, and Welf Löwe. Comparing software metrics tools. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 131–142, New York, NY, USA, 2008. ACM.

[115] Mark Lorenz and Jeff Kidd. *Object-oriented software metrics: a practical guide*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.

[116] Panagiotis Louridas, Diomidis Spinellis, and Vasileios Vlachos. Power laws in software. *ACM Transactions on Software Engineering and Methodology*, 18(1):2:1–2:26, October 2008.

[117] Ulrike Luxburg. A tutorial on spectral clustering. *Statistics and Computing*, 17(4): 395–416, December 2007.

[118] Ying Ma, Guangchun Luo, Xue Zeng, and Aiguo Chen. Transfer learning for cross-company software defect prediction. *Information and Software Technology*, 54(3): 248–256, March 2012.

[119] T.J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*

(TSE), SE-2(4):308 – 320, December 1976.

[120] Thilo Mende and Rainer Koschke. Revisiting the evaluation of defect prediction models. In *Proceedings of the 5th International Conference on Predictor Models in Software Engineering*, PROMISE '09, pages 7:1–7:10, 2009.

[121] Thilo Mende and Rainer Koschke. Effort-aware defect prediction models. In *Proceedings of the 2010 14th European Conference on Software Maintenance and Reengineering*, CSMR '10, pages 107–116. IEEE Computer Society, 2010.

[122] T. Menzies, A. Dekhtyar, J. Distefano, and J. Greenwald. Problems with precision: A response to "comments on 'data mining static code attributes to learn defect predictors'". *IEEE Transactions on Software Engineering (TSE)*, 33(9):637–640, 2007.

[123] T. Menzies, J. Greenwald, and A. Frank. Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering (TSE)*, 33(1):2–13, 2007.

[124] Tim Menzies, Andrew Butcher, Andrian Marcus, Thomas Zimmermann, and David Cok. Local vs. global models for effort estimation and defect prediction. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 343–351. IEEE Computer Society, 2011.

[125] A. Mockus. Amassing and indexing a large sample of version control systems: Towards the census of public source code history. In *6th IEEE International Working Conference on Mining Software Repositories*, MSR'09, pages 11 –20, May 2009.

[126] A. Mockus and L.G. Votta. Identifying reasons for software changes using historic

databases. In *Proceedings of the 16th International Conference on Software Maintenance*, ICSM '00, pages 120–130, 2000.

[127] Bojan Mohar. The laplacian spectrum of graphs. In *Graph Theory, Combinatorics, and Applications*, pages 871–898. Wiley, 1991.

[128] R. Moser, W. Pedrycz, and G. Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *30th International Conference on Software Engineering*, pages 181–190. ACM, 2008.

[129] R.E. Mullen and S.S. Gokhale. Software Defect Rediscoveries: A Discrete Lognormal Model. In *16th IEEE International Symposium on Software Reliability Engineering (ISSRE'05)*, pages 203–212. IEEE, 2005.

[130] Meiyappan Nagappan, Thomas Zimmermann, and Christian Bird. Diversity in software engineering research. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 466–476, 2013.

[131] Nachiappan Nagappan and Thomas Ball. Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 284–292, 2005.

[132] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. Mining metrics to predict component failures. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 452–461, New York, NY, USA, 2006. ACM.

[133] Jaechang Nam and Sunghun Kim. Clami: Defect prediction on unlabeled datasets. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, ASE '15, 2015.

[134] Jaechang Nam and Sunghun Kim. Heterogeneous defect prediction. In *Proceedings of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ESEC/FSE '15, 2015.

[135] Jaechang Nam, Sinno Jialin Pan, and Sunghun Kim. Transfer defect learning. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 382–391, Piscataway, NJ, USA, 2013. IEEE Press.

[136] NASA. Metrics Data Program. http://openscience.us/repo/defect/mccabehalsted, 2015. [Online; accessed 25-August-2015].

[137] Andrew Y. Ng, Michael I. Jordan, and Yair Weiss. On spectral clustering: Analysis and an algorithm. In *Advances in Neural Information Processing Systems*, pages 849–856. MIT Press, 2001.

[138] T.H.D. Nguyen, B. Adams, and Ahmed E. Hassan. Studying the impact of dependency network measures on software quality. In *Proceedings of the 26th IEEE International Conference on Software Maintenance*, ICSM '10, pages 1–10, September 2010.

[139] Tung Thanh Nguyen, Tien N. Nguyen, and Tu Minh Phuong. Topic-based defect prediction (nier track). In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 932–935, New York, NY, USA, 2011. ACM.

[140] MC Ohlsson and P Runeson. Experience from replicating empirical studies on prediction models. *EIGHTH IEEE SYMPOSIUM ON SOFTWARE METRICS, PROCEEDINGS*, pages 217–226, 2002.

[141] Jason W. Osborne. Improving your data transformations: Applying the box-cox

transformation. *Practical Assessment, Research & Evaluation*, 15(12), 2010.

[142] Thomas J. Ostrand and Elaine J. Weyuker. The distribution of faults in a large industrial software system. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '02, pages 55–64. ACM, 2002.

[143] T.J. Ostrand and E.J. Weyuker. On the automation of software fault prediction. In *Testing: Academic and Industrial Conference - Practice And Research Techniques, 2006. TAIC PART 2006. Proceedings*, pages 41–48, August 2006.

[144] T.J. Ostrand, E.J. Weyuker, and R.M. Bell. Predicting the location and number of faults in large software systems. *Software Engineering, IEEE Transactions on*, 31 (4):340–355, April 2005.

[145] Sinno Jialin Pan and Qiang Yang. A survey on transfer learning. *IEEE Transactions on Knowledge and Data Engineering*, 22(10):1345–1359, Oct 2010.

[146] Sinno Jialin Pan, I.W. Tsang, J.T. Kwok, and Qiang Yang. Domain adaptation via transfer component analysis. *IEEE Transactions on Neural Networks*, 22(2):199–210, February 2011.

[147] Annibale Panichella, Rocco Oliveto, and Andrea De Lucia. Cross-project defect prediction models: L'Union fait la force. In *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pages 164–173. IEEE, February 2014.

[148] L. Pelayo and S. Dick. Applying novel resampling strategies to software defect prediction. In *Annual Conference of the North American Fuzzy Information processing Society*, NAFIPS '07, pages 69–72, June 2007.

[149] Lourdes Pelayo and Scott Dick. Evaluating Stratification Alternatives to Improve Software Defect Prediction. *IEEE Transactions on Reliability*, 61(2):516–525, June 2012.

[150] F. Peters, T. Menzies, L. Gong, and H. Zhang. Balancing privacy and utility in cross-company defect prediction. *IEEE Transactions on Software Engineering*, 39 (8):1054–1068, August 2013.

[151] Fayola Peters, Tim Menzies, and Andrian Marcus. Better cross company defect prediction. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 409–418, Piscataway, NJ, USA, 2013. IEEE Press.

[152] Martin Pinzger, Nachiappan Nagappan, and Brendan Murphy. Can developer-module networks predict failures? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, pages 2–12, New York, NY, USA, 2008. ACM.

[153] Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. Ecological inference in empirical software engineering. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 362–371, Washington, DC, USA, 2011. IEEE Computer Society.

[154] R. Premraj and K. Herzig. Network versus code metrics to predict defects: A replication study. In *2011 International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 215–224, 2011.

[155] Marie-Therese Puth, Markus Neuhäuser, and Graeme D. Ruxton. Effective use of spearman's and kendall's correlation coefficients for association between two measured traits. *Animal Behaviour*, 102(0):77 – 84, 2015.

[156] 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, 2013.

[157] Foyzur Rahman, Daryl Posnett, and Premkumar Devanbu. Recalling the "imprecision" of cross-project defect prediction. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 61:1–61:11, New York, NY, USA, 2012. ACM.

[158] Foyzur Rahman, Daryl Posnett, Israel Herraiz, and Premkumar Devanbu. Sample size vs. bias in defect prediction. In *Proceedings of the 21th ACM SIGSOFT Symposium and the 15th European Conference on Foundations of Software Engineering*, ESEC/FSE '13, New York, New York, USA, 2013. ACM.

[159] Rakesh Rana, Miroslaw Staron, Christian Berger, Jorgen Hansson, Martin Nilsson, and Wilhelm Meding. The Adoption of Machine Learning Techniques for Software Defect Prediction: An Initial Industrial Validation. *KNOWLEDGE-BASED SOFTWARE ENGINEERING, JCKBSE 2014*, 466:270–285, 2014.

[160] Jeanine Romano, Jeffrey D. Kromrey, Jesse Coraggio, and Jeff Skowronek. Appropriate statistics for ordinal level data: Should we really be using t-test and cohen's d for evaluating group differences on the nsse and other surveys? In *Annual Meeting of the Florida Association of Institutional Research*, pages 1–33, February 2006.

[161] Laura Sánchez-González, Félix García, Francisco Ruiz, and Jan Mendling. A study of the effectiveness of two threshold definition techniques. In *Evaluation Assessment in Software Engineering (EASE 2012), 16th International Conference on*, pages 197–205, May 2012.

[162] F. Sarro, S. Di Martino, F. Ferrucci, and C. Gravino. A further analysis on the use of genetic algorithm to configure support vector machines for inter-release fault prediction. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 1215–1220, New York, NY, USA, 2012. ACM.

[163] Scitools. Metrics computed by understand, 2015. `http://www.scitools.com/documents/metricsList.php`.

[164] SciTools. Understand 3.1 build 726. `https://scitools.com`, 2015. [Online; accessed 15-June-2015].

[165] G.M.K. Selim, L. Barbour, Weiyi Shang, B. Adams, Ahmed E. Hassan, and Ying Zou. Studying the impact of clones on software defects. In *Proceeddings of the 17th Working Conference on Reverse Engineering*, pages 13–21, Oct 2010.

[166] Naeem Seliya and Taghi M. Khoshgoftaar. Software quality estimation with limited fault data: a semi-supervised learning perspective. *Software Quality Journal*, 15(3): 327–344, August 2007.

[167] A. Serebrenik and M. Van Den Brand. Theil index for aggregation of software metrics values. In *Proceedings of the 26th IEEE International Conference on Software Maintenance*, pages 1–9, September 2010.

[168] HanLin Shang. Selection of the optimal box-cox transformation parameter for modelling and forecasting age-specific fertility. *Journal of Population Research*, pages 1–11, 2014.

[169] Claude Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27:379–423, 623–656, 1948.

[170] R. Shatnawi. A quantitative investigation of the acceptable risk levels of object-oriented metrics in open-source systems. *IEEE Transactions on Software Engineering*, 36(2):216 –225, March 2010.

[171] Raed Shatnawi and Wei Li. The effectiveness of software metrics in identifying error-prone classes in post-release software evolution process. *Journal of Systems and Software*, 81(11):1868–1882, November 2008.

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

[173] David J. Sheskin. *Handbook of Parametric and Nonparametric Statistical Procedures, Fourth Edition*. Chapman & Hall/CRC, January 2007.

[174] Jianbo Shi and Jitendra Malik. Normalized cuts and image segmentation. *IEEE Trans. Pattern Anal. Mach. Intell.*, 22(8):888–905, August 2000.

[175] E. Shihab. Practical software quality prediction. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 639–644, Sept 2014.

[176] Emad Shihab, Zhen Ming Jiang, Walid M. Ibrahim, Bram Adams, and Ahmed E. Hassan. Understanding the impact of code and process metrics on post-release defects: a case study on the eclipse project. In *Proceedings of the 2010 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '10, pages 4:1–4:10, New York, NY, USA, 2010. ACM.

[177] F. Shull, V. Basili, B. Boehm, A.W. Brown, P. Costa, M. Lindvall, D. Port, I. Rus, R. Tesoriero, and M. Zelkowitz. What we have learned about fighting defects. In

Proceedings of the 8th IEEE Symposium on Software Metrics, pages 249–258, 2002.

[178] Susan Elliott Sim, Steve Easterbrook, and Richard C. Holt. Using benchmarking to advance research: a challenge to software engineering. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 74–83, Washington, DC, USA, 2003. IEEE Computer Society.

[179] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? In *Proceedings of the 2nd International Workshop on Mining Software Repositories*, MSR '05, pages 1–5, New York, NY, USA, 2005. ACM.

[180] Qinbao Song, Zihan Jia, M. Shepperd, Shi Ying, and Jin Liu. A general software defect-proneness prediction framework. *IEEE Transactions on Software Engineering*, 37(3):356–370, May 2011.

[181] R. Subramanyam and M.S. Krishnan. Empirical analysis of ck metrics for object-oriented design complexity: implications for software defects. *IEEE Transactions on Software Engineering*, 29(4):297–310, April 2003.

[182] Giancarlo Succi, Witold Pedrycz, Snezana Djokic, Paolo Zuliani, and Barbara Russo. An empirical exploration of the distributions of the chidamber and kemerer object-oriented metrics suite. *Empirical Software Engineering*, 10(1):81–104, 2005.

[183] T. Systa and H. Muller. Predicting fault-proneness using OO metrics. An industrial case study. In *Proceedings of the Sixth European Conference on Software Maintenance and Reengineering*, pages 99–107. IEEE Comput. Soc, 2002.

[184] G. Tassey. The economic impacts of inadequate infrastructure for software testing.

Technical Report Planning Report 02-3, National Institute of Standards and Technology, May 2002.

[185] Henri Theil. *Economics and Information Theory*. North-Holland Pub. Co., Amsterdam, 1967.

[186] Piotr Tomaszewski, Jim Håkansson, Håkan Grahn, and Lars Lundberg. Statistical models vs. expert estimation for fault prediction in modified code - an industrial case study. *Journal of Systems and Software*, 80(8):1227–1238, August 2007.

[187] Ayşe Tosun, Ayşe Bener, Burak Turhan, and Tim Menzies. Practical considerations in deploying statistical methods for defect prediction: A case study within the Turkish telecommunications industry. *Information and Software Technology*, 52(11): 1242–1257, November 2010.

[188] 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, October 2009.

[189] C. van Koten and A.R. Gray. An application of bayesian network for predicting object-oriented software maintainability. *Information and Software Technology*, 48 (1):59 – 67, 2006.

[190] R. Vasa, M. Lumpe, P. Branch, and O. Nierstrasz. Comparative analysis of evolving software systems using the gini coefficient. In *Proceedings of the 25th IEEE International Conference on Software Maintenance*, pages 179–188, September 2009.

[191] Bogdan Vasilescu. Analysis of advanced aggregation techniques for software metrics. Master's thesis, Eindhoven University of Technology, 2011.

[192] Bogdan Vasilescu, Alexander Serebrenik, and Mark Van den Brand. By no means: A study on aggregating software metrics. In *Proceedings of the 2Nd International Workshop on Emerging Trends in Software Metrics*, WETSoM '11, pages 23–26, New York, NY, USA, 2011. ACM.

[193] Shuo Wang and Xin Yao. Using Class Imbalance Learning for Software Defect Prediction. *IEEE Transactions on Reliability*, 62(2):434–443, 2013.

[194] Andrew R. Webb and Keith D. Copsey. *Statistical Pattern Recognition, Third Edition*. John Wiley & Sons, Inc., 2011.

[195] Elaine J. Weyuker, Thomas J. Ostrand, and Robert M. Bell. Comparing the effectiveness of several modeling methods for fault prediction. *Empirical Software Engineering*, 15(3):277–295, June 2009.

[196] Rongxin Wu, Hongyu Zhang, Sunghun Kim, and Shing-Chi Cheung. Relink: recovering links between bugs and changes. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 15–25, New York, NY, USA, 2011. ACM.

[197] Harikesh Bahadur Yadav and Dilip Kumar Yadav. A fuzzy logic based approach for phase-wise software defects prediction using software metrics. *Information and Software Technology*, 63:44–57, July 2015.

[198] Bingbing Yang, Qian Yin, Shengyong Xu, and Ping Guo. Software quality prediction using affinity propagation algorithm. In *Neural Networks, 2008. IJCNN 2008. (IEEE World Congress on Computational Intelligence). IEEE International Joint Conference on*, pages 1891–1896, June 2008.

[199] Robert K. Yin. *Case Study Research: Design and Methods - Third Edition*. SAGE Publications, 3 edition, 2002.

[200] Kyung-A Yoon, Oh-Sung Kwon, and Doo-Hwan Bae. An approach to outlier detection of software measurement data using the k-means clustering method. In *Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on*, pages 443 –445, September 2007.

[201] Feng Zhang, Audris Mockus, Ying Zou, Foutse Khomh, and Ahmed E. Hassan. How does context affect the distribution of software maintainability metrics? In *Proceedings of the 29th IEEE International Conference on Software Maintainability*, ICSM '13, pages 350 – 359, 2013.

[202] Feng Zhang, Audris Mockus, Iman Keivanloo, and Ying Zou. Towards building a universal defect prediction model. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR '14, pages 41–50, Piscataway, NJ, USA, 2014. IEEE Press.

[203] Feng Zhang, Audris Mockus, Iman Keivanloo, and Ying Zou. Towards building a universal defect prediction model with rank transformed predictors. *Empirical Software Engineering*, pages 1–39, 2015.

[204] Feng Zhang, Quan Zheng, Ying Zou, and Ahmed E. Hassan. Defect prediction without training data via spectral clustering. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages –, may. 2016.

[205] Hongyu Zhang. Discovering power laws in computer programs. *Information Processing & Management*, 45(4):477–483, July 2009.

[206] Shi Zhong, T.M. Khoshgoftaar, and N. Seliya. Unsupervised learning for expert-based software quality estimation. In *Proceedings of the 8th IEEE International Symposium on High Assurance Systems Engineering.*, pages 149–155, March 2004.

[207] Yuming Zhou and Hareton Leung. Predicting object-oriented software maintainability using multivariate adaptive regression splines. *Journal of Systems and Software*, 80(8):1349 – 1361, 2007.

[208] T. Zimmermann, R. Premraj, and A. Zeller. Predicting defects for eclipse. In *Proceedings of the International Workshop on Predictor Models in Software Engineering*, PROMISE '07, page 9, May 2007.

[209] Thomas Zimmermann and Nachiappan Nagappan. Predicting defects using network analysis on dependency graphs. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 531–540, 2008.

[210] Thomas Zimmermann, Nachiappan Nagappan, Harald Gall, Emanuel Giger, and Brendan Murphy. Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC/FSE '09, pages 91–100, 2009.

[211] Thomas Zimmermann, Nachiappan Nagappan, Philip J. Guo, and Brendan Murphy. Characterizing and predicting which bugs get reopened. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 1074 –1083, June 2012.

[212] Ying Zou and K. Kontogiannis. Migration to object oriented platforms: a state transformation approach. In *Proceedings of the 18th International Conference on Software Maintenance*, ICSM '02, pages 530 – 539, 2002.

Additional Analysis

A.1 R Implementation of Our Spectral Classifier

Listing 1 presents the R implementation of our connectivity-based unsupervised spectral classifier

Listing A.1: R implementation of our approach.

```
connectivity_based_defect_prediction <- function(A) {
  # Normalize software metrics.
  normA = apply(A, 2, function(x){(x-mean(x))/sd(x)})
  # Construct the weighted adjacency matrix.
  W = normA %*% t(normA)
  # Set all negative values to zero.
  W[W<0] = 0
  # Set the self-similarity to zero.
  W = W - diag(diag(W))
  # Construct the graph Laplacian matrix.
  Dnsqrt = diag(1/sqrt(rowSums(W)))
  I = diag(rep(1, nrow(W)))
  Lsym = I - Dnsqrt %*% W %*% Dnsqrt
  # Perform the eigendecomposition.
  ret_egn = eigen(Lsym, symmetric=TRUE)
  # Pick up the second smallest eigenvector.
  v1 = Dnsqrt %*% ret_egn$vectors[, nrow(W)-1]
  v1 = v1 / sqrt(sum(v1^2))
  # Divide the data set into two clusters.
  defect_proneness = (v1>0)
  # Label the defective and clean clusters.
  rs = rowSums(normA)
  if(mean(rs[v1>0])<mean(rs[v1<0]))
      defect_proneness = (v1<0)
  # Return the defect proneness.
  defect_proneness
}
```

A.2 Matrices in Spectra Clustering

1) Affiliation matrix A**.** In the context of defect prediction, an affiliation matrix A describes the properties of all software entities (e.g., files or classes) in a software project. Each software entity is represented using a row of the matrix, and its properties (e.g., lines of code and other software metrics) are described in columns. If there are n entities in a software project and each entity is measured using m metrics, then A is an $n \times m$ matrix. We denote $A = \{a_{ij}\}$, where a_{ij} is the value of metric j of software entity i.

2) Adjacency matrix W**.** In the context of defect prediction, an adjacency matrix W describes the network among software entities in a software project. Each software entity is a node, and edges represent the relations between nodes. The weight of an edge measures the similarity between the corresponding pair of nodes. If there are n entities in a software project, then W is an $n \times n$ matrix. We denote $W = \{w_{ij}\}$, where w_{ij} is the weight of edge between nodes i and j, and w_{ij} measures the similarity between the two nodes. As edges have no direction, W is symmetric. An adjacency matrix W can be easily obtained from a affiliation matrix A through a function that computes the similarity between nodes.

3) Graph Laplacian matrix L**.** Performing an eigendecomposition on a graph Laplacian matrix L is the major step of spectral clustering. There is no exact definition of "graph Laplacian matrix" in the literature [117]. A basic form is $L = D - W$ [35], where D is a diagonal matrix of row sums of W. The matrix L is unnormalized. In the normalized cut algorithm [174], a symmetric graph Laplacian matrix L_{sym} is used to achieve a better and more robust performance. It is defined as $L_{sym} = D^{-\frac{1}{2}}LD^{-\frac{1}{2}} = I - D^{-\frac{1}{2}}WD^{-\frac{1}{2}}$, where I is the unit matrix with size n. The diagonal matrix $D^{-\frac{1}{2}} = Diag(d_1^{-\frac{1}{2}}, \ldots, d_n^{-\frac{1}{2}})$, where $d_i^{-\frac{1}{2}} = (\sum_{j=1}^{n} w_{ij})^{-\frac{1}{2}}$.