NAIST-IS-DD1461019

Doctoral Dissertation

Towards a Better Understanding of the Impact of Experimental Components on Defect Prediction Models

Chakkrit Tantithamthavorn

September 5, 2016

Graduate School of Information Science
Nara Institute of Science and Technology

A Doctoral Dissertation
submitted to Graduate School of Information Science,
Nara Institute of Science and Technology
in partial fulfillment of the requirements for the degree of
Doctor of ENGINEERING

Chakkrit Tantithamthavorn

Thesis Committee:

Professor Kenichi Matsumoto	(Nara Institute of Science and Technology)
Professor Hajimu Iida	(Nara Institute of Science and Technology)
Assistant Professor Akinori Ihara	(Nara Institute of Science and Technology)
Professor Ahmed E. Hassan	(Queen's University)
Dr. Thomas Zimmermann	(Microsoft Research)

Towards a Better Understanding of the Impact of Experimental Components on Defect Prediction Models*

Chakkrit Tantithamthavorn

Abstract

Software Quality Assurance (SQA) teams play a critical role in the software development process to ensure the absence of software defects. It is not feasible to perform exhaustive SQA tasks (i.e., software testing and code review) on a large software product given the limited SQA resources that are available. Thus, the prioritization of SQA efforts is an essential step in all SQA efforts.

Defect prediction models are used to prioritize risky software modules and understand the impact of software metrics on the defect-proneness of software modules. The predictions and insights that are derived from defect prediction models can help software teams allocate their limited SQA resources to the modules that are most likely to be defective and avoid common past pitfalls that are associated with the defective modules of the past. However, the predictions and insights that are derived from defect prediction models may be inaccurate and unreliable if practitioners do not control for the impact of experimental components (e.g., datasets, metrics, and classifiers) on defect prediction models, which could lead to erroneous decision-making in practice.

*Doctoral Dissertation, Graduate School of Information Science,
Nara Institute of Science and Technology, NAIST-IS-DD1461019, September 5, 2016.

In this thesis, we investigate the impact of experimental components on the performance and interpretation of defect prediction models. More specifically, we investigate the impact of the three often overlooked experimental components (i.e., issue report mislabelling, parameter optimization of classification techniques, and model validation techniques) have on defect prediction models. Through case studies of systems that span both proprietary and open-source domains, we demonstrate that (1) issue report mislabelling does not impact the precision of defect prediction models, suggesting that researchers can rely on the predictions of defect prediction models that were trained using noisy defect datasets; (2) automated parameter optimization for classification techniques substantially improve the performance and stability of defect prediction models, as well as they change their interpretation, suggesting that researchers should no longer shy from applying parameter optimization to their models; and (3) the out-of-sample bootstrap validation technique produces a good balance between bias and variance of performance estimates, suggesting that the single holdout and cross-validation families that are commonly-used nowadays should be avoided.

Keywords:

Software Quality Assurance, Defect Prediction Modelling

Related Publications

Early versions of the work in this thesis were published as listed below.

- **(Chapter 1 and Chapter 4) Towards a Better Understanding of the Impact of Experimental Components on Defect Prediction Modelling**.
  Chakkrit Tantithamthavorn. In the Doctoral Symposium of the International Conference on Software Engineering (ICSE 2016), pp. 867–870.
  Acceptance Rate: 22% (8/36).

- **(Chapter 3) Comments on "Researcher Bias: The Use of Machine Learning in Software Defect Prediction"**.
  Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E. Hassan, and Kenichi Matsumoto. IEEE Transactions on Software Engineering (TSE), 4 pages, In Press. Impact Factor: 1.6 (2015).

- **(Chapter 5) The Impact of Mislabelling on the Performance and Interpretation of Defect Prediction Models**.
  Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E. Hassan, Akinori Ihara, and Kenichi Matsumoto. In Proceedings of the International Conference on Software Engineering (ICSE 2015), pp. 812–823.
  Acceptance Rate: 19% (89/455).
  This paper received an Outstanding Paper Award for Young C&C Researchers by NEC C&C, Japan.

- **(Chapter 6) Automated Parameter Optimization of Classification Techniques for Defect Prediction Models**.
  Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E. Hassan, and Kenichi Matsumoto. In Proceedings of the International Conference on Software Engineering (ICSE 2016), pp. 321–332. Acceptance Rate: 19% (101/530).

- **(Chapter 7) An Empirical Comparison of Model Validation Techniques for Defect Prediction Models**.
  Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E. Hassan, and Kenichi Matsumoto. IEEE Transactions on Software Engineering (TSE), 21 pages, In Press. Impact Factor: 1.6 (2015).

The following publications are not directly related to the material in this thesis, but were produced in parallel to the research performed for this thesis.

- **Who Should Review My Code? A File Location-Based Code-Reviewer Recommendation Approach for Modern Code Review**.
  Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Raula Gaikovina Kula, Norihiro Yoshida, Hajimu Iida, Kenichi Matsumoto. In Proceedings of The International Conference on Software Analysis, Evolution, and Reengineering (SANER 2015), pp 141–150. Acceptance Rate: 32% (46/144).

- **A Study of Redundant Metrics in Defect Prediction Datasets**.
  Jirayus Jiarpakdee, Chakkrit Tantithamthavorn, Akinori Ihara, Kenichi Matsumoto. In Proceedings of the International Symposium on Software Reliability Engineering (ISSRE 2016), To Appear.

Acknowledgements

This thesis would not have been possible without support and encouragement from many important people. To the following people, I owe an enormous debt of gratitude.

First and foremost, there are no proper words to convey my deep gratitude and respect for my thesis supervisor, Kenichi Matsumoto, for his supervision and continuous support throughout my Ph.D. journey.

I have been amazingly fortunate to work closely with and mentored by Ahmed E. Hassan and Shane McIntosh. Their technical and editorial advice was essential to the completion of this dissertation. They have taught me innumerable lessons and insights on the workings of academic research in general. They also demonstrated what a brilliant and hard-working researcher can accomplish. I am also thankful for their feedback and countless revisions of this dissertation. I could not have imagined having better mentors for my Ph.D study. I hope that one day I would become as good an advisor to my students as they have been to me.

Besides my supervisor and mentors, I would like to thank the rest of my thesis committee Thomas Zimmermann, Hajimu Iida, and Akinori Ihara, for their encouragement, insightful comments, and hard questions. I am particularly grateful to Yasutaka Kamei, Meiyappan Nagappan, Weiyi Shang and Hideaki Hata, who have been like older brothers to me, offering advice and sharing their experiences.

I am very fortunate to work with the brightest network of collaborators on topics outside of the scope of this thesis: Surafel Lemma Abebe, Norihiro Yoshida,

Raula G. Kula, Safwat Ibrahim Hassan, Ranjan Kumar Rahul, Cor-Paul Bazemer, Jirayus Jiarpakdee, Shade Ruangwan, Nonthaphat Wongwattanakij, and Phuwit Vititayanon.

I would like to thank Frank E. Harrell Jr. for fruitful discussion on the state of the art and practice of model validation techniques, as well as performance measures. He helped me sort out the technical details of my work during the Regression Modelling Strategies workshop 2015 at the School of Medicine, Vanderbilt University, USA.

I wish to thank my labmates both at NAIST and Queen's University. Also, Daniel Alenca da Costa and Hanyang Hu for sharing their advice and sharing their experiences.

I also thank Compute Canada for providing me access to the High Performance Computing (HPC) systems and the Centre for Advanced Computing at Queen's University.

My Ph.D. journey is generously supported by NAIST Excellent Student Scholarship Award 2014, NEC C&C Research Grant for Non-Japanese Researchers 2014, the JSPS Research Fellowship for Young Scientist (DC2), the Grant-in-Aid for JSPS Fellows (No. 16J03360), and the Natural Sciences and Engineering Research Council of Canada (NSERC).

Special thanks to Patanamon Thongtanunam. I feel so lucky and thankful that we were able to share another journey together as we pursued our Master's and Ph.D. degrees. Thank you for the patient, understanding, and humorous jokes that always make me forget my tiredness. You also made my days bright (even the really tough ones) and filled every second of my time with happiness and joy.

Finally and most importantly, I would like to thank my parents for their never ending encouragement and support. Their love is the driving force that has pushed me this far. Thank you for allowing me to chase my dreams and be as ambitious as I wanted.

Dedication

I dedicate this thesis to my beloved parents Chan and Mayuree. I hope that this achievement will complete the dream that you had for me all those many years when you chose to give me the best education you could.

Contents

Abstract **i**

Related Publications **iii**

Acknowledgements **v**

Dedication **vii**

List of Figures **xiii**

List of Tables **xix**

1. Introduction **1**

1.1. Problem Statement 6
1.2. Thesis Overview 6
1.3. Thesis Contribution 10
1.4. Thesis Organization 11

I. Background and Definitions **13**

2. Defect Prediction Modelling **15**

2.1. Introduction 15

2.2. Data Preparation 16
2.3. Model Construction 23
2.4. Model Validation 24
2.5. Chapter Summary 27

II. A Motivating Study and Related Research 29

3. The Experimental Components that Impact Defect Prediction Models 31
3.1. Introduction 32
3.2. The Presence of Collinearity 32
3.3. The Interference of Collinearity 34
3.4. Mitigating Collinearity 36
3.5. Chapter Summary 38

4. Related Research 41
4.1. Introduction 42
4.2. Data Preparation Stage 42
4.3. Model Construction Stage 45
4.4. Model Validation Stage 46
4.5. Chapter Summary 47

III. Empirical Investigations of the Impact of Experimental Components 49

5. The Impact of Issue Report Mislabelling 51
5.1. Introduction 52
5.2. Related Work & Research Questions 54
5.3. Case Study Design 59
5.4. Case Study Results 68

5.5. Discussion & Threats to Validity 78
5.6. Chapter Summary 80

6. The Impact of Automated Parameter Optimization 83
6.1. Introduction 84
6.2. The Relevance of Parameter Settings for Defect Prediction Models 87
6.3. Related Work & Research Questions 96
6.4. Case Study Design 101
6.5. Case Study Results 110
6.6. Revisiting the Ranking of Classification Techniques for Defect Prediction Models 125
6.7. Discussion 130
6.8. Threats to Validity 133
6.9. Chapter Summary 134

7. The Impact of Model Validation Techniques 137
7.1. Introduction 138
7.2. Motivating Examples 141
7.3. Model Validation Techniques in Defect Prediction Literature . . . 149
7.4. Related Work & Research Questions 154
7.5. Case Study Design 156
7.6. Case Study Results 167
7.7. Discussion 180
7.8. Practical Guidelines 183
7.9. Threats to Validity 184
7.10. Chapter Summary 187

IV. Conclusion and Future Work 191

8. Conclusion and Future Work 193

8.1. Contributions and Findings . . . 194

8.2. Opportunities for Future Research . . . 197

References 203

Appendix A. Replication Package for Chapter 3 237

A.1. Download dataset and scripts . . . 237

A.2. Load miscellaneous functions . . . 237

A.3. The Presence of Collinearity . . . 237

A.4. The Interference of Collinearity . . . 238

A.5. Mitigating the Collinearity . . . 241

Appendix B. A List of 101 Defect Datasets. 247

B.1. Install and load necessary R packages . . . 247

B.2. A list of 101 datasets . . . 247

Appendix C. An example R script for Caret parameter optimization 253

C.1. Install necessary R packages . . . 253

C.2. Caret parameter optimization . . . 253

C.3. Results . . . 256

Appendix D. An R implementation of the generic variable importance computation 257

D.1. Install necessary R packages . . . 257

Appendix E. An R Implementation of the Scott-Knott ESD Test 261

Appendix F. An example R script for out-of-sample bootstrap validation 263

F.1. Install necessary R packages . . . 263

List of Figures

1.1. An overview of the typical defect prediction modelling and its related experimental components. 4

1.2. An overview of the scope of this thesis. 7

2.1. A graphical illustration of the defect prediction modelling process. 18

2.2. A graphical illustration of data preparation stage. 21

2.3. An example issue report that describes software defect. A red box is used to highlight the category of the issue report, while a blue box is used to highlight the modules that are changed to address the issue report. 22

2.4. A Git commit that is related to the given example issue report of Figure 2.3. 22

3.1. [A Motivating Study] Distribution of the partial η^2 values for each explanatory variable when it appears at each position in the model formula. 35

5.1. [Empirical Study 1] The construction of defect prediction datasets. 55

5.2. [Empirical Study 1] An overview of our data extraction and analysis approaches. 58

5.3. [Empirical Study 1] A comparison of the performance of our models that are trained to identify mislabelled issue reports (blue) against random guessing (white). Error bars indicate the 95% confidence interval based on 1,000 bootstrap iterations. 70

5.4. [Empirical Study 1] The difference in performance between models trained using realistic noisy samples and clean samples. All models are tested on clean samples (defect mislabelling). 72

5.5. [Empirical Study 1] The difference in performance between models trained using random noisy and realistic noisy samples. All models are tested on clean samples (defect mislabelling). 74

5.6. [Empirical Study 1] An overview of our approach to study the impact of issue report mislabelling. 76

5.7. [Empirical Study 1] The difference in the ranks for the metrics according to their variable importance scores among the clean and noisy models. The bars indicate the percentage of variables that appear in that rank in the clean model while also appearing in that rank in the noisy models. 77

6.1. [Empirical Study 2] An overview of our case study approach for studying the impact of automated parameter optimization on defect prediction models. 104

6.2. [Empirical Study 2] An overview of Caret parameter optimization. 106

6.3. [Empirical Study 2] An overview of our generic variable importance calculation that can be applied to any classification techniques. . . 109

6.4. [Empirical Study 2] The performance improvement and its Cohen's d effect size for each of the studied classification techniques. . . . 112

6.5. [Empirical Study 2] The AUC performance difference of the top-20 most sensitive parameters. 113

6.6. [Empirical Study 2] The stability ratio of the classifiers that are trained using Caret-optimized settings compared to the classifiers that are trained using default settings for each of the studied classification techniques. 116

6.7. [Empirical Study 2] The stability ratio of the top-20 most sensitive parameters. 117

6.8. [Empirical Study 2] The difference in the ranks for the variables according to their variable importance scores among the classifiers that are trained using Caret-optimized settings and classifiers that are trained using default settings. The bars indicate the percentage of variables that appear in that rank in the Caret-optimized model while also appearing in that rank in the default models. 119

6.9. [Empirical Study 2] Four types of transferability for each of the top-20 most sensitive parameters. Higher frequency for each of the Caret-suggested settings that appear across datasets indicates high transferability of a parameter. 121

6.10. [Empirical Study 2] Computational cost of Caret optimization techniques (hours). 125

6.11. [Empirical Study 2] An overview of our statistical comparison over multiple datasets. 127

6.12. [Empirical Study 2] The likelihood of each technique appearing in the top Scott-Knott ESD rank. Circle dots and triangle dots indicate the median likelihood, while the error bars indicate the 95% confidence interval of the likelihood of the bootstrap analysis. A likelihood of 80% indicates that a classification technique appears at the top-rank for 80% of the studied datasets. 128

7.1. [Empirical Study 3] The distribution of Events Per Variable (EPV) values in publicly-available defect prediction datasets. The black line indicates the median value. The vertical red line indicates the rule-of-thumb EPV value of 10 that is recommended by Peduzzi *et al.* [186]. The red-shaded area indicates the high-risk datasets that do not meet this recommendation (EPV $\leq$ 10). . . . 142

7.2. [Empirical Study 3] The distribution of performance estimates that are produced by the repeated 10-fold cross validation at different EPV contexts when the experiment is repeated 100 times. 145

7.3. [Empirical Study 3] An overview of the design of our case study experiment. 159

7.4. [Empirical Study 3] An overview of our ranking and clustering approach. 166

7.5. [Empirical Study 3] The Scott-Knott ESD ranking of the *bias* of model validation techniques. The technique in a bracket indicates the top-performing technique for each classifier type. The red diamond indicates the average amount of bias across our studied datasets. 170

7.5. [Empirical Study 3] The Scott-Knott ESD ranking of the *bias* of model validation techniques. The technique in a bracket indicates the top-performing technique for each classifier type. The red diamond indicates the average amount of bias across our studied datasets. (Cont.) 171

7.5. [Empirical Study 3] The Scott-Knott ESD ranking of the *bias* of model validation techniques. The technique in a bracket indicates the top-performing technique for each classifier type. The red diamond indicates the average amount of bias across our studied datasets. (Cont.) 172

7.6. [Empirical Study 3] The Scott-Knott ESD ranking of the *variance* of model validation techniques. The technique in a bracket indicates the top-performing technique for each classifier type. The red diamond indicates the average amount of variance across our studied datasets. 176
7.6. [Empirical Study 3] The Scott-Knott ESD ranking of the *variance* of model validation techniques. The technique in a bracket indicates the top-performing technique for each classifier type. The red diamond indicates the average amount of variance across our studied datasets. (Cont.) 177
7.6. [Empirical Study 3] The Scott-Knott ESD ranking of the *variance* of model validation techniques. The technique in a bracket indicates the top-performing technique for each classifier type. The red diamond indicates the average amount of variance across our studied datasets. (Cont.) 178
7.7. [Empirical Study 3] A scatter plot of the mean Scott-Knott ESD ranks in terms of bias and variance among 5 performance metrics, 3 studied classifiers, and 18 studied systems for the high-risk EPV context (EPV= 3) when using two different types of unseen data, i.e., Figure 7.7a uses the observations from the input dataset that do not appear in the sample dataset; and Figure 7.7b uses the next software release. The techniques that appear in the upper-right corner are top-performers. 181

List of Tables

2.1. Confusion matrix for predicting defect-prone modules. 25

3.1. [A Motivating Study] The association among explanatory variables. 33

3.2. [A Motivating Study] Partial η^2 values of the multi-way ANOVA analysis with respect to the Eclipse dataset family. 37

5.1. [Empirical Study 1] An overview of the studied systems. Those above the double line satisfy our criteria for analysis. 58

5.2. [Empirical Study 1] Factors used to study the nature of mislabelled issue reports (RQ1). 61

5.3. [Empirical Study 1] The factors that we use to build our defect models (RQ2, RQ3). 63

5.4. [Empirical Study 1] Example confusion matrices. 67

6.1. [Empirical Study 2] Overview of studied parameters of classification techniques. [N] denotes a numeric value; [L] denotes a logical value; [F] denotes a factor value. The default values are shown in bold typeface and correspond to the default values of the Caret R package. 88

6.2. [Empirical Study 2] An overview of the studied systems. 100

7.1. [Empirical Study 3] Summary of model validation techniques. . . 146

7.2. [Empirical Study 3] An overview of the studied systems. 158

Listings

B.1. A List of 101 Defect Datasets. 251

C.1. An example R script for grid-search parameter optimization. . . . 253

D.1. An R implementation of the generic variable importance function. 257
D.2. An example usage of the generic variable importance function. . . 258

E.1. An R implementation of the Scott-Knott ESD test. 261

F.1. An example R script for out-of-sample bootstrap validation. . . . 263

CHAPTER 1

Introduction

Key Concept

Defect prediction models play a critical role in the prioritization of software quality assurance effort. Yet, the accuracy and reliability of such predictions and associated insights have never been explored in depth.

An earlier version of the work in this chapter appears in the Doctoral Symposium of the International Conference on Software Engineering (ICSE 2016) [229].

Software Quality Assurance (SQA) teams play a critical role in the software development process to ensure the absence of software defects. Therefore, several modern software companies (like Amazon [4], Facebook [2], Mozilla [1], Blackberry [211]) often have a dedicated SQA department. There are various SQA tasks that software engineers must perform. First and foremost, they design, implement, and execute tests to verify and validate that software systems satisfy their functional and non-functional requirements, as well as meet user expectations to ensure that systems are of sufficient quality before their release to customers [38, 120]. Furthermore, they also review designs, look closely at code quality and risk, and refactoring code to make it more testable [19].

Prior work raises several concerns that SQA activities are expensive and time-consuming [8, 235]. For example, Alberts *et al.* [8] point out that SQA activities require almost 50% of the software development resources. Thus, it is unlikely feasible to exhaustively test and review such a large software product given the limited SQA resources (in terms of team size and time). For example, Facebook allocates about 3 months to test a new product [3]. A case study of Mozilla project by Mantyla *et al.* [142] shows that the adoption of rapid release software development substantially increase the workload of software testers. Therefore, inadequate software testing can ultimately result in software defects that cost billions dollars [235].

Defect prediction models play a critical role in the prioritization of SQA effort. Defect prediction models are trained using historical data to identify defect-prone software modules. From an SQA perspective, defect prediction models serve two main purposes. First, defect prediction models can be used to predict modules that are likely to be defect-prone in the future [7, 50, 90, 119, 163, 167, 175, 195, 258]. SQA teams can use defect prediction models in a prediction setting to effectively allocate their limited resources to the modules that are most likely to be defective. Second, defect models can be used to understand the impact of various software metrics on the defect-proneness of a module [42, 145, 161, 163, 212, 214]. The

insights that software teams derive from defect prediction models can help them avoid past pitfalls that are associated with the defective modules of the past.

In the last decade, there has been a spike in the adoption of defect prediction models in practice. Some recent examples of adopters of defect prediction include (but are not limited to): Bell Labs [163], AT&T [182], Turkish Telecommunication [256], Microsoft Research [171,172,175,255,257], Google [134], Blackberry [211], Cisco [227], IBM [39], and Sony Mobile [215]. Such companies often report their successful adoption of defect prediction models and their lessons learned.

Figure 1.1 provides an overview of the defect prediction modelling process. The process is composed of the following stages: data preparation, model construction, and model validation. To develop a defect prediction model, we first need to *prepare a metrics dataset* of software modules (e.g., module size, module complexity), which are typically collected from a version control system (VCS). Second, we need to *label defective modules* if they have been affected by a code change that addresses an issue report that is classified as a defect. Third, we *train defect models* using a machine learning technique. Forth, we need to *configure the parameter settings* of such machine learning techniques that control their characteristics (e.g., the number of trees in a random forest classifier). Fifth, we *select performance measures* to measure the performance of defect prediction models. Finally, we *validate the models* in order to estimate the model performance when it is applied to new software modules and *interpret the models* in order to understand past pitfalls that lead to defective modules.

Prior work raises several concerns about inconsistent conclusions of several defect prediction studies. For example, a literature survey by Hall *et al.* [82] points out that several defect prediction studies that evaluate the top-performing defect prediction models arrive at different conclusions. Thus, the lack of consistency in the conclusions of prior work makes it hard to derive practical guidelines about the most appropriate defect prediction modelling process to use in practice.

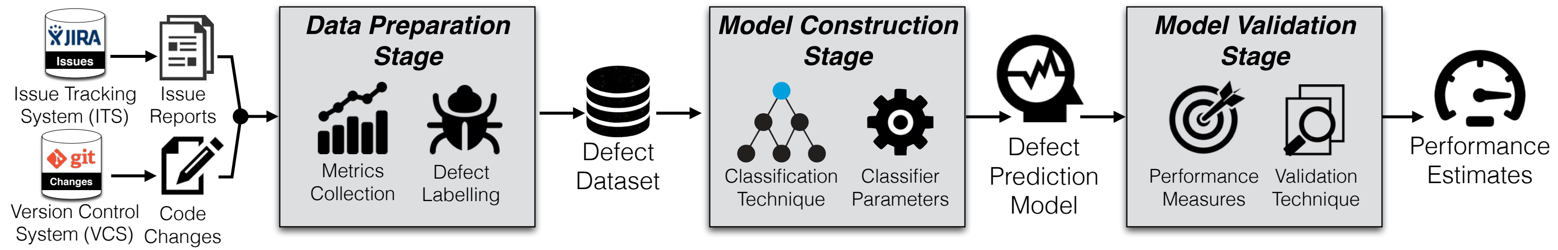


Figure 1.1.: An overview of the typical defect prediction modelling and its related experimental components.

Troublingly, a meta-analysis by Shepperd *et al.* [207] points out that the reported performance of defect prediction models shares a strong relationship with the research group who performed the study. This observation raises several concerns about the state of the defect prediction field. However, we suspect that research groups are likely to reuse experimental components (e.g., datasets, metrics, and classifiers) across their various studies. This tendency to reuse experimental components would introduce a strong relationship among the explanatory variables, which calls into question the validity of their findings.

Before delving into the main body of this thesis, we first set out to investigate the components that impact the conclusions of defect prediction models using the data provided by Shepperd *et al.* [207]. More specifically, we investigate (1) the strength of the relationship among the explanatory variables, i.e., research group and the experimental components (i.e., dataset family, metric family, and classifier family); (2) the interference that these relationships introduce when interpreting the impact of explanatory variables on the reported performance; and (3) the impact of the explanatory variables on the reported performance after we mitigate the strong relationship among the explanatory variables. Through a case study of 42 primary defect prediction studies (see Chapter 3), we find that (1) research group shares a strong relationship with the dataset and metrics families that are used in building models; (2) the strong relationship among explanatory variables introduces interference when interpreting the impact of research group on the reported model performance; and (3) after mitigating the interference, we find that the experimental components (e.g., metric family) that are used to construct defect prediction models share a stronger relationship with the reported performance than research group does. Our findings suggest that experimental components of defect prediction modelling may influence the conclusions of defect prediction models.

1.1. Problem Statement

The empirical evidence of our motivating analysis (see Chapter 3) leads us to the formation of our thesis statement, which we state as follows.

> **Thesis Statement:** The experimental components of defect prediction modelling impact the predictions and associated insights that are derived from defect prediction models. Empirical investigations on the impact of overlooked experimental components are needed to derive practical guidelines for defect prediction modelling.

Indeed, there exists a plethora of research that raise concerns about the impact of experimental components on defect prediction models. For example, Menzies and Shepperd [154] point out that a variety of experimental components may have an impact on the conclusions of defect prediction models. Yet, the **accuracy** (i.e., how much do estimates (e.g., predictions, insights, performance) differ from the ground-truth?) and **reliability** (i.e., how do such estimates vary when an experiment is repeated?) of such model predictions and associated insights have never been explored in depth.

Hence, in this thesis, we set out to empirically investigate the impact of the three often overlooked experimental components (i.e., noise generated by issue report mislabelling, parameter settings of classification techniques, and model validation techniques) of the three stages of defect prediction modelling, as well as provide practical guidelines for defect prediction modelling.

1.2. Thesis Overview

We now provide a brief overview of the thesis. Figure 1.2 provides an overview sketch of this thesis, which is broken into three parts. We describe each part below.

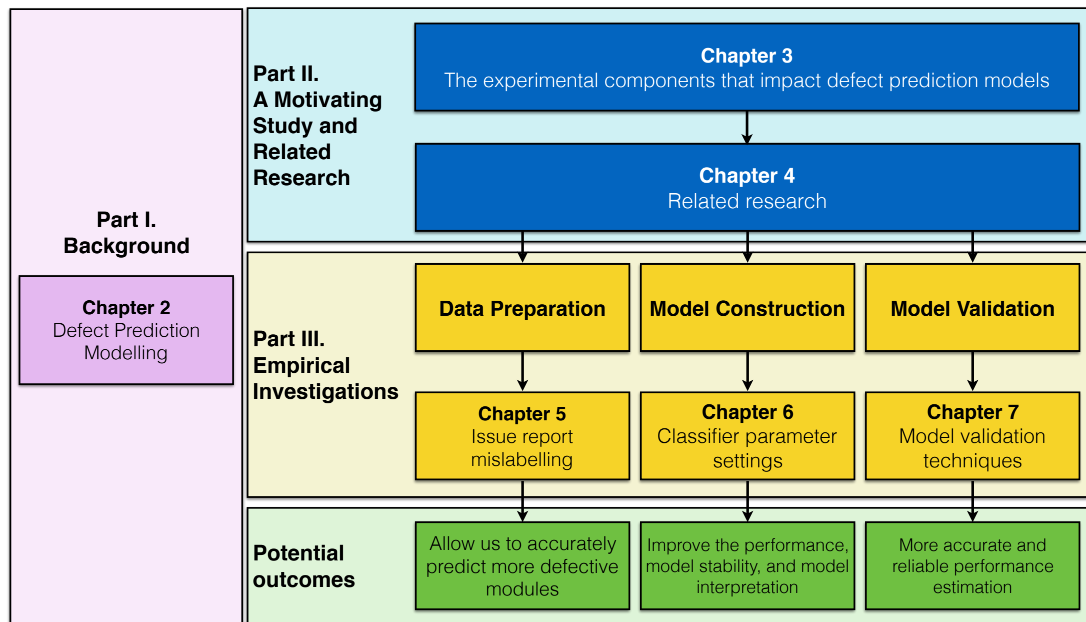


Figure 1.2.: An overview of the scope of this thesis.

Part I: Background and Definitions

Chapter 2 Defect Prediction Modelling.

In this chapter, we first provide necessary background material of the defect prediction modelling process (purple boxes). More specifically, we provide definitions of a defect prediction model and the defect prediction modelling process. Then, we provide a detailed description of the defect prediction modelling process with respect to the data preparation, model construction, and model validation stages.

Part II: A Motivating Study and Related Research

In order to situate the thesis, we perform a motivating analysis on the relationship between the reported performance of a defect prediction model and the experimental components that are used to construct the models (blue boxes). Then, we discuss several concerns with respect to prior research in order to situate the

three empirical studies.

Chapter 3 The Experimental Components that Impact Defect Prediction Models.

In this chapter, we investigate the relationship between the reported performance of a defect prediction model and the explanatory variables, i.e., research group and the experimental components (i.e., dataset family, metric family, and classifier family) using the data provided by Shepperd *et al.* [207]. More specifically, we investigate (1) the strength of the association among the explanatory variables, i.e., research group and the experimental components; (2) the interference that these associations introduce when interpreting the impact of the explanatory variables on the reported performance; and (3) the impact of the explanatory variables on the reported performance after we mitigate the strong associations among the explanatory variables.

Chapter 4 Related Research.

In order to situate this thesis with respect to prior research, we present a survey of prior research on defect prediction modelling.

Part III: Empirical Investigations of the Impact of Experimental Components

In this part, we shift our focus to the main body of the thesis. In this thesis, we focus on the three often overlooked experimental components of defect prediction modelling process (yellow boxes).

Chapter 5 The Impact of Issue Report Mislabelling.

The accuracy and reliability of a prediction model depends on the quality of the data from which it was trained. Therefore, defect prediction models may be inaccurate and unreliable if they are trained

using noisy data [94, 118]. Recent research shows that noise that is generated by *issue report mislabelling*, i.e., issue reports that describe defects but were not classified as such (or vice versa), may impact the performance of defect models [118]. Yet, while issue report mislabelling is likely influenced by characteristics of the issue itself — e.g., novice developers may be more likely to mislabel an issue than an experienced developer — the prior work randomly generates mislabelled issues. In this chapter, we investigate whether mislabelled issue reports can be accurately explained using characteristics of the issue reports themselves, and what is the impact of a realistic amount of noise on the predictions and insights derived from defect models.

Chapter 6 The Impact of Automated Parameter Optimization.

Defect prediction models are classifiers that are trained to identify defect-prone software modules. Such classifiers have *configurable parameters* that control their characteristics (e.g., the number of trees in a random forest classifier). Recent studies show that these classifiers may underperform due to the use of suboptimal default parameter settings [82]. However, it is likely feasible that parameter optimization may increase the risk of model overfitting (i.e., producing a classifier that is too specialized for the data from which it was trained to apply to other datasets). In this chapter, we investigate the performance, stability, and interpretation of defect prediction models where Caret — an automated parameter optimization technique — is applied.

Chapter 7 The Impact of Model Validation Techniques.

Defect prediction models may provide an unrealistically optimistic estimation of model performance when (re)applied to the same sample with which that they were trained. To address this problem, *Model Validation Techniques (MVTs)* (e.g., k-fold cross-validation) are used

to estimate how well a model will perform on unseen data [51,133,140, 257]. Recent research has raised concerns about the *bias* (i.e., how much do the performance estimates differ from the ground truth?) and *variance* (i.e., how much do performance estimates vary when an experiment is repeated?) of model validation techniques when applied to defect prediction models [152, 159, 169, 208, 246]. An optimal MVT would not overestimate or underestimate the ground truth performance. Moreover, the performance estimates should not vary broadly when the experiment is repeated. However, little is known about how bias and variance the performance estimates of MVTs tend to be. In this chapter, we set out to investigate the bias and variance of model validation techniques in the domain of defect prediction.

1.3. Thesis Contribution

In this thesis, we investigate the impact of experimental components of the three stages of defect prediction modelling on the performance and interpretation of defect prediction models. This thesis makes a variety of contributions to the research field. We highlight the key contributions as follow:

1. We demonstrate that research group shares a strong relationship with the dataset and metrics families that are used in building models. Such a strong relationship makes it difficult to discern the impact of the research group on model performance. After mitigating the impact of this strong relationship, we find that the research group has a smaller impact than metrics families. (Chapter 3)

2. We demonstrate that the noise that is generated by issue report mislabelling is non-random. Unlike prior work that shows that noise that are artificially generated has a large negative impact on the performance of defect pre-

diction models, we find that noise that are realistically generated by issue report mislabelling has little impact on the performance and interpretation of defect prediction models. (Chapter 5)

3. We demonstrate that defect prediction models substantially improve the performance and stability of defect prediction models, as well as they change their interpretation, when automated parameter optimization is applied. (Chapter 6)

4. We demonstrate that 78 out of 101 publicly-available defect datasets (77%) are highly susceptible to producing inaccurate and unstable results. (Chapter 7)

5. We demonstrate that the choice of model validation techniques has an impact on the bias and variance of the performance estimates that are produced by defect prediction models. (Chapter 7)

1.4. Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 provides background information and defines key terms. Chapter 3 presents the results of an analysis of the experimental factors. Chapter 4 presents research related to overlooked experimental components. Chapter 5 presents an investigation approach of the impact of issue report mislabelling and discusses their results. Chapter 6 presents an investigation approach of the impact of automated parameter optimization and discusses their results. Chapter 7 presents an investigation approach of the impact of model validation techniques and discusses their results. Finally, Chapter 8 draws conclusions and discusses promising avenues for future work.

Part I.

Background and Definitions

CHAPTER 2

Defect Prediction Modelling

Key Concept

A defect prediction model is a classifier that is trained to identify defect-prone software modules.

In this chapter, we provide a background and definitions of defect prediction model. We also provide a detailed explanation of defect prediction modelling.

2.1. Introduction

Software quality assurance (SQA) teams plays a critical role in software companies to ensure that software products are of sufficient quality before their release to customers [38, 120]. However, SQA teams often have limited resources in terms of team size and time.

Defect prediction models can help SQA teams focus their limited SQA resources on the most risky software modules. Broadly speaking, defect prediction models leverage historical software development data, such as, issue reports, historical code changes, to support decisions and help evaluate current development practices. Thus, insights that are derived from defect prediction models could lead to more informed and empirically-supported decisions.

In this thesis, we define *a defect prediction model* as a classifier that is trained to identify defect-prone software modules. Thus, defect prediction modelling is the process of preparing a defect dataset, applying a statistical model or a machine learning classifier to defect dataset, and estimating the performance of defect prediction models. Figure 2.1 provides a graphical illustration of the defect prediction modelling process.

2.1.1. Chapter Organization

In the remaining of this chapter, we provide a detailed explanation of the three stages of defect prediction modelling (i.e., data preparation, model construction, and model validation) which are the focus of this thesis.

2.2. Data Preparation

Module metrics and classes are typically mined from historical repositories, such as Issue Tracking Systems (ITSs) and Version Control Systems (VCSs). Issue Tracking Systems (ITSs) are a software application that keeps track of issue reports from users, developers, testers (e.g., JIRA[1], Bugzilla[2]). Version Control Systems (VCSs) are a software application that keeps track of revisions of source code. Some examples of VCSs that are widely-used today (but are not limited

[1] https://www.atlassian.com/software/jira

[2] https://www.bugzilla.org/

to): Git[3], Mercurial SCM[4], and SVN.[5]

To prepare a defect dataset, we first *extract metrics* of software modules (e.g., size, complexity, process metrics) from a version control system (VCS). We focus on the development activity that occurs prior to a release that corresponds to the `release` branch of a studied system. With this strictly controlled setting, we ensure that changes that we study correspond to the development and maintenance of official software releases. Since modern issue tracking systems, like JIRA, often provide traceable links between issue reports (i.e., a report described defects or feature requests) and code changes, we can identify the modules that are changed to address a particular issue report. Finally, we *label defective modules* if they have been affected by a code change that addresses an issue report that is classified as a defect. Similar to prior studies [51, 111, 213, 258], we define post-release defects as modules that are changed to address an issue report within the six-month period after the release date.

2.2.1. Metric Collection

To understand the characteristics of defect-proneness, prior studies proposed a variety of metrics that are related to software quality. We provide a brief summary of software metrics below.

Code metrics describe the relationship between code properties and software quality. For example, in 1971, Akiyama *et al.* [7] is among the first research to show that the size of modules (e.g., lines of code) shares a relationship with defect-proneness. McCabe *et al.* [144] and Halstead *et al.* [83] also show that the high complexity of modules (e.g., the number of distinct operators and operands) are likely to be more defective. Chidamber and Kemerer [43] also propose CK metrics suites, a set of six metrics that describe the structure of a class (e.g., the number of

[3] https://git-scm.com/
[4] https://www.mercurial-scm.org/
[5] https://subversion.apache.org/

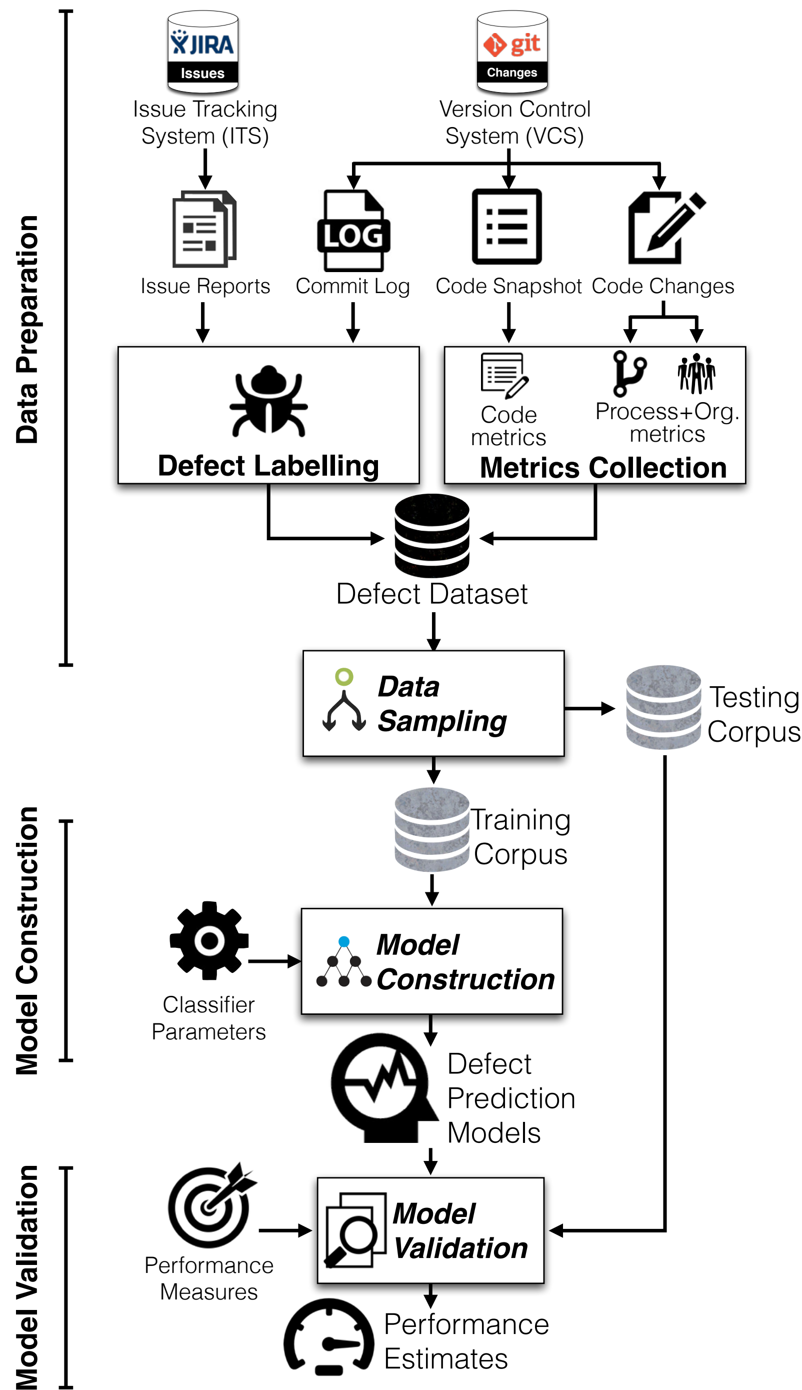


Figure 2.1.: A graphical illustration of the defect prediction modelling process.

methods defined in a class), that may share a relationship with defect-proneness. An independent evaluation by Basili *et al.* [22] and Gyimothy *et al.* [80] confirms that the CK metrics suites can be effectively used as software quality indicators. An in-depth investigation on the size-defect relationship by Syer *et al.* [226] shows that defect density has an inverted "U" shaped pattern (i.e., defect density increases in smaller files, peaks in the largest small-sized files/smallest medium-sized files, then decreases in medium and larger files). Such code metrics are typically extracted from a code snapshot at a release using various available static analysis tools (e.g., SLOCcount[6], Understand tool[7], the `mccabe` python package[8], and the `radon` python package[9])

Process metrics describe the relationship between change activities during software development process and software quality. Prior work has shown that historical software development process can be used to describe the characteristics of defect-proneness. For example, Graves *et al.* [76] find that the software defects may still exist in modules that were recently defective. Nagappan and Ball [170, 171] find that modules that have undergone a lot of change are likely to be more defective. Hassan [90] find that modules with a volatile change process, where changes are spread amongst several files are likely defective. Hassan and Holt [91] find that modules that are recently changed are more likely to be defective. Zimmermann and Nagappan [255, 257, 258] find that program dependency graphs share a relationship with software quality. Zimmermann *et al.* [258] find that the number of defects that were reported in the last six months before release for a module shares a relationship with software quality. Such process metrics are typically extracted using code changes that are stored in VCSs.

---

[6] https://sourceforge.net/projects/sloccount/
[7] https://scitools.com/
[8] https://pypi.python.org/pypi/mccabe
[9] https://pypi.python.org/pypi/radon

Organization metrics describe the relationship between organization structure and software quality [173]. For example, Graves *et al.* [76] find that modules that are changed by a large number of developers are likely to be more defective. Despite of the organization structure, prior studies have shown that code ownership shares a relationship with software quality. Thus, we focus on the two dimensions for an approximation of code ownership. First, *authoring activity* has been widely used to approximate code ownership. For example, Mockus and Herbslerb [162] identify developers who responsible for a module using the number of tasks that a developer has completed. Furthermore, Bird *et al.* [29] find that modules that are written by many experts are less likely to be defect-prone. Second, *reviewing activity* has been also used to approximate code ownership. For example, Thongtanunam *et al.* [238] find that reviewing expertise also shares a relationship with defect-proneness (e.g., modules that are reviewed by experts are less likely to be defect-prone).

2.2.2. Defect Labelling

Since modern issue tracking systems, like JIRA, often provide traceable links between issue reports (i.e., an issue report that describes a software defect or a feature request) and commit logs, we can identify the modules that are changed to address an issue report. First, the issue reports are extracted from the ITS of each studied system. Next, the references to code changes are extracted from those issue reports. Then, the commit information is extracted for the referenced code changes from the VCS. Finally, modules that are changed to address an issue report in the six-month period after the release date are labelled as a defective module. On the other hand, modules that are not changed to address any issue reports after the release date are labelled as clean (i.e., not defective).

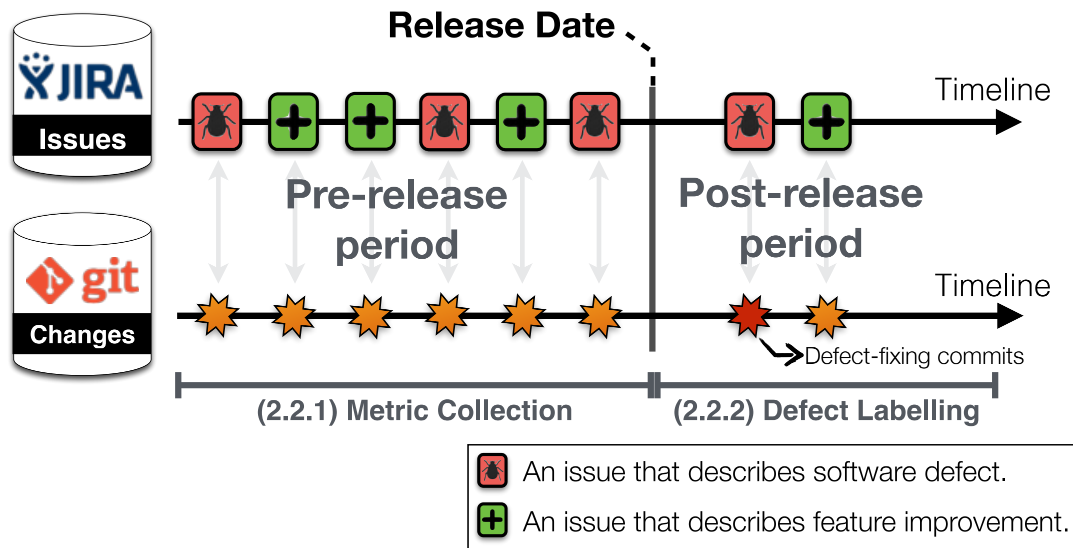


Figure 2.2.: A graphical illustration of data preparation stage.

To illustration, Figure 2.3 provides an example of a JIRA issue report that describes a software defect that affected the 4.0-ALPHA release of the LUCENE project.[10] A red box is used to highlight the category of the issue report (e.g., Bug, Improvement, Documentation, Maintenance, and Test), while a blue box is used to highlight the modules that are changed to address the issue report. Figure 2.4 shows an example Git commit that is related to the given example issue report.[11] Such commit information allows us to understand the characteristics of historical development process (e.g., the numbers of changed modules, the numbers of added or removed lines of code) that may share a relationship with software quality. Take this example, we label the `lucene/core/src/.../index/SegmentInfos.java` module as a defective module, since it is changed to address an issue report that describes a software defect.

[10] https://issues.apache.org/jira/browse/LUCENE-4128
[11] https://github.com/apache/lucene-solr/commit/662f8dd3423b3d56e9e1a197fe816393a33155e2

Lucene - Core / LUCENE-4128

add safety to preflex segmentinfo "upgrade"

Agile Board

Details

Type:	Bug	Status:	CLOSED
Priority:	Major	Resolution:	Fixed
Affects Version/s:	4.0-ALPHA	Fix Version/s:	4.0-ALPHA
Component/s:	None		
Labels:	None		
Lucene Fields:	New		

Description

Currently the one-time-upgrade depends on whether the upgraded .si file exists. And the writing is done in a try/finally so its removed if ioexception happens.

but I think there could be a power-loss or something else in the middle of this, the upgraded .si file could be bogus, then the user would have to manually remove it (they probably wouldnt know).

i think instead we should just have a marker file on completion, that we create after we successfully fsync the upgraded .si file. this way if something happens we just rewrite the thing.

Attachments

LUCENE-4128.patch	4 kB	11/Jun/12 12:41
LUCENE-4128.patch	3 kB	11/Jun/12 11:34

› **Activity**

People

Assignee:
Michael McCandless

Reporter:
Robert Muir

Votes:
0 Vote for this issue

Watchers:
1 Start watching this issue

Dates

Created:
10/Jun/12 17:29

Updated:
10/May/13 10:41

Resolved:
11/Jun/12 17:48

Development

1 commit

Latest 11/Jun/12 17:47

Figure 2.3.: An example issue report that describes software defect. A red box is used to highlight the category of the issue report, while a blue box is used to highlight the modules that are changed to address the issue report.

LUCENE-4128: be more careful on upgrading a 3.x segments_N file to in… Browse files

…dividual .si files so that OS or JVM crash, or machine power loss, doesn't leave index unusable

git-svn-id: https://svn.apache.org/repos/asf/lucene/dev/branches/branch_4x@1348942 13f79535-47bb-0310-9956-ffa450edef68

mikemccand committed on Jun 12, 2012 1 parent 15736d3 commit 662f8dd3423b3d56e9e1a197fe816393a33155e2

Showing **1 changed file** with **50 additions** and **6 deletions**. Unified Split

56 lucene/core/src/java/org/apache/lucene/index/SegmentInfos.java View

Figure 2.4.: A Git commit that is related to the given example issue report of Figure 2.3.

2.3. Model Construction

A defect prediction model is a classifier that is trained to identify defect-prone software modules. Defect prediction models are typically trained using the relationship between module's software metrics (e.g., size and complexity) that are extracted from historical development data that is recorded in software repositories and their defectiveness (e.g., the number of post-release defects) using a statistical regression or a machine learning classifier. In order to understand the relationship between such software metrics and their defectiveness, much research often uses supervised learning approaches (i.e., a statistical learning technique that is trained on a training data in order to produce a predicted outcome) [82, 157, 210, 251]. Below, we discuss two common types of outcomes (e.g., continuous outcome and binary outcome) in defect prediction research.

Continuous Outcome. There is a plethora of research that is focused on predicting the number of post-release defects [29, 51, 145, 151, 163, 180]. Linear regression is a common technique that is used for predicting the number of post-release defects. For example, Mockus and Weiss [163] use linear regression models to predict the number of software defects. Maneely *et al.* [151] use linear regression models to understand the relationship between developers social networks and software quality. Bird *et al.* [29] use linear regression models to understand the relationship between code ownership and software quality. McIntosh *et al.* [145] use linear regression models to understand the relationship between code review practices and software quality.

Binary Outcome. Despite predicting the number of post-release defects, much research is also focused on classifying modules as defective or clean. A variety of classification techniques have been explored in defect prediction research. For example, logistic regression [22, 36], rule-based techniques [16], boosting techniques [16, 105, 204], Multivariate Adaptive Regression Splines (MARS) [26, 100], and discrimination analysis techniques [110]. Such techniques can be accessed us-

ing common research toolkits (such as R [190], Weka [81], and Scikit-learn [185]).

2.4. Model Validation

Once a defect prediction model is constructed, it is often unknown how accurate is a defect prediction model performs when it is applied to a new software modules (i.e., unseen dataset). To address this problem, *model validation techniques* (e.g., k-fold cross-validation) are commonly used to estimate the model performance. The model performance is used to (1) indicate how well a model will perform on unseen data [51, 54, 140, 187, 245, 257]; (2) select the top-performing prediction model [74, 115, 133, 159, 169, 232, 247]; and (3) combine several prediction models [20, 206, 241, 252]. Below, we briefly discuss the common performance measures that are used to evaluate defect prediction models and model validation techniques that are used to generate sample datasets for model construction.

2.4.1. Performance Measures

The performance of defect prediction models can be quantified using a variety of threshold-dependent (e.g., precision, recall) and threshold-independent (e.g, Area Under the receiver operating characteristic Curve (AUC)) performance measures. We describe each type of performance measure below.

Threshold-Dependent Performance Measures. When applied to a module from the testing corpus, a defect prediction model will report the probability of that module being defective. In order to calculate the threshold-dependent performance measures, these probabilities are transformed into a binary classification (defective or clean) using a threshold value of 0.5, i.e., if a module has a predicted probability above 0.5, it is considered defective; otherwise, the module is considered clean.

Using the threshold of 0.5, we compute the precision and recall performance

Table 2.1.: Confusion matrix for predicting defect-prone modules.

	Actual	
Classified as	**Defective**	**Non-Defective**
Defective	TP	FP
Non-defective	FN	TN

measures. These measures are calculated using the confusion matrix of Table 2.1. *Precision* measures the proportion of modules that are classified as defective, which are actually defective ($\frac{TP}{TP+FP}$). *Recall* measures the proportion of actually defective modules that were classified as such ($\frac{TP}{TP+FN}$).

Threshold-Independent Performance Measures. Prior research has argued that precision and recall are unsuitable for measuring the performance of defect prediction models because they: (1) depend on an arbitrarily-selected threshold (typically 0.5) [9, 17, 133, 192, 210], and (2) are sensitive to imbalanced data [52, 93, 139, 147, 227]. Thus, we also discuss three threshold-independent performance measures to quantify the performance of our defect prediction models.

First, the *Area Under the receiver operator characteristic Curve* (AUC) [84] is used to measure the discrimination power of our models. The AUC measures a classifier's ability to discriminate between defective and clean modules (i.e., do the defective modules tend to have higher predicted probabilities than clean modules?). AUC is computed by measuring the area under the curve that plots true positive rate against the false positive rate while varying the threshold that is used to determine whether a module is classified as defective or not. Values of AUC vary from 0 (worst classifier performance), 0.5 (random classifier performance) to 1 (best classifier performance).

In addition to the discrimination power, practitioners often use the predicted probabilities to rank defect-prone files [163, 170, 214, 258]. Shihab *et al.* [214] point out that practitioners often use the predicted probability to make decisions. Mockus *et al.* [163] point out that the appropriate range of probability values is

important to make an appropriate decision (e.g., high-reliability systems may require a lower cutoff value than 0.5). However, the AUC does not capture all of the dimensions of a prediction model [55, 87, 222, 224]. To measure the accuracy of the predicted probabilities, we use the Brier score and the calibration slope.

The *Brier score* [37, 201] is used to measure the distance between the predicted probabilities and the outcome. The Brier score is calculated as:

$$\mathrm{B} = \frac{1}{N}\sum_{i=1}^{N}(f_t - o_t)^2 \tag{2.1}$$

where f_t is the predicted probability, o_t is the outcome for module t encoded as 0 if module t is clean and 1 if it is defective, and N is the total number of modules. The Brier score varies from 0 (best classifier performance), 0.25 (random classifier performance), to 1 (worst classifier performance).

Finally, the *calibration slope* is used to measure the direction and spread of the predicted probabilities [48, 55, 87, 89, 158, 223, 224]. The calibration slope is the slope of a logistic regression model that is trained using the predicted probabilities of our original defect prediction model to predict whether a module will be defective or not [48]. A calibration slope of 1 indicates the best classifier performance and a calibration slope of 0 indicates the worst classifier performance.

2.4.2. Model Validation Techniques

We often *validate the defect prediction models* in order to estimate the model performance when it is applied to new (i.e., unseen) software modules. There is a variety of model validation techniques that can be used to estimate the model performance. *Holdout validation* randomly splits a dataset into training and testing corpora according to a given proportion (e.g., 30% holdout for testing). *Cross-validation* extends the idea of holdout validation by repeating the splitting process several times, which randomly partitions the data into k folds of

roughly equal size where each fold contains roughly the same proportions of the defective ratio [73, 225]. We then use the training corpus to train a defect prediction model, while the testing corpus is used to estimate the model performance. While such cross-validation techniques are commonly used in defect prediction research, more powerful approach like *bootstrap validation* that leverages aspects of statistical inference [62, 87, 92, 222] has been rarely explored in software engineering research. Unlike cross-validation techniques, a model is still trained using a bootstrap sample (i.e., a sample that is randomly drawn with replacement from a defect dataset), and the model is tested using the rows that do not appear in the bootstrap sample [58]. The process of resampling is repeated several times (e.g., 100 repetitions). The key intuition is that the relationship between the studied dataset and the theoretical population from which it is derived is asymptotically equivalent to the relationship between the bootstrap samples and the studied dataset. More detailed explanation is discussed in Section 7.3.

2.5. Chapter Summary

This chapter provides a definition of defect prediction models and describes foundation concepts of the typical defect prediction modelling process. In the next chapter, we perform a motivational analysis to investigate what are the factors that impact the predictions and insights that are derived from defect prediction models.

Part II.

A Motivating Study and Related Research

CHAPTER 3

The Experimental Components that Impact Defect Prediction Models

Key Question

Which experimental components have the biggest impact on the conclusion of a defect prediction study?

An earlier version of the work in this chapter appears in the Journal of the IEEE Transactions on Software Engineering (TSE) [233].

3.1. Introduction

Recently, Shepperd *et al.* [207] study the extent to which the research group that performs a defect prediction study associates with the reported performance of defect prediction models. Through a meta-analysis of 42 primary studies, they find that the reported performance of a defect prediction model shares a strong relationship with the group of researchers who construct the models. Shepperd *et al.*'s findings raise several concerns about the current state of the defect prediction field. Indeed, their findings suggest that many published defect prediction studies are biased (i.e., producing inconsistent conclusions), and calls their validity into question.

In this chapter, we perform an alternative investigation of Shepperd *et al.*'s data. More specifically, we set out to investigate (1) the strength of the association among the explanatory variables, e.g., research group and metric family (Section 3.2); (2) the interference that these associations introduce when interpreting the impact of explanatory variables on the reported performance (Section 3.3); and (3) the impact of the explanatory variables on the reported performance after we mitigate the strong associations among the explanatory variables (Section 3.4). We also provide a replication package of the following experiment in Appendix A.

3.2. The Presence of Collinearity

We suspect that research groups are likely to reuse experimental components (e.g., datasets, metrics, and classifiers) in several studies. This tendency to reuse experimental components would introduce a strong association among the explanatory variables of Shepperd *et al.* [207]. To investigate our suspicion, we measure the strength of the association between each pair of the explanatory variables that are used by Shepperd *et al.* [207], i.e., ResearcherGroup, DatasetFamily, MetricFamily, and ClassifierFamily.

Table 3.1.: [A Motivating Study] The association among explanatory variables.

Pair	**Cramer's V**	**Magnitude**
ResearcherGroup & MetricFamily	0.65***	Strong
ResearcherGroup & DatasetFamily	0.56***	Relatively strong
MetricFamily & DatasetFamily	0.55***	Relatively strong
ResearcherGroup & ClassifierFamily	0.54***	Relatively strong
DatasetFamily & ClassifierFamily	0.34***	Moderate
MetricFamily & ClassifierFamily	0.21***	Moderate

Statistical significance of the Pearson χ^2 test:
$\circ p \geq .05$; * $p < .05$; ** $p < .01$; *** $p < .001$

Approach. Since the explanatory variables are categorical, we first use a Pearson χ^2 test [5] to check whether a statistically significant association exists between each pair of explanatory variables ($\alpha = 0.05$). Then, we compute Cramer's V [49] to quantify the strength of the association between each pair of two categorical variables. The value of Cramer's V ranges between 0 (no association) and 1 (strongest association). We use the convention of Rea *et al.* [197] for describing the magnitude of an association. To compute the Pearson's χ^2 and Cramer's V values, we use the implementation provided by the `assocstats` function of the `vcd` R package [156].

Results. **Research group shares a strong association with the dataset and metrics that are used.** Table 3.1 shows the Cramer's V values and the p-value of the Pearson χ^2 test for each pair of explanatory variables. The Cramer's V values indicate that research group shares a strong association with the dataset and metrics that are used. Indeed, we find that 13 of the 23 research groups (57%) only experiment with one dataset family, where 9 of them only use one NASA dataset, which contains only one family of software metrics (i.e., static metrics). Moreover, 39% of researcher groups only use the static metric family of the NASA dataset in several studies. The strong association among research groups, dataset, and metrics confirms our suspicion that researchers often reuse experimental components.

3.3. The Interference of Collinearity

The strong association among explanatory variables that we observe in Section 3.2 may introduce interference when one studies the impact of these explanatory variables on the outcome [77,243]. Furthermore, this interference among variables may cause impact analyses, such as ANOVA, to report spurious relationships that are dependent on the ordering of variables in the model formula. Indeed, ANOVA is a hierarchical model that first attributes as much variance as it can to the first variable before attributing residual variance to the second variable in the model formula [189]. If two variables share a strong association, the variable that appear first in the model formula will got the brunt of the variance associated with it. Hence, we set out to investigate the interference that is introduced by the strong association among explanatory variables.

Approach. To investigate this interference, we use a bootstrap analysis approach, which leverages aspects of statistical inference [62]. We first draw a bootstrap sample of size N that is randomly drawn with replacement from an original dataset that is also of size N. We train linear regression models with the data of the bootstrap sample using the implementation provided by the `lm` function of the `stats` R package [190]. For each bootstrap sample, we train multi-way ANOVA models with all of the 24 possible ordering of the explanatory variables (e.g., ANOVA(ResearcherGroup × DatasetFamily × MetricFamily × ClassifierFamily) versus ANOVA(DatasetFamily × ResearcherGroup × MetricFamily × ClassifierFamily)). Following the prior study [207], we compute the partial η^2 values [199], which describe the proportion of the total variance that is attributed to an explanatory variable for each of the 24 models. We use the implementation provided by the `etasq` function of the `heplots` R package [67]. We repeat the experiment 1,000 times for each of the 24 models to produce a distribution of the partial η^2 values for each explanatory variable.

Results. **The strong association among the explanatory variables introduces**

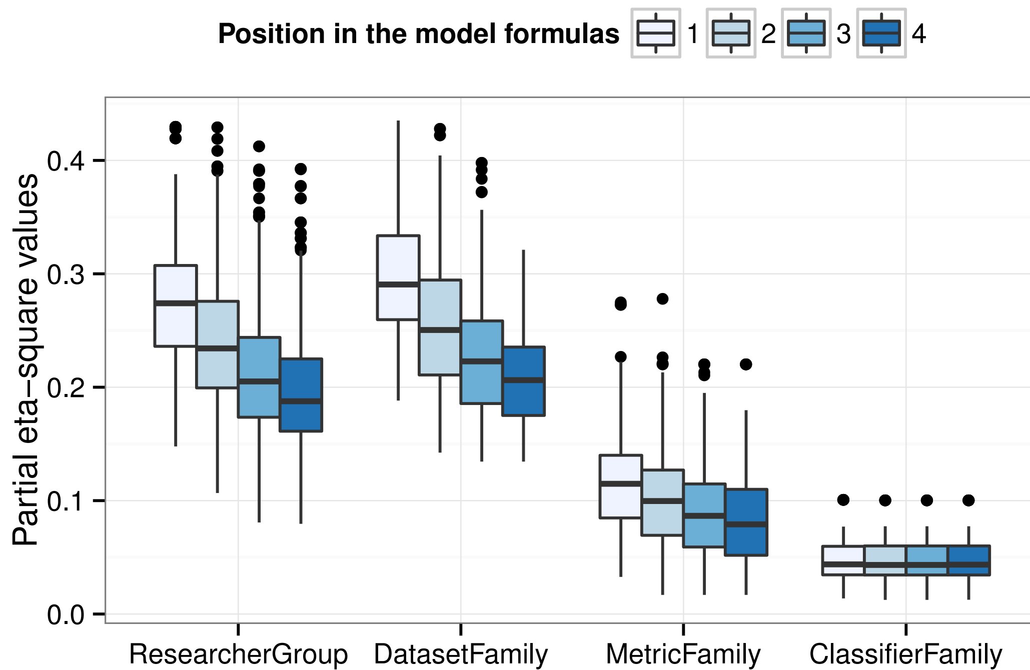


Figure 3.1.: [A Motivating Study] Distribution of the partial η^2 values for each explanatory variable when it appears at each position in the model formula.

interference when interpreting the impact of research group on model performance. Figure 3.1 shows the distributions of the partial η^2 values for each explanatory variable when it appears at each position in the model formula. Each boxplot is derived from the models of the 24 possible variable ordering combinations. The results show that there is a decreasing trend in partial eta-squared values when collinearity is not mitigated. Indeed, research group and dataset family tend to have the largest impact when they appear in earlier positions in the model formula, indicating that the impact of explanatory variables on the outcome depends on the ordering of variables in the model formula. Moreover, we observe that research group and dataset family tend to have comparable partial

η^2 values, indicating that the strong association introduces interference. In particular, a model with DatasetFamily in the first position and ResearcherGroup in the second position in the model formula would lead us to believe that DatasetFamily plays a bigger role than the ResearcherGroup. However, once we flip the positions of DatasetFamily and ResearcherGroup, we would reach a different conclusion, i.e., that ResearcherGroup plays a bigger role in the model. On the other hand, the moderate association between the metric and classifier families introduces much less interference.

3.4. Mitigating Collinearity

In the study of Shepperd *et al.*, Table 13 is derived from a one-way ANOVA analysis ($y = x_n$) for each of the explanatory variables and Table 14 is derived from a multi-way ANOVA analysis ($y = x_1 \times ... \times x_n$). One of the main assumption of ANOVA analysis is that the explanatory variables must be independent. The prior sections show that such strong association among the explanatory variables (Section 3.2) introduces interference when we interpret the impact of the explanatory variables on the outcome (Section 3.3). However, Shepperd *et al.*'s multi-way ANOVA analysis did not mitigate for the collinearity between the explanatory variables.

In this section, we set out to investigate the impact of the explanatory variables on the reported performance after mitigating the interference that is introduced by strongly associated explanatory variables. Thus, our earlier analysis indicates that we cannot include all three of the explanatory variables in the same model. Instead, we should only include two of these three variables in our models. Hence, we opt to focus on one studied dataset in order to control the dataset family metric by holding it constant.

Approach. To mitigate the interference, we first select the Eclipse dataset family, which is the second-most popular dataset family in the studied dataset. We

Table 3.2.: [A Motivating Study] Partial η^2 values of the multi-way ANOVA analysis with respect to the Eclipse dataset family.

	ResearcherGroup Model	**MetricFamily Model**
Adjusted R^2	0.19	0.36
AIC	-77	-105
	Partial η^2	Partial η^2
Research Group	0.127 (medium)	†
Metric Family	†	0.235 (large)
Classifier Family	0.122 (medium)	0.113 (medium)
Research Group:Classifier Family	0.022 (small)	-
Metric Family:Classifier Family	-	0.162 (large)

†Strong association variables

choose the Eclipse dataset family instead of selecting the NASA dataset family because Section 3.2 reveals that 39% of the research groups only use the static metric family of the NASA dataset in several studies. Overall, the Eclipse dataset family contains 6 metrics families, which are used by 7 research groups who fit defect prediction model using 6 classifier families. While controlling for the dataset family metric, only research group, metric family, and classifier family will be included in the model formulas. Since the dataset family metric is now a constant, it is excluded from our model formula.

Since Table 3.1 shows that research group shares a strong association with the metric family, we build two different linear regression models by removing one of the two strongly associated variables, i.e., one model uses the research group and classifier family variables, while another model uses the metric family and classifier family variables.

To confirm the absence of collinearity in the models, we perform a redundancy analysis [87] in order to detect redundant variables prior to constructing the models. We use the implementation provided by the `redun` function in the `rms` R package [88].

To assess the fit of our models, we compute the adjusted R^2 and the Akaike Information Criterion (AIC) [6]. The adjusted R^2 measures the amount of variance, while AIC measures the goodness-of-fit based on information entropy. In general, higher adjusted R^2 and lower AIC values correspond to a better fit of the model to the underlying data.

Finally, we perform a multi-way ANOVA analysis and compute the partial η^2 values. As suggested by Richardson *et al.* [199] and Mittas *et al.* [159], we use the convention of Cohen [44] for describing the effect size of the partial η^2 values — values below 0.01, 0.06, and 0.14 describe small, medium, and large effect sizes, respectively.

Results. **When we mitigate the interference of strongly associated explanatory variables, we find that the research group has a smaller impact than the metric family with respect to the Eclipse dataset family.** Table 3.2 shows partial η^2 values of the ANOVA analysis with respect to the Eclipse dataset family. The results show that the MetricFamily model, which achieves a higher adjusted R^2 and a lower AIC, tends to represent the underlying data better than the ResearcherGroup model. The redundancy analysis also confirms that there are no redundant variables in the MetricFamily model. Unlike Shepperd *et al.*'s earlier observations, our ANOVA analysis of the MetricFamily model shows that the choice of metrics that are used to build defect prediction models tends to have a large impact on the reported performance with respect to the Eclipse dataset family. Moreover, since the interference has been mitigated, the ANOVA results still hold when the explanatory variables are reordered.

3.5. Chapter Summary

The prior work of Shepperd *et al.* [207] suggests that the reported performance of a defect prediction model shares a strong relationship with the research group who conducted the study. This observation raises several concerns about the state of

the defect prediction field. In this chapter, we investigate (1) the strength of the association among the explanatory variables of Shepperd *et al.*'s study [207]; (2) the interference that these associations introduce when interpreting the impact of the explanatory variables on the reported performance; and (3) the impact of the explanatory variables on the reported performance after we mitigate the interference that is introduced by strongly associated explanatory variables. We make the following observations:

- Research group shares a strong association with the dataset and metrics families that are used in building models, suggesting that researchers should experiment with a broader selection of datasets and metrics in order to maximize external validity.

- The strong association among explanatory variables introduces interference when interpreting the impact of research group on the reported model performance, suggesting that researchers should carefully mitigate collinearity issues prior to analysis in order to maximize internal and construct validity.

- After mitigating the interference, we find that the research group has a smaller impact than metric family with respect to the Eclipse dataset family, suggesting that researchers should carefully examine the choice of metrics when building defect prediction models.

These observations lead us to conclude that the relationship between research groups and the performance of a defect prediction model have more to do with the tendency of researchers to reuse experimental components (e.g., datasets and metrics). Hence, a threat of bias exists if authors fixate on studying the same datasets with the same metrics. We recommend that research groups experiment with different datasets and metrics rather than relying entirely on reusing experimental components.

When adhering to our recommendation, researchers should be mindful of the inherent trade-off between maximizing internal and external validity in empirical research [216]. For example, maximizing external validity by studying a large corpus of datasets may raise threats to the internal validity of the study (i.e., the insights may be difficult to discern due to a broad selection of the studied systems). On the other hand, maximizing internal validity by focusing on a highly controlled experiment may raise threats to the external validity of the study (i.e., the insights may be too specific to the studied systems to generalize to other systems).

3.5.1. Concluding Remarks

In this chapter, we perform a meta-analysis to investigate which experimental components have the biggest impact on the conclusion of a defect prediction study. We find that the reported performance shares a stronger relationship with the choice of metrics than research group who perform studies, suggesting that experimental design choices of defect prediction modelling may influence the conclusions of defect prediction studies.

In the next chapter, we survey prior research on defect prediction modelling in order to situate our empirical studies.

CHAPTER 4

Related Research

Key Concept

Experimental components may impact the conclusions of defect prediction models.

An earlier version of the work in this chapter appears in the Doctoral Symposium of the International Conference on Software Engineering (ICSE 2016). [229]

4.1. Introduction

There exists a plethora of research that raise concerns about the impact of experimental components on defect prediction models. For example, Menzies and Shepperd [154] point out that a variety of experimental components may have an impact on the conclusions of defect prediction models. Kim *et al.* [118] point out that noise in defect datasets has a large negative impact on the performance of defect prediction models. Hall *et al.* [82] point out that defect prediction models may underperform because the suboptimal default parameter settings are used. Mittas *et al.* [159] and Turhan *et al.* [246] point out that the random nature of sampling used by model validation techniques may produce inaccurate performance estimates.

In this chapter, we survey the related research on the concerns about the impact of experimental components on defect prediction models. In the following sections, we organize the concerns along the 3 stages of defect prediction modelling (i.e., data preparation, model construction, and model validation) that are the focus of this thesis. More specifically, we describe how the related work motivates our three empirical studies.

4.2. Data Preparation Stage

Data preparation stage involves a variety of experimental components, such as metrics selection and the quality of data preparation process. Prior work shows that the use of various choice of metrics may have an impact on the performance of defect prediction models. For example, Kamei *et al.* [109] and Rahman *et al.* [192] show that defect prediction models that are trained with process metrics outperform defect prediction models that are trained with code metrics. Furthermore, the motivating analysis of Chapter 3 also shows that the selected software metrics shares the strongest relationship with the reported performance of defect prediction models.

Even if the best set of metrics are selected, decision-making based on poor data quality (i.e., data that is inaccurate) can easily do more harm than good. Several studies show that poor data quality can lead to a large negative impact on all segments of the economy (e.g., companies, governments, and academia and their customers). For example, Redman [198] argues that poor data quality increases operational costs and can impact operations, tactics and strategies.

Prior studies raise several concerns about data quality for software engineering research. For example, Mockus [160] argues that poor data quality can lead to biased conclusions. Aranda and Venolia [12] argue that information that is recorded in issue reports is often inaccurate. Tian *et al.* [239] point out that severity levels (e.g., critical or minor) of issue reports are often misclassified. While such software engineering data is inaccurate, Liebchen and Shepperd [136] point out that very few software engineering studies are aware of data quality issues.

The predictions and insights that are derived from defect prediction models depends on the quality of the data from which these models are trained. Prior work has investigated various potential sources of biases that can produce noise in the defect data that are used to train defect models. Below, we discuss related research along the two dimensions.

4.2.1. Noise generated by linkage process.

Defect models are trained using datasets that connect issue reports recorded in an Issue Tracking System (ITS) with the software modules that are impacted by the associated code changes that address these issue reports. The code changes are in turn recorded in a Version Control System (VCS). Thus, the quality of the data recorded in the ITS and VCS impacts the quality of the data that is used to train defect models [11, 21, 28, 94, 178].

Prior work points out the process of linking issue reports with code changes can generate noise in defect prediction datasets, since the linkage process often depends on manually-entered links that are provided by developers [21, 28, 118]. For example, Koru and Tian [127] point out that the defect linking practices are different from project to project. Bird *et al.* [28] find that the issue reports of several defects are not identified in the commit logs. Bachmann *et al.* [21] find that the noise generated by missing links in defect prediction datasets introduces bias. Kim *et al.* [118] find that the noise has a large negative impact on the performance of defect prediction models. Recently, Rahman *et al.* [193] find that the size of the dataset matters more than the amount of injected noise. Hence, various techniques that are proposed by Wu *et al.* [253] and Nguyen *et al.* [178] are needed to detect and cope with biases in defect datasets.

4.2.2. Noise generated by issue report mislabelling.

Even if all of the links between issue reports and code changes are correctly recovered, noise may creep into defect prediction datasets if the issue reports themselves are mislabelled. For example, Herzig *et al.* [94] find that 43% of all issue reports are mislabelled, and this mislabelling impacts the ranking of the most defect-prone files. While Kim *et al.* [118] find that noise has a large negative impact on the performance of defect prediction models, they artificially generated noise. Yet, noise is likely non-random—novice developers might tend to mislabel issue reports more than experienced developers.

Observation 1. Little is known about the characteristics of noise that is generated by issue report mislabelling and its impact on the performance and interpretation of defect prediction models.

4.3. Model Construction Stage

Model construction stage involves various experimental components, such as, the choice of classification techniques, and the choice of classifier parameter settings.

4.3.1. Selecting Classification Techniques

To build defect prediction models, prior research explored the use of various classification techniques in order to train defect prediction models [74, 133, 221]. For example, in early research, researchers tend to apply statistical learning techniques (e.g., logistic regression) to build defect prediction models [22, 36]. In recent research, researchers tend to used more advanced machine learning techniques, such as, rule-based techniques [16], boosting techniques [16, 105, 204], Multivariate Adaptive Regression Splines (MARS) [26, 100], and discrimination analysis techniques [110].

Even if the noise in defect datasets has been mitigated, the performance of defect prediction models still relies heavily on the used classification techniques. Indeed, Song *et al.* [221] suggests that different data sets should use different classification techniques. Panichella *et al.* [184] also find that using different classification techniques can identify different defective modules. Ghotra *et al.* [74] also confirm that the choice of classification techniques has a large impact on the performance of defect prediction models for both proprietary and open-source systems. Hence, researchers should explore many of the readily-available classification techniques from toolkits such as R and Weka.

4.3.2. Selecting Classifier Parameter Settings

Even if the best classification techniques are used to construct defect prediction models, such classification techniques often have *configurable parameters* that control their characteristics (e.g., the number of trees in a random forest classi-

fier). Thus, training defect prediction models involves selecting a classification technique and its most appropriate parameters.

Even if the best classifier is selected, the performance of defect prediction models may also be fluctuated based on different classifier parameters. For example, prior research points out that selecting different classifier parameters may impact the performance of defect models [147, 148]. Hall *et al.* [82] also point out that defect prediction models may under-perform due to the use of suboptimal default parameter settings. Mittas *et al.* [159] also point out that highly sensitive classifiers will make defect prediction studies more difficult to reproduce.

Observation 2. Little is known about the impact of parameter settings on the performance and interpretation of defect prediction models and the benefits of automated parameter optimization techniques.

4.4. Model Validation Stage

Model validation stage involves various experimental components, such as, the choice of performance measures and the choice of model validation techniques.

4.4.1. Selecting Model Validation Techniques.

Prediction models may provide an unrealistically optimistic estimation of model performance when (re)applied to the same sample with which were trained. To address this problem, *model validation techniques* (e.g., k-fold cross-validation) are commonly used to estimate the model performance. The model performance is used to (1) indicate how well a model will perform on unseen data [51, 54, 140, 187, 245, 257]; (2) select the top-performing prediction model [74, 115, 133, 159, 169, 232, 247]; and (3) combine several prediction models [20, 206, 241, 252].

Recent research has raised concerns about the accuracy and reliability of model validation techniques when applied to defect prediction models [72, 152, 154, 159, 169, 246]. Indeed, a perfectly unbiased model validation technique should produce a performance estimate that is equivalent to the model performance on unseen data.

> **Observation 3.** Little is known about the accuracy and reliability of the performance estimates that are derived from commonly used model validation techniques for defect prediction models.

4.5. Chapter Summary

In this chapter, we briefly survey prior research along the three stages of defect prediction modelling. We find that while the related work supports our thesis hypothesis that the choice of experimental components may have an impact on the conclusions of defect prediction studies, it is not yet clear: (1) what are the characteristics of noise that is generated by issue report mislabelling and what is the impact of such a noise on the performance and interpretation of defect prediction models; (2) the impact of parameter settings on the performance, model stability, and interpretation of defect prediction models; and (3) the accuracy and reliability of the performance estimates that are derived from the commonly-used model validation techniques. Broadly speaking, the remainder of this thesis describes our empirical studies that set out to tackle these three gaps in the literature. We begin, in the next chapter, by studying the impact of issue report mislabelling on the performance and interpretation of defect prediction models.

Part III.

Empirical Investigations of the Impact of Experimental Components

CHAPTER 5

The Impact of Issue Report Mislabelling

Key Finding

Noise generated by issue report mislabelling has little impact on the performance and interpretation of defect prediction models.

An earlier version of the work in this chapter appears in the Proceedings of the International Conference on Software Engineering (ICSE), 2015 [230].

5.1. Introduction

Defect models, which identify defect-prone software modules using a variety of software metrics [82, 191, 210], serve two main purposes. First, defect models can be used to *predict* [7, 50, 90, 119, 163, 167, 175, 195, 258] modules that are likely to be defect-prone. Software Quality Assurance (SQA) teams can use defect models in a prediction setting to effectively allocate their limited resources to the modules that are most likely to be defective. Second, defect models can be used to *understand* [42, 145, 161, 163, 212, 214] the impact of various software metrics on the defect-proneness of a module. The insights derived from defect models can help software teams to avoid pitfalls that have often led to defective software modules in the past.

The accuracy of the predictions and insights derived from defect models depends on the quality of the data from which these models are trained. Indeed, Mockus argues that poor data quality can lead to biased conclusions [160]. Defect models are trained using datasets that connect issue reports recorded in an Issue Tracking System (ITS) with the software modules that are impacted by the associated code changes that address these issue reports. The code changes are in turn recorded in a Version Control System (VCS). Thus, the quality of the data recorded in the ITS and VCS impacts the quality of the data used to train defect models [11, 21, 28, 94, 178].

Recent research shows that the noise that is generated by *issue report mislabelling*, i.e., issue reports that describe defects but were not classified as such (or vice versa), may impact the performance of defect models [118, 193]. Yet, while issue report mislabelling is likely influenced by characteristics of the issue itself — e.g., novice developers may be more likely to mislabel an issue than an experienced developer — the prior work randomly generates mislabelled issues.

In this chapter, we set out to investigate whether mislabelled issue reports can be accurately explained using characteristics of the issue reports themselves,

and what impact a realistic amount of noise has on the predictions and insights derived from defect models. Using the manually-curated dataset of mislabelled issue reports provided by Herzig *et al.* [94], we generate three types of defect datasets: (1) *realistic noisy* datasets that contain mislabelled issue reports as classified manually by Herzig *et al.*, (2) *random noisy* datasets that contain the same proportion of mislabelled issue reports as contained in the realistic noisy dataset, however the mislabelled issue reports are selected at random, and (3) *clean* datasets that contain no mislabelled issues (we use Herzig *et al.*'s data to reassign the mislabelled issue reports to their correct categories). Through a case study of 3,931 issue reports spread across 22 releases of the Apache Jackrabbit and Lucene systems, we address the following three research questions:

(RQ1) Is mislabelling truly random?
Issue report mislabelling is not random. Our models can predict mislabelled issue reports with a mean F-measure that is 4-34 times better than that of random guessing. The tendency of a reporter to mislabel issues in the past is consistently the most influential metric used by our models.

(RQ2) How does mislabelling impact the performance of defect models?
We find that the precision of our defect models is rarely impacted by mislabelling. Hence, practitioners can rely on the accuracy of modules labelled as defective by defect models that are trained using noisy data. However, cleaning the data prior to training the defect models will likely improve their ability to identify all defective modules.

(RQ3) How does mislabelling impact the interpretation of defect models?
We find that 80%-85% of the metrics in the top influence rank of the clean models also appear in the top influence rank of the noisy models, indicating that the most influential metrics are not heavily impacted by issue report mislabelling. On the other hand, as little as 18% of the metrics

in the second and third influence rank of the clean models appear in the same rank in the noisy models, which suggests that the less influential metrics are more unstable.

Furthermore, we find that randomly injecting mislabelled defects tends to overestimate the impact of truly mislabelling on model performance and model interpretation.

5.1.1. Chapter organization

The remainder of this chapter is organized as follows. Section 5.2 situates this paper with respect to the related work. Section 5.3 discusses the design of our case study, while Section 5.4 presents the results with respect to our three research questions. Section 5.5 discloses the threats to the validity of our work. Finally, Section 5.6 draws conclusions.

5.2. Related Work & Research Questions

Given a software module, such as a source code file, a defect model classifies it as either likely to be defective or clean. Defect models do so by modelling the relationship between module metrics (e.g., size and complexity), and module class (defective or clean).

As shown in Figure 5.1, module metrics and classes are typically mined from historical repositories, such as ITSs and VCSs. First, issue reports, which describe defects, feature requests, and general maintenance tasks, are extracted from the ITS. Next, the historical code changes that are recorded in a VCS are extracted. Finally, these issue reports are linked to the code changes that have been performed in order to address them. For example, a module's class is set to defective if it has been affected by a code change that addresses an issue report that is classified as a defect.

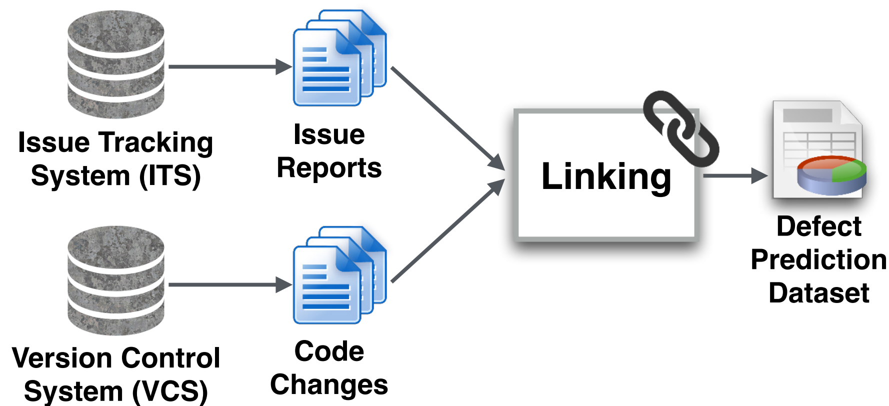


Figure 5.1.: [Empirical Study 1] The construction of defect prediction datasets.

Various data quality issues can arise when constructing defect prediction datasets. Specifically, prior work has investigated data quality issues with respect to the linkage process and the issue reports themselves. We describe the prior work with respect to each data quality issue below.

5.2.1. Linkage of Issue Reports with Code Changes

The process of linking issue reports with code changes can generate noise in defect prediction datasets, since the linkage process often depends on manually-entered links that are provided by developers. Bachmann *et al.* find that the issue reports of several defects are not identified in the commit logs [21], and thus are not visible to the automated linking tools that are used to extract defect datasets. Wu *et al.* [253] and Nguyen *et al.* [178] use the textual similarity between issue reports and version control logs to recover the missing links between the ITS and VCS repositories.

The noise generated by missing links in defect prediction datasets introduces bias. Bird *et al.* find that more experienced developers are more likely to explicitly link issue reports to the corresponding code changes [28]. Nguyen *et al.* find that such biases also exist in commercial datasets [179], which were suspected

to be "near-ideal." Rahman *et al.* examined the impact of bias on defect models by generating artificially biased datasets [192], reporting that the size of the generated dataset matters more than the amount of injected bias.

Linkage noise and bias are addressed by modern tools like JIRA[1] and IBM Jazz[2] that automatically link issue reports with code changes. Nevertheless, recent work by Nguyen *et al.* shows that even when such modern tools are used, bias still creeps into defect datasets [179]. Hence, techniques are needed to detect and cope with biases in defect prediction datasets.

5.2.2. Mislabelled Issue Reports

Even if all of the links between issue reports and code changes are correctly recovered, noise may creep into defect prediction datasets if the issue reports themselves are mislabelled. Aranda and Venolia find that ITS and VCS repositories are noisy sources of data [12]. Antoniol *et al.* find that textual features can be used to classify issue reports [11], e.g., the term "crash" is more often used in the issue reports of defects than other types of issue reports. Herzig *et al.* find that 43% of all issue reports are mislabelled, and this mislabelling impacts the ranking of the most defect-prone files [94].

Mislabelled issue reports generate noise that impacts defect prediction models. Yet, little is known about the nature of mislabelling. For example, do mislabelled issue reports truly appear at random throughout defect prediction datasets, or are they explainable using characteristics of code changes and issue reports? Knowledge of the characteristics that lead to mislabelling would help researchers to more effectively filter (or repair) mislabelled issue reports in defect prediction datasets. Hence, we formulate the following research question:

[1] https://issues.apache.org/jira/
[2] http://www.jazz.net/

(RQ1) Is mislabelling truly random?

Prior work has shown that issue report mislabelling may impact the performance of defect models. Kim *et al.* find that defect models are considerably less accurate when they are trained using datasets that have a 20%-35% mislabelling rate [118]. Seiffert *et al.* conduct a comprehensive study [205], and the results confirm the prior findings of Kim *et al.* [118].

However, prior work assumes that issue report mislabelling is random, which is not necessarily true. For example, novice developers may be more likely to mislabel an issue report than experienced developers. Hence, we set out to address the following research question:

(RQ2) How does mislabelling impact the performance of defect models?

In addition to being used for prediction, defect models are also used to understand the characteristics of defect-prone modules. Mockus *et al.* study the relationship between developer-centric measures of organizational change and the probability of customer-reported defects in the context of a large software system [161]. Cataldo *et al.* study the impact of software and work dependencies on software quality [42]. Shihab *et al.* study the characteristics of high-impact and surprise defects [214]. McIntosh *et al.* study the relationship between software quality and modern code review practices [145]. Such an understanding of defect-proneness is essential to chart quality improvement plans.

Mislabelled issue reports likely impact the interpretation of defect models as well. To investigate this, we formulate the following research question:

(RQ3) How does mislabelling impact the interpretation of defect models?

Table 5.1.: [Empirical Study 1] An overview of the studied systems. Those above the double line satisfy our criteria for analysis.

Overview				**Studied Issue Reports**				**Releases & Source Code Information**		
System Name	**Tracker Type**	**#Issues**	**Link Rate**	**#Defective Issues**	**%Mislabelled**	**#Non-Defective Issues**	**%Mislabelled**	**Releases**	**#Files**	**%Defective Files**
Jackrabbit	JIRA	2,402	79%	966	24%	922	2%	11	1,236 - 2,931	<1% - 7%
Lucene	JIRA	2,443	84%	838	29%	1205	1%	11	517 - 4,820	2% - 6%
HTTPClient	JIRA	746	31%	125	27%	106	2%	-	-	-
Rhino	Bugzilla	584	-	-	-	-	-	-	-	-
Tomcat5	Bugzilla	1,077	-	-	-	-	-	-	-	-

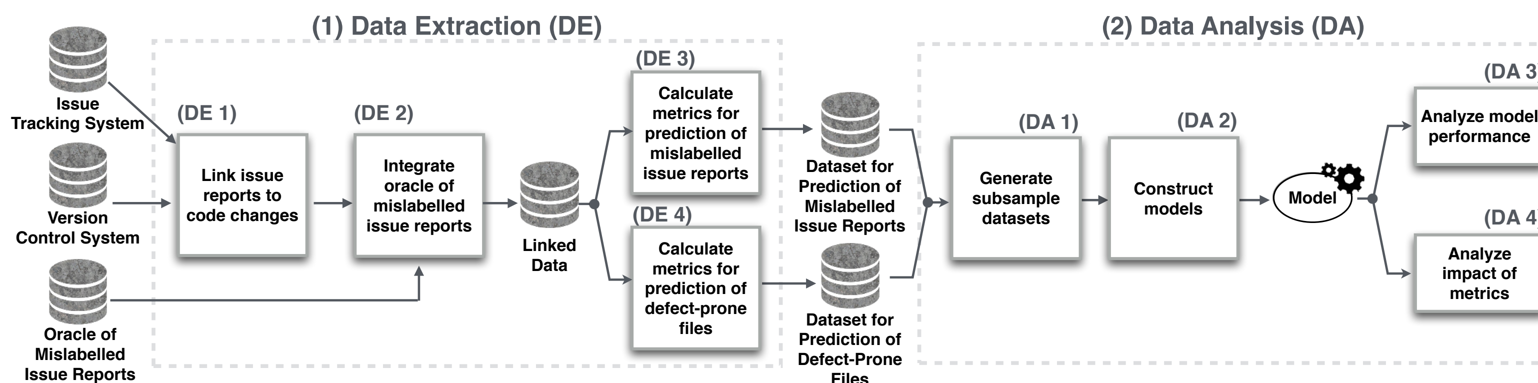


Figure 5.2.: [Empirical Study 1] An overview of our data extraction and analysis approaches.

5.3. Case Study Design

In this section, we outline our criteria for selecting the studied systems, and our data extraction and analysis approaches.

5.3.1. Studied Systems

To address our research questions, we need a dataset of mislabelled issue reports. In selecting the studied systems, we identified two important criteria that needed to be satisfied:

– **Criterion 1 — Mislabelled issue report oracle**: In order to study the impact of mislabelling on defect prediction models, we need an oracle of which issues have been mislabelled.

– **Criterion 2 — Issue report linking rate**: The issue reports for each studied system must be *traceable*, i.e., an issue report must establish a link to the code change that addresses it. Systems with low rates of traceable issue reports will introduce too many missing links [21, 28], which may impact the performance of our defect models [192]. Hence, we only study systems where a large proportion of issue reports can be mapped to the code changes that address them.

To satisfy criterion 1, we began our study using the corpus of mislabelled issue reports that was manually-curated by Herzig *et al.* [95]. Table 5.1 provides an overview of the five systems in the corpus.

To satisfy criterion 2, we first select the set of systems in the corpus of Herzig *et al.* that use the JIRA ITS.[1] JIRA explicitly links code changes to the issue reports that they address. Since Rhino and Tomcat5 do not use JIRA, we removed them from our analysis. Next, we discard systems that do not have a high linkage

rate. We discard HTTPClient, since fewer than half of the issue reports could be linked to the code changes that address them.

Table 5.1 shows that the Jackrabbit and Lucene systems satisfied our criteria for analysis. Jackrabbit is a digital content repository that stores versioned entries in a hierarchy.[3] Lucene is a library offering common search indexing functionality.[4]

5.3.2. Data Extraction

In order to produce the datasets necessary for our study, we first need to extract data from the ITS of each studied system. Next, we need to link the extracted ITS data with entries from the respective VCS repositories, as well as with the oracle of mislabelled issue reports. Figure 5.2 provides an overview of our data extraction approach, which is further divided into the four steps that we describe below.

(DE 1) Link issue reports to code changes. We first extract the issue reports from the ITS of each studied system. Then, we extract the references to code changes from those issue reports. Finally, we extract the commit information for the referenced code changes from the VCS.

(DE 2) Integrate oracle of mislabelled issue reports. We link the oracle of mislabelled issue reports with our defect datasets for two purposes. First, we record the mislabelled issues in order to train models that predict and explain the nature of mislabelling (*cf.* RQ1). Second, we use the oracle to correct mislabelled issues in order to produce clean (mislabel-free) versions of our defect prediction datasets. We use this data to study the impact of mislabelling on the performance and interpretation of our models (*cf.* RQ2 and RQ3).

(DE 3) Calculate metrics for the prediction of mislabelled issue reports. In order to address RQ1, we train models that classify whether an issue report is

[3] http://jackrabbit.apache.org/
[4] http://lucene.apache.org/

Table 5.2.: [Empirical Study 1] Factors used to study the nature of mislabelled issue reports (RQ1).

Metrics	**Description**
Diffusion Dimension	
# Files, # Components, # Subsystems	The number of unique files, components, and subsystems that are involved in the code changes that address an issue report.
Entropy	The dispersion of a change across the involved files.
Size Dimension	
# Commits	The number of commits made to address an issue report.
Churn	The sum of the added and removed lines in the code changes made to address an issue report.
History Dimension	
Reporter tendency	The proportion of prior issue reports that were previously filed by the reporter of this issue and that were mislabelled.
Code tendency	For each file involved in the code changes that address an issue report, we calculate the proportion of its prior issue reports that were mislabelled. For each issue report, we select the maximum of the proportions of each of its files.
Communication Dimension	
Discussion length	The number of comments that were posted on the issue report.

mislabelled or not. Table 5.2 shows the nine metrics that we use to predict whether an issue report is mislabelled or not. These nine metrics capture four dimensions of an issue report that we briefly describe below.

Diffusion metrics measure the dispersion of a change across modules. Since broadly-dispersed code changes may contain several different concepts, they are likely difficult to accurately label. We use four metrics to measure diffusion as described below. The *# Subsystems*, *# Components*, and *# Files* metrics measure the spread of a change at different granularities. For example, for

a file `org/apache/lucene/index/values/Reader.java`, the subsystem is `org.apache.lucene.index` and the component is `org/apache/lucene/index/values`. We count the number of unique subsystems, components, and files that are modified by a change by analyzing the file paths as described above. We also measure the *entropy* (i.e., disorder) of a change. We use the entropy definition of prior work [90, 111], i.e., the entropy of a change C is $H(C) = -\sum_{k=1}^{n}(p_k \times \log_2 p_k)$, where N is the number of files included in a change, p_k is the proportion of change C that impacts file k. The larger the entropy value, the more broadly that a change is dispersed among files.

<u>*Size*</u> metrics measure how much code change was required to address an issue report. Similar to diffusion, we suspect that larger changes may contain more concepts, which likely makes the task of labelling more difficult. We measure the size of a change by the *# commits* (i.e., the number of changes in the VCS history that are related to this issue report) and the *churn* (i.e., the sum of the added and removed lines).

<u>*History*</u> metrics measure the tendency of files and reporters to be involved with mislabelled issue reports. Files and reporters that have often been involved with mislabelled issue reports in the past are likely to be involved with mislabelled issue reports in the future. The *reporter tendency* is the proportion of prior issue reports that were created by a given reporter and were mislabelled. To calculate the *code tendency* for an issue report r, we first compute the tendency of mislabelling for each involved file f_k, i.e., the proportion of prior issue reports that involve f_k that were mislabelled. We select the maximum of the mislabelling tendencies of f_k to represent r.

<u>*Communication*</u> metrics measure the degree of discussion that occurred on an issue report. Issue reports that are discussed more are likely better understood, and hence are less likely to be mislabelled. We represent the communication dimension with the *discussion length* metric, which counts the number of comments posted on an issue report.

Table 5.3.: [Empirical Study 1] The factors that we use to build our defect models (RQ2, RQ3).

Metrics	Description
Process Metrics	
# Commits	Number of commits made to a file during a studied release.
Normalized lines added	Number of added lines in this file normalized by the sum of all lines added to all files during a studied release.
Normalized lines deleted	Number of deleted lines in this file normalized by the sum of all lines deleted from all files during a studied release.
Churn	The sum of added and removed lines in a file during a studied release.
Developer Metrics	
Active developer	Number of developers who made a change to a file during a studied release.
Distinct developer	Number of distinct developers who made a change to a file during or prior to a studied release.
Minor contributor	Number of developers who have authored less than 5% of the changes to a file.
Ownership Metrics	
Ownership ratio	The proportion of lines written by the author who made the most changes to a file.
Owner experience	The experience (i.e., the proportion of all of the lines in a project that have been written by an author) of the most active contributor to a file.
Committer experience	The geometric mean of the experiences of all of the developers that contributed to a file.

(DE 4) Calculate metrics for the prediction of defect-prone files. In order to address RQ2 and RQ3, we train defect models that identify defect-prone files. Table 5.3 shows the ten metrics that are spread across three dimensions that we use to predict defect-prone files. These metrics have been used in several previous defect prediction studies [17,29,109,163,167,174,192,212,250]. We briefly describe

each dimension below.

Process metrics measure the change activity of a file. We count the number of *commits*, *lines added*, *lines deleted*, and *churn* to measure change activity of each file. Similar to Rahman *et al.* [192], we normalize the lines added and lines deleted of a file by the total lines added and lines deleted.

Developer metrics measure the size of the team involved in the development of each file [29]. *Active developers* counts the developers who have made changes to a file during the studied release period. *Distinct developers* counts the developers who have made changes to a file up to (and including) the studied release period. *Minor developers* counts the number of developers who have authored less than 5% of the changes to a file in the studied release period.

Ownership metrics measure how much of the change to a file has been contributed by a single author [29]. *Ownership ratio* is the proportion of the changed lines to a file that have been contributed by the most active author. We measure the experience of an author using the proportion of changed lines in all of the system files that have been contributed by that author. *Owner experience* is the experience of the most active author of a file. *Committer experience* is the geometric mean of the experiences of the authors that contributed to a file.

5.3.3. Data Analysis

We train models using the datasets that we extracted from each studied system. We then analyze the performance of these models, and measure the influence that each of our metrics has on model predictions. Figure 5.2 provides an overview of our data analysis approach, which is divided into four steps. We describe each step below.

(DA 1) Generate bootstrap datasets. In order to ensure that the conclusions that we draw about our models are robust, we use the bootstrap resampling technique [63]. The bootstrap randomly samples K observations with replacement

from the original dataset of size K. Using the bootstrap technique, we repeat our experiments several times, i.e., once for each bootstrap sample. We use implementation of the bootstrap algorithm provided by the `boot` R package [41].

Unlike k-fold cross-validation, the bootstrap technique fits models using the entire dataset. Cross-validation splits the data into k equal parts, using k - 1 parts for fitting the model, setting aside 1 fold for testing. The process is repeated k times, using a different part for testing each time. Notice, however, that models are fit using k - 1 folds (i.e., a subset) of the dataset. Models fit using the full dataset are not directly tested when using k-fold cross-validation. Previous research demonstrates that the bootstrap leads to considerably more stable results for unseen data points [63, 87]. Moreover, the use of bootstrap is recommended for high-skewed datasets [87], as is the case in our defect prediction datasets.

(DA 2) Construct models. We train our models using the random forest classification technique [34]. Random forest is an accurate classification technique that is robust to noisy data [103, 217], and has been used in several previous studies [70, 75, 103, 109, 133]. The random forest technique constructs a large number of decision trees at training time. Each node in a decision tree is split using a random subset of all of the metrics. Performing this random split ensures that all of the trees have a low correlation between them. Since each tree in the forest may report a different outcome, the final class of a work item is decided by aggregating the votes from all trees and deciding whether the final score is higher than a chosen threshold. We use the implementation of the random forest technique provided by the `bigrf` R package [137].

We use the approach described by Harrell Jr. to train and test our models using the bootstrap and original samples [87]. In theory, the relationship between the bootstrap samples and the original data is asymptotically equivalent to the relationship between the original data and its population [87]. Since the population of our datasets is unknown, we cannot train a model on the original

dataset and test it on the population. Hence, we use the bootstrap samples to approximate this by using several thousand bootstrap samples to train several models, and test each of them using the original data.

Handling skewed metrics: Analysis of the distributions of our metrics reveals that they are right-skewed. To mitigate this skew, we log-transform each metric prior to training our models ($\ln(x + 1)$).

Handling redundant metrics: Correlation analysis reduces collinearity among our metrics, however it would not detect all of the redundant metrics, i.e., metrics that do not have a unique signal with respect to the other metrics. Redundant metrics will interfere with each other, distorting the modelled relationship between the module metrics and its class. We, therefore, remove redundant metrics prior to constructing our defect models. In order to detect redundant metrics, we fit preliminary models that explain each metric using the other metrics. We use the R^2 value of the preliminary models to measure how well each metric is explained by the others.

We use the implementation of this approach provided by the `redun` function of the `rms` R package. The function builds preliminary models for each metric for each bootstrap iteration. The metric that is most well-explained by the other metrics is iteratively dropped until either: (1) no preliminary model achieves an R^2 above a cutoff threshold (for this paper, we use the default threshold of 0.9), or (2) removing a metric would make a previously dropped metric no longer explainable, i.e., its preliminary model will no longer achieve an R^2 exceeding our 0.9 threshold.

Handling imbalanced categories: Table 5.1 shows that our dependent variables are imbalanced, e.g., there are more correctly labelled issue reports than mislabelled ones. If left untreated, the models trained using imbalanced data will favour the majority category, since it offers more predictive power. In our case, the models will more accurately identify correctly-labelled issue reports than mislabelled ones.

Table 5.4.: [Empirical Study 1] Example confusion matrices.

	Actual	
Classified as	**Mislabel**	**Correct**
Mislabelled	TP	FP
Correct	FN	TN

(a) [Empirical Study 1] Prediction of mislabelled issue reports.

	Actual	
Classified as	**Defective**	**Non-Defective**
Defective	TP	FP
Non-defective	FN	TN

(b) [Empirical Study 1] Prediction of defect-prone files.

To combat the bias of imbalanced categories, we re-balance the training corpus to improve the performance of the minority category. We re-balance the data using a re-sampling technique that removes samples from the majority category (under-sampling) and repeats samples in the minority category (over-sampling). We only apply re-balancing to bootstrap samples (training data) — the original (testing) data is not re-balanced.

(DA 3) Analyze model performance. To evaluate the performance of the prediction models, we use the traditional evaluation metrics in defect prediction, i.e., precision, recall, and F-measure. These metrics are calculated using the confusion matrices of Table 5.4. Precision measures the proportion of classified entities that are correct ($\frac{TP}{TP+FP}$). Recall measures the proportion of correctly classified entities ($\frac{TP}{TP+FN}$). F-measure is the harmonic mean of precision and recall ($\frac{2 \times Precision \times Recall}{Precision + Recall}$).

(DA 4) Analyze influence of metrics. To study the most influential metrics in our random forest models, we compute Breiman's variable importance score [34] for each studied metric. The larger the score, the greater the influence of the metric on our models. We use the `varimp` function of the `bigrf` R package [137] to compute the variable importance scores of our metrics.

To study the influence that the studied metrics have on our models, we apply the Scott-Knott test [203]. Each metric will have several variable importance scores (i.e., one from each of the releases). The Scott-Knott test will cluster

the metrics according to statistically significant differences in their mean variable importance scores ($\alpha = 0.05$). We use the implementation of the Scott-Knott test provided by the `ScottKnott` R package [98]. The Scott-Knott test ranks each metric exactly once, however several metrics may appear within one rank.

5.4. Case Study Results

In this section, we present the results of our case study with respect to our three research questions.

5.4.1. (RQ1) Is mislabelling truly random?

To address RQ1, we train models that indicate whether or not an issue report was mislabelled. We build two types of mislabelling models — one to predict issue reports that were incorrectly labelled as defects (defect mislabelling, i.e., *false positives*), and another to predict issue reports that should have been labelled as defects, but were not (non-defect mislabelling, i.e., *false negatives*). We then measure the performance of these models (RQ1-a) and study the impact of each of our issue report metrics in Table 5.2 (RQ1-b).

(RQ1-a) Model performance. Figure 5.3 shows the performance of 1,000 bootstrap-trained models. The error bars indicate the 95% confidence interval of the performance of the bootstrap-trained models, while the height of the bars indicates the mean performance of these models. We compare the performance of our models to random guessing.

Our models achieve a mean F-measure of 0.38-0.73, which is 4-34 times better than random guessing. Figure 5.3 also shows that our models also achieve a mean precision of 0.68-0.78, which is 6-75 times better than random guessing. Due to the scarcity of non-defect mislabelling (see Table 5.1), we observe broader ranges covered by the confidence intervals of the performance values in Figure 5.3b.

Nonetheless, the ranges covered by the confidence intervals of the precision and F-measure of all of our models does not overlap with those of random guessing. Given the skewed nature of the distributions at hand, we opt to use a bootstrap t-test, which is distribution independent. The results show that the differences are statistically significant ($\alpha = 0.05$).

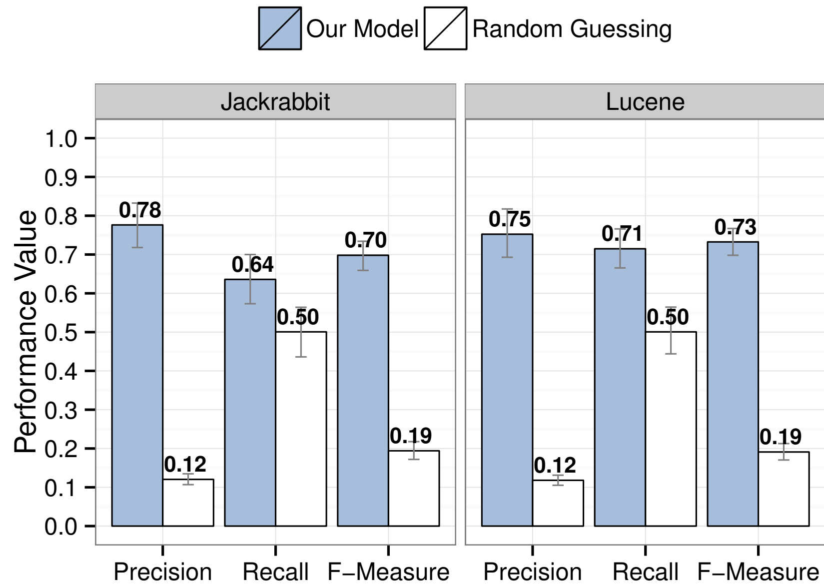
Figure 5.3b shows that the only case where our models under-perform with respect to random guessing is the non-defect mislabelling model on the Jackrabbit system. Although the mean recall of our model is lower in this case, the mean precision and F-measure are still much higher than that of random guessing.

(RQ1-b) Influence of metrics. We calculate the variable importance scores of our metrics in 1,000 bootstrap-trained models, and cluster the results using the Scott-Knott test.

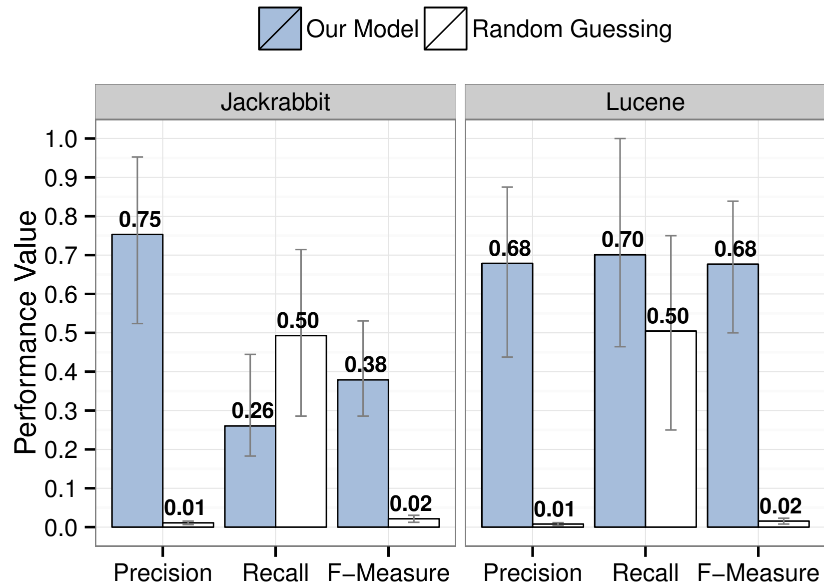
A reporter's tendency to mislabel issues in the past is the most influential metric for predicting mislabelled issue reports. We find that reporter tendency is the only metric in the top Scott-Knott cluster, indicating that it is consistently the most influential metric for our mislabelling models. Moreover, for defect mislabelling, reporter tendency is the most influential metric in 94% of our bootstrapped Jackrabbit models and 86% of our Lucene models.

Similar to RQ1-a, we find that there is more variability in the influential metrics of our non-defect mislabelling models than our defect mislabelling ones. Nonetheless, reporter tendency is still the only metric in the top Scott-Knott cluster. Furthermore, reporter tendency is the most influential metric in 46% of our Jackrabbit models and 73% of our Lucene models.

Summary. Issue report mislabelling is not random. Our models can predict mislabelled issue reports with a mean F-measure that is 4-34 times better than that of random guessing. The tendency of a reporter to mislabel issues in the past is consistently the most influential metric used by our models.



(a) [Empirical Study 1] Defect mislabelling (false positive)



(b) [Empirical Study 1] Non-defect mislabelling (false negative)

Figure 5.3.: [Empirical Study 1] A comparison of the performance of our models that are trained to identify mislabelled issue reports (blue) against random guessing (white). Error bars indicate the 95% confidence interval based on 1,000 bootstrap iterations.

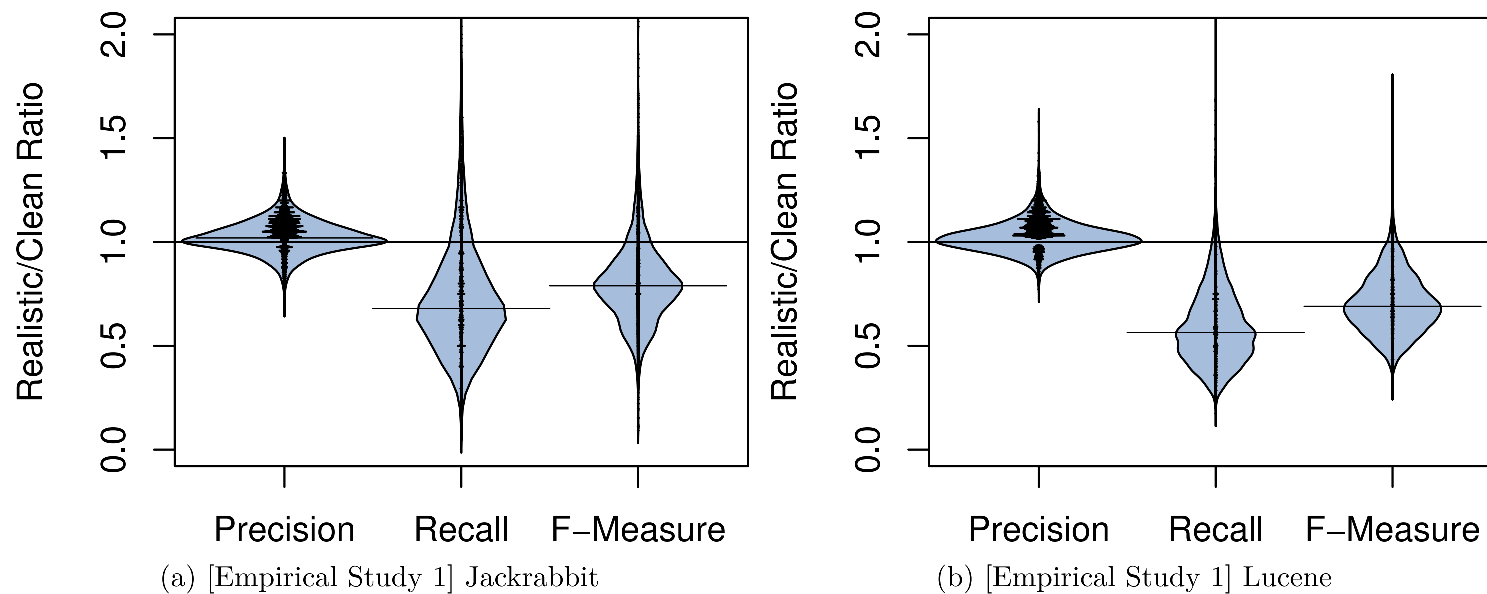
5.4.2. (RQ2) How does mislabelling impact the performance of defect models?

Approach. We use the same high-level approach to address RQ2 and RQ3. Figure 5.6 provides an overview of the steps in that approach. We describe how we implement each step to address RQ2 in particular below.

(Step 1) Construct models: For each bootstrap iteration, we train models using clean, realistic noisy, and random noisy samples. The clean sample is the unmodified bootstrap sample. The realistic noisy sample is generated by re-introducing the mislabelled issue reports in the bootstrap sample. To generate the random noisy sample, we randomly inject mislabelled issue reports in the bootstrap sample until the rate of mislabelled issue reports is the same as the realistic noisy sample. Finally, we train models on each of the three samples.

(Step 2) Analyze models: We want to measure the impact of real mislabelling and random mislabelling on defect prediction. Thus, we compute the ratio of the performance of models that are trained using the noisy samples to that of the clean sample. Since we have three performance measures, we generate six ratios for each bootstrap iteration, i.e., the precision, recall, and F-measure ratios for realistic noisy and random noisy samples compared to the clean sample.

(Step 3) Interpret results: We repeat the bootstrap experiment for each studied release individually. Finally, we compare the distributions of each performance ratio using beanplots [112]. Beanplots are boxplots in which the vertical curves summarize the distributions of different data sets. The horizontal lines indicate the median values. We choose beanplots over boxplots, since beanplots show contours of the data that are hidden by the flat edges of boxplots.



(a) [Empirical Study 1] Jackrabbit

(b) [Empirical Study 1] Lucene

Figure 5.4.: [Empirical Study 1] The difference in performance between models trained using realistic noisy samples and clean samples. All models are tested on clean samples (defect mislabelling).

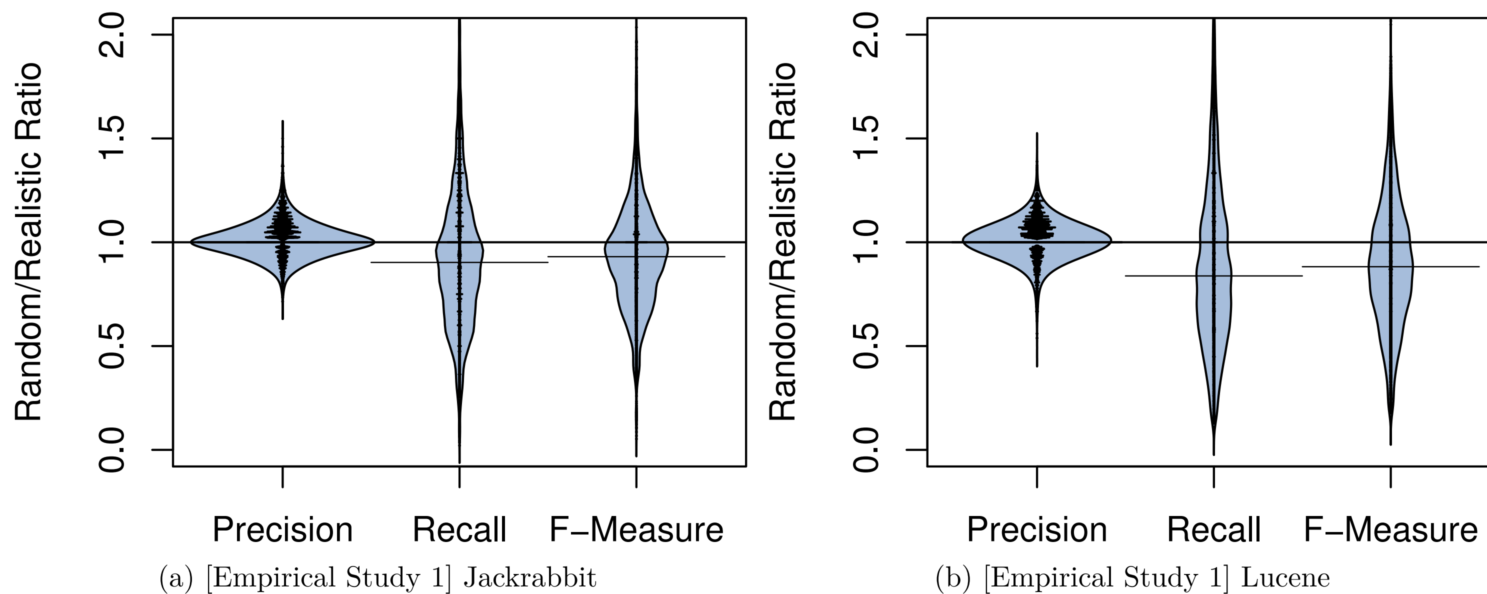
Results. Figure 5.4 shows the distribution of the ratios of our performance metrics in all of the studied releases.

Similar to RQ1, we perform experiments for defect mislabelling and non-defect mislabelling individually. We find that, likely due to scarcity, non-defect mislabelled issue reports have little impact on our models. Hence, we focus on defect mislabelling for the remainder of this section.

The modules classified as defective by models trained using noisy data are typically as reliable as the modules classified as defective by models trained using clean data. Figure 5.4 shows that there is a median ratio of one between the precision of models trained using the realistic noisy and clean samples for both of the studied systems. Furthermore, we find that the 95% confidence interval for the distributions are 0.88-1.20 (Jackrabbit) and 0.90-1.19 (Lucene). This tight range of values that are centred at one suggests that the precision of our models is not typically impacted by mislabelled defects.

On the other hand, models trained using noisy data tend to miss more defective modules than models trained using clean data. Figure 5.4 shows that the median ratio between the recall of models trained using the realistic noisy and clean samples is 0.68 (Jackrabbit) and 0.56 (Lucene). This indicates that models trained using data with mislabelled defects typically achieve 56%-68% of the recall that models trained on clean data would achieve when tested on clean data.

Summary. While defect mislabelling rarely impacts the precision of defect models, the recall is often impacted. Practitioners can rely on the modules classified as defective by defect models trained on noisy data. However, cleaning historical data prior to training defect models will likely improve their recall.



(a) [Empirical Study 1] Jackrabbit

(b) [Empirical Study 1] Lucene

Figure 5.5.: [Empirical Study 1] The difference in performance between models trained using random noisy and realistic noisy samples. All models are tested on clean samples (defect mislabelling).

Random mislabelling issue reports tends to overestimate the impact of realistic mislabelling on model performance. Figure 5.5 shows that while the median ratio between the precision of realistic and random noisy models is 1 for both studied systems, the median recall and F-measure ratios are 0.84-0.90 and 0.88-0.93 respectively. In fact, 64%-66% of the recall and F-measure ratios are below 1 in our studied systems, indicating that models trained using randomly mislabelled issues tend to overestimate the impact of real mislabelling on the recall and F-measure of our models.

Summary. When randomly injecting mislabelled defects, our results suggest that the impact of the mislabelling will be overestimated by 7-16 percentage points.

5.4.3. (RQ3) How does mislabelling impact the interpretation of defect models?

Approach. We again use the high-level approach of Figure 5.6 to address RQ3. While Step 1 of the approach is identical for RQ2 and RQ3, Steps 2 and 3 are performed differently. We describe the different Steps 2 and 3 below.

(Step 2) Analyze models: For each bootstrap iteration, we calculate the variable importance score for each metric in each type of model (i.e., clean, realistic noisy, and random noisy). Hence, the variable importance score for each metric is calculated three times in each bootstrap iteration.

(Step 3) Interpret results: We cluster the variable importance scores of metrics in each type of model using Scott-Knott tests to produce statistically distinct ranks of metrics for clean, realistic noisy, and random noisy models. Thus, each metric has a rank for each type of model.

To estimate the impact of random and realistic mislabelling have on model interpretation, we compute the difference in the ranks of the metrics that appear in the top-three ranks of the clean models. For example, if a metric m appears in

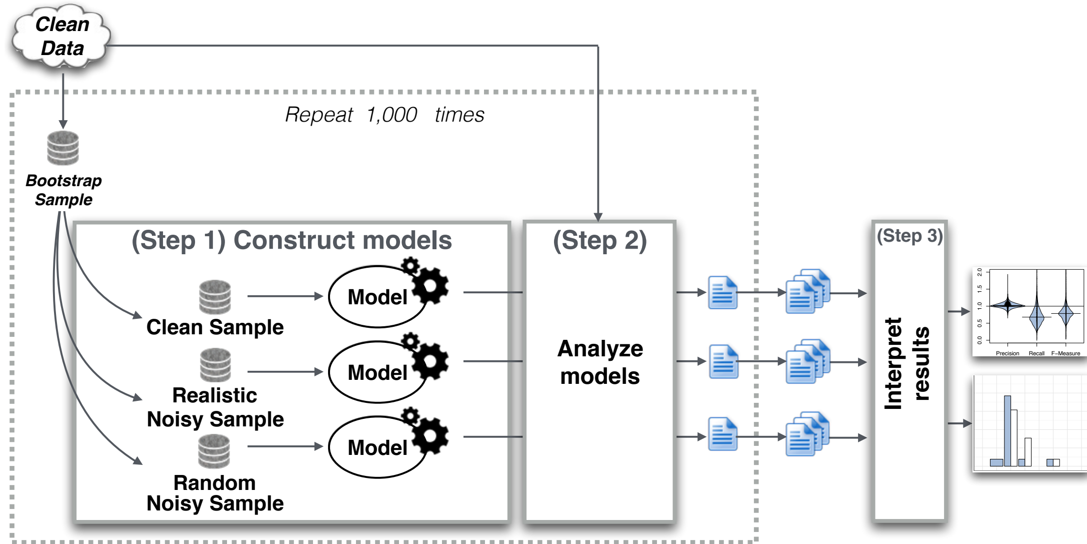


Figure 5.6.: [Empirical Study 1] An overview of our approach to study the impact of issue report mislabelling.

the top rank in the clean and realistic noisy models then the metric would have a rank difference of zero. However, if m appears in the third rank in the random noisy model, then the rank difference of m would be negative two.

Similar to RQ2, we repeat the whole experiment for each studied release individually.

Results. Figure 5.7 shows the rank differences for all of the studied releases. We again perform experiments for defect mislabelling and non-defect mislabelling individually. The scarcity of non-defect mislabelling limits the impact of it can have on model interpretation. Indeed, we find that there are very few rank differences in the non-defect mislabelling results. Hence, we focus on defect mislabelling for the remainder of this section.

The most influential metrics are generally robust to the noise that is introduced by defect mislabelling. Figure 5.7 shows that 80% (Lucene) to 85% (Jackrabbit) of the metrics in the top rank of the clean model (most often, the committer experience) also appear in the top rank of the realistic noisy model.

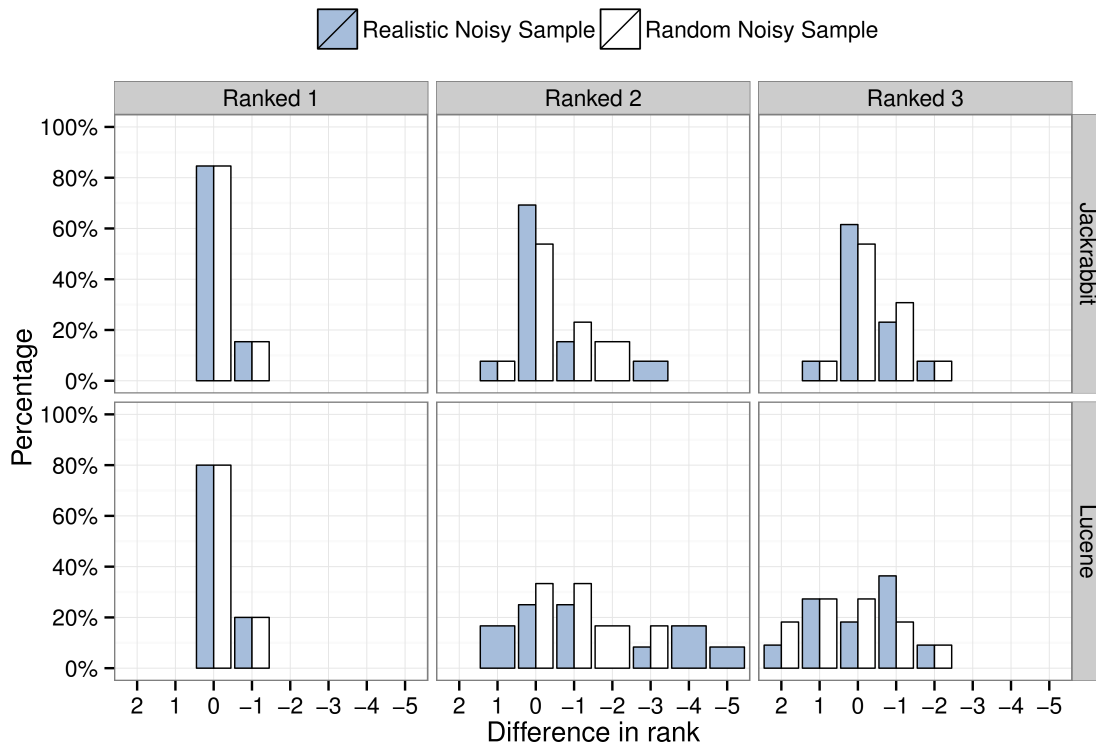


Figure 5.7.: [Empirical Study 1] The difference in the ranks for the metrics according to their variable importance scores among the clean and noisy models. The bars indicate the percentage of variables that appear in that rank in the clean model while also appearing in that rank in the noisy models.

Similarly, 80% (Lucene) to 85% (Jackrabbit) of the metrics in the top rank of the clean model appear in the top rank of the random noisy model. Moreover, the 10%-15% of metrics in the top rank of the clean model that do not appear in the top rank of the noisy models only decrease by one rank.

Conversely, the metrics in the second and third ranks are less stable. Figure 5.7 shows that 31% (Jackrabbit) to 75% (Lucene) of the metrics in the second rank and 38% (Jackrabbit) to 82% (Lucene) of the metrics in the third rank of the clean model (most often, the process and developer metrics) do not appear in the second rank of the realistic noisy model, indicating that these metrics are

influenced by defect mislabelling. Furthermore, 8%-33% of the second and third rank variables drop by two or more ranks in the noisy models.

Summary. The most influential metrics are generally robust to defect mislabelling, with 80%-85% of the most influential metrics from the clean models appearing in the top rank of the noisy models as well. On the other hand, the second and third ranks are unstable, with as little as 18% of the metrics from the clean models appearing in those ranks in the noisy models.

Randomly injected mislabelled defects have a more damaging impact on model interpretation than real mislabelled defects do. Figure 5.7 shows that a smaller percentage of the metrics of the clean models are found at the same rank in the random noisy models than the realistic noisy models.

Summary. Randomly injecting mislabelled defects tends to distort the interpretation of influential metrics more than truly mislabelled defects do.

5.5. Discussion & Threats to Validity

We now discuss the results of our case study with respect to other work on issue report mislabelling, as well as the threats to the validity of our case study.

5.5.1. Discussion

In prior work, Herzig *et al.* show that issue report mislabelling has a drastic impact on the relative order of the most defect-prone files [94] — 16%-40% of the top-10% most defect-prone files do not belong in that group. The impact of issue report mislabelling on the ordering of the most defect-prone files suggests that defect models (such as the ones that we build in this study) will also be drastically impacted, both in terms of precision and recall.

Yet in this study, we find that issue report mislabelling has little impact on the precision of defect models, which may seem to be incongruent with the prior work. We suspect that the differences in the conclusions that we draw have to do with the differences in our defect prediction experiments.

In the study of Herzig *et al.*, files are ranked according to the number of defect reports that are mapped to a file. The files at the top of this ranked list are the most defect-prone, and would yield the most benefit from additional quality assurance effort [91]. Instability in the top-10% of files in this ranked list occurs if these highly defect-prone file have several mislabelled defects mapped to them.

On the other hand, our defect models classify whether a file is defective or clean. In order for a file to be remapped from defective to clean, all of the defects that are mapped to a file must be mislabelled, reducing the number of defects to zero. Otherwise, a file would still be considered defective. Hence, the instability that Herzig *et al.* observe with respect to the most defect-prone files may not have as much of an impact on the files that our defect models will consider defective.

5.5.2. Threats to Validity

External validity. We focus our study on two subject systems, due to the low number of systems that satisfied our analysis criteria (*cf.* Section 5.3). The lack of a curated oracle of mislabelled issue reports presented a major challenge. Nonetheless, additional replication studies are needed.

Construct validity. Although the studied datasets have high link rates of issue reports and code changes, we make the implicit assumption that these links are correct. On the other hand, we rely on JIRA links from issue reports to code changes, which others have noted lead to more accurate links than links constructed from code changes to issue reports [192].

Internal validity. We use nine metrics to train models that identify mislabelled issue reports, and ten metrics to train models that identify defective files. We se-

lected metrics that cover a variety of dimensions for each type of model. However, other metrics that we may have overlooked could also improve the performance of our models.

We focus on the random forest classification technique. Although prior studies have also used random forest [70,75,103,109,133], our findings are entirely bound to this technique. Future research should explore the impact of issue report mislabelling on other classification techniques.

5.6. Chapter Summary

Defect models identify potentially defective software modules. However, the accuracy of the predictions and the insights derived from defect models depend on the quality of the data from which these models are trained. While recent work shows that issue report mislabelling may impact the performance of defect prediction models [118,192], the mislabelled issue reports were generated randomly.

In this chapter, we study the nature of mislabelled issue reports and the impact of truly mislabelled issue reports on the performance and interpretation of defect models. Through a case study of two large and successful open source systems, we make the following observations:

– Mislabelling is not random. Models trained to identify mislabelled issue reports achieve a mean F-measure that is 4-34 times better than that of random guessing. A reporter's tendency to mislabel issues in the past is consistently the most influential metric used by our models.

– Since we observe that the precision of our defect models is rarely impacted by defect mislabelling, practitioners can rely on the accuracy of modules labelled as defective by defect models that are trained using noisy data — the files that are classified as defect-prone by models trained using noisy data are often just as accurate as the defect-prone predictions of models

trained using clean data (i.e., mislabel-free). However, cleaning the data prior to training defect models will likely allow them to identify more of the truly defective modules.

– The most influential metrics are generally robust to defect mislabelling. 80%-85% of the most influential metrics from the clean models appear in the top ranks of the noisy models as well.

– On the other hand, the second- and third-most influential metrics are more unstable than the most influential ones. As little as 18% of the metrics in the second and third influence rank of the clean models also appear in the same rank in the noisy models.

– Randomly injecting mislabelled defects tends to overestimate the impact of defect mislabelling on the performance and interpretation of defect models.

5.6.1. Concluding Remarks

The focus of this chapter is on the impact of the noise generated by issue report mislabelling. Unlike prior study assumes that mislabelling is random and such random noise has a large negative impact [118], we find that realistic noise that is generated by issue report mislabelling is non-random and such realistic noise has little impact on the performance and interpretation of defect prediction models.

Even if the noise in defect datasets has been mitigated, the performance of defect prediction models still relies heavily on the used classification techniques. Such classification techniques often have configurable parameters that control the characteristics of classification techniques. We begin, in the next chapter, by studying the impact of the parameter settings of classification techniques on the performance, model stability, and interpretation of defect prediction models.

CHAPTER 6

The Impact of Automated Parameter Optimization

Key Finding

Automated parameter optimization substantially improves the performance, model stability, as well as, the interpretation of defect prediction models.

An earlier version of the work in this chapter appears in the Proceedings of the International Conference on Software Engineering (ICSE), 2016 [232].

6.1. Introduction

The limited Software Quality Assurance (SQA) resources of software organizations must focus on software modules (e.g., source code files) that are likely to be defective in the future. To that end, defect prediction models are trained to identify defect-prone software modules using statistical or machine learning classification techniques.

Such classification techniques often have configurable parameters that control the characteristics of the classifiers that they produce. For example, the number of decision trees of which a random forest classifier is comprised can be configured prior to training the forest. Furthermore, the number of non-overlapping clusters of which a k-nearest neighbours classifier is comprised must be configured prior to using the classification technique.

Since the optimal settings for these parameters are not known ahead of time, the settings are often left at default values. Prior work suggests that defect prediction models may underperform if they are trained using suboptimal parameter settings. For example, Jiang *et al.* [103] and Tosun *et al.* [242] also point out that the default parameter settings of random forest and naïve bayes are often suboptimal. Koru *et al.* [78] and Mende *et al.* [147, 148] show that selecting different parameter settings can impact the performance of defect models. Hall *et al.* [82] show that unstable classification techniques may underperform due to the use of default parameter settings. Mittas *et al.* [159] and Menzies *et al.* [154] argue that unstable classification techniques can make replication of defect prediction studies more difficult.

Indeed, we perform a literature analysis that reveals that 26 of the 30 most commonly used classification techniques (87%) require at least one parameter setting. Since such parameter settings may impact the performance of defect prediction models, the settings should be carefully selected. However, it is impractical to assess all of the possible settings in the parameter space of a single classification

technique [24, 86, 122]. For example, Kocaguneli *et al.* [122] point out that there are at least 17,000 possible settings to explore when training k-nearest neighbours classifier.

In this chapter, we investigate the performance, stability, and interpretation of defect prediction models where Caret [131] — an off-the-shelf automated parameter optimization technique — has been applied. Caret evaluates candidate parameter settings and suggests the optimized setting that achieves the highest performance. Through a case study of 18 datasets from systems that span both proprietary and open source domains, we record our observations with respect to five dimensions:

(1) Performance improvement: Caret improves the AUC performance of defect prediction models by up to 40 percentage points. Moreover, the performance improvement provided by applying Caret is non-negligible for 16 of the 26 studied classification techniques (62%).

(2) Performance stability: Caret-optimized classifiers are at least as stable as classifiers that are trained using the default parameter settings. Moreover, the Caret-optimized classifiers of 9 of the 26 studied classification techniques (35%) are significantly more stable than classifiers that are trained using the default parameter settings.

(3) Model interpretation: Caret substantially shifts the ranking of the importance of software metrics, with as little as 28% of the variables from the Caret-optimized models appearing in those ranks in the default models. Moreover, classification techniques where Caret has a larger impact on the performance are also subject to large shifts in interpretation.

(4) Parameter transferability: The Caret-suggested setting of 85% of the 20 most sensitive parameters can be applied to datasets that share a similar set of metrics without a statistically significant drop in performance.

(5) Computational cost: Caret adds less than 30 minutes of additional computation time to 65% of the studied classification techniques.

Since we find that parameter settings can have such a substantial impact on model performance and model interpretation, we revisit prior analyses that rank classification techniques by their ability to yield top-performing defect prediction models. We find that Caret increases the likelihood of producing a top-performing classifier by as much as 83%, suggesting that automated parameter optimization can substantially shift the ranking of classification techniques.

Our results lead us to conclude that parameter settings can indeed have a large impact on the performance, stability, and interpretation of defect prediction models, suggesting that researchers should experiment with the parameters of the classification techniques. Since automated parameter optimization techniques like Caret yield substantial benefits in terms of performance improvement, stability, and interpretation, while incurring a manageable additional computational cost, they should be included in future defect prediction studies.

6.1.1. Contributions

This chapter makes the following contributions:

- A large collection of 43 parameters that are derived from 26 of the most frequently-used classification techniques in the context of defect prediction.
- The improvement and stability of the performance of defect prediction models when automated parameter optimization is applied.
- An investigation of the impact of parameter settings of classification techniques on the interpretation of defect prediction models when automated parameter optimization is applied.

– An investigation of parameter transferability in a cross-context prediction setting when automated parameter optimization is applied.

– An in-depth discussion of the broader implications of our findings with respect to software defect prediction modelling and search-based software engineering (Section 6.7).

– The introduction of a generic variable importance calculation that applies to the 26 studied classification techniques (Section 6.4.6).

6.1.2. Chapter Organization

The remainder of the chapter is organized as follows. Section 6.2 illustrates the importance of parameter settings of classification techniques for defect prediction models. Section 6.3 positions this chapter with respect to the related work. Section 6.4 presents the design and approach of our case study. Section 6.5 presents the results of our case study with respect to our five research questions. Section 6.6 revisits prior analyses that rank classification techniques by their likelihood of producing top-performing defect prediction models. Section 6.7 discusses the broader implications of our findings with related work. Section 6.8 discloses the threats to the validity of our study. Finally, Section 6.9 draws conclusions.

6.2. The Relevance of Parameter Settings for Defect Prediction Models

A variety of classification techniques are used to train defect prediction models. Since some classification techniques do not require parameter settings (e.g., logistic regression), we first assess whether the most commonly used classification techniques require parameter settings.

Table 6.1.: [Empirical Study 2] Overview of studied parameters of classification techniques. [N] denotes a numeric value; [L] denotes a logical value; [F] denotes a factor value. The default values are shown in bold typeface and correspond to the default values of the Caret R package.

Family	**Family Description**	**Parameter Name**	**Parameter Description**	**Techniques that apply with their default (in bold typeface) and candidate parameter values.**
Naive Bayes	Naive Bayes is a probability model that assumes that predictors are independent of each other [128]. **Techniques:** Naive Bayes (NB).	Laplace Correction	[N] Laplace correction (0 indicates no correction).	NB={**0**}
		Distribution Type	[L] TRUE indicates a kernel density estimation, while FALSE indicates a normal density estimation.	NB={TRUE, **FALSE**}
Nearest Neighbour	Nearest neighbour is an algorithm that stores all available observations and classifies new observations based on its similarity to prior observations [128]. **Techniques:** k-Nearest Neighbour (KNN).	#Clusters	[N] The numbers of non-overlapping clusters to produce.	KNN={**1**, 5, 9, 13, 17}

Continued on next page

Table 6.1 – continued from previous page

Family	**Family Description**	**Parameter Name**	**Parameter Description**	**Techniques that apply with their default (in bold typeface) and candidate parameter values.**
Regression	Logistic regression is a technique for explaining binary dependent variables. MARS is a non-linear regression modelling technique [66]. **Techniques:** GLM and MARS.	Degree Interaction	[N] The maximum degree of interaction (Friedman's mi). The default is 1, meaning build an additive model (i.e., no interaction terms).	MARS={**1**}
Partial Least Squares	Partial Least Squares regression generalizes and combines features from principal component analysis and multiple regression [236]. **Techniques:** Generalized Partial Least Squares (GPLS).	#Components	[N] The number of PLS components.	GPLS={**1**, 2, 3, 4, 5}

Continued on next page

Table 6.1 – continued from previous page

Family	**Family Description**	**Parameter Name**	**Parameter Description**	**Techniques that apply with their default (in bold typeface) and candidate parameter values.**
Neural Network	Neural network techniques are used to estimate or approximate functions that can depend on a large number of inputs and are generally unknown [218]. **Techniques:** Standard (NNet), Model Averaged (AVNNet), Feature Extraction (PCANNet), Radial Basis Functions (RBF), Multi-layer Perceptron (MLP), Voted-MLP (MLPWeightDecay), and Penalized Multinomial Regression (PMR).	Bagging	[L] Should each repetition apply bagging?	AVNNet={TRUE, **FALSE**}
		Weight Decay	[N] A penalty factor to be applied to the errors function.	MLPWeightDecay, PMR, AVNNet, NNet, PCANNet={**0**, 0.0001, 0.001, 0.01, 0.1}, SVMLinear={**1**}
		#Hidden Units	[N] Numbers of neurons in the hidden layers of the network that are used to produce the prediction.	MLP, MLPWeightDecay, AVNNet, NNet, PCANNet={**1**, 3, 5, 7, 9}, RBF={**11**, 13, 15, 17, 19}

Continued on next page

Table 6.1 – continued from previous page

Family	Family Description	Parameter Name	Parameter Description	Techniques that apply with their default (in bold typeface) and candidate parameter values.
Discrimination Analysis	Discriminant analysis applies different kernel functions (e.g., linear) to classify a set of observations into predefined classes based on a set of predictors [65]. **Techniques:** Linear Discriminant Analysis (LDA), Penalized Discriminant Analysis (PDA), and Flexible Discriminant Analysis (FDA).	Product Degree	[N] The number of degrees of freedom that are available for each term.	FDA={**1**}
		Shrinkage Penalty Coefficient	[N] A shrinkage parameter to be applied to each tree in the expansion (a.k.a., learning rate or step-size reduction).	PDA={**1**, 2, 3, 4, 5}
		#Terms	[N] The number of terms in the model.	FDA={**10**, 20, 30, 40, 50}
Rule	Rule-based techniques transcribe decision trees using a set of rules for classification [128]. **Techniques:** Rule-based classifier (Rule), and Ripper classifier (Ripper).	#Optimizations	[N] The number of optimization iterations.	Ripper={1, **2**, 3, 4, 5}

Continued on next page

Table 6.1 – continued from previous page

Family	**Family Description**	**Parameter Name**	**Parameter Description**	**Techniques that apply with their default (in bold typeface) and candidate parameter values.**
Decision Trees-Based	Decision trees use feature values to classify instances [128]. **Techniques:** C4.5-like trees (J48), Logistic Model Trees (LMT), and Classification And Regression Trees (CART).	Complexity	[N] A penalty factor to be applied to the error rate of the terminal nodes of the tree.	CART={0.0001, 0.001, **0.01**, 0.1, 0.5}
		Confidence	[N] The confidence factor used for pruning (smaller values incur more pruning).	J48={**0.25**}
		#Iterations	[N] The numbers of iterations.	LMT={**1**, 21, 41, 61, 81}
SVM	Support Vector Machines (SVMs) use a hyperplane to separate two classes (i.e., defective or not) [128]. **Techniques:** SVM with Linear kernel (SVMLinear), and SVM with Radial basis function kernel (SVMRadial).	Sigma	[N] The width of Gaussian kernels.	SVMRadial={0.1, 0.3, **0.5**, 0.7, 0.9}
		Cost	[N] A penalty factor to be applied to the number of errors.	SVMRadial={0.25, 0.5, **1**, 2, 4}, SVMLinear={**1**}

Continued on next page

Table 6.1 – continued from previous page

Family	**Family Description**	**Parameter Name**	**Parameter Description**	**Techniques that apply with their default (in bold typeface) and candidate parameter values.**
Bagging	Bagging methods combine different base learners together to solve one problem [254]. **Technique:** Random Forest (RF), Bagged CART (BaggedCART)	#Trees	[N] The numbers of classification trees.	RF={**10**, 20, 30, 40, 50}
Boosting	Boosting performs multiple iterations, each with different example weights, and makes predictions using voting of classifiers [33]. **Techniques:** Gradient Boosting Machine (GBM), Adaptive Boosting (AdaBoost), Generalized linear and Additive Models Boosting (GAMBoost), Logistic Regression Boosting (LogitBoost), eXtreme Gradient Boosting Tree (xGBTree), and C5.0.	#Boosting Iterations	[N] The numbers of iterations that are used to construct models.	C5.0={**1**, 10, 20, 30, 40}, GAMBoost={**50**, 100, 150, 200, 250}, LogitBoost={**11**, 21, 31, 41, 51}, GBM,xGBTree={50, **100**, 150, 200, 250}
		#Trees	[N] The numbers of classification trees.	AdaBoost={**50**, 100, 150, 200, 250}

Continued on next page

Table 6.1 – continued from previous page

Family	Family Description	Parameter Name	Parameter Description	Techniques that apply with their default (in bold typeface) and candidate parameter values.
		Shrinkage	[N] A shrinkage factor to be applied to each tree in the expansion (a.k.a., learning rate reduction).	GBM={**0.1**}, xGB-Tree={**0.3**}
		Max Tree Depth	[N] The maximum depth per tree.	AdaBoost, GBM, xGB-Tree={**1**, 2, 3, 4, 5}
		Min. Terminal Node Size	[N] The minimum terminal nodes in trees.	GBM={**10**}
		Winnow	[L] Should predictor winnowing (i.e feature selection) be applied?	C5.0={**FALSE**, TRUE}
		AIC Prune?	[L] Should pruning using stepwise feature selection be applied?	GAMBoost={**FALSE**, TRUE}

Continued on next page

Table 6.1 – continued from previous page

Family	**Family Description**	**Parameter Name**	**Parameter Description**	**Techniques that apply with their default (in bold typeface) and candidate parameter values.**
		Model Type	[F] Either tree for the predicted class or rules for model confidence values.	C5.0={**rules**, tree}

We begin with the 6 families of classification techniques that are used by Lessmann *et al.* [133]. Based on a recent literature review of Laradji *et al.* [132], we add 5 additional families of classification techniques that have been recently used in defect prediction studies. In total, we study 30 classification techniques that span 11 classifier families. Table 6.1 provides an overview of the 11 families of classification techniques.

Our literature analysis reveals that 26 of the 30 most commonly used classification techniques require at least one parameter setting. Table 6.1 provides an overview of the 25 unique parameters that apply to the studied classification techniques.

Summary. 26 of the 30 most commonly used classification techniques require at least one parameter setting, indicating that selecting an optimal parameter setting for defect prediction models is an important experimental design choice.

6.3. Related Work & Research Questions

Recent research has raised concerns about parameter settings of classification techniques when applied to defect prediction models. For example, Koru *et al.* [78] and Mende *et al.* [147, 148] point out that selecting different parameter settings can impact the performance of defect models. Jiang *et al.* [103] and Tosun *et al.* [242] also point out that the default parameter settings of research toolkits (e.g., R [190], Weka [81], Scikit-learn [185], MATLAB [143]) are suboptimal.

Although prior work suggests that defect prediction models may underperform if they are trained using suboptimal parameter settings, parameters are often left at their default values. For example, Mende *et al.* [150] use the default number of decision trees to train a random forest classifier (provided by an R package). Weyuker *et al.* [250] also train defect models using the default setting of C4.5 that

is provided by Weka. Jiang *et al.* [101] and Bibi *et al.* [27] also use the default value of the k-nearest neighbours classification technique ($k = 1$). In our prior work [74, 230], we ourselves have also used default classification settings.

In addition, the implementations of classification techniques that are provided by different research toolkits often use different default settings. For example, for the number of decision trees of the random forest technique, the default varies among settings of 10 for the `bigrf` R package [137], 50 for MATLAB [143], 100 for Weka [81], and 500 for the `randomForest` R package [135]. Moreover, for the number of hidden layers of the neural networks techniques, the default varies among settings of 1 for the `neuralnet` R package [68], 2 for Weka [81] and the `nnet` R package [200], and 10 for MATLAB [143]. Such variation among default settings of different research toolkits may influence conclusions of defect prediction studies [229].

There are many empirical studies in the area of Search-Based Software Engineering (SBSE) [24, 85, 86, 122] that aim to optimize software engineering tasks (e.g., software testing [99]). However, little is known about how the performance difference of defect prediction models tend to be when automated parameter optimization is applied. Thus, we formulate the following research question:

(RQ1) How much does the performance of defect prediction models change when automated parameter optimization is applied?

Recent research voices concerns about the stability of performance estimates that are obtained from classification techniques when applied to defect prediction models. For example, Menzies *et al.* [154] and Mittas *et al.* [159] argue that unstable classification techniques can make replication of defect prediction studies more difficult. Shepperd *et al.* [207], and Jorgensen *et al.* [107] also point out that the unstable performance estimates that are produced by classification techniques may introduce bias, which can mislead different research groups to

draw erroneous conclusions. Mair *et al.* [141] and Myrtveit *et al.* [169] show that high variance in performance estimates from classification techniques is a critical problem in comparative studies of prediction models. Song *et al.* [220] also show that applying different settings to unstable classification techniques will provide different results.

Like any form of classifier optimization, automated parameter optimization may increase the risk of *overfitting*, i.e., producing a classifier that is too specialized for the data from which it was trained to apply to other datasets. To investigate whether parameter optimization is impacting the stability of defect prediction models, we formulate the following research question:

(RQ2) How stable is the performance of defect prediction models when automated parameter optimization is applied?

In addition to being used for prediction, defect models are also used to understand the characteristics of defect-prone modules. For example, Bettenburg *et al.* [25] study the relationship between social interactions and software quality. Shihab *et al.* [214] study the characteristics of high impact and surprising defects. McIntosh *et al.* [145, 146], Thongtanunam *et al.* [238], Morales *et al.* [166], and Kononenko *et al.* [126] study the relationship between code review practices and software quality. Such an understanding of defect-proneness is essential to design effective quality improvement plans.

Recent research has draws into question about the accuracy of insights that are derived from defect prediction models. For example, our recent work shows that issue report mislabelling has a large impact on the interpretation of defect prediction models [230]. To investigate whether default parameter settings are impacting the interpretation of defect prediction models, we formulate the following research question:

(RQ3) How much does the interpretation of defect prediction models change when automated parameter optimization is applied?

Recent research applies parameter settings of classification techniques that perform well on one dataset to another. For example, Tan *et al.* [227] explore different settings to identify the optimal setting and apply it to several datasets. Jiang *et al.* [100] also experiment with various settings on one dataset, since the total execution time of a classification technique (i.e., MARS) can take several hours. However, Ghotra *et al.* [74], Khoshgoftaar *et al.* [115], and Song *et al.* [221] show that the performance of defect prediction models often depend on the characteristics of defect datasets that are used to train it. Gao *et al.* [71] also argue that classification techniques are context-specific. Yet, little is known about whether the optimal parameter settings that are obtained for one dataset are transferable to another. Knowing the transferability of the optimal parameter settings of the most sensitive classification techniques will shed light onto whether automated parameter optimization can be safely omitted (by opting to use these transferable setting instead) without incurring a significant drop in performance. To investigate how well the optimal parameter settings transfer from one context to another, we formulate the following research question:

(RQ4) How well do optimal parameter settings transfer from one context to another?

In addition to the benefits of applying automated parameter optimization, one must also consider the additional computational cost to provide a balance understanding of the practicality of the optimization. Menzies *et al.* [152] point out that some classification techniques can be very slow and exploring various settings would require a large computational cost. Ma *et al.* [140] point out that

the computational cost depends on to the size of training dataset. Yet, little is known about how the cost of applying automated parameter optimization tend to be. Hence, we formulate the following research question:

(RQ5) What is the cost of applying automated parameter optimization?

Project	System	Defective Ratio	#Files	#Metrics	EPV
NASA	JM1[1]	21%	7,782	21	80
	PC5[1]	28%	1,711	38	12
Proprietary	Prop-1[2]	15%	18,471	20	137
	Prop-2[2]	11%	23,014	20	122
	Prop-3[2]	11%	10,274	20	59
	Prop-4[2]	10%	8,718	20	42
	Prop-5[2]	15%	8,516	20	65
Apache	Camel 1.2[2]	36%	608	20	11
	Xalan 2.5[2]	48%	803	20	19
	Xalan 2.6[2]	46%	885	20	21
Eclipse	Platform 2.0[3]	14%	6,729	32	30
	Platform 2.1[3]	11%	7,888	32	27
	Platform 3.0[3]	15%	10,593	32	49
	Debug 3.4[4]	25%	1,065	17	15
	SWT 3.4[4]	44%	1,485	17	38
	JDT[5]	21%	997	15	14
	Mylyn[5]	13%	1,862	15	16
	PDE[5]	14%	1,497	15	14

[1]Provided by Shepperd *et al.* [209].
[2]Provided by Jureczko *et al.* [108].
[3]Provided by Zimmermann *et al.* [258].
[4]Provided by Kim *et al.* [118].
[5]Provided by D'Ambros *et al.* [50, 51].

Table 6.2.: [Empirical Study 2] An overview of the studied systems.

6.4. Case Study Design

In this section, we discuss our selection criteria for the studied systems and then describe the design of the case study experiment that we perform in order to address our research questions.

6.4.1. Studied Datasets

In selecting the studied datasets, we identified three important criteria that needed to be satisfied:

Criterion 1 — Publicly-available defect datasets from different corpora

Our recent work [233] Chapter 3 shows that researchers tend to reuse experimental components (e.g., datasets, metrics, and classifiers). Song *et al.* [221] and Ghotra *et al.* [74] also show that the performance of defect prediction models can be impacted by the dataset from which they are trained. To combat potential bias in our conclusions and to foster replication of our experiments, we choose to train our defect prediction models using datasets from different corpora and domains that are hosted in publicly-available data repositories. To satisfy criterion 1, we began our study using 101 publicly-available defect datasets. 76 datasets are downloaded from the Tera-PROMISE Repository,[1] 12 clean NASA datasets are provided by Shepperd *et al.* [209], 5 are provided by Kim *et al.* [118,253], 5 are provided by D'Ambros *et al.* [50,51], and 3 are provided by Zimmermann *et al.* [258].

Criterion 2 — Dataset robustness

Mende *et al.* [147] show that models that are trained using small datasets may produce unstable performance estimates. An influential characteristic in the performance of a classification technique is the number of *Events Per Variable*

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

(EPV) [186,231], i.e., the ratio of the number of occurrences of the least frequently occurring class of the dependent variable (i.e., the events) to the number of independent variables that are used to train the model (i.e., the variables). Our recent work shows that defect prediction models that are trained using datasets with a low EPV value are especially susceptible to unstable results [231]. To mitigate this risk, we choose to study datasets that have an EPV above 10, as suggested by Peduzzi *et al.* [186]. To satisfy criterion 2, we exclude the 78 datasets that we found to have EPV values below 10.

Criterion 3 — Sane defect data

Since it is unlikely that more software modules have defects than are free of defects, we choose to study datasets that have a rate of defective modules below 50%. To satisfy criterion 3, we exclude an additional 5 datasets because they have a defective rate above 50%.

Table 6.2 provides an overview of the 18 datasets that satisfy our criteria for analysis. To strengthen the generalizability of our results, the studied datasets include proprietary and open source systems of varying size and domain.

Figure 6.1 provides an overview of the approach that we apply to each studied system. We describe each step in the approach below.

6.4.2. Generate Bootstrap Sample

In order to ensure that the conclusions that we draw about our models are robust, we use the out-of-sample bootstrap validation technique [58,233], which leverages aspects of statistical inference [62]. The out-of-sample bootstrap is made up of two steps:

(Step 1) A bootstrap sample of size N is randomly drawn with replacement from an original dataset, which is also of size N.

(Step 2) A model is trained using the bootstrap sample and tested using the rows that do not appear in the bootstrap sample. On average, 36.8% of the rows will not appear in the bootstrap sample, since it is drawn with replacement [58].

The out-of-sample bootstrap process is repeated 100 times, and the average out-of-sample performance is reported as the performance estimate.

Unlike the ordinary bootstrap, the out-of-sample bootstrap technique fits models using the bootstrap samples, but rather than testing the model on the original sample, the model is instead tested using the rows that do not appear in the bootstrap sample [233]. Thus, the training and testing corpora do not share overlapping observations.

Unlike k-fold cross-validation, the out-of-sample bootstrap technique fits models using a dataset that is of equal length to the original dataset. Cross-validation splits the data into k equal parts, using k - 1 parts for fitting the model, setting aside 1 fold for testing. The process is repeated k times, using a different part for testing each time. However, Mende *et al.* [147] point out that the scarcity of defective modules in the small testing corpora of 10-fold cross validation may produce biased and unstable results. Prior studies have also shown that 10-fold cross validation can produce unstable results for small samples [32]. On the other hand, our recent research demonstrates that the out-of-sample bootstrap tends to produce the least biased and most stable performance estimates [233]. Moreover, the out-of-sample bootstrap is recommended for highly-skewed datasets [87], as is the case in our defect prediction datasets.

6.4.3. Generate Caret-optimized setting

Since it is impractical to assess all of the possible parameter settings of the parameter spaces, we use the optimized parameter settings suggested by the `train` function of the `caret` R package [131]. Caret suggests candidate settings for

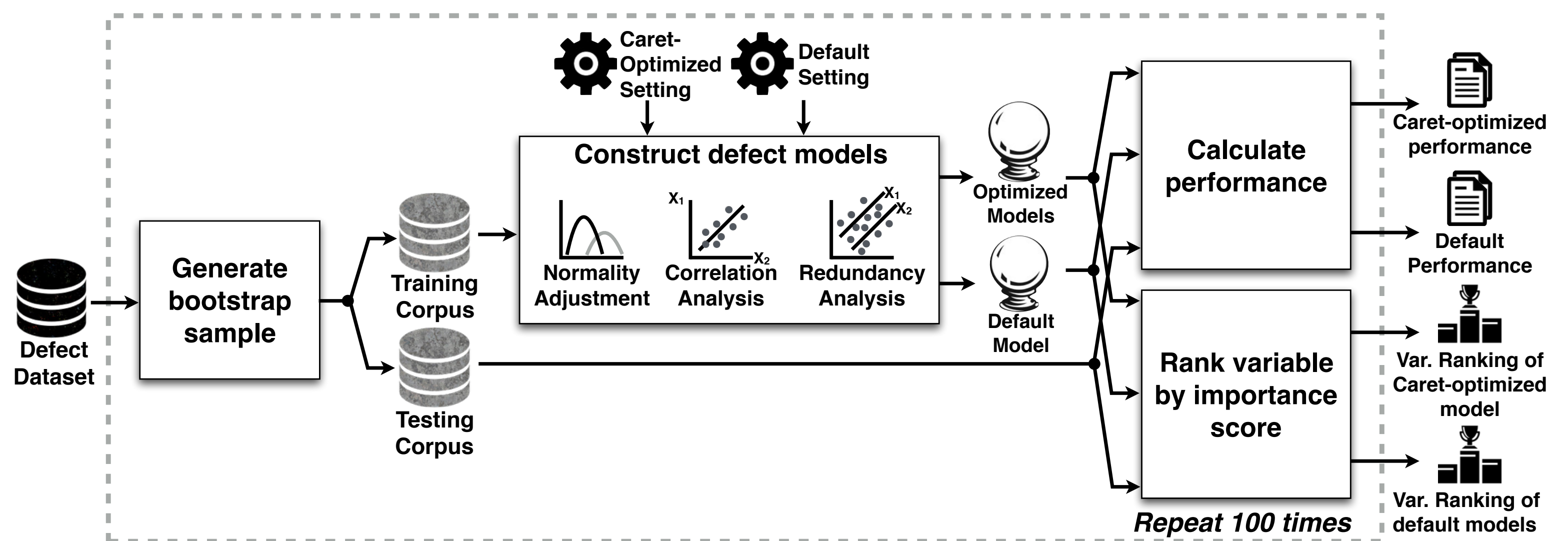


Figure 6.1.: [Empirical Study 2] An overview of our case study approach for studying the impact of automated parameter optimization on defect prediction models.

each of the studied classification techniques, which can be checked using the `getModelInfo` function of the `caret` R package [131]. Figure 6.2 provides an overview of Caret parameter optimization. The optimization process is made up of three steps.

(Step 1) Generate candidate parameter settings: The `train` function will generate candidate parameter settings based on a given budget threshold (i.e., tune length) for evaluation. The budget threshold indicates the number of different values to be evaluated for each parameter. As suggested by Kuhn [129], we use a budget threshold of 5. For example, the number of boosting iterations of the C5.0 classification technique is initialized to 1 and is increased by 10 until the number of candidate settings reaches the budget threshold (e.g., 1, 10, 20, 30, 40). Table 6.1 shows the candidate parameter settings for each of the studied parameters. The default settings are shown in bold typeface.

(Step 2) Evaluate candidate parameter settings: Caret evaluates all of the potential combinations of the candidate parameter settings. For example, if a classification technique accepts 2 parameters with 5 candidate parameter settings for each, Caret will explore all 25 potential combinations of parameter settings (unless the budget is exceeded). We use 100 repetitions of the out-of-sample bootstrap to estimate the performance of classifiers that are trained using each of the candidate parameter settings. For each candidate parameter setting, a classifier is fit to a subsample of the training corpus and we estimate the performance of a model using those rows in the training corpus that do not appear in the subsample that was used to trained the classifier.

(Step 3) Identify the Caret-optimized setting: Finally, the performance estimates are used to identify which parameter settings are the most opti-

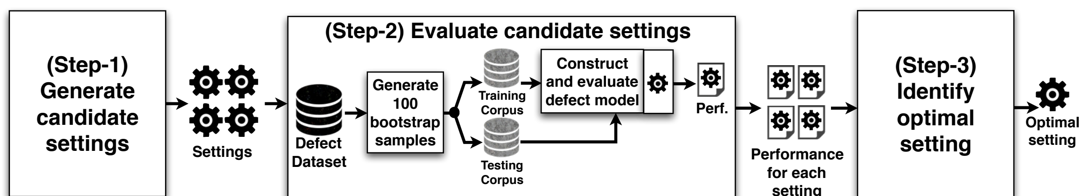


Figure 6.2.: [Empirical Study 2] An overview of Caret parameter optimization.

mal. The Caret-optimized setting is the one that achieves the highest performance estimate.

6.4.4. Construct Defect Models

In order to measure the impact of automated parameter optimization on defect prediction models, we train defect models using Caret-optimized settings and default settings. To ensure that the training and testing corpora have similar characteristics, we do not re-balance or re-sample the training data.

Normality Adjustment. Analysis of the distributions of our independent variables reveals that they are right-skewed. As suggested by previous research [103], we mitigate this skew by log-transforming each independent variable ($ln(x+1)$) prior to using them to train our models.

Correlation Analysis. Independent variables that are highly correlated with each other when a model is being interpreted. Indeed, our recent work [233] (see Chapter 3) has demonstrated that collinearity and multicollinearity issues can artificially inflate (or deflate) the impact of software metrics in the interpretation of defect prediction models. Jiarpakdee *et al.* [106] also point out that 10%-67% of software metrics of the publicly defect datasets are redundant. We measure the correlation between explanatory variables using Spearman rank correlation tests (ρ). We then use a variable clustering analysis [202] to construct a hierarchical overview of the correlation and remove explanatory variables with a high correlation. We select $|\rho| = 0.7$ as a threshold for removing highly correlated variables,

as a rule of thumb for interpreting the magnitude of the correlation [113]. We perform this analysis iteratively until all clusters of surviving variables have $|\rho|$ values below 0.7.

Redundancy Analysis. While correlation analysis reduces collinearity among our variables, it does not detect all of the *redundant variables*, i.e., variables that do not have a unique signal with respect to the other variables. Redundant variables will interfere with each other, distorting the modelled relationship between the explanatory variables and its class. We, therefore, remove redundant variables prior to constructing our defect prediction models. In order to detect redundant variables, we fit preliminary models that explain each variable using the other variables. We use the R^2 value of the preliminary models to measure how well each variable is explained by the others.

We use the implementation of this approach provided by the `redun` function of the `rms` R package [88]. The variable that is most well-explained by the other variables is iteratively dropped until either: (1) no preliminary model achieves an R^2 above a cutoff threshold (for this paper, we use the default threshold of 0.9), or (2) removing a variable would make a previously dropped variable no longer explainable, i.e., its preliminary model will no longer achieve an R^2 exceeding the threshold.

6.4.5. Calculate Performance

Prior studies have argued that threshold-dependent performance metrics (i.e., precision and recall) are problematic because they: (1) depend on an arbitrarily-selected threshold [9,17,133,192,211] and (2) are sensitive to imbalanced data [52, 93,139,147,227]. Instead, we use the *Area Under the receiver operator characteristic Curve (AUC)* to measure the discrimination power of our models as suggested by recent research [55, 87, 96, 133, 222, 224].

The AUC is a threshold-independent performance metric that measures a clas-

sifier's ability to discriminate between defective and clean modules (i.e., do the defective modules tend to have higher predicted probabilities than clean modules?). AUC is computed by measuring the area under the curve that plots the true positive rate against the false positive rate, while varying the threshold that is used to determine whether a file is classified as defective or not. Values of AUC range between 0 (worst performance), 0.5 (random guessing performance), and 1 (best performance).

6.4.6. Rank Variables by Importance Score

To identify the most important variables, we compute variable importance for each variable in our models. To do so, we introduce a generic variable importance score that can be applied to any classifier. Figure 6.3 provides an overview of the calculation of our variable importance measurement to generate ranks of important variables for each of the Caret-optimized and default models.

Generic Variable Importance Score

The calculation of our variable importance score is made up of 2 steps.

(Step 1) For each testing dataset, we first randomly permute values of a variable under test in order to produce a randomly-permuted dataset.

(Step 2) We then compute the misclassification rate of a defect prediction model that we train using randomly-permuted dataset. The larger the misclassification rate, the greater the importance of the variable.

We repeat the Steps 1 and 2 in order to produce a variable important score for all variables. Since the experiment is repeated 100 times, each variable will have several variable importance scores (i.e., one from each of the repetitions). An example `R` implementation of the generic variable importance calculation is provided in Appendix D.

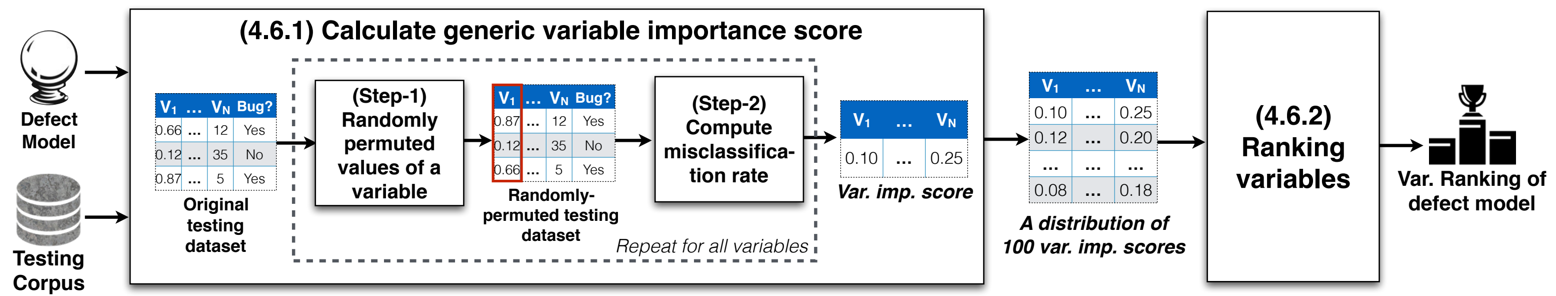


Figure 6.3.: [Empirical Study 2] An overview of our generic variable importance calculation that can be applied to any classification techniques.

Ranking Variables

To study the impact of the studied variables on our models, we apply the Scott-Knott Effect Size Difference (ESD) test [231] (see Chapter 7). The Scott-Knott ESD test will cluster variables according to statistically significant differences in their mean variable importance scores ($\alpha = 0.05$). The Scott-Knott ESD test ranks each variable exactly once, however several variables may appear within one rank. The Scott-Knott ESD test is a variant of the Scott-Knott test that is effect size aware. The Scott-Knott ESD test uses hierarchical cluster analysis to partition the set of treatment means into statistically distinct groups.

Unlike the traditional Scott-Knott test [98], the Scott-Knott ESD test will merge any two statistically distinct groups that have a negligible Cohen's d effect size [44] into one group. The Scott-Knott ESD test also overcomes the confounding factor of overlapping groups that are produced by several other post-hoc tests [74,159], such as Nemenyi's test [177], which were used in prior studies [133]. We implement the Scott-Knott ESD test based on the implementation of the Scott-Knott test provided by the `ScottKnott` R package [98] and the implementation of Cohen's d provided by the `effsize` R package [240].

Finally, we produce rankings variables in the Caret-optimized and default models. Thus, each variable has a rank for each type of model.

6.5. Case Study Results

In this section, we present the results of our case study with respect to our three research questions.

(RQ1) How much does the performance of defect prediction models change when automated parameter optimization is applied?

Approach. To address RQ1, we start with the AUC performance distribution of the 26 classification techniques that require at least one parameter setting (see Section 6.2). For each classification technique, we compute the difference in the performance of classifiers that are trained using default and Caret-optimized parameter settings. We then use boxplots to present the distribution of the performance difference for each of the 18 studied datasets. To quantify the magnitude of the performance improvement, we use Cohen's d effect size [44], which is the difference between the two means divided by the standard deviation of the two datasets ($d = \frac{\bar{x}_1 - \bar{x}_2}{s.d.}$). The magnitude is assessed using the thresholds provided by Cohen [45]:

$$\text{effect size} = \begin{cases} negligible & \text{if Cohen's } d \leq 0.2 \\ small & \text{if } 0.2 < \text{Cohen's } d \leq 0.5 \\ medium & \text{if } 0.5 < \text{Cohen's } d \leq 0.8 \\ large & \text{if } 0.8 < \text{Cohen's } d \end{cases}$$

Furthermore, understanding the most influential parameters would allow researchers to focus their optimization effort. To this end, we investigate the performance difference for each of the studied parameters. To quantify the individual impact of each parameter, we train a classifier with all of the studied parameters set to their default settings, except for the parameter whose impact we want to measure, which is set to its Caret-optimized setting. We estimate the impact of each parameter using the difference of its performance with respect to a classifier that is trained entirely using default parameter settings.

Results. **Caret improves the AUC performance by up to 40 percentage points.** Figure 6.4 shows the performance improvement for each of the 18 studied datasets

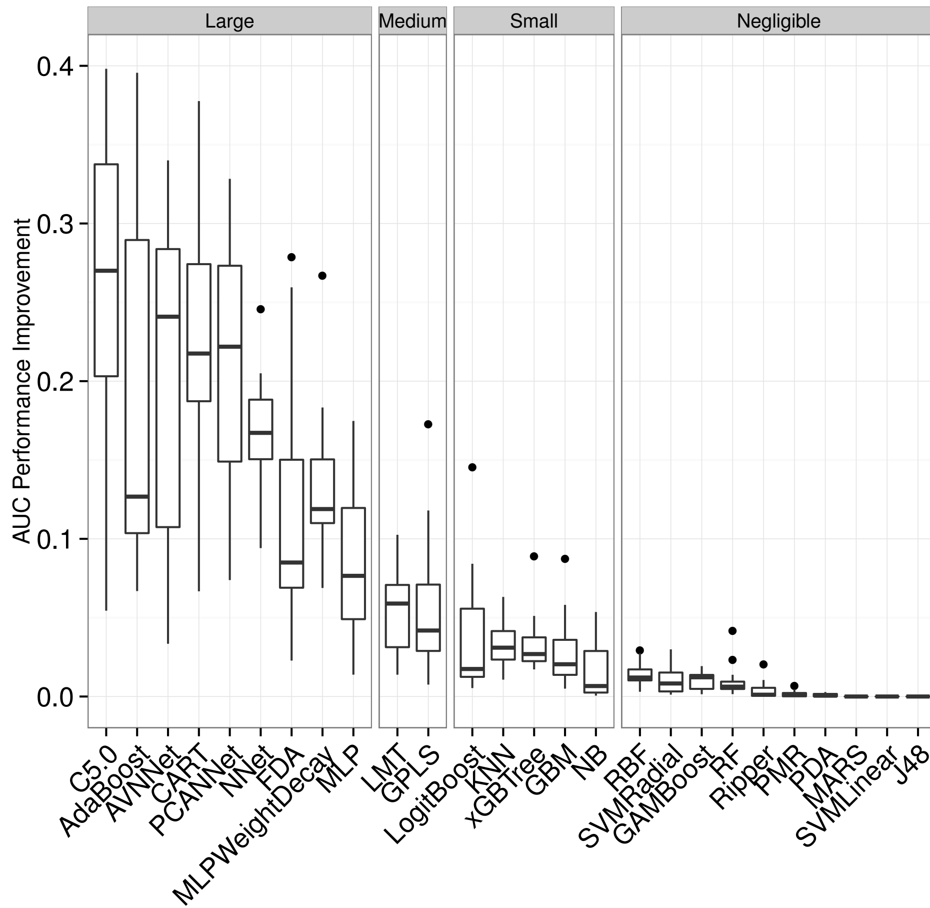


Figure 6.4.: [Empirical Study 2] The performance improvement and its Cohen's d effect size for each of the studied classification techniques.

and for each of the classification techniques. The boxplots show that Caret can improve the AUC performance by up to 40 percentage points. Moreover, the performance improvement provided by applying Caret is non-negligible (i.e., $d > 0.2$) for 16 of the 26 studied classification techniques (62%). This indicates that parameter settings can substantially influence the performance of defect prediction models.

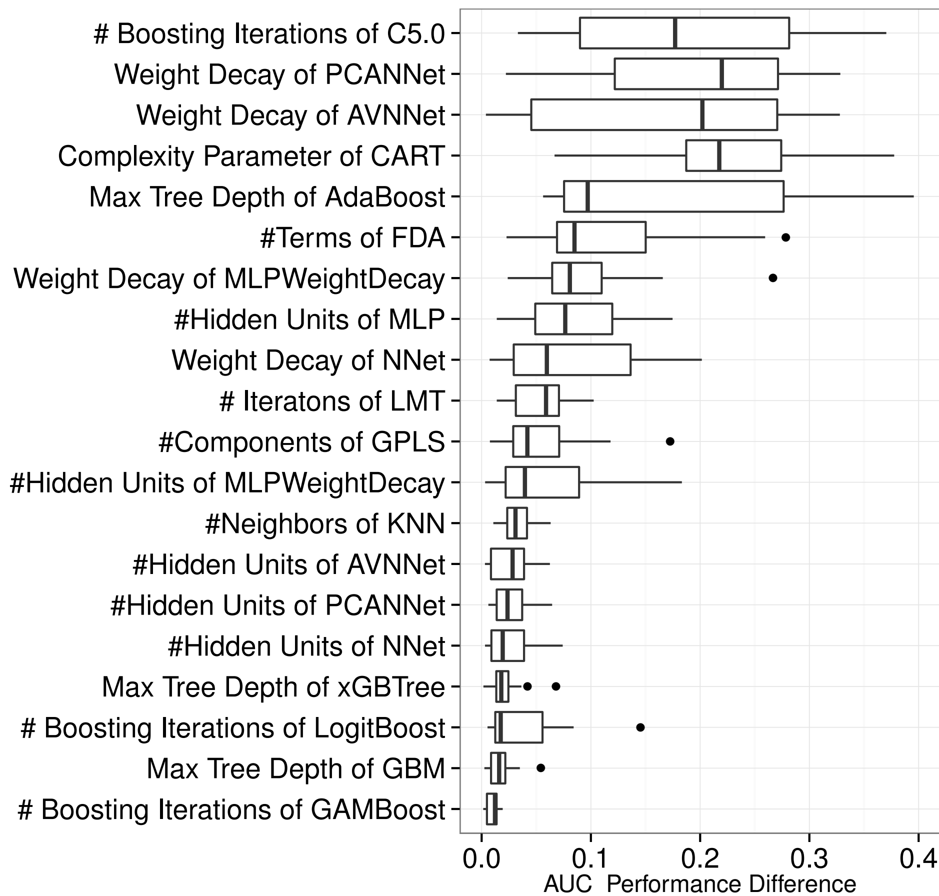


Figure 6.5.: [Empirical Study 2] The AUC performance difference of the top-20 most sensitive parameters.

C5.0 boosting yields the largest performance improvement when Caret is applied. According to Cohen's d, the performance improvement provided by applying Caret is large for 9 of the 26 studied classification techniques (35%). On average, Figure 6.4 shows that the C5.0 boosting classification technique benefits most by applying Caret, with a median performance improvement of 27 percentage points. Indeed, the C5.0 boosting classification technique improves from 6 to 40 percentage points.

Figure 6.5 shows that the `#boosting iterations` parameter of the C5.0 classification technique is the most influential parameter, while the `winnow` and `model type` parameters tend to have less of an impact. Indeed, the default `#boosting iterations` setting that is provided by the `C5.0` R package [130] is 1, indicating that only one C5.0 tree model is used for prediction. Nevertheless, we find that the optimal `#boosting iterations` parameter is 40, suggesting that the default parameter settings of the research toolkits are suboptimal for defect prediction datasets. This finding provides supporting evidence of the suspicion of prior studies [82, 103, 220, 242].

In addition to C5.0 boosting, other classifiers also yield a considerably large benefit. Figure 6.4 shows that the performance of the adaptive boosting (i.e., AdaBoost), advanced neural networks (i.e., AVNNet, PCANNet, NNet, MLP, and MLPWeightDecay), CART, and Flexible Discriminant Analysis (FDA) classification techniques also have a large effect size with a median performance improvement from 13-24 percentage points. Indeed, Figure 6.5 shows that the fluctuation of the performance of the advanced neural network techniques is largely caused by changing the `weight decay`, but not the `#hidden units` or `bagging` parameters. Moreover, the `complexity` parameter of CART and `max tree depth` of adaptive boosting classification techniques are also sensitive to parameter optimization.

Summary. Caret improves the AUC performance of defect prediction models by up to 40 percentage points. Moreover, the performance improvement provided by applying Caret is non-negligible for 16 of the 26 studied classification techniques (62%).

(RQ2) How stable is the performance of defect prediction models when automated parameter optimization is applied?

Approach. To address RQ2, we start with the AUC performance distribution of the 26 studied classification techniques on each of the 18 studied datasets. The stability of a classification technique is measured in terms of the variability of the performance estimates that are produced by the 100 iterations of the out-of-sample bootstrap. For each classification technique, we compute the standard deviation (S.D.) of the bootstrap performance estimates of the classifiers where Caret-optimized settings have been used and the S.D. of the bootstrap performance estimates of the classifiers where the default settings have been used. To analyze the difference of the stability between two classification techniques, we present distribution of the stability ratio (i.e., S.D. of the optimized classifier divided by the S.D. of the default classifier) of the two classifiers when applied to the 18 studied datasets.

Similar to RQ1, we analyze the parameters that have the largest impact on the stability of the performance estimates. To this end, we investigate the stability ratio for each of the studied parameters. To quantify the individual impact of each parameter, we train a classifier with all of the studied parameters set to their default settings, except for the parameter whose impact we want to measure, which is set to its Caret-optimized setting. We estimate the impact of each parameter using the stability ratio of its S.D. of performance estimates with respect to a classifier that is trained entirely using default settings.

Results. **Caret-optimized classifiers are at least as stable as classifiers that are trained using the default settings.** Figure 6.6 shows that there is a median stability ratio of at least one for all of the studied classification techniques. Indeed, we find that the median ratio of one tends to appear for the classification techniques that yield negligible performance improvements in RQ1. These tight stability ratio ranges that are centered at one indicate that the stability of classifiers is

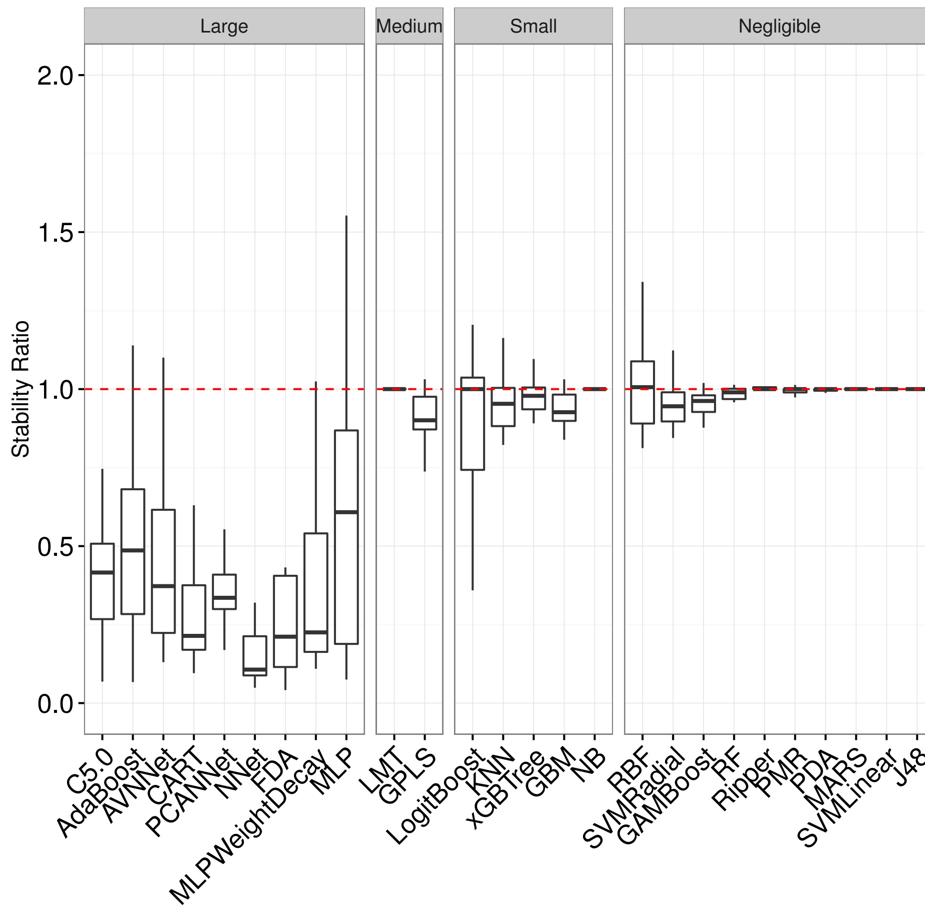


Figure 6.6.: [Empirical Study 2] The stability ratio of the classifiers that are trained using Caret-optimized settings compared to the classifiers that are trained using default settings for each of the studied classification techniques.

not typically impacted by Caret-optimized settings.

Moreover, the Caret-optimized classifiers of 9 of the 26 studied classification techniques (35%) are more stable than classifiers that are trained using the default values. Indeed, Figure 6.6 shows that there is a median stability ratio of 0.11 (NNet) to 0.61 (MLP) among the 9 classification techniques where the stability has improved. This equates to a 39%-89% stability improvement for these Caret-optimized classifiers. Indeed, Figure 6.7 shows that the stability of

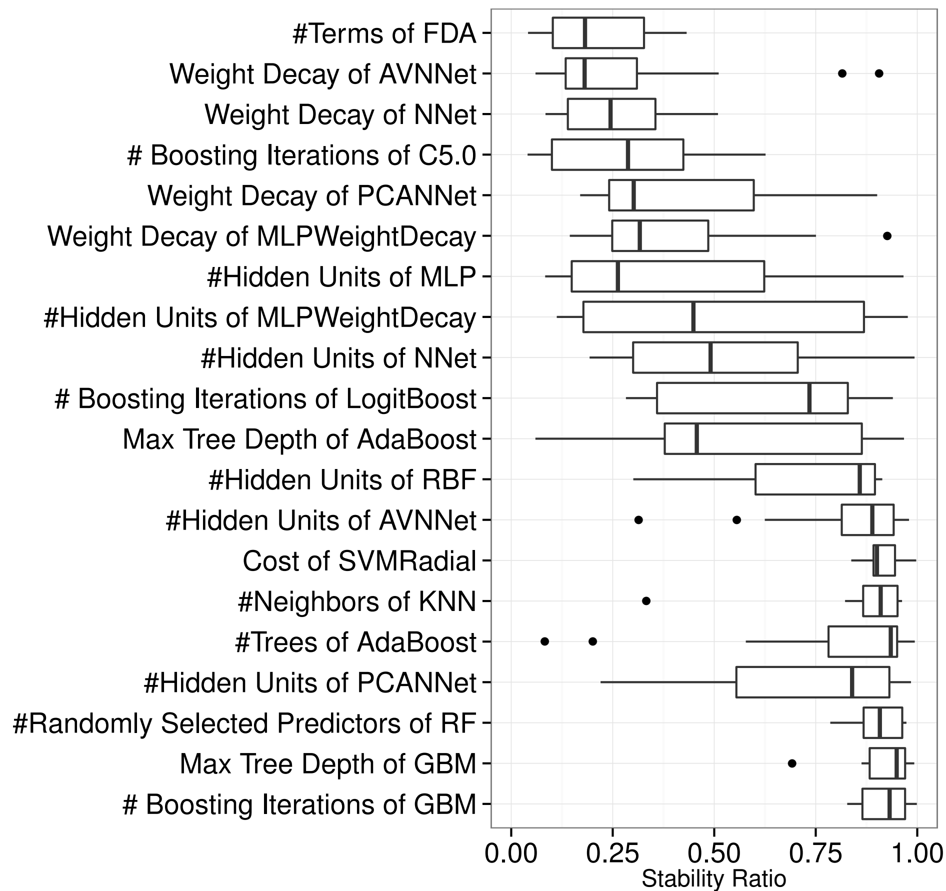


Figure 6.7.: [Empirical Study 2] The stability ratio of the top-20 most sensitive parameters.

the performance of the advanced neural network techniques is largely caused by changing the `weight decay`, but not the `#hidden units` or `bagging` parameters, which is consistent with our findings in RQ1.

<u>Summary.</u> Caret-optimized classifiers are at least as stable as classifiers that are trained using the default parameter settings. Moreover, the Caret-optimized classifiers of 9 of the 26 studied classification techniques (35%) are significantly more stable than classifiers that are trained using the default parameter settings.

(RQ3) How much does the interpretation of defect prediction models change when automated parameter optimization is applied?

Approach. To address RQ3, we start with the variable ranking of the 26 studied classification techniques on each of the 18 studied datasets for both Caret-optimized and default models. For each classification technique, we compute the difference in the ranks of the variables that appear in the top-three ranks of the classifiers that are trained using Caret-optimized and default settings. For example, if a variable v appears in the top rank in both the Caret-optimized and default models, then the variable would have a rank difference of zero. However, if v appears in the third rank in the default model, then the rank difference of v would be negative two.

Results. **The insights that are derived from 42% of the Caret-optimized classifiers are different from the insights that are derived from the classifiers that are trained using the default values.** Figure 6.8 shows the rank differences for all of the studied classifiers. In the 11 studied classification techniques that yield a medium to large performance improvement, we find that 64%-65% of the variables in the top importance rank of the Caret-optimized models also appear in the top importance rank of the default models. Indeed, in the 9 studied classification techniques that yield a large performance improvement, as little as 28% of the variables in the third rank of the Caret-optimized models also appear in the third rank of the default models. The low percentage indicates that the insights are heavily influenced by the parameter settings of classification techniques.

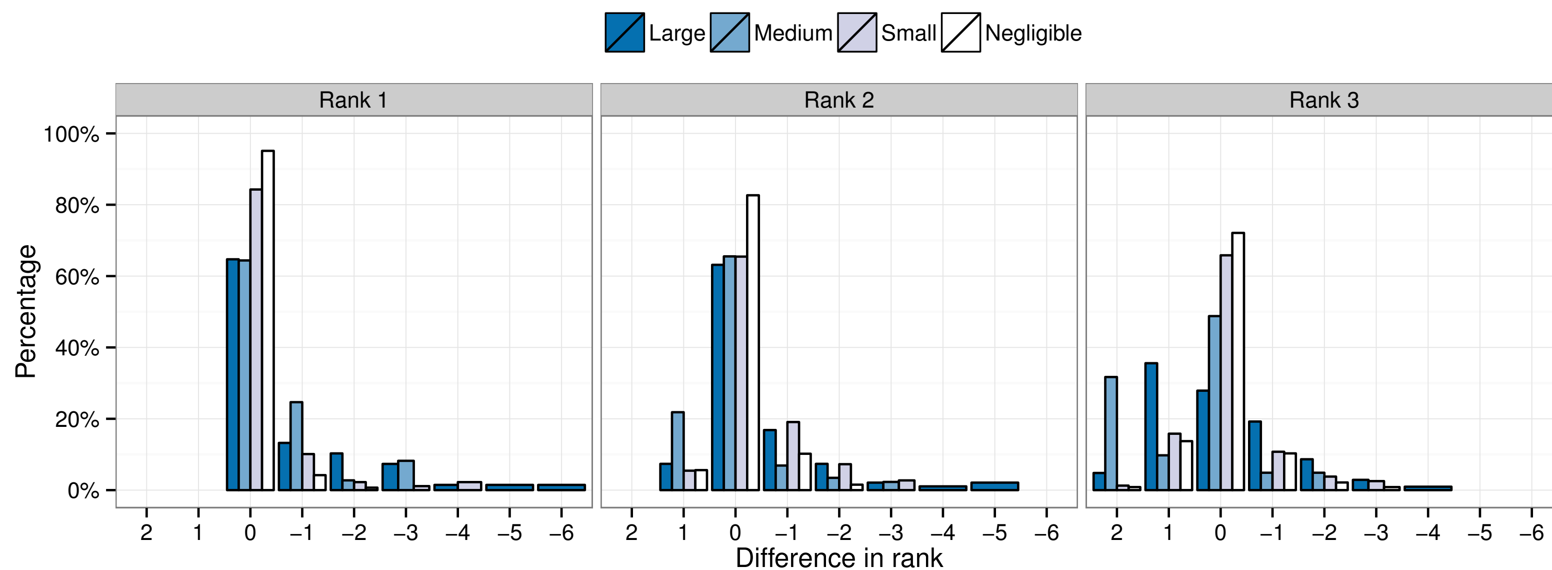


Figure 6.8.: [Empirical Study 2] The difference in the ranks for the variables according to their variable importance scores among the classifiers that are trained using Caret-optimized settings and classifiers that are trained using default settings. The bars indicate the percentage of variables that appear in that rank in the Caret-optimized model while also appearing in that rank in the default models.

Classifiers that yield a large performance improvement tends to have a larger impact on interpretation than classifiers that yield a negligible performance improvement do. Figure 6.8 shows that a smaller percentage of the variables of the Caret-optimized models are found at the same rank in the classification techniques that have a large performance improvement than the classification techniques that have a negligible performance improvement. Indeed, 65% of the variables in the top importance rank of classification techniques that yield a large performance improvement are found at the same rank. On the other hand, 95% of the variables appear at the same rank in the Caret-optimized and default models of classification techniques where Caret yields a negligible performance improvement.

Summary. Caret substantially shifts the ranking of the importance of software metrics, with as little as 28% of the variables from the Caret-optimized models appearing in those ranks in the default models. Moreover, classification techniques where Caret has a larger impact on the performance are also subject to large shifts in interpretation.

(RQ4) How well do optimal parameter settings transfer from one context to another?

Approach. We apply Caret parameter optimization on all datasets. For each classification technique, we obtain a list of Caret-suggested settings for every studied dataset. We then estimate the transferability of these settings using their frequency of appearance in the Caret suggestions across datasets. We call the setting values that appear in the Caret suggestions of several datasets transferable parameters.

We analyze 4 different types of transferability, i.e., across the 18 datasets, across the 5 proprietary datasets, across the 3 Eclipse datasets, and across the 2 NASA

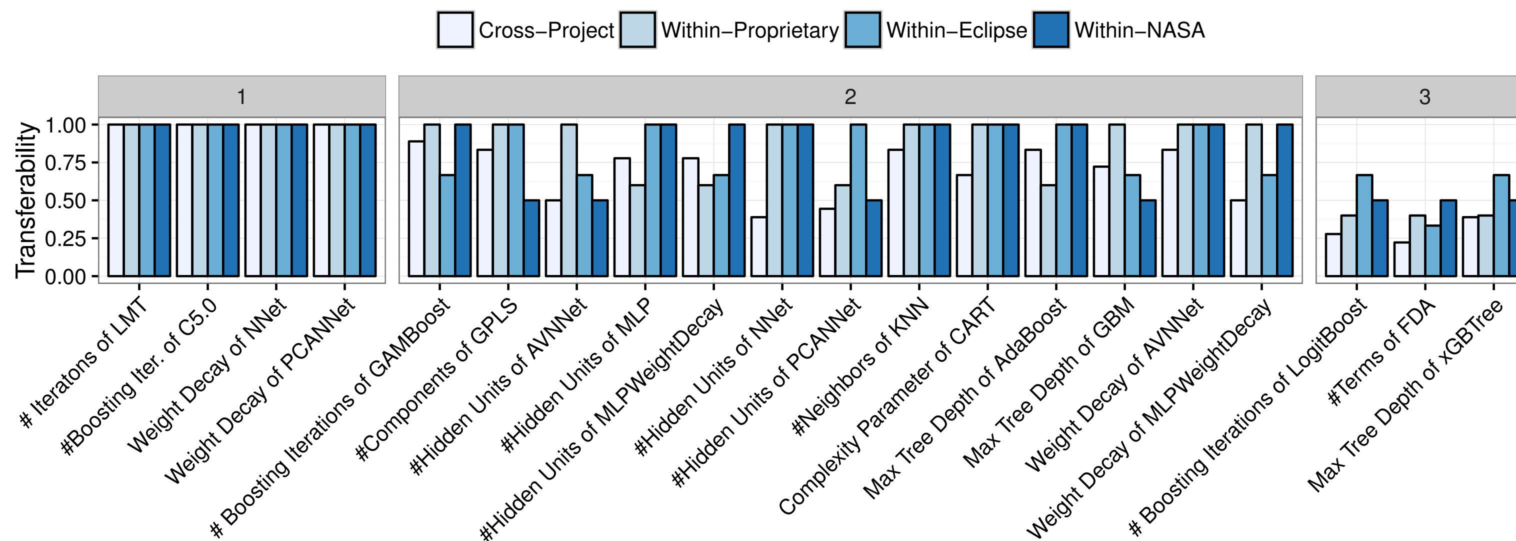


Figure 6.9.: [Empirical Study 2] Four types of transferability for each of the top-20 most sensitive parameters. Higher frequency for each of the Caret-suggested settings that appear across datasets indicates high transferability of a parameter.

datasets. Since the transferability of Caret-suggested parameters has more of an impact on the most sensitive parameters, we focus our analysis on the 20 most sensitive parameters (*cf.* Figure 6.5).

Results. **The Caret-suggested setting of 85% of the 20 most sensitive parameters can be applied to datasets that share a similar set of metrics without a statistically significant drop in performance.** Figure 6.9 shows the 4 types of transferability for each of the 20 most sensitive parameters. We divide the transferable parameters into three groups. The first group is used to indicate the parameters that can be transfered across the studied contexts. The second group is used to indicate the parameters that can be transfered across some of the studied contexts. The third group is used to indicate the parameters that can not be transfered across any of the studied contexts.

We find that the Caret-suggested setting of 17 of the 20 most sensitive parameters can be applied to other datasets that share a similar set of metrics without a statistically significant drop in performance with respect to the optimal performance for that dataset (i.e., the first and the second groups). For example, the value 40 for the `# boosting iterations` parameter of the C5.0 boosting classification technique can be applied to all of the 18 studied datasets without a statistically significant drop in performance. Moreover, the Caret-suggested value 9 for `#hidden units` and value 0.1 for `weight decay` parameters of the advanced neural networks (i.e., AVNNet) can be applied to 9 and 15 of the 18 studied datasets, respectively, that have a similar set of metrics without a statistically significant drop in performance. Indeed, the parameters of C5.0 and LMT are always transferable across the studied contexts, indicating that researchers and practitioners can safely adopt the Caret-suggested parameters that are obtained using a dataset with similar metrics. On the other hand, we find that only the parameters of LogitBoost, FDA, and xGBTree cannot be transfered across any of the studied contexts, indicating that researchers and practitioners should re-apply automated parameter optimization.

<u>Summary.</u> The Caret-suggested setting of 85% of the 20 most sensitive parameters can be applied to datasets that share a similar set of metrics without a statistically significant drop in performance.

(RQ5) What is the cost of applying automated parameter optimization?

Approach. Our case study approach is computationally-intensive (i.e., 450 parameter settings × 100 out-of-sample bootstrap repetitions × 18 systems = 810,000 results). However, the results can be computed in parallel. Hence, we design our experiment using a High Performance Computing (HPC) environment. Our experiments are performed on 43 high performance computing machines with 2x Intel Xeon 5675 @3.1 GHz (24 hyper-threads) and 64 GB memory (i.e., in total, 24 hyper-threads × 43 machines = 1,032 hyper-threads). Each machine connects to a 2 petabyte shared storage array via a dual 10-gigabit fibre-channel connection.

For each of the classification techniques, we compute the average amount of execution time that was consumed by Caret when producing suggested parameter settings for each of the studied datasets.

Results. **Caret adds less than 30 minutes of additional computation time to 65% of the studied classification techniques.** Figure 6.10 shows the computational cost of Caret optimization techniques for each of the 18 studied classification techniques. The optimization cost of 17 of the 26 studied classification techniques (65%) is less than 30 minutes. C5.0 and extreme gradient boosting classification techniques, which yield top-performing classifiers more frequently than other classification techniques, fall into this category. This indicates that applying Caret tends to improve the performance of defect models while incurring a manageable additional computational cost.

On the other hand, 12% of the studied classification techniques require more than 3 additional hours of computation time to apply Caret. Only AdaBoost, MLPWeightDecay, and RBF incur this large overhead. Nonetheless, the computation could still be completed if it was run overnight. Since defect prediction models do not need to be (re)trained very often in practice, this cost should still be manageable.

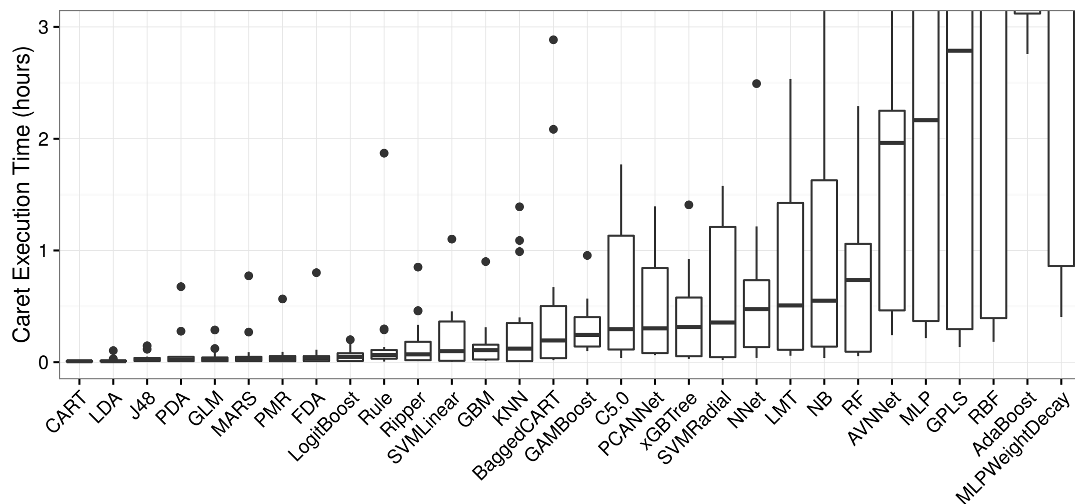


Figure 6.10.: [Empirical Study 2] Computational cost of Caret optimization techniques (hours).

Summary. Caret adds less than 30 minutes of additional computation time to 65% of the studied classification techniques.

6.6. Revisiting the Ranking of Classification Techniques for Defect Prediction Models

Prior studies have ranked classification techniques according to their performance on defect prediction datasets. For example, Lessmann *et al.* [133] demonstrate that 17 of 22 studied classification techniques are statistically indistinguishable. On the other hand, Ghotra *et al.* [74] argue that classification techniques can have a large impact on the performance of defect prediction models.

However, these studies have not taken parameter optimization into account. Since we find that parameter settings can improve the performance of the classifiers that are produced (see RQ1), we set out to revisit the findings of prior studies when Caret-optimized settings have been applied.

6.6.1. Approach

As Keung *et al.* [114] point out, dataset selection can be a source of bias in an analysis of top-performing classification techniques. To combat the bias that may be introduced by dataset selection, we perform a bootstrap-based Ranking Likelihood Estimation (RLE) experiment. Figure 6.11 provides an overview of our RLE experiment. The experiment uses a statistical comparison approach over multiple datasets that leverages both effect size differences and aspects of statistical inference [62]. The experiment is divided into two steps that we describe below.

(Step 1) Ranking Generation. We first start with the AUC performance distribution of the 26 studied classification techniques with the Caret-optimized parameter settings and the default settings. To find statistically distinct ranks of classification techniques within each dataset, we provide the AUC performance distribution of the 100 bootstrap iterations of each classification technique with both parameter settings to a Scott-Knott ESD test ($\alpha = 0.05$) [233].

We use the Scott-Knott ESD test in order to control for dataset-specific model performance, since some datasets may have a tendency to produce over- or underperforming classifiers. Finally, for each classification technique, we have 18 different Scott-Knott ranks (i.e., one from each dataset).

(Step 2) Bootstrap Analysis. We then perform a bootstrap analysis to approximate the empirical distribution of the likelihood that a technique will appear in the top Scott-Knott ESD rank [58]. The key intuition is that the relationship between the likelihood that is derived from studied datasets and the true likelihood that would be derived from the population of defect datasets is asymptotically equivalent to the relationship between the likelihood that is derived from bootstrap samples and the likelihood that is derived from studied datasets. We first input the ranking of the studied classification techniques on 18 studied datasets to the bootstrap analysis, which is comprised of two steps:

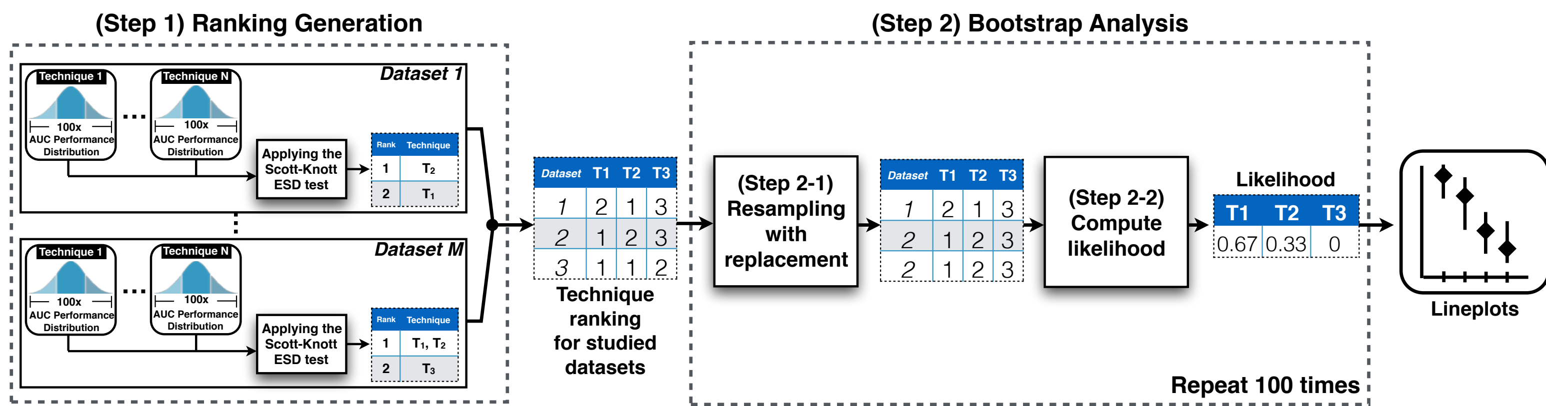


Figure 6.11.: [Empirical Study 2] An overview of our statistical comparison over multiple datasets.

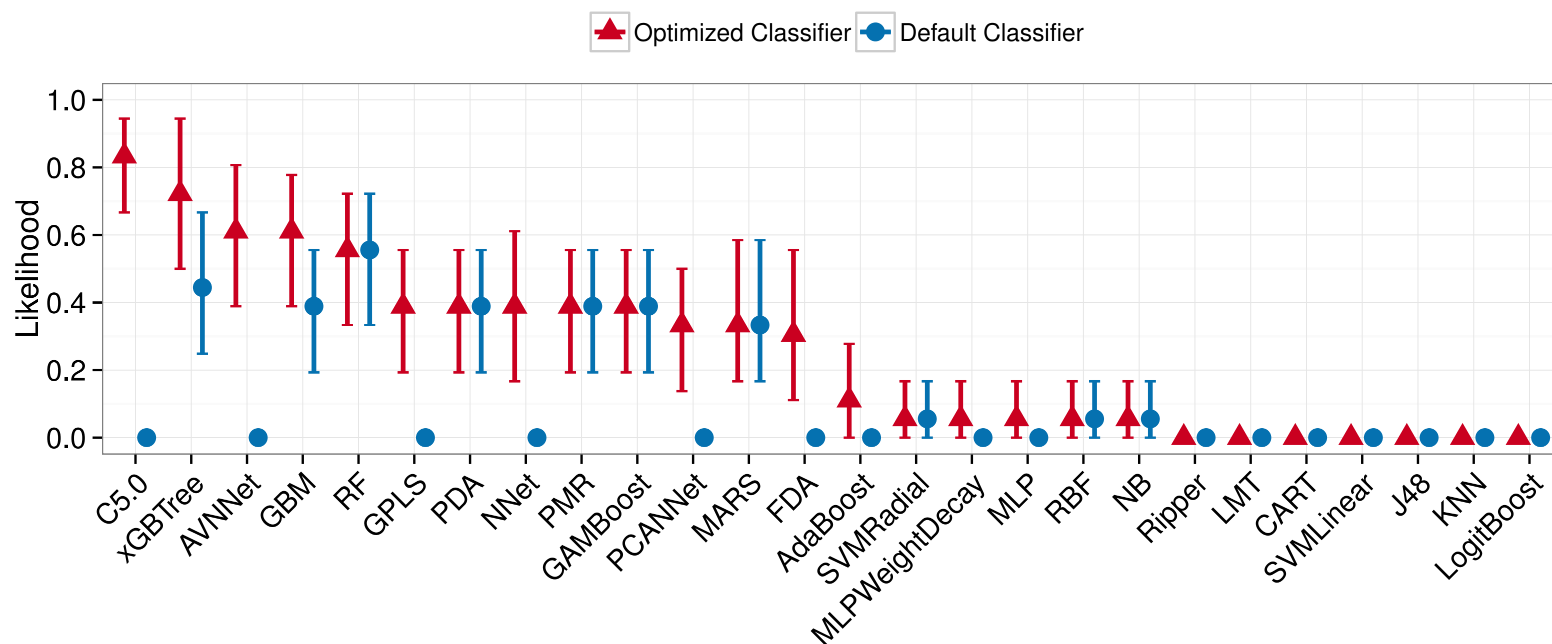


Figure 6.12.: [Empirical Study 2] The likelihood of each technique appearing in the top Scott-Knott ESD rank. Circle dots and triangle dots indicate the median likelihood, while the error bars indicate the 95% confidence interval of the likelihood of the bootstrap analysis. A likelihood of 80% indicates that a classification technique appears at the top-rank for 80% of the studied datasets.

(Step 2-1) A bootstrap sample of 18 datasets is randomly drawn with replacement from the ranking table, which is also of comprised of size 18 studied datasets.

(Step 2-2) For each classification technique, we compute the likelihood that a technique appears in the top Scott-Knott ESD rank in the bootstrap sample.

The bootstrap analysis is repeated 100 times. We then present the results with its 95% confidence interval, which is derived from the bootstrap analysis.

6.6.2. Results

C5.0 boosting tends to yield top-performing defect prediction models more frequently than the other studied classification techniques. Figure 6.12 shows the likelihood of each technique appearing in the top Scott-Knott ESD rank. We find that there is a 83% likelihood of C5.0 appearing in the top Scott-Knott rank. Furthermore, the bootstrap-derived 95% confidence interval ranges from 67% to 94%. On the other hand, when default settings are applied, C5.0 boosting has a 0% likelihood of appearing in the top rank. This echoes the findings of RQ1, where C5.0 boosting was found to be the classification technique that is most sensitive to parameter optimization.

Unlike prior work in the data mining domain, we find that random forest is not the most frequent top performer in our defect prediction datasets. Indeed, we find that there is a 55% likelihood of random forest appearing in the top Scott-Knott rank with a bootstrap-derived 95% confidence interval that ranges from 33% to 72%. A one-tailed bootstrap t-test reveals that the likelihood of C5.0 producing a top performing classifier is significantly larger than the likelihood of random forest producing a top-performing classifier ($\alpha = 0.05$). This contradicts the conclusions of Fernandez-Delgado *et al.* [64], who found that random forest

tends to yield top-performing classifiers the most frequently. The contradictory conclusions indicate that the domain-specifics play an important role.

Automated parameter optimization increases the likelihood of appearing in the top Scott-Knott ESD rank by as much as 83%. Figure 6.12 shows that automated parameter optimization increases the likelihood of 11 of the studied 26 classification techniques by as much as 83% (i.e., C5.0 boosting). This suggests that automated parameter optimization can substantially shift the ranking of classification techniques.

Summary. C5.0 boosting tends to yield top-performing defect prediction models more frequently than the other studied classification techniques. This disagrees with prior studies in the data mining domain, suggesting that domain-specifics play a key role. Furthermore, automated parameter optimization increases the likelihood of appearing in the top Scott-Knott ESD rank by as much as 83%.

6.7. Discussion

In this section, we discuss the broader implications of our findings and how our findings fit with the software defect prediction modelling and search-based software engineering literature.

6.7.1. Defect Prediction Modelling

Plenty of research raises concerns about the impact of experimental components of defect prediction modelling on research conclusions [229]. For example, concerns about the quality of defect datasets have been raised [28,179,193,230,253]. Bird *et al.* [28] find that the issue reports of several defects are not identified in commit logs. Bachmann *et al.* [21] find that the noise generated by missing links in defect prediction datasets introduces bias. Kim *et al.* [118] find that the randomly-generated noise has a large negative impact on the performance of de-

fect models. On the other hand, our recent work [230] shows that realistic noise (i.e., noise generated by actually mislabelled issue reports [94]) does not typically impact the precision of defect prediction models.

Recent research also raises concerns about that the choice of classification techniques and model validation techniques. For example, Ghotra *et al.* [74] find that there are statistically significant differences in the performance of defect prediction models that are trained using different classification techniques. Panichella *et al.* [184] and Bowes *et al.* [30] also find that using different classification techniques can identify different defective modules. Our recent work [231] (see Chapter 7) shows that the choice of model validation techniques also has an impact on the accuracy and stability of performance estimates.

Implications. In addition to the concerns about data quality, classification techniques, and model validation techniques, we find that parameter settings of classification techniques can also have a large impact on the performance, stability, and interpretation of defect prediction models. Our findings suggest that researchers should experiment with the parameters of the classification techniques. Given the large parameter space of many classification techniques, automated parameter optimization techniques like Caret offer an efficient way to conduct such an exploration.

6.7.2. Parameter Optimization for Defect Prediction Models

The most closely research related to our paper is the work by Fu *et al.*, who demonstrate that automated parameter optimization has an impact on the performance, model interpretation, and the ranking of top-performing classification techniques [69]. However, their experimental design is different from our paper in numerous ways. First, we investigate the impact of parameter settings of the 26 classification techniques, while they focus only 4 classification techniques. Second, we measure the performance using threshold-independent measures (i.e., AUC),

while they measure using threshold-dependent measures (i.e., Precision, Recall, and F-measure). Third, we apply Caret—an off-the-shelf parameter optimization technique, while they apply Differential Evolution (DE) algorithm. Forth, we evaluate defect prediction models using a within-release prediction setting, while they evaluate using a cross-project prediction setting.

Implications. Despite their experimental design is different from our paper in numerous ways, their conclusions are consistent to the conclusions of our study (i.e., RQ1, RQ3, and Section 6.6). Relative to the contributions of prior work, this paper makes the two additional contributions: an investigation of the automated parameter optimization on the model stability (RQ2), and an investigation on the parameter transferability in a cross-context prediction setting (RQ4).

6.7.3. Search-Based Software Engineering

Prior studies in the area of the search-based software engineering have shown that search algorithms can be used to find useful solution to software engineering problems [85,86]. For example, Panichella *et al.* [183] and Lohar *et al.* [138] tune the parameters of topic modelling techniques (i.e., Latent Dirichlet Allocation) for software engineering tasks. Thomas *et al.* [237] tune off-the-shelf Information Retrieval techniques for bug localization. Song *et al.* [220] tune classification techniques for software effort estimation models. Wang *et al.* [248] tune the settings of code clone detection techniques. Arcuri *et al.* [15] investigate a minimal test suite that maximizes branch coverage in the context of test data generation. Jia *et al.* [99] investigate an optimal test suite for combinatorial interaction testing.

Implications. In addition to the area of topic modelling, information retrieval, effort estimation, code clone detections, test data and suite generation, we find that Caret, which is a grid-search technique, can also be used to efficiently tune parameter settings of classification techniques for defect prediction models. Fur-

thermore, we find that Caret yields a large benefit in terms of performance improvement of defect prediction models, suggesting that automated parameter optimization like Caret should be used in future defect prediction studies.

6.8. Threats to Validity

We now discuss the threats to the validity of our study.

6.8.1. Construct Validity

The datasets that we analyze are part of several collections (e.g., NASA and PROMISE), which each provide different sets of metrics. Since the metrics vary, this is a point of variation between the studied systems that could impact our results. However, our within-family datasets analysis shows that the number and type of predictors do not influence our findings. Thus, we conclude that the variation of metrics does not pose a threat to our study. On the other hand, the variety of metrics also strengthens the generalization of our results, i.e., our findings are not bound to one specific set of metrics.

The Caret budget, which controls the number of settings that we evaluate for each parameter, limits our exploration of the parameter space. Although our budget setting is selected based on the literature [131], selecting a different budget may yield different results. To combat this bias, we repeat the experiment with different budgets of exploration (i.e., 3, 5, 7) and find consistent conclusions. Thus, we believe that an increase in the budget would not alter the conclusions of our study.

6.8.2. Internal Validity

We measure the performance of our classifiers using AUC. Other performance measures may yield different results. Future research to expand the set of mea-

sures that we adopt in our future work.

Prior work has shown that noisy data may influence conclusions that are drawn from defect prediction studies [74, 229, 230]. While Chapter 5 shows that noise generated by issue report mislabelling has little impact on the precision of defect prediction models, they do indeed impact the ability of identifying defective modules (i.e., recall). Hence, noisy data may be influencing our conclusions. However, we conduct a highly-controlled experiment where known-to-be noisy NASA data [209] has been cleaned. Nonetheless, dataset cleanliness should be inspected in future work.

6.8.3. External Validity

We study a limited number of systems in this paper. Thus, our results may not generalize to all software systems. However, the goal of this paper is not to show a result that generalizes to all datasets, but rather to show that there are datasets where parameter optimization matters. Nonetheless, additional replication studies may prove fruitful.

The generalizability of the bootstrap-based Ranking Likelihood Estimation (RLE) is dependent on how representative our sample is. To combat potential bias in our samples, we analyze datasets of different sizes and domains. Nonetheless, a larger sample may yield more robust results.

6.9. Chapter Summary

Defect prediction models are classifiers that are trained to identify defect-prone software modules. The characteristics of the classifiers that are produced are controlled by configurable parameters. Recent studies point out that classifiers may under-perform because they were trained using suboptimal default parameter settings. However, it is impractical to explore all of the possible settings in the

parameter space of a classification technique.

In this chapter, we investigate the performance, stability, and interpretation of defect prediction models where Caret [131] — an automated parameter optimization technique — has been applied. Through a case study of 18 datasets from systems that span both proprietary and open source domains, we make the following observations:

– Caret improves the AUC performance of defect prediction models by up to 40 percentage points. Moreover, the performance improvement provided by applying Caret is non-negligible for 16 of the 26 studied classification techniques (62%).

– Caret-optimized classifiers are at least as stable as classifiers that are trained using the default parameter settings. Moreover, the Caret-optimized classifiers of 9 of the 26 studied classification techniques (35%) are significantly more stable than classifiers that are trained using the default parameter settings.

– Caret substantially shifts the ranking of the importance of software metrics, with as little as 28% of the variables from the Caret-optimized models appearing in those ranks in the default models. Moreover, classification techniques where Caret has a larger impact on the performance are also subject to large shifts in interpretation.

– Caret increases the likelihood of producing a top-performing classifier by as much as 83%, suggesting that automated parameter optimization can substantially shift the ranking of classification techniques.

Our results lead us to conclude that parameter settings can indeed have a large impact on the performance, stability, and interpretation of defect prediction models. Since automated parameter optimization techniques like Caret yield

benefits in terms of performance, stability, and interpretation, they should be considered for use in future defect prediction studies.

Finally, we would like to emphasize that we do not seek to claim the generalization of our results. Instead, the key message of our study is that there are datasets where there are statistically significant differences between the performance of classification techniques that are trained using default and Caret-optimized parameter settings. Hence, we recommend that software engineering researchers experiment with the automated parameter optimization (e.g., Caret) instead of relying on the default parameter setting of the research toolkits, assuming that other parameter settings are not likely to lead to statistically significant improvements in their reported results. Given the availability of automated parameter optimization in commonly-used research toolkits (e.g., Caret for R [131], MultiSearch for Weka [81], GridSearch for Scikit-learn [185]), we believe that our recommendation is a rather simple and low-cost recommendation to adopt.

6.9.1. Concluding Remarks

In this chapter, we investigate the impact of the choice of parameter settings of classification techniques on defect prediction models. Our findings suggest that automated parameter optimization should be applied to produce more accurate and more reliable defect prediction models.

Even if defect prediction models are trained on a clean defect dataset and their parameter settings are optimized, defect prediction models may produce inaccurate performance estimates if we do not consider the impact of model validation techniques (e.g., k-fold cross-validation). In the next chapter, we investigate the impact of model validation techniques on the performance estimates.

CHAPTER 7

The Impact of Model Validation Techniques

<u>Key Finding</u>

Model validation techniques produces statistically different performance estimates.

An earlier version of the work in this chapter appears in the IEEE Transactions on Software Engineering (TSE) [231].

7.1. Introduction

Defect prediction models help Software Quality Assurance (SQA) teams to effectively focus their limited resources on the most defect-prone software modules. Broadly speaking, a defect prediction model is a statistical regression model or a machine learning classifier that is trained to identify defect-prone software modules. These defect prediction models are typically trained using software metrics that are mined from historical development data that is recorded in software repositories.

Prediction models may provide an unrealistically optimistic estimation of model performance when (re)applied to the same sample with which were trained. To address this problem, *model validation techniques* (e.g., k-fold cross-validation) are commonly used to estimate the model performance. The model performance is used to (1) indicate how well a model will perform on unseen data [51, 54, 140, 187, 245, 257]; (2) select the top-performing prediction model [74, 115, 133, 159, 169, 232, 247]; and (3) combine several prediction models [20, 206, 241, 252].

The conclusions that are derived from defect prediction models may not be sound if the estimated performance is unrealistic or unstable. Such unrealistic and unstable performance estimates could lead to incorrect model selection in practice and inaccurate conclusions for defect prediction studies [154, 229].

Recent research has raised concerns about the *bias* (i.e., how much do the performance estimates differ from the model performance on unseen data?) and *variance* (i.e., how much do performance estimates vary when the experiment is repeated on the same data?) of model validation techniques when they are applied to defect prediction models [72, 152, 159, 169, 208, 246]. An optimal model validation technique would not overestimate or underestimate the model performance on unseen data. Moreover, the performance estimates should not vary greatly when the experiment is repeated. Mittas *et al.* [159] and Turhan *et al.* [246] point out that the random nature of sampling used by model validation techniques may

introduce bias. Myrtveit *et al.* [169] point out that a high variance in the performance estimates that are derived from model validation techniques is a critical problem in comparative studies of prediction models.

To assess the risk that defect prediction datasets pose with respect to producing unstable results, we analyze the number of *Events Per Variable* (EPV) in publicly-available defect datasets. Models that are trained using datasets where the EPV is low (i.e., below 10) are especially susceptible to unstable results. We find that 77% of defect prediction datasets have EPV values below 10, and thus are highly susceptible to producing unstable results. Hence, selecting an appropriate model validation technique is a critical experimental design choice.

Therefore, in this chapter, we explore the bias and variance of model validation techniques in both high-risk (i.e., EPV=3) and low-risk (i.e., EPV=10) contexts. Based on an analysis of 256 defect prediction studies that were published from 2000-2011, we select the 12 most popular model validation techniques for our study. The selected techniques include holdout, cross-validation, and bootstrap family techniques. We evaluate 3 types of classifiers, i.e., probability-based (i.e., naïve bayes), regression-based (i.e., logistic regression) and machine learning (i.e., random forest) classifiers. Through a case study of 18 systems spanning both proprietary and open source domains, we address the following two research questions:

(RQ1) Which model validation techniques are the least biased for defect prediction models?

Irrespective of the type of classifier, the out-of-sample bootstrap tends to provide the least biased performance estimates in terms of both threshold-dependent (e.g., precision) and threshold-independent performance measures (e.g., AUC).

(RQ2) Which model validation techniques are the most stable for defect prediction models?

Irrespective of the type of classifier, the ordinary bootstrap is the most stable model validation technique in terms of threshold-dependent and threshold-independent performance measures.

Furthermore, we derive the following practical guidelines for future defect prediction studies:

1. **The single holdout validation techniques should be avoided**, since we find that the single holdout validation tends to produce performance estimates with 46%-229% more bias and 53%-863% more variance than the top ranked model validation techniques.

2. **Researchers should use the out-of-sample bootstrap techniques instead of cross-validation or holdout techniques**, since we find that out-of-sample bootstrap validation is less prone to bias and variance in the sparse data contexts that are present in many publicly-available defect datasets.

To the best of our knowledge, this chapter is the first work to examine:

1. A large collection of model validation techniques, especially bootstrap validation, which has only rarely been explored in the software engineering literature.

2. The bias and variance of model validation techniques for defect prediction models.

3. The distribution of Events Per Variable (EPV) of publicly-available defect datasets (Section 7.2).

4. Furthermore, we introduce the Scott-Knott Effect Size Difference (ESD) test—an enhancement of the standard Scott-Knott test (which cluster distributions (e.g., distributions of the importance score of variables) into statistically distinct ranks [203]), which makes no assumptions about the underlying distribution and takes the effect size into consideration (Section 7.5.8).

7.1.1. Chapter Organization

Section 7.2 introduces the Events Per Variable (EPV) in a dataset, i.e., a metric that quantifies the risk of producing unstable results, and presents realistic examples to illustrate its potential impact. Section 7.3 introduces the studied model validation techniques. Section 7.4 situates this chapter with respect to the related work. Section 7.5 discusses the design of our case study, while Section 7.6 presents the results with respect to our two research questions. Section 7.7 provides a broader discussion of the implications of our results, while Section 7.8 derives practical guidelines for future research. Section 7.9 discloses the threats to the validity of our study. Finally, Section 7.10 draws conclusions.

7.2. Motivating Examples

Mende [147] and Jiang *et al.* [105] point out that model validation techniques may not perform well when using a small dataset. An influential characteristic in the performance of a model validation technique is the number of *Events Per Variable* (EPV) [10, 18, 186, 223], i.e., the ratio of the number of occurrences of the least frequently occurring class of the dependent variable (i.e., the events) to the number of independent variables used to train the model (i.e., the variables). Models that are trained using datasets where the EPV is low (i.e., too few events are available relative to the number of independent variables) are especially sus-

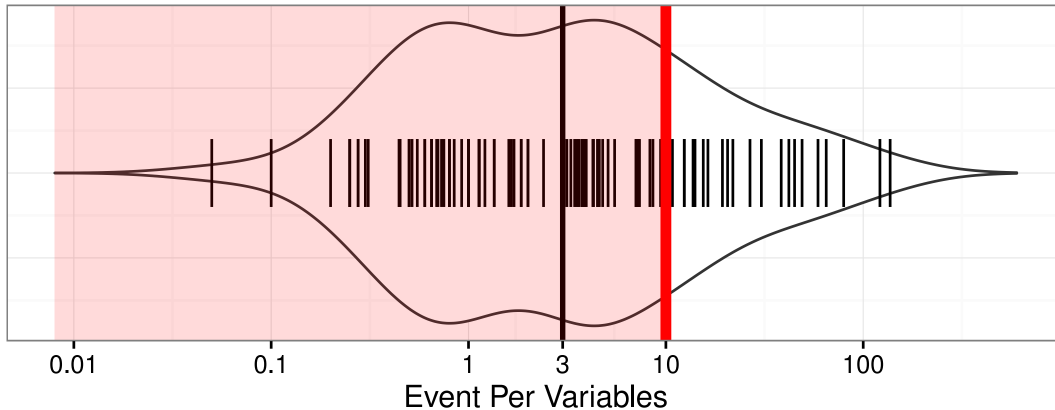


Figure 7.1.: [Empirical Study 3] The distribution of Events Per Variable (EPV) values in publicly-available defect prediction datasets. The black line indicates the median value. The vertical red line indicates the rule-of-thumb EPV value of 10 that is recommended by Peduzzi *et al.* [186]. The red-shaded area indicates the high-risk datasets that do not meet this recommendation (EPV $\leq$ 10).

ceptible to overfitting (i.e., being fit too closely to the training data) and produce unstable results [46, 186].

In order to assess whether low EPV values are affecting defect prediction studies, we analyze 101 publicly-available defect datasets (see Section 7.5.1 for details on our selection process for the studied datasets). 76 datasets are downloaded from the Tera-PROMISE repository [153], 12 clean NASA datasets are provided by Shepperd *et al.* [209], 5 datasets are provided by Kim *et al.* [118, 253], 5 datasets are provided by D'Ambros *et al.* [50, 51], and 3 datasets are provided by Zimmermann *et al.* [258].

As is often done in defect prediction studies [50, 51, 152, 192, 230, 258], we create our dependent variable by classifying the modules in these datasets as defective (i.e., #defects > 0) or clean (i.e., #defects $= 0$). We then calculate the EPV using the number of independent variables that are offered by the studied datasets.

Figure 7.1 shows the distribution of EPV values in the studied datasets using a

beanplot [112]. Beanplots are boxplots in which the horizontal curves summarize the distribution of a dataset. The long vertical black line indicates the median value. Peduzzi *et al.* [186] argue that, in order to avoid unstable results, the EPV of a dataset should be at least 10. Thus, we shade the dataset that fall into the high-risk area in red.

Summary. 78 out of 101 publicly-available defect datasets (77%) are highly susceptible to producing inaccurate and unstable results. Hence, selecting an appropriate model validation technique is a critical experimental design choice.

Figure 7.1 shows that 77% of the studied datasets have an EPV value below 10. Furthermore, the median EPV value that we observe is 3, indicating that half of the studied datasets are at a high risk of producing inaccurate and unstable results. Indeed, only 23% of the studied datasets have EPV values that satisfy the recommendation of Peduzzi *et al.* [186].

Below, we present realistic examples to illustrate impact that model validation techniques and EPV can have on the performance of defect prediction models.

7.2.1. The Risk of Producing Inaccurate Performance Estimate

To assess the risk that model validation techniques pose with respect to producing inaccurate performance estimates, we analyze the performance of a defect prediction model when it is produced by model validation techniques. We select the `JM1` NASA dataset as the subject of our analysis, since it is widely used in different defect prediction studies [79, 101, 102, 105, 133, 149, 221, 249]. We focus on the high-risk EPV context (i.e., EPV= 3) because Figure 7.1 shows that 50% of the 101 studied defect datasets have an EPV value below 3. Thus, we select a sample from the `JM1` dataset such that the EPV is 3. This sampling yields a dataset with 300 observations (i.e., 63 defective and 237 clean modules). We train a defect prediction model using logistic regression [48] with the implementation that is provided by the `glm` R function [190]. We measure the performance of the

defect prediction models using the Area Under the receiver operator characteristic Curve (AUC) [84]. Finally, we apply 12 different model validation techniques in order to produce performance estimates (see Section 7.3 for details on our selection process for the studied model validation techniques).

The performance estimates that are produced by model validation techniques vary by up to 15 percentage points. We find that the AUC performance estimates that are produced by model validation techniques vary from 0.58 to 0.73. Indeed, the ordinary bootstrap produces an estimated AUC of 0.73, while the holdout 50% validation produces an estimated AUC of 0.58. This suggests that defect prediction models may produce inaccurate performance estimates if the wrong different model validation technique is selected. However, it is not clear which model validation techniques provide the most accurate performance estimates in low EPV contexts. Hence, we evaluate each of our research questions across a variety of model validation techniques.

7.2.2. The Risk of Producing Unstable Results in Defect Datasets

To assess the risk that defect datasets pose with respect to producing unstable performance estimates, we analyze the variation in performance estimates that are produced by defect prediction models when the experiments are repeated at different EPV contexts. Similar to Section 7.2.1, we use the `JM1` NASA dataset as the subject of our analysis. We train defect prediction models using logistic regression [48] and measure the AUC performance. In order to assess whether datasets at different EPV values are affecting performance estimates, we draw sample defect datasets of different EPV values (i.e., EPV$= 1, 3, 5, 10$). Since we preserve the original rate of defective modules in these samples (i.e., 21%), our samples are of 100, 300, 500, and 1,000 modules for an EPV value of 1, 3, 5, and 10, respectively (see Section 7.5.2 for details on sample size generation). We apply

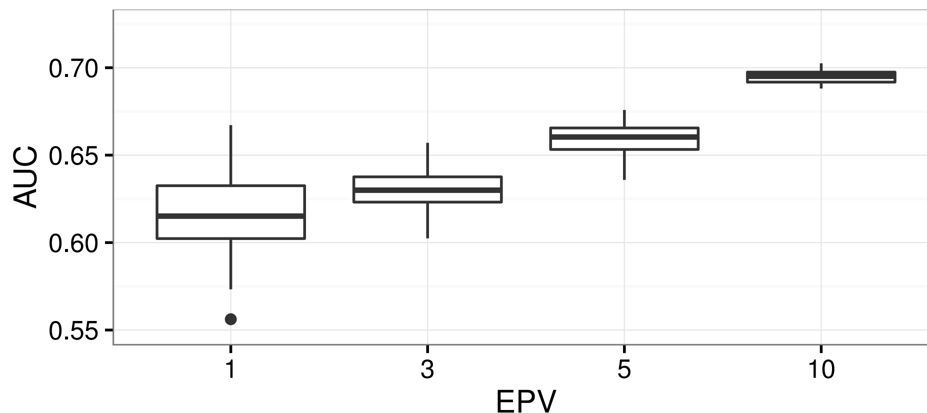


Figure 7.2.: [Empirical Study 3] The distribution of performance estimates that are produced by the repeated 10-fold cross validation at different EPV contexts when the experiment is repeated 100 times.

the repeated 10-fold cross-validation, since it is one of the most commonly-used model validation techniques in defect prediction studies (see Section 7.3 for details on our literature analysis). In order to investigate the variation of performance estimates, we repeat the experiment 100 times. Figure 7.2 shows the distribution of the 100 AUC performance estimates for each EPV context.

Performance estimates that are produced by a model that is trained in a low-risk EPV context (i.e., EPV= 10) are more stable than that of a model that is trained in a high-risk EPV context (i.e., EPV= 3). Figure 7.2 shows that the performance estimates produced by the model that is trained in a low-risk EPV context (i.e., EPV= 10) vary from 0.69 to 0.70, while the performance estimates produced by the model that is trained in a high-risk EPV context (i.e., EPV= 3) vary from 0.56 to 0.67. Thus, model validation techniques may help to counter the risk of low EPV datasets through built-in repetition (e.g., M bootstrap iterations or k folds of cross-validation). However, it is not clear which model validation techniques provide the most stable performance estimates in such low EPV contexts. Hence, we evaluate each of our research questions in both high-risk (i.e., EPV=3) and low-risk (i.e., EPV=10) contexts.

Table 7.1.: [Empirical Study 3] Summary of model validation techniques.

Family	**Technique**	**Training sample**	**Testing sample**	**Estimated performance**	**Iteration(s)**
Holdout	Holdout 0.5	50% of original	Independent: 50% of original	A single estimate	1
	Holdout 0.7	70% of original	Independent: 30% of original	A single estimate	1
	Repeated Holdout 0.5	50% of original	Independent: 50% of original	Average of performance of several samples	100
	Repeated Holdout 0.7	70% of original	Independent: 30% of original	Average of performance of several samples	100

Continued on next page

Table 7.1 – continued from previous page

Family	Technique	Training sample	Testing sample	Estimated performance	Iteration(s)
Cross Validation	Leave-one-out	N-1 of original	Independent: Subject that is not included in the training sample	Average of performance of several samples	N
	2-Fold	50% of original	Independent: 50% of original	Average of performance of several samples	2
	10-Fold	90% of original	Independent: 10% of original	Average of performance of several samples	10
	10 × 10-Fold	90% of original	Independent: 10% of original	Average of performance of several samples	100

Continued on next page

Table 7.1 – continued from previous page

Family	Technique	Training sample	Testing sample	Estimated performance	Iteration(s)
Bootstrapping	Ordinary	Bootstrap	Original	Average of performance of several samples	100
	Optimism-reduced	Bootstrap	Original	Apparent† - optimism	100
	Out-of-sample	Bootstrap	Independent: the training subjects that are not sampled in bootstrap	Average of performance of several samples	100
	.632 Bootstrap	Bootstrap	Independent: the training subjects that are not sampled in bootstrap	0.368 × Apparent† + 0.632 × average(out-of-sample)	100

† *Apparent performance* is computed from a model that is trained and tested on the original sample.

7.3. Model Validation Techniques in Defect Prediction Literature

There are a plethora of model validation techniques available. Since it is impractical to study all of these techniques, we would like to select a manageable, yet representative set of model validation techniques for our study. To do so, we analyze the defect prediction literature in order to identify the commonly used model validation techniques.

We begin by selecting 310 papers that were published between 2000-2011 from two literature surveys of defect prediction—208 papers from the survey of Hall *et al.* [82] and 102 papers from the survey of Shihab [210]. After eliminating duplicate papers, we are left with 256 unique defect prediction studies for analysis.

We read the 256 papers in order to identify the most commonly-used model validation techniques in defect prediction research. We find that 38 of the studied papers needed to be excluded from our analysis because they did not train defect prediction models. For example, many of the excluded papers performed correlation analyses, which does not require a model validation technique. Furthermore, another 35 papers also needed to be excluded because they did not use any model validation technique. Finally, our analysis below describes our findings with respect to the 183 papers that used model validation techniques.

Summary. 89 studies (49%) use k-fold cross-validation, 83 studies (45%) use hold-out validation, 10 studies (5%) use leave-one-out cross-validation, and 1 study (0.5%) uses bootstrap validation.

Below, we describe each studied model validation technique. Table 7.1 provides an overview of the three families of model validation techniques that we select based on our study of the defect prediction literature.

7.3.1. Holdout Validation

Holdout validation randomly splits a dataset into training and testing corpora according to a given proportion (e.g., 30% holdout for testing). The training corpus is only used to train the model, while the testing corpus is only used to estimate the performance of the model. Prior work has shown that holdout validation is statistically inefficient because much of the data is not used to train the prediction model [60, 62, 87, 164, 222, 225]. Moreover, an unfortunate split of the training and testing corpora may cause the performance estimate of holdout validation to be misleading. To reduce the bias and variance of holdout validation results, prior studies suggest that it be applied in a repeated fashion [29, 174, 234, 255, 258]. In this chapter, we study the commonly used variant of repeated holdout validation, where the entire process is repeated 100 times.

7.3.2. Cross-Validation

Cross-validation extends the idea of holdout validation by repeating the splitting process several times. In this chapter, we study the *k-fold cross-validation* technique, which randomly partitions the data into k folds of roughly equal size where each fold contains roughly the same proportions of the defective ratio [73, 225]. A single fold is used for the testing corpus, and the remaining $k - 1$ folds are used for the training corpus. The process of splitting is repeated k times, using each of the k folds as the testing corpus once. The average of the k results is used to estimate the true model performance.

The advantage of k-fold cross-validation is that all of the data is at one point used for both training and testing. However, selecting an appropriate value for k presents a challenge. In this chapter, we explore two popular k values (i.e., $k = 2$ and $k = 10$).

While the cross-validation technique is known to be nearly unbiased, prior studies have shown that it can produce unstable results for small samples [32,

97, 105]. To improve the variance of cross-validation results, the entire cross-validation process can be repeated several times. In this chapter, we study the commonly used variant of the 10-fold cross-validation where the entire process is repeated 10 times (i.e., 10 $\times$ 10-fold cross-validation).

Leave-One-Out Cross Validation (LOOCV) is the extreme case of k-fold cross-validation, where k is equal to the total number of observations (n). A classifier is trained n times using $n - 1$ observations, and the one remaining observation is used for testing. In a simulation experiment [35], and an experiment using effort estimation datasets [121], prior work has shown that LOOCV is the most unbiased model validation technique.

We considered two approaches to estimate model performance when using LOOCV in the defect prediction context (i.e., classifying if a module is defective or clean):

(1) Computing performance metrics once for each iteration. However, threshold-dependent performance measures achieve unrealistic results. Take, for example, the precision metric—when the one testing observation is a defective module, the precision value will either 0% (meaning the model suggests that the module is clean) or 100% (the model suggests that the module is defective). Alternatively, when the one testing observation is a clean module, the precision value is undefined because the denominator (i.e., #true positives + #false positives) is zero.

(2) Computing performance metrics using the N predicted values all at once. While this approach avoids the pitfalls of approach 1, this approach will produce a single performance value. Thus, using this approach, we cannot measure the variance of performance estimates. Furthermore, this approach yields bias values that are not comparable to the other studied model validation techniques, since they are not based on a distribution.

By considering the trade-offs of the considered approaches, we opt to apply LOOCV using approach 1 to only the Brier score (see Section 7.5.6) because it can be computed using a single observation for testing [87, 92, 188].

7.3.3. Bootstrap Validation

The bootstrap is a powerful model validation technique that leverages aspects of statistical inference [62, 87, 92, 222]. In this chapter, we study four variants of the bootstrap. We describe each variant below.

The *ordinary bootstrap* was proposed by Efron *et al.* [60]. The bootstrap process is made up of two steps:

(Step 1) A bootstrap sample of size N is randomly drawn with replacement from an original dataset that is also of size N.

(Step 2) A model is trained using the bootstrap sample and tested using the original sample.

These two steps are repeated M times to produce a distribution of model performance measurements from which the average is reported as the performance estimate. The key intuition is that the relationship between the studied dataset and the theoretical population from which it is derived is asymptotically equivalent to the relationship between the bootstrap samples and the studied dataset.

The *optimism-reduced bootstrap* is an enhancement to the ordinary bootstrap that is used to correct for upward bias [59]. The enhancement alters Step 2 of the ordinary bootstrap procedure. A model is still trained using the ith bootstrap sample, but the model is tested twice—once using the original sample and again using the bootstrap sample from which the model was trained. The optimism of the model is estimated by subtracting the performance (P) of the model when it is applied to the ith bootstrap sample from the performance of the model when it is applied to the original sample (see Equation 7.1). Optimism calculations are

repeated M times to produce a distribution from which the average optimism is derived.

$$\text{Optimism} = \frac{1}{M}\sum_{i=1}^{M}(P^{\text{boot}}_{\text{boot}(i)} - P^{\text{orig}}_{\text{boot}(i)}) \tag{7.1}$$

Finally, a model is trained using the original sample and tested on the original sample, which yields an unrealistically inflated performance value. The average optimism is subtracted from that performance value to obtain the performance estimate (see Equation 7.2).

$$\text{Optimism-reduced} = P^{\text{orig}}_{\text{orig}} - \text{Optimism} \tag{7.2}$$

The *out-of-sample bootstrap*, another enhancement to the ordinary bootstrap, is used to leverage the unused rows from the bootstrap samples. Similar to the optimism-reduced bootstrap, Step 2 of the ordinary bootstrap procedure is altered. A model is still trained using the drawn bootstrap sample, but rather than testing the model on the original sample, the model is instead tested using the rows that do not appear in the bootstrap sample [58]. On average, approximately 36.8% of the rows will not appear in the bootstrap sample, since the bootstrap sample is drawn with replacement. Again, the entire bootstrap process is repeated M times, and the average out-of-sample performance is reported as the performance estimate.

The *.632 bootstrap*, an enhancement to the out-of-sample bootstrap, is designed to correct for the downward bias in performance estimates [58]. Similar to the optimism-reduced bootstrap, a model is first trained using the original sample and tested on the same original sample to obtain an "apparent" performance value. This yields an unrealistically inflated performance value, which is combined with the upwardly-biased estimate from the out-of-sample bootstrap as follows:

$$.632 \text{ Bootstrap} = \frac{1}{M}\sum_{i=1}^{M}(0.368 \times P_{\text{orig}}^{\text{orig}} + 0.632 \times P_{\text{boot}(i)}^{-\text{boot}(i)}) \qquad (7.3)$$

The two constants in Equation 7.3 (i.e., 0.632 and 0.368) are carefully selected. The constants are selected because the training corpus of the out-of-sample bootstrap will have approximately 63.2% of the unique observations from the original dataset and the testing corpus will have 36.8% (i.e., 100% − 63.2%) of the observations. Furthermore, since the out-of-sample bootstrap tends to underestimate and the apparent performance tends to overestimate, 0.632 is used to correct the downward bias of the out-of-sample bootstrap and 0.368 is used to correct the upward bias of the apparent performance.

7.4. Related Work & Research Questions

Defect prediction models may produce an unrealistic estimation of model performance when inaccurate and unreliable model validation techniques are applied. Such inaccurate and unreliable model validation techniques could lead to incorrect model selection in practice and unstable conclusions of defect prediction studies.

Recent research has raised concerns about the bias of model validation techniques when applied to defect prediction models [72, 152, 154, 159, 169, 246]. The *bias* of a model validation technique is often measured in terms of the difference between a performance estimate that is derived from a model validation technique and the model performance on unseen data. A perfectly unbiased model validation technique will produce a performance estimate that is equivalent to the model performance on unseen data. Mittas *et al.* [159], Turhan *et al.* [246], and Myrtveit *et al.* [169] point out that the random nature of sampling that is employed by model validation techniques is a potential source of bias. Gao *et al.* [72] point out that an unfortunate division of training and testing corpora may also introduce bias.

Plenty of prior studies have explored the bias of model validation techniques in other research domains. However, these studies have arrived at contradictory conclusions. Indeed, some studies conclude that the bootstrap achieves the least biased estimate of model performance [23, 32], while others conclude that 10-fold cross validation achieves the least biased estimate of model performance [124,164]. Other work concludes that LOOCV should be used to achieve the most accurate estimate of model performance [35, 121].

We suspect that the contradictory conclusions of prior work are largely related to changes in experimental context. For example, some studies conduct simulation experiments [32, 53, 57, 61, 97, 116], while others use empirical data [32, 57, 121, 124, 164].

The lack of consistency in the conclusions of prior work makes it hard to derive practical guidelines about the most appropriate model validation technique to use in defect prediction research. To address this, we formulate the following research question:

(RQ1) Which model validation techniques are the least biased for defect prediction models?

In addition to the bias of a model validation technique, it is also important to evaluate its variance [105, 147, 154, 168, 169, 208]—the performance estimates should not vary broadly when the experiment is repeated. The *variance* of a model validation technique is typically measured in terms of the variability of the estimated performance, i.e., how much do performance estimates vary when the experiment is repeated on the same data. Myrtveit *et al.* [169] point that high variance in performance estimates from model validation techniques is a critical challenge in comparative studies of prediction models. Shepperd and Kadoda [208] show that the performance of defect prediction models that are trained using different subsamples that are drawn from the same underlying

dataset vary widely. To structure our analysis of the variance of model validation techniques, we formulate the following research question:

(RQ2) Which model validation techniques are the most stable for defect prediction models?

7.5. Case Study Design

In this section, we discuss our selection criteria for the studied systems and then describe the design of our case study experiment that we perform in order to address our research questions.

7.5.1. Studied Datasets

In selecting the studied datasets, we identified three important criteria that needed to be satisfied:

- **Criterion 1 — Different corpora**: Recent research points out that the performance of defect prediction models may be limited to the dataset from which they are trained [221, 233]. To extend the generality of our conclusions, we choose to train our defect prediction models using systems from different corpora and domains.

- **Criterion 2 — Sufficient EPV**: Since we would like to study cases where EPV is low-risk (i.e, = 10) and high-risk (i.e, = 3), the systems that we select for analysis should begin with a low-risk EPV. Our rationale is that we prefer to control for dataset-specific model performance by generating low and high-risk EPV settings using the same dataset. This ensures that a comparison of EPV is derived from the same context. Essentially, we

disregard datasets with low initial EPV because we prefer to undersample to generate low and high-risk EPV datasets from an initially high EPV dataset. For example, if we were to select datasets with an initial EPV of 5, we would need to over-sample the defective class in order to raise the EPV to 10. However, the defective class of a system with an initial EPV of 15 can be under-sampled in order to lower the EPV to 10.

- **Criterion 3 — Sane defect data**: Since it is unlikely that more software modules have defects than are free of defects, we choose to study datasets that have a rate of defective modules below 50%.

To satisfy criterion 1, we begin our study using the 101 publicly-available defect datasets described in Section 7.2.2. We provide a list of the 101 defect datasets in Appendix A. To satisfy criterion 2, we exclude the 78 datasets that we found to have an EPV value lower than 10 in Section 7.2.2. To satisfy criterion 3, we exclude an additional 5 datasets because they have a defective ratio above 50%.

Table 7.2 provides an overview of the 18 datasets that satisfy our criteria for analysis. To combat potential bias in our conclusions, the studied datasets include proprietary and open source systems with varying size, domain, and defective ratio.

Figure 7.3 provides an overview of the approach that we apply to each studied system. The crux of our approach is that we calculate model performance on unseen data such that the performance estimates that are derived from model validation techniques can be compared to the model performance on unseen data. The approach is repeated 1,000 times to ensure that the results are robust and that they converge

Project	**System**	**Defective Ratio**	**#Files**	**#Metrics**	**EPV**
NASA	JM1[1]	21%	7,782	21	80
	PC5[1]	28%	1,711	38	12
Proprietary	Prop-1[2]	15%	18,471	20	137
	Prop-2[2]	11%	23,014	20	122
	Prop-3[2]	11%	10,274	20	59
	Prop-4[2]	10%	8,718	20	42
	Prop-5[2]	15%	8,516	20	65
Apache	Camel 1.2[2]	36%	608	20	11
	Xalan 2.5[2]	48%	803	20	19
	Xalan 2.6[2]	46%	885	20	21
Eclipse	Platform 2.0[3]	14%	6,729	32	30
	Platform 2.1[3]	11%	7,888	32	27
	Platform 3.0[3]	15%	10,593	32	49
	Debug 3.4[4]	25%	1,065	17	15
	SWT 3.4[4]	44%	1,485	17	38
	JDT[5]	21%	997	15	14
	Mylyn[5]	13%	1,862	15	16
	PDE[5]	14%	1,497	15	14

[1]Provided by Shepperd *et al.* [209].
[2]Provided by Jureczko *et al.* [108].
[3]Provided by Zimmermann *et al.* [258].
[4]Provided by Kim *et al.* [118].
[5]Provided by D'Ambros *et al.* [50, 51].

Table 7.2.: [Empirical Study 3] An overview of the studied systems.

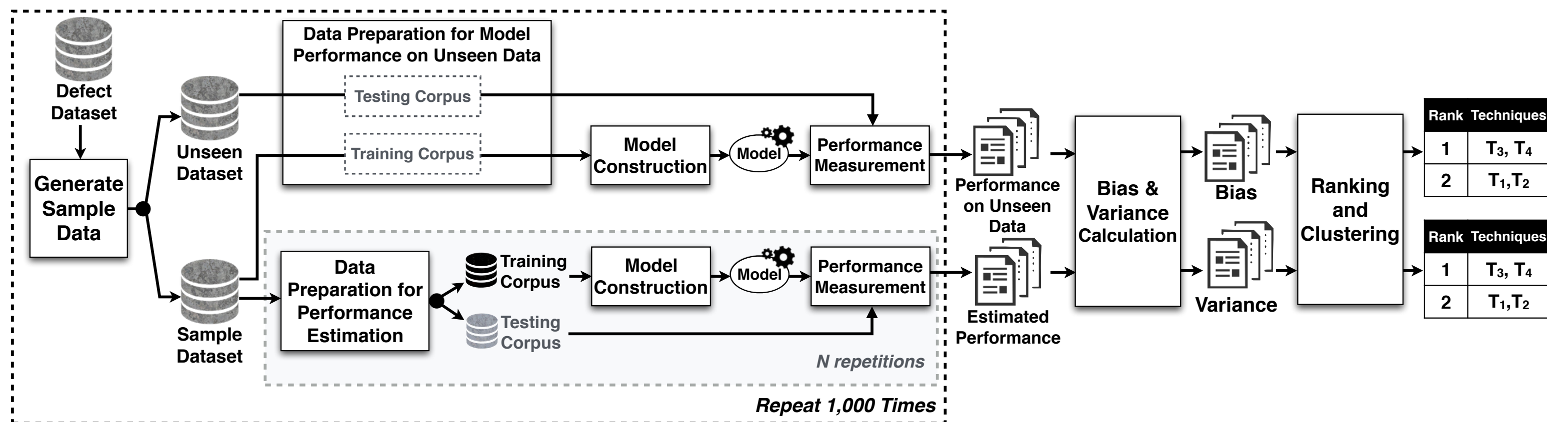


Figure 7.3.: [Empirical Study 3] An overview of the design of our case study experiment.

7.5.2. Generate Sample Data

In order to compare the studied model validation techniques, we begin by creating sample and unseen datasets using historical data from a studied dataset. The sample dataset is created in order to train our model for performance on unseen data, while the unseen dataset is used to test it. The sample dataset is also used to train and test models using the studied model validation techniques. The performance on the unseen dataset is compared to the performance estimates that are derived from the studied model validation techniques.

Sample Dataset

In order to produce our sample datasets, we select observations with replacement from the input dataset, while controlling for two confounding factors:

(C1) The defective ratio: While generating sample datasets, we preserve the defective ratio of the original dataset to ensure that the sample and unseen datasets are representative of the original dataset. Thus, the defective ratio of the unseen datasets are the same as the defective ratio of the original datasets.

(C2) The EPV: As mentioned above, we explore high-risk (EPV = 3) and low-risk (EPV = 10) contexts.

Note that by controlling for C1 and C2, we have specified the number of defective modules (and indirectly, the number of clean modules) in the sample dataset. For example, the size of the original JM1 dataset is 7,782 modules. To generate a sample of the JM1 dataset with EPV values of 3 and 10, we need to preserve (1) the defective ratio of 21%; and (2) the 21 variables. Thus, a sample size of the JM1 dataset with an EPV of 3 is 300 modules (i.e., 63 defective and 237 clean modules). Similarly, a sample size of the JM1 dataset with an EPV of 10 is 1000 (i.e., 210 defective and 790 clean modules).

7.5.3. Data Preparation for Model Performance on Unseen Data

Unfortunately, the population of the input dataset is unknown, so the actual model performance on unseen data cannot be directly measured. Inspired by previous studies [32, 97, 116, 124, 125], we estimate the model performance using *unseen data*, i.e., the observations from the input dataset that do not appear in the sample dataset. To do so, we use the sample dataset as the training corpus for training a defect prediction model and we use the unseen dataset as the testing corpus to measure the performance of that defect prediction model.

7.5.4. Data Preparation for Performance Estimation

In order to compare the estimates of the studied model validation techniques to the model performance on unseen data, we apply the model validation techniques to the sample dataset. To split the sample dataset into training and testing corpora, we use: (1) the `createDataPartition` function of the `caret` R package for the holdout family of model validation techniques [129, 131], (2) the `createFolds` function of the `caret` R package for the cross-validation family of model validation techniques [129, 131], and (3) the `boot` function of the `boot` R package for the bootstrap family of model validation techniques [41].

7.5.5. Model Construction

We select three types of classifiers that are often used in defect prediction literature. These types of classifiers include probability-based (i.e., naïve bayes [56]), regression-based (i.e., logistic regression [48]) and machine learning-based (i.e., random forest [34]) classifiers.

Naïve bayes is a probability-based technique that assumes that all of the predictors are independent of each other [56]. We use the implementation of naïve bayes that is provided by the `naiveBayes` R function [155].

Logistic regression measures the relationship between a categorical dependent variable and one or more independent variables [48]. We use the implementation of logistic regression that is provided by the `glm` R function [190].

Random forest is a machine learning classifier that constructs multiple decision trees from bootstrap samples [34]. Since each tree in the forest may return a different outcome, the final class of a software module is computed by aggregating the votes from all trees. We use the implementation of random forest that is provided by the `randomForest` R function [135].

To ensure that the training and testing corpora have similar characteristics, we do not re-balance or re-sample the training data, as suggested by Turhan [244].

Normality Adjustment

Analysis of the distributions of our independent variables reveals that they are right-skewed. As suggested by previous research [103, 152], we mitigate this skew by log-transforming each independent variable ($ln(x+1)$) prior to using them to train our models.

7.5.6. Performance Measurement

We apply the defect prediction models that we train using the training corpus to the testing corpus in order to measure their performance. We use both threshold-dependent and threshold-independent performance measures to quantify the performance of our models. We describe each performance metric below.

Threshold-Dependent Performance Measures

When applied to a module from the testing corpus, a defect prediction model will report the probability of that module being defective. In order to calculate the threshold-dependent performance measures, these probabilities are transformed into a binary classification (defective or clean) using a threshold value of 0.5,

i.e., if a module has a predicted probability above 0.5, it is considered defective; otherwise, the module is considered clean.

Using the threshold of 0.5, we compute the precision and recall performance measures. *Precision* measures the proportion of modules that are classified as defective, which are actually defective ($\frac{TP}{TP+FP}$). *Recall* measures the proportion of actually defective modules that were classified as such ($\frac{TP}{TP+FN}$). We use the `confusionMatrix` function of the `caret` R package [129, 131] to compute the precision and recall of our models.

Threshold-Independent Performance Measures

Prior research has argued that the precision and recall are unsuitable for measuring the performance of defect prediction models because they: (1) depend on an arbitrarily-selected threshold (typically 0.5) [9, 17, 133, 192, 211], and (2) are sensitive to imbalanced data [52, 93, 139, 147, 227]. Thus, we also use three threshold-independent performance measures to quantify the performance of our defect prediction models.

First, we use the *Area Under the receiver operator characteristic Curve* (AUC) [84] to measure the discrimination power of our models. The AUC measures a classifier's ability to discriminate between defective and clean modules (i.e., do the defective modules tend to have higher predicted probabilities than clean modules?). AUC is computed by measuring the area under the curve that plots true positive rate against the false positive rate while varying the threshold that is used to determine whether a module is classified as defective or not. Values of AUC range between 0 (worst classifier performance) and 1 (best classifier performance).

In addition to the discrimination power, practitioners often use the predicted probabilities to rank the defect-prone files [163, 170, 214, 258]. Shihab *et al.* [214] point out that practitioners often use the predicted probability to make decisions.

Mockus *et al.* [163] point out that the appropriate range of probability values is important to make an appropriate decision (e.g., high-reliability systems may require a lower cutoff value than 0.5). However, the AUC does not capture all of the dimensions of a prediction model [55, 87, 222, 224]. To measure the accuracy of the predicted probabilities, we use the Brier score and the calibration slope.

We use the *Brier score* [37, 201] to measure the distance between the predicted probabilities and the outcome. The Brier score is calculated as:

$$\mathrm{B} = \frac{1}{N}\sum_{i=1}^{N}(f_t - o_t)^2 \tag{7.4}$$

where f_t is the predicted probability, o_t is the outcome for module t encoded as 0 if module t is clean and 1 if it is defective, and N is the total number of modules. The Brier score ranges from 0 (best classifier performance) to 1 (worst classifier performance).

Finally, we use the *calibration slope* to measure the direction and spread of the predicted probabilities [48, 55, 87, 89, 158, 223, 224]. The calibration slope is the slope of a logistic regression model that is trained using the predicted probabilities of our original defect prediction model to predict whether a module will be defective or not [48]. A calibration slope of 1 indicates the best classifier performance and a calibration slope of 0 (or less) indicates the worst classifier performance. We use the `val.prob` function of the `rms` R package [88] to calculate the Brier score, AUC, and calibration slope.

7.5.7. Bias and Variance Calculation

We calculate each performance measure using the model performance on unseen data and the model validation techniques. In order to address RQ1, we calculate the *bias*, i.e., the absolute difference between the performance that we derive from the model validation techniques and the model performance on unseen data. In order to address RQ2, we calculate the *variance* of the performance measures that we derive from the model validation techniques (in terms of standard deviation).

7.5.8. Ranking and Clustering

Finally, we group the model validation techniques into statistically distinct ranks according to the their bias and variance using the Scott-Knott Effect Size Difference (ESD) test.

The Scott-Knott Effect Size Difference (ESD) test

The Scott-Knott test [203] uses hierarchical cluster analysis to partition the set of treatment means into statistically distinct groups ($\alpha = 0.05$). Two major limitations of the Scott-Knott test are that (1) it assumes that the data should be normally distributed; and (2) it may create groups that are trivially different from one another. To strengthen the Scott-Knott test, we propose the Scott-Knott Effect Size Difference (ESD) test—a variant of the Scott-Knott test that is normality and effect size aware. The Scott-Knott ESD test will (1) correct the non-normal distribution of an input dataset; and (2) merge any two statistically distinct groups that have a negligible effect size into one group.

Normality correction. The Scott-Knott test assumes that the data under analysis are normally distributed. Thus, we mitigate the skewness by log-transforming each treatment ($ln(x+1)$), since it is a commonly-used transformation technique in software engineering research [103, 152].

Effect size correction. To quantify the effect size, we use Cohen's delta (d) [44], which is the difference between two means divided by the standard deviation of the data:

$$d = \frac{\bar{x}_1 - \bar{x}_2}{s.d.} \tag{7.5}$$

where the magnitude is assessed using the thresholds provided in Cohen [45], i.e. $|d| < 0.2$ "negligible", $|d| < 0.5$ "small", $|d| < 0.8$ "medium", otherwise "large".

We implement the Scott-Knott ESD test [228] based on the implementation

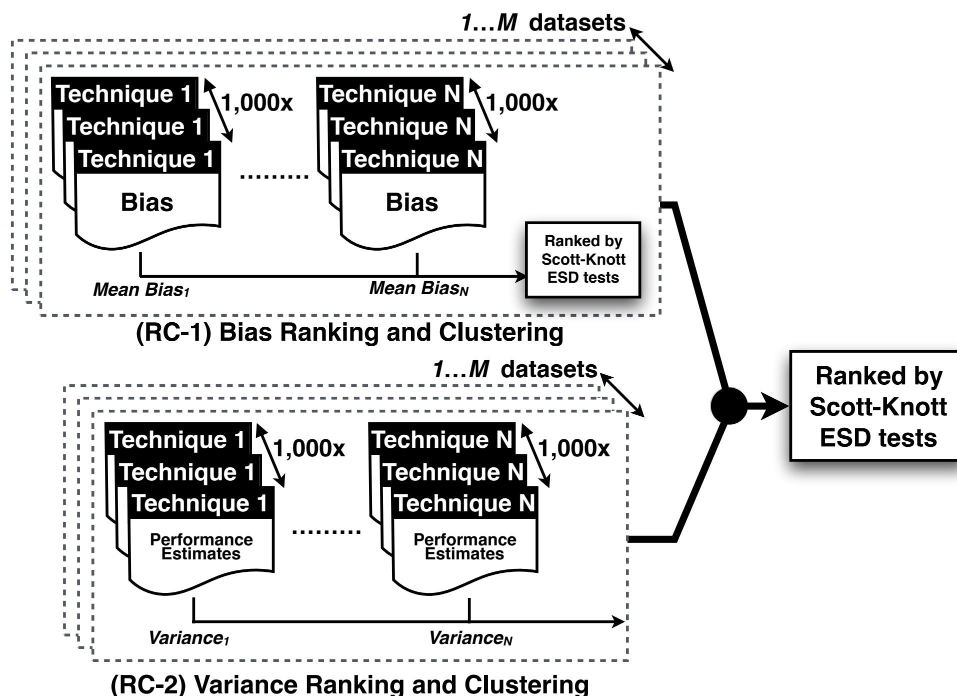


Figure 7.4.: [Empirical Study 3] An overview of our ranking and clustering approach.

of the Scott-Knott test that is provided by the `ScottKnott` R package [98] and the implementation of Cohen's delta provided by the `effsize` R package [240]. The `R` implementation of the ScottKnott ESD test is provided in Appendix E and published online as an R package.[1]

Ranking and Clustering Approach

Figure 7.4 provides an overview of our approach. First, to identify the least biased model validation techniques, similar to our prior work [74], we perform a double Scott-Knott test. We apply a Scott-Knott ESD test on the bias results from the 1,000 iterations that are performed for each studied dataset individually. After performing this first set of Scott-Knott ESD tests, we have a list of ranks

[1] https://github.com/klainfo/ScottKnottESD

for each model validation technique (i.e., one rank from each studied dataset). We provide these lists of ranks to a second run of the Scott-Knott's ESD test, which produces a ranking of model validation techniques across all of the studied datasets. We perform the first set of Scott-Knott ESD tests in order to control for dataset-specific model performance, since some datasets may be more or less susceptible to bias than others. Since only one variance value is computed for the 1,000 runs of one technique, only a single Scott-Knott ESD test needs to be applied to the variance values.

7.6. Case Study Results

In this section, we present the results of our case study with respect to our two research questions.

(RQ1) Which model validation techniques are the least biased for defect prediction models?

In order to address RQ1, we compute the bias in terms of precision, recall, Brier score, AUC, and calibration slope. We then present the results with respect to probability-based (RQ1-a), regression-based (RQ1-b), and machine learning-based (RQ1-c) classifiers. Figure 7.5 shows the statistically distinct Scott-Knott ESD ranks of the bias of the studied model validation techniques.

RQ1-a: Probability-Based Classifiers

The .632, out-of-sample, and optimism-reduced bootstraps are the least biased model validation techniques for naïve bayes classifiers. Figure 7.5a shows that the .632, out-of-sample, and optimism-reduced bootstraps are the only model validation techniques that consistently appear in the top Scott-Knott ESD rank

in terms of precision, recall, Brier score, AUC, and calibration slope bias in both EPV contexts.

Other techniques appear consistently in the top rank for some metrics, but not across all metrics. For example, the repeated 10-fold and 10-fold cross-validation techniques appear in the top rank in terms of precision, recall, AUC, and Brier score, but not in calibration slope for the high-risk EPV context. We suspect that the calibration slope that we derived from the 10-fold cross-validation is upwardly-biased because of the scarcity of defective modules in the small testing corpus of 10-fold cross-validation (i.e., 10% of the input dataset).

While advanced model validation techniques with built-in repetitions (i.e., repeated holdout validation, cross-validation and bootstrap validation) tend to produce the least biased performance estimates, single-repetition holdout validation tends to produce the most biased performance estimates. Indeed, single-repetition holdout validation tends to produce performance estimates with 46%-229% more bias than the top-ranked model validation techniques, suggesting that researchers should avoid using single holdout validation, since it tends to produce overly optimistic or pessimistic performance estimates. However, repeated holdout validation produces up to 43% less biased performance estimates than single holdout validation does. Indeed, by repeating the holdout validation technique, the amount of bias of the high-risk EPV context is substantially reduced by 30%, 20%, 32%, and 43% for precision, recall, AUC, and calibration slope, respectively. Indeed, single-repetition holdout validation should be avoided.

RQ1-b: Regression-Based Classifier

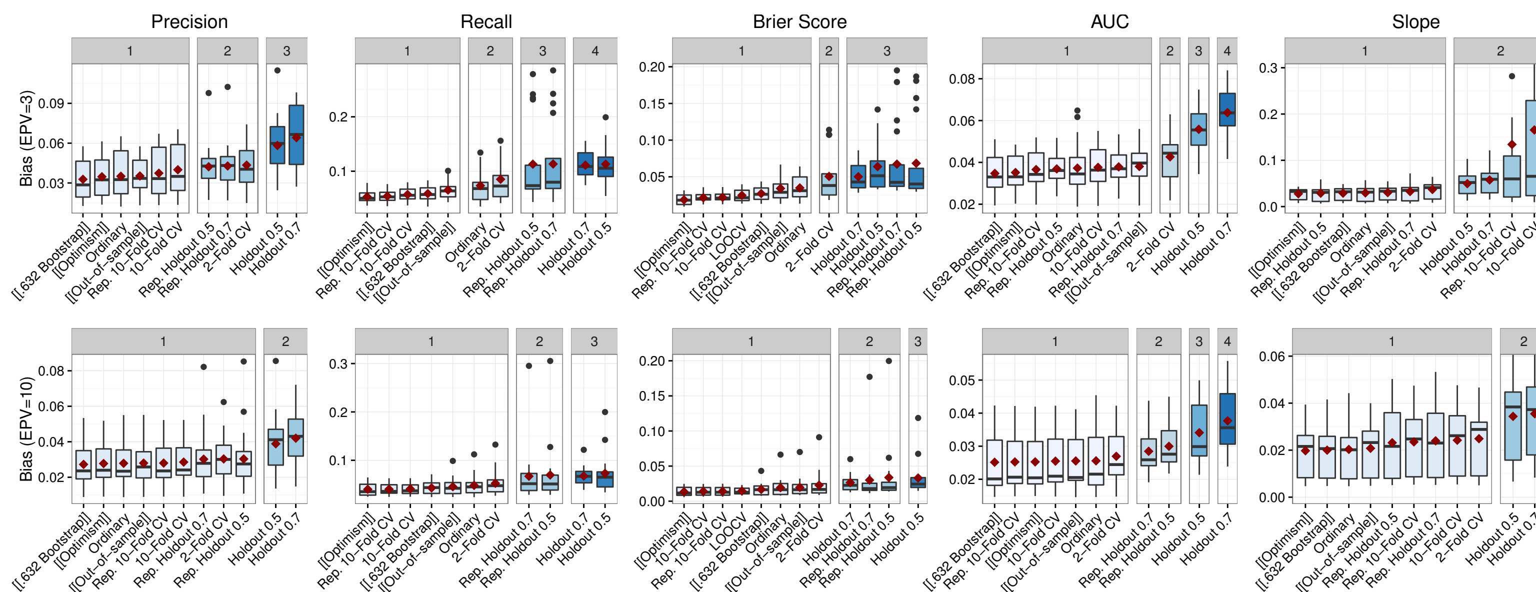
The out-of-sample bootstrap is the least biased model validation technique for logistic regression classifiers. Figure 7.5b shows that the out-of-sample bootstrap is the only model validation technique that consistently appears in the top Scott-Knott ESD rank in terms of precision, recall, Brier score, AUC, and calibration

slope bias in both EPV contexts.

Other techniques appear consistently in the top rank with respect to the low-risk EPV context, but not the high-risk EPV context. For example, the .632 and optimism-reduced bootstraps, and the 10-fold and repeated 10-fold cross-validation techniques consistently appear in the top rank in the low-risk EPV context, but do not consistently appear in the top ranks of the high-risk EPV context. Indeed, Figure 7.5b indicates that the .632 bootstrap tends to produce recall estimates with 21% more bias than the top-ranked technique in the high-risk EPV context, while the optimism-reduced bootstrap, 10-fold, and repeated 10-fold cross-validation techniques tend to produce precision estimates that are 29% more biased than the top ranked technique in the high-risk EPV context. Furthermore, the bias in performance estimates in the high-risk EPV context is larger than those of the low-risk EPV context. This indicates that the performance estimates that are derived from the least biased model validation techniques in the high-risk EPV context tend to produce 46% more bias than the least biased model validation techniques in the low-risk EPV context. *A lack of generality across EPV contexts of some model validation techniques and an increase of performance bias in the high-risk EPV context, suggests that selecting a robust model validation technique is an especially important experimental design choice in the high-risk EPV contexts that are commonplace in defect datasets (cf. Section 7.2).*

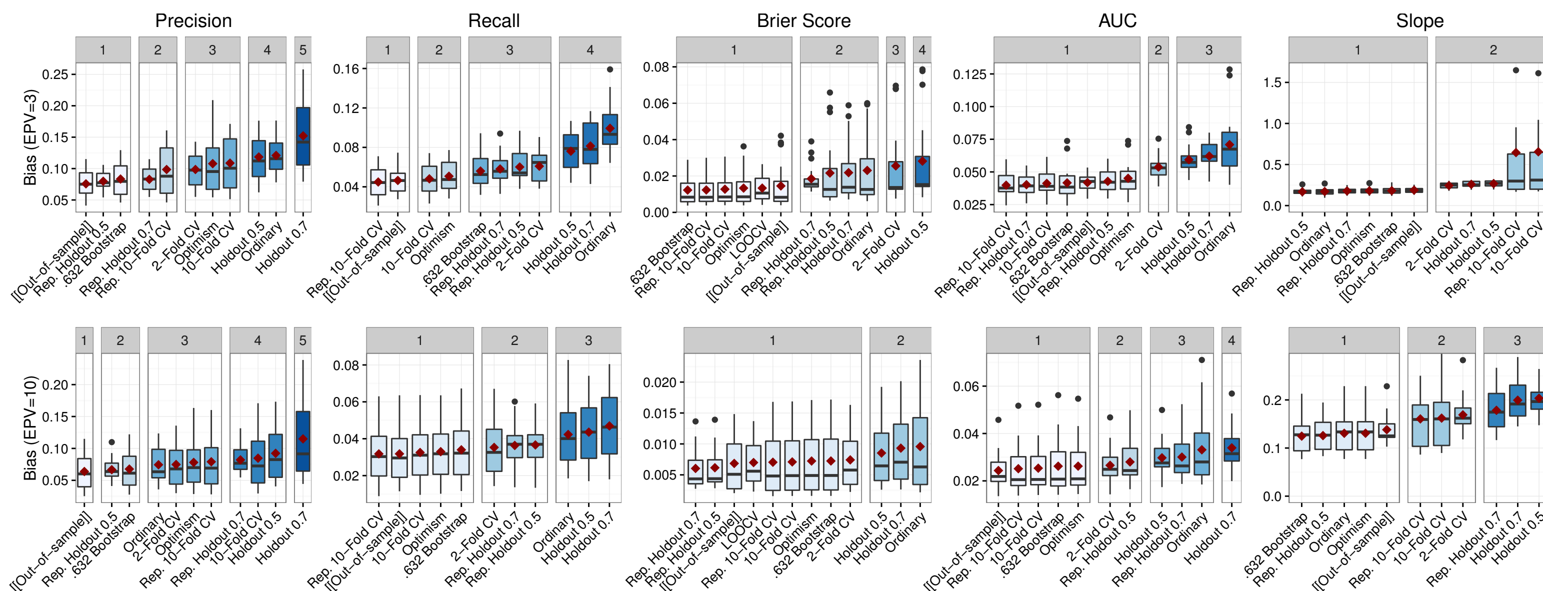
RQ1-c: Machine Learning-Based Classifier

The out-of-sample bootstrap and 2-fold cross-validation are the least biased model validation techniques for random forest classifiers. Figure 7.5c shows that the out-of-sample bootstrap and 2-fold cross-validation techniques consistently appear in the top Scott-Knott ESD rank in terms of precision, recall, Brier score, AUC, and calibration slope bias in both EPV contexts.



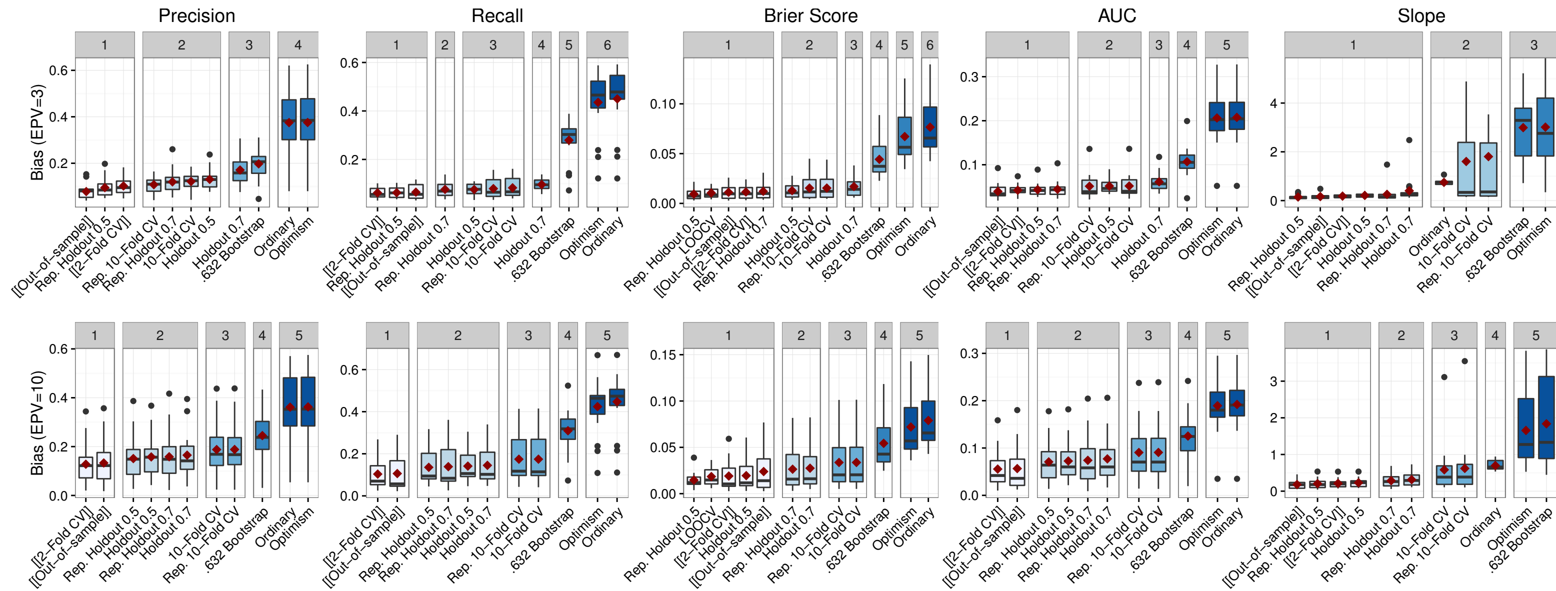
(a) [Empirical Study 3] Probability-Based Classifiers

Figure 7.5.: [Empirical Study 3] The Scott-Knott ESD ranking of the *bias* of model validation techniques. The technique in a bracket indicates the top-performing technique for each classifier type. The red diamond indicates the average amount of bias across our studied datasets.



(b) [Empirical Study 3] Regression-Based Classifiers

Figure 7.5.: [Empirical Study 3] The Scott-Knott ESD ranking of the *bias* of model validation techniques. The technique in a bracket indicates the top-performing technique for each classifier type. The red diamond indicates the average amount of bias across our studied datasets. (Cont.)



(c) [Empirical Study 3] Machine Learning-Based Classifiers

Figure 7.5.: [Empirical Study 3] The Scott-Knott ESD ranking of the *bias* of model validation techniques. The technique in a bracket indicates the top-performing technique for each classifier type. The red diamond indicates the average amount of bias across our studied datasets. (Cont.)

While the .632 and optimism-reduced bootstraps are less biased for the naïve bayes and logistic regression classifiers, the .632 and optimism-reduced bootstraps are quite biased for random forest classifiers. We suspect that the upward-bias in the .632 and optimism-reduced bootstraps have to do with the low training error rate of random forest. Since the low training error rate often produces a high apparent performance, the calculation of .632 and optimism-reduced bootstraps, which are partly computed using the apparent performance, are biased. This suggests that the .632 and optimism-reduced bootstraps are not appropriate for classifiers that have low training error rates, such as random forest. This finding is also complementary to Kohavi *et al.* [124], who suggests that repeated 10-fold cross-validation should be used. On the other hand, Kohavi *et al.* [124] did not evaluate the out-of-sample bootstrap. In our analysis, we find that the out-of-sample bootstrap is less-prone to bias than repeated 10-fold cross-validation is.

Summary. Irrespective of the type of classifier, the out-of-sample bootstrap tends to provide the least biased performance estimates in terms of both threshold-dependent and threshold-independent performance metrics.

(RQ2) Which model validation techniques are the most stable for defect prediction models?

In order to address RQ2, we compute the variance in terms of precision, recall, Brier score, AUC, and calibration slope. We then present the results with respect to probability-based (RQ2-a), regression-based (RQ2-b), and machine learning-based (RQ2-c) classifiers. Figure 7.6 shows the statistically distinct Scott-Knott ESD ranks of the variance of the studied model validation techniques.

RQ2-a: Probability-Based Classifier

All variants of the bootstrap, repeated 10-fold cross validation, and repeated holdout validation techniques are the most stable model validation techniques for naïve bayes classifiers. Figure 7.6a shows that, in addition to the repeated 10-fold cross-validation and the repeated holdout techniques, the .632, optimism-reduced, out-of-sample, and ordinary bootstrap techniques are the only model validation techniques that consistently appear in the top Scott-Knott ESD rank in terms of both precision and recall variance in both EPV contexts.

While advanced model validation techniques with built-in repetitions (i.e., repeated holdout validation, cross-validation and bootstrap validation) tend to yield the most stable performance estimates, single holdout validation, which only uses a single iteration, tends to yield the least stable performance estimates. Indeed, single holdout validation tends to produce performance estimates with 53%-863% more variance than the most stable model validation techniques. Moreover, the repeated holdout validation produces 40%-68% more stable performance estimates than single holdout validation does. Indeed, by repeating the holdout validation technique, the amount of variance in the high-risk EPV context is substantially reduced by 45%, 45%, 48%, 40%, 68% for precision, recall, Brier score, AUC, and calibration slope, respectively. Indeed, holdout validation should be avoid unless it is repeated.

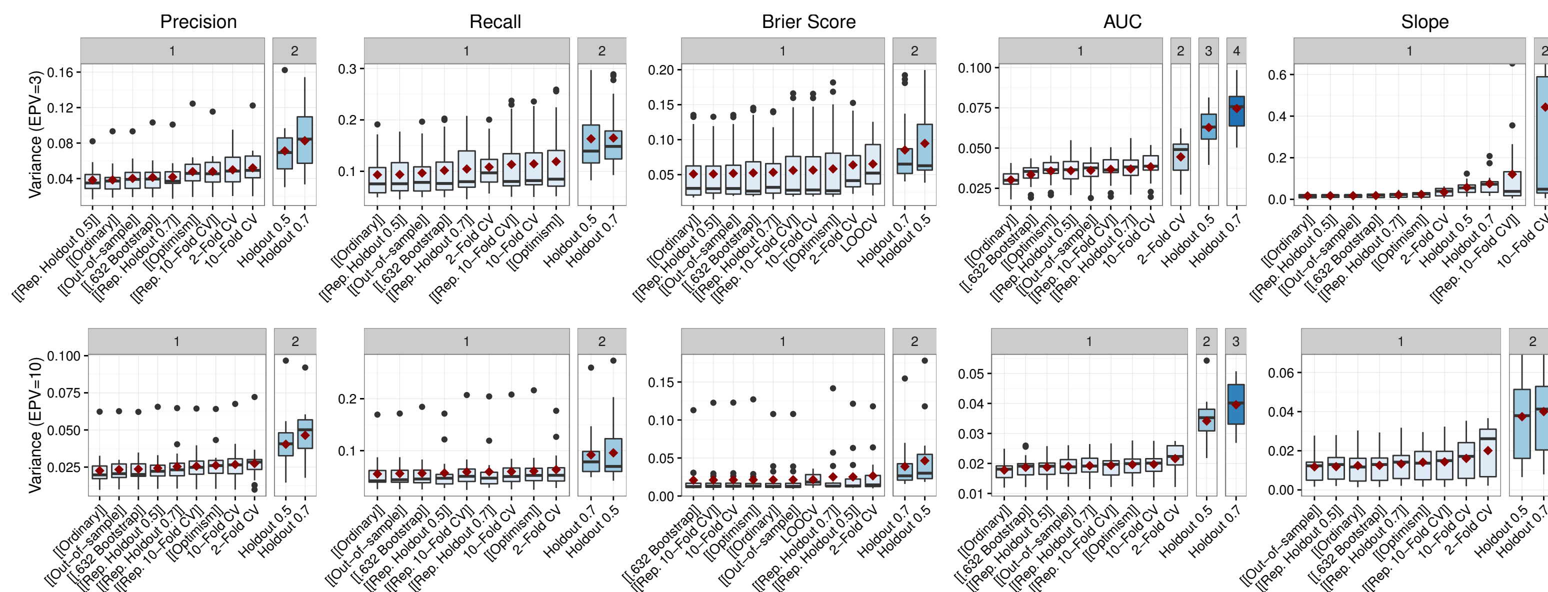
RQ2-b: Regression-Based Classifier

The .632 and ordinary bootstraps are the most stable model validation techniques for logistic regression classifiers. Figure 7.6b shows that the .632 and ordinary bootstraps are the only model validation techniques that consistently appear in the top Scott-Knott ESD rank in terms of precision, recall, Brier score, AUC, and calibration slope variance in both EPV contexts.

Other techniques appear consistently in the top rank for the low-risk EPV

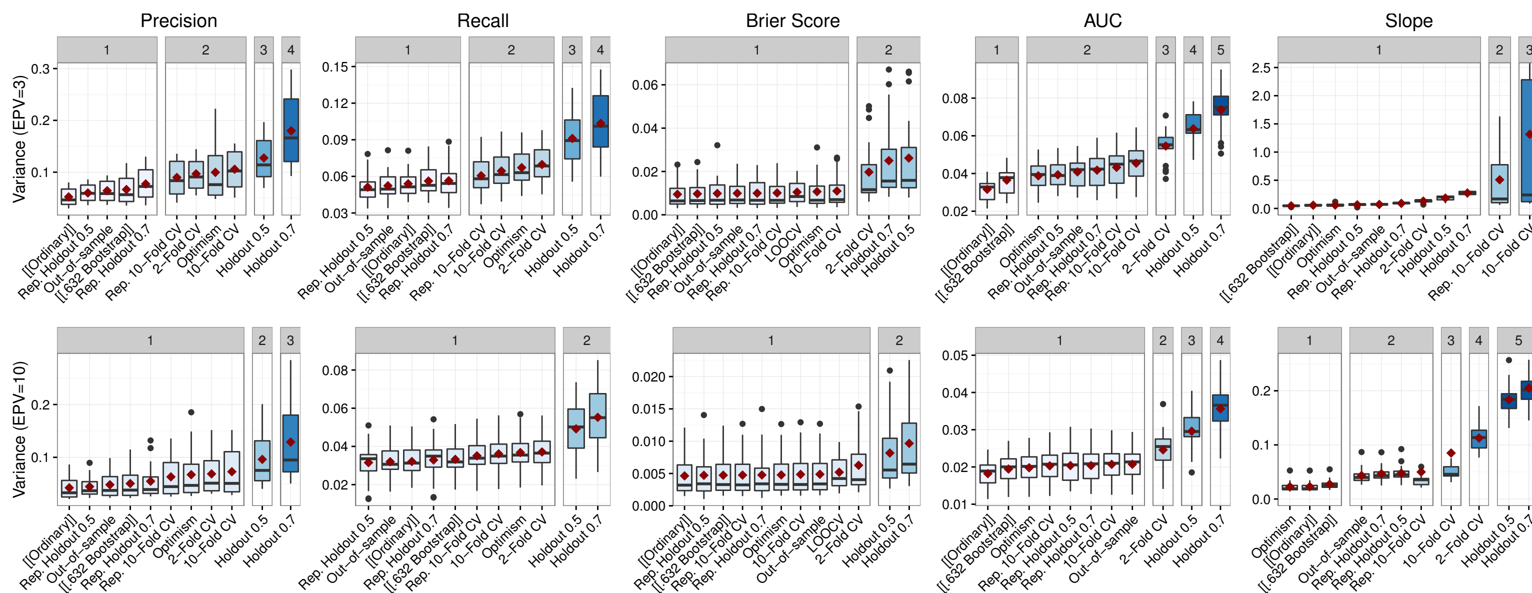
context, but not for high-risk EPV context. For example, the optimism-reduced bootstrap and the 10-fold cross-validation techniques appear consistently in the top rank in terms of threshold-dependent metrics in the low-risk EPV context, but not in the high-risk EPV context. Indeed, Figure 7.6b shows that there is (on average) a 66% increase in terms of precision variance and 91% increase in terms of recall variance when those techniques are used in the high-risk EPV context. We suspect that the precision and recall that are derived from the cross-validation family are less stable in the high-risk EPV context because an unfortunate split of the training and testing corpora may result in too few defective modules appearing in the testing corpus. Hence, very few correct (or incorrect) predictions can have a very large impact on cross-validation performance estimates.

Furthermore, the optimism-reduced and out-of-sample bootstraps, as well as the repeated 10-fold cross-validation techniques appear consistently in the top rank in terms of threshold-independent metrics in the low-risk EPV context, but not in the high-risk EPV context. Indeed, Figure 7.6b shows that there is (on average) a 110% increase in terms of AUC variance when those techniques are used in the high-risk EPV context. We suspect that the AUC is less stable because the AUC derived from these techniques are calculated from a small amount of data in the testing corpus. For example, 10-fold cross-validation is validated on 10% of the original data and the optimism-reduced and out-of-sample bootstraps are validated on 36.8% of the original data. Conversely, the performance estimates when using the ordinary bootstrap that are computed using a sample that has the same size as the original data is likely to produce the most stable performance estimates. On the other hand, the .632 bootstrap, which is an enhancement of the out-of-sample bootstrap (*cf.* Section 7.4), is also generally robust to the AUC variance. A lack of generality across EPV contexts of some model validation techniques and an increase of performance variance in the high-risk EPV context, indeed, suggest that the EPV plays a major role in the stability of performance estimates.



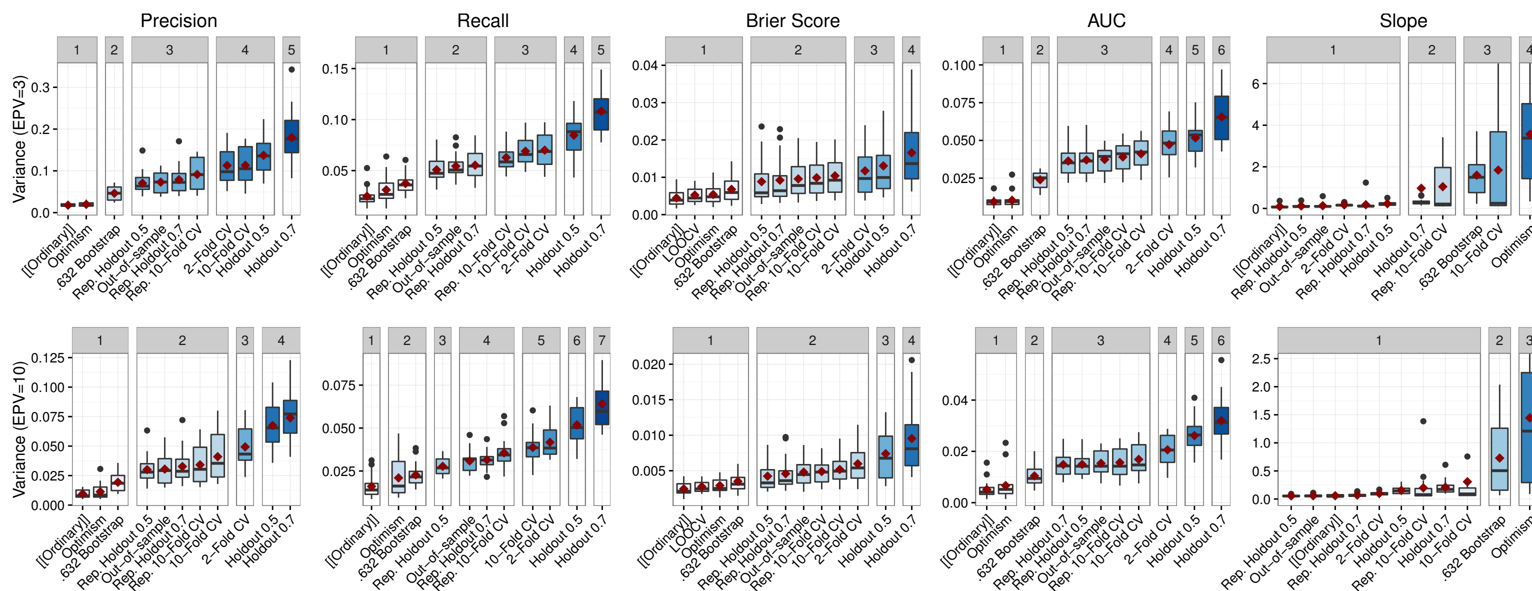
(a) [Empirical Study 3] Probability-Based Classifiers

Figure 7.6.: [Empirical Study 3] The Scott-Knott ESD ranking of the *variance* of model validation techniques. The technique in a bracket indicates the top-performing technique for each classifier type. The red diamond indicates the average amount of variance across our studied datasets.



(b) [Empirical Study 3] Regression-Based Classifiers

Figure 7.6.: [Empirical Study 3] The Scott-Knott ESD ranking of the *variance* of model validation techniques. The technique in a bracket indicates the top-performing technique for each classifier type. The red diamond indicates the average amount of variance across our studied datasets. (Cont.)



(c) [Empirical Study 3] Machine Learning-Based Classifiers

Figure 7.6.: [Empirical Study 3] The Scott-Knott ESD ranking of the *variance* of model validation techniques. The technique in a bracket indicates the top-performing technique for each classifier type. The red diamond indicates the average amount of variance across our studied datasets. (Cont.)

RQ2-c: Machine Learning-Based Classifier

The ordinary bootstrap is the most stable model validation technique for random forest classifiers. Figure 7.6c shows that the ordinary bootstrap is the only model validation technique that consistently appears in the top Scott-Knott ESD rank in terms of precision, recall, Brier score, AUC, and calibration slope variance in both EPV contexts.

We suspect that the most stable performance estimates that are being produced by the ordinary bootstrap because of the statistical inference properties of the bootstrap itself. Indeed, the key intuition is that the relationship between the performance that is derived from a studied dataset and the true performance that would be derived from the population of defect datasets is asymptotically equivalent to the relationship between the performance that is derived from a bootstrap sample and the performance that is derived from the studied dataset.

In addition to yielding highly biased results for random forest classifiers (*cf.* RQ1-c), the .632 and optimism-reduced bootstraps also tend to produce the least stable performance estimates in terms of calibration slope. Indeed, this further supports our suspicion that the .632 and optimism-reduced bootstraps are not appropriate for classifiers that have low training error rates, such as random forest.

Summary. Irrespective of the type of classifier, the ordinary bootstrap is the most stable model validation technique in terms of threshold-dependent and threshold-independent metrics.

7.7. Discussion

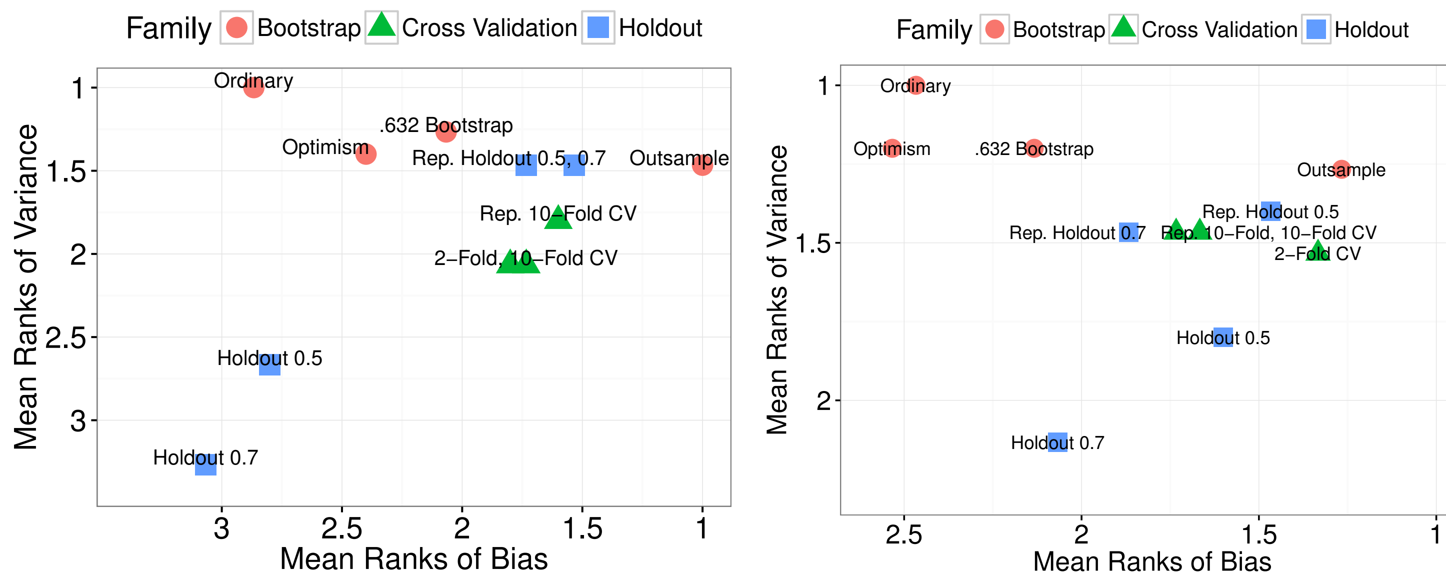
7.7.1. Leave-One-Out Cross-Validation

In Section 7.3, we find that 5% of the analyzed defect prediction literature uses the LOOCV technique. However, since LOOCV only performs one prediction per fold, it is incompatible with the majority of our performance metrics. Thus, we exclude it from our core analysis of our research questions. On the other hand, the bias and variance of the LOOCV technique can be assessed using the Brier score.

Leave-one-out cross-validation is among the least biased and most stable model validation techniques in terms of Brier score. Indeed, Figures 7.5 and 7.6 show that LOOCV appears in the top Scott-Knott ESD ranks in terms of Brier score bias and variance. This finding is very much complementary to recent work on software effort estimation [121], which argues for the use of LOOCV for assessing software effort estimation models.

7.7.2. External Validation

We estimate the model performance using *unseen data*, which may not be realistic, since it is derived from a random sampling. While this random splitting approach is common in other research areas [32, 97, 116, 124, 125], recent studies in software engineering tend to estimate the performance of defect models using data from subsequent releases [104, 192–194]. We perform an additional analysis in order to investigate whether the performance of classifiers that is derived from model validation techniques is similar to the performance of classifiers that are trained using one release and tested on the next one. We then repeat all of our experiments to compute the bias and variance in terms of precision, recall, Brier score, AUC, and calibration slope of naïve bayes, logistic regression, and random forest classifiers.



(a) [Empirical Study 3] Internal Validation (Section 7.6) (b) [Empirical Study 3] External Validation (Section 7.7.2)

Figure 7.7.: [Empirical Study 3] A scatter plot of the mean Scott-Knott ESD ranks in terms of bias and variance among 5 performance metrics, 3 studied classifiers, and 18 studied systems for the high-risk EPV context (EPV= 3) when using two different types of unseen data, i.e., Figure 7.7a uses the observations from the input dataset that do not appear in the sample dataset; and Figure 7.7b uses the next software release. The techniques that appear in the upper-right corner are top-performers.

The out-of-sample bootstrap still yields the best balance between bias and variance of performance estimates. Based on our analysis of 5 releases of Proprietary systems, 2 releases of Apache Xalan, and 3 releases of the Eclipse Platform, Figure 7.7b shows that, similar to Section 7.6, the out-of-sample bootstrap tends to provide a good balance between bias and variance of performance estimates in the high-risk EPV context. We also find a consistent finding in the low-risk EPV context, indicating that internally validated estimates of model performance could accurately be obtained with the out-of-sample bootstrap.

7.7.3. The Computational Cost of Model Validation Techniques

Note that the computational cost (i.e., the number of iterations) of each model validation technique varies (see Table 7.1). For example, the performance that is derived from holdout validation requires a single iteration, while the performance derived from bootstrap validations requires several iterations. Thus, for complex classifiers (e.g., random forest), the advanced model validation techniques (e.g., bootstrap validation and repeated 10-fold cross-validation) may require a large amount of total execution time. However, since each iteration is independent, they can be computed simultaneously. Therefore, researchers can speed up the process using multi-core or multi-thread processors.

In addition to the impact of the computational cost of model validation techniques on the total execution time, it may affect the bias and variance of performance estimates.

The number of iterations impacts the variance of the performance estimates, but not the bias. Figure 7.6 and Figure 7.7b show that the bootstrap family and repeated 10-fold cross-validation, which requires several iterations, tend to yield the most stable performance estimates, indicating that increasing the number of iterations tends to produce more stable performance estimates. This suggests that the repeated 10-fold cross-validation is more preferable than the 10-fold cross-

validation. Conversely, we find that the number of iterations tends to have less of an impact on the bias. For example, all variants of bootstrap and repeated 10-fold cross-validation that have the same amount of computational cost (i.e., 100 iterations), have different amounts of bias on performance estimates. Hence, we suspect that the impact of model validation techniques on the bias of performance estimates may have more to do with their calculation techniques (e.g., type or size of training and testing data) than the number of iterations.

7.8. Practical Guidelines

Our experimental results indicate that the choice of model validation technique can influence model performance estimates, especially for complex classifiers. An inappropriate model validation technique could lead to misleading conclusions. In this section, we offer practical guidelines for future defect prediction studies:

(1) Single-repetition holdout validation should be avoided.
Section 7.3 shows that 45% of the surveyed defect prediction literature uses the holdout model validation technique. However, Section 7.6 shows that the single holdout family is consistently the most biased and least stable model validation technique. Indeed, our results show that the single holdout family tends to produce performance estimates with 46%-229% more bias and 53%-863% more variance than the top-ranked model validation techniques. Hence, researchers should avoid using the holdout family, since it may produce overly optimistic or pessimistic performance estimates and yield results that are difficult to reproduce. Section 7.6 shows that the repetitions of holdout validation technique substantially reduce the amount bias and variance, suggesting that the repetition must be applied for the holdout validation. Nonetheless, the repeated holdout validation techniques still produce performance estimates with more variance than the ordinary bootstrap.

(2) Researchers should use out-of-sample bootstrap validation instead of cross-validation or holdout.

Although other families of model validation techniques are comparable to the out-of-sample bootstrap techniques in the low-risk EPV context, Figure 7.7a shows that, in high-risk EPV contexts, the out-of-sample bootstrap technique is less biased (Figure 7.5) and more stable (Figure 7.6) than the other studied model validation techniques. Moreover, Figure 7.7b also confirms that internally validated estimates of model performance can accurately be obtained with the out-of-sample bootstrap. Furthermore, since Section 7.4 shows that many publicly-available defect datasets suffer from a high-risk EPV, we recommend that researchers use the out-of-sample bootstrap in future defect prediction studies.

7.9. Threats to Validity

Like any empirical study design, experimental design choices may impact the results of our study [229]. However, we perform a highly-controlled experiment to ensure that our results are robust. Below, we discuss threats that may impact the results of our study.

7.9.1. Construct Validity

The datasets that we analyze are part of several collections (e.g., NASA and PROMISE), which each provide different sets of software metrics. Since the metrics vary, this is another point of variation between the studied systems that could impact the results. On the other hand, the variation of metrics also strengthens the generalization of our results—our findings are not bound to one specific set of software metrics.

The conclusions of Section 7.2.2 are based on a rule-of-thumb EPV value that

is suggested by Peduzzi *et al.* [186], who argue that, in order to avoid unstable results, the EPV of a dataset should be at least 10. However, their conclusions are not derived in a defect prediction context. Thus, future research should explore an optimal EPV value for defect prediction context.

The design our experiments of Section 7.5 take the approaches that have been used in the other research areas into consideration [32,97,116,124,125]. Although our approach is built upon these successful previous studies, we made several improvements. For example, Kohavi *et al.* [124] adopt a simple holdout approach to generate sample and unseen corpora. However, our approach is based on the bootstrap concept, which leverages aspects of statistical inference. Moreover, we also maintain the same defective ratio as the original dataset to ensure that the sample and unseen datasets are representative of the dataset from which they were generated. We also repeat the experiment several times to ensure that the results are robust and that they converge.

Randomness may introduce bias. To combat this bias, we repeat the experiment 1,000 times to ensure that the results converge. While we have reported the results using 1,000 repetitions, we also repeated the experiment using 300 repetitions, and found consistent results. Thus, we believe the results have converged, and an increase in the number of repetitions would not alter the conclusions of our study.

In our experiment, parameter settings of classification techniques may impact the performance of defect prediction models [14,47,219,229,232]. However, only 2 out of the 3 studied classification techniques (i.e., naïve bayes and random forest) require at least one parameter setting. Our recent work [232] (see Chapter 6) finds that both naïve bayes and random forest classifiers are relatively insensitive to the choice of parameter settings. Hence, we believe that the choice of parameter settings would not alter the conclusions of our study.

Although the Scott-Knott Effect Size Difference (ESD) test (Section 7.5.8) uses log-transformations [31] to mitigate the skewness of data distribution, we

are aware that other data transformation techniques (i.e., Box-Cox transformation [20, 31], Blom transformation [159]) may yield different results [181]. To ensure that the results are robust, we repeat the experiment 1,000 times, as suggested by Arcuri *et al.* [13]. Therefore, according to the Central Limit Theorem, the distribution of bias and variance results will be approximately normally distributed if the sample size is large enough [165]. We believe that other data transformation techniques would not alter the conclusions of our study.

Finally, all variants of bootstrap validation techniques are repeated 100 times (see Table 7.1). While our results show that the 100 repetitions of the out-of-sample bootstrap validation produce the least biased and most stable performance estimates, we also repeated the experiment using 200 repetitions, and found consistent results. Hence, we believe that 100 repetitions are sufficient.

7.9.2. Internal Validity

While we find that the out-of-sample bootstrap tends to provide the least bias and most stable performance estimates across measures in both EPV contexts, our findings are limited to 5 performance measures—2 threshold-dependent performance measures (i.e., precision and recall) and 3 threshold-independent performance measures (i.e., AUC, Brier score, calibration slope). Other performance measures may yield different results. However, 3 of the studied performance measures (i.e., precision, recall, and AUC) are commonly used measures in defect prediction studies. We also explore another 2 performance measures (i.e., Brier score and calibration slope) that capture other important dimensions of the performance of a classifier (*cf.* Section 7.5.6), which have not yet been explored in the software engineering literature. Nonetheless, other performance measures can be explored in future work. This chapter provides a detailed methodology for others who would like to re-examine our findings using unexplored performance measures.

Prior work has shown that noisy data may influence conclusions that are drawn from defect prediction studies [74, 229, 230]. While Chapter 5 shows that noise generated by issue report mislabelling has little impact on the precision of defect prediction models, they do indeed impact the ability of identifying defective modules (i.e., recall). Hence, noisy data may be influencing our conclusions. However, we conduct a highly-controlled experiment where known-to-be noisy NASA data [209] has been cleaned. Nonetheless, dataset cleanliness should be inspected in future work.

7.9.3. External Validity

We study a limited number of datasets in this chapter. Thus, our results may not generalize to all software systems. To combat potential bias, we analyze 18 datasets from both proprietary and open source domains. Nonetheless, additional replication studies are needed.

We study only one technique for each classification family, i.e., probability-based, regression-based, and machine learning-based families. Thus, our results may not generalize to other classification techniques of each family. Hence, additional evaluation of families of classification techniques are needed.

7.10. Chapter Summary

Defect prediction models help software quality assurance teams to effectively allocate their limited resources to the most defect-prone software modules. Model validation techniques are used to estimate how well a model will perform on unseen data, i.e., data other than that which was used to train the model. However, the validity and the reliability of performance estimates rely heavily on the employed model validation techniques. Yet, little is known about which model validation techniques tend to produce the least biased and most stable perfor-

mance estimates.

To that end, this chapter investigates the bias and the variance of 12 model validation techniques in terms of 2 threshold-dependent performance measures (i.e., precision and recall) and 3 threshold-independent performance measures (i.e., Brier score, AUC, and calibration slope). Since many publicly available defect datasets are at a high risk of producing unstable results (see Section 7.2.2), we explore the bias and variance of model validation techniques in both high-risk (i.e., EPV=3) and low-risk (i.e., EPV=10) contexts. We evaluate 3 types of classifiers that include probability-based (i.e., naïve bayes), regression-based (i.e., logistic regression) and machine learning-based (i.e., random forest) classifiers. Through a case study of 18 datasets spanning both open source and proprietary domains, we make the following observations:

– The out-of-sample bootstrap is the least biased model validation technique in terms of both threshold-dependent and threshold-independent performance measures.

– The ordinary bootstrap is the most stable model validation technique in terms of both threshold-dependent and threshold-independent performance measures.

– In datasets with a high-risk of producing unstable results (i.e., the EPV is low), the out-of-sample bootstrap family yields a good balance between bias and variance of model performance.

– The single-repetition holdout validation consistently yields the most biased and least stable estimates of model performance.

  To mitigate the risk of result instability that is present in many defect datasets, we recommend that future defect prediction studies avoid using the single-repetition holdout validation and instead opt to use the out-of-sample bootstrap model validation technique.

7.10.1. Concluding Remarks

In the past two chapters, we focus on the impact of the two common overlooked experimental components (i.g., noise generated by issue report mislabelling and the choice of parameter settings of classification techniques). In this chapter, we focus on the impact of model validation techniques.

Part IV.

Conclusion and Future Work

CHAPTER 8

Conclusion and Future Work

Key Concept

The experimental components of defect prediction modelling do indeed impact defect prediction models.

The limited Software Quality Assurance (SQA) resources of software organizations must focus on software modules (e.g., source code files) that are likely to be defective in the future. To that end, defect prediction models are trained to identify defect-prone software modules using statistical or machine learning classification techniques. However, the predictions and insights derived from defect prediction models may not be accurate and reliable, if practitioners do not consider the impact of the experimental design choices on defect prediction modelling.

8.1. Contributions and Findings

The goal of this thesis is to better understand the impact of experimental components on defect prediction models. To do so, we set out to investigate the impact of the three often overlooked experimental components (i.e., noise generated by issue report mislabelling, parameter settings of classification techniques, and model validation techniques) of the 3 stages of defect prediction modelling. Broadly speaking, we find that:

> The experimental components of defect prediction modelling impact the predictions and associated insights that are derived from defect prediction models. Empirical investigations on the impact of these overlooked experimental components are needed to derive practical guidelines for defect prediction modelling.

Below, we reiterate the main findings and its implication of the thesis:

1. Research group shares a strong relationship with the dataset and metrics families that are used in building models.

   Implications: Researchers should experiment with a broader selection of datasets and metrics in order to maximize external validity (Chapter 3).

2. The strong association among explanatory variables introduces interference when interpreting the impact of research group on the reported model performance.

   Implications: Researchers should carefully mitigate collinearity issues prior to analysis in order to maximize internal and construct validity (Chapter 3).

3. After mitigating the interference, we find that the research group has a smaller impact than metric family with respect to the Eclipse dataset family.

Implications: Researchers should carefully examine the choice of metrics when building defect prediction models in order not to produce under-performing models (Chapter 3).

4. The precision of our defect prediction models is rarely impacted by the noise generated by issue report mislabelling.

   Implications: Practitioners and researchers can rely on the accuracy of modules labelled as defective by defect prediction models that are trained using such noisy data (Chapter 5).

5. Defect prediction models that are trained on such noisy data typically achieve 56%-68% of the recall of models that are trained on clean data.

   Implications: Researchers should clean mislabelled issue reports in order to improve the ability of defect prediction models to identify defective modules (Chapter 5).

6. The most important variables are generally robust to defect mislabelling. On the other hand, the second- and third-most important variables are more unstable than the most important ones.

   Implications: Practitioners and researchers should only interpret or make decisions based on the most important metrics of defect prediction models when they are trained on noisy data (Chapter 5).

7. Automated parameter optimization substantially improves the performance and stability of defect prediction models, as well as they change their interpretation.

   Implications: Researchers should apply automated parameter optimization in order to improve the performance and reliability of defect prediction models (Chapter 6).

8. The single-repetition holdout validation consistently yields the most biased and least stable estimates of model performance.

Implications: Practitioners and researchers should avoid using the single-repetition holdout validation and instead opt to use the out-of-sample bootstrap model validation technique in order to produce more accurate and reliable performance estimates (Chapter 7).

Furthermore, this thesis is the first to introduce a variety of approaches that future software engineering research can use in their studies:

1. An evaluation approach of the impact of collinearity and multicollinearity on the insights derived from an ANOVA analysis (Chapter 3). A replication package of the approach is provided in Appendix A.

2. An evaluation approach of the impact of mislabelling on the performance and interpretation of defect prediction models (Chapter 5).

3. An evaluation approach of the impact of automated parameter optimization on the performance, stability, and interpretation of defect prediction models (Chapter 6).

4. An introduction of a generic variable importance calculation that can be applied to a variety of classification techniques (Chapter 6). An example `R` implementation of the generic variable importance calculation is provided in Appendix D.

5. An introduction of Caret parameter optimization (Chapter 6). An example `R` usage of the Caret parameter optimization is provided in Appendix C.

6. An introduction of a bootstrap sensitivity analysis that can be used to estimate empirical distributions from an unknown population (e.g., an estimation of the likelihood that a technique will appear in the top-rank). This approach can be used a statistical comparison approach over multiple datasets (Chapter 6).

7. An evaluation approach of the bias and variance of model validation techniques for defect prediction models (Chapter 7).

8. The distribution of Events Per Variable (EPV) of publicly-available defect datasets (Chapter 7). A list of the datasets and their EPV values is provided in Appendix B.

9. An introduction of the Scott-Knott Effect Size Difference (ESD) test—an enhancement of the standard Scott-Knott test (which cluster distributions (e.g., distributions of the importance score of variables) into statistically distinct ranks [203]), which makes no assumptions about the underlying distribution and takes the effect size into consideration (Chapter 7). The `R` implementation of the ScottKnott ESD test is provided in Appendix E and published online as an R package.[1]

10. An introduction of the two new threshold-independent performance measures (i.e., Brier score and calibration slope) that capture other important dimensions of the performance of a classifier (*cf.* Section 7.5.6), which have not yet been explored in the software engineering literature (Chapter 7).

11. An introduction of a large collection of model validation techniques, especially bootstrap validation, which has only rarely been explored in the software engineering literature (Chapter 7). An example `R` implementation of the out-of-sample bootstrap validation is provided in Appendix F.

8.2. Opportunities for Future Research

We believe that our thesis makes a positive contribution towards providing empirical evidence of the impact of experimental components for defect prediction modelling. However, we believe that our work also opens a number of research

[1] `https://github.com/klainfo/ScottKnottESD`

opportunities. In the following, we highlight potential future work to extend our results.

8.2.1. Improving early classification approach of issue report mislabelling

Our approach in Chapter 5 shows that the use of nine characteristics of issue reports can be used to early detect issue report mislabelling. Even if we select the characteristics that cover a variety of dimensions. However, future research should leverage other information and other techniques in order improve the performance of the detection approach. For example, we can use a text mining approach with the title and description of issue reports to classify into categories (e.g., bug, feature improvement).

8.2.2. Understanding the impact of mislabelling on other classification techniques

Our investigation in Chapter 5 focus only on the random forest classification technique. Although prior studies have also used random forest [70, 75, 103, 109, 133], our findings are entirely bound to this technique. Future research should explore the impact of issue report mislabelling on other classification techniques (e.g., logistic regression).

8.2.3. Understanding the benefits of other parameter optimization techniques for defect prediction models

Our investigation in Chapter 6 focuses only on Caret parameter optimization. Caret relies on a grid search technique which exhaustively searches for an optimal parameter setting for a classification technique. Yet, little is known about the benefits that other parameter optimization techniques (e.g., random search) have

on defect prediction models. Thus, future research should explore the benefits of other parameter optimization techniques for defect prediction models.

8.2.4. Revisiting defect prediction studies when automated parameter optimization is applied

Our investigation in Chapter 6 shows that automated parameter optimization impacts the performance, model stability, and interpretation of defect prediction models. However, little research applies automated parameter optimization techniques. Thus, future research should revisit prior studies if automated parameter optimization techniques impacts the performance, model stability, and model interpretation of various settings of defect prediction models, such as, cross-project defect prediction [257], change classification [117], Just-In-Time (JIT) defect prediction [70, 111], transfer learning models [123, 140], and heterogeneous defect prediction [176].

Furthermore, recent research applies search-based techniques to identify the best set of classification techniques for defect prediction models [40]. Search-based techniques also require a set of configurable parameters, e.g., crossover and mutation rate. Arcuri *et al.* [15] point out that such search-based parameters impact the performance of defect prediction models, which is consistent to the findings of Chapter 6. Thus, future research should explore if automated parameter optimization can help to configure parameters of search-based techniques for defect prediction model and improve the performance of defect prediction models.

8.2.5. Investigating a rule-of-thumb EPV value for a defect prediction context

The conclusions of our investigation in Chapter 7 are based on a rule-of-thumb EPV value that is suggested by Peduzzi *et al.* [186], who argue that, in order to

avoid unstable results, the EPV of a dataset should be at least 10. However, their conclusions are not derived in a defect prediction context. Thus, future research should explore an optimal EPV value for defect prediction context.

8.2.6. Understanding the impact of normalization techniques for the Scott-Knott ESD test

The Scott-Knott Effect Size Difference (ESD) test (Section 7.5.8) uses log transformations [31] to mitigate the skewness of data distribution. However, we aware that other data transformation techniques (i.e., Box-Cox transformation [20, 31], Blom transformation [159]) may yield different results [181]. Thus, future research should investigate the impact of normalization techniques for the Scott-Knott ESD test.

8.2.7. Revisiting defect prediction studies when out-of-sample bootstrap is applied

Our investigation in Chapter 7 shows that model validation techniques impacts the accuracy and reliability of the performance estimates that are derived from defect prediction models. However, little research adopts the out-of-sample bootstrap validation, suggesting that the reported performance may be inaccurate and unstable. Thus, future research should revisit prior studies if the out-of-sample bootstrap validation impacts the conclusions of prior studies, such as, personalized defect prediction [100], online defect prediction [227], change classification [117], Just-In-Time (JIT) defect prediction [70, 111], transfer learning models [123, 140], and heterogeneous defect prediction [176].

8.2.8. Revisiting prior literature analyses when the overlooked experimental components are taken into consideration

Our investigation in Chapter 5, Chapter 6, and Chapter 7 show that a variety of experimental components (i.e., noise cleaning techniques, parameter optimization techniques, and model validation techniques) impacts the accuracy and reliability of predictions, insights, and performance estimates that are derived from defect prediction models. However, prior literature analyses [82,207] does not take into a consideration these overlooked experimental components. Thus, future literature analyses should include families of these overlooked experimental components in order to provide a more complete understanding of defect prediction literature.

8.2.9. Inspecting the cleanliness of publicly-available defect prediction datasets

Prior work has shown that noisy data may influence conclusions that are drawn from defect prediction studies [74, 229, 230]. While Chapter 5 shows that noise generated by issue report mislabelling has little impact on the precision of defect prediction models, they do indeed impact the ability of identifying defective modules (i.e., recall). However, the conclusions of several defect prediction studies heavily rely on the use of publicly-available datasets [82, 191, 210]. Thus, dataset cleanliness of the publicly-available defect prediction datasets should be inspected in future work.

8.2.10. Investigating the impact of research toolkits on defect prediction models

The findings of this thesis rely on one research toolkit (i.e., R). Ramirez *et al.* [196] show that different versions of the `randomForest` R package provide different

results. Thus, tools may impact the conclusions of defect prediction studies. Therefore, future research should investigate if research toolkits impact the conclusions of defect prediction models. Also, future research should report the used functions and tools in order to increase validity and reproducibility of research.

References

[1] Mozilla quality assurance. https://quality.mozilla.org/, 2016. (Cited on page 2.)

[2] Product quality analyst jobs. http://www.facebook.jobs/product-quality-analyst/jobs-in/, 2016. (Cited on page 2.)

[3] Quality assurance. https://code.facebook.com/posts/624281754360665/quality-assurance/, 2016. (Cited on page 2.)

[4] Systems, quality, & security engineering jobs. https://www.amazon.jobs/en/job_categories/systems-quality-security-engineering, 2016. (Cited on page 2.)

[5] A. Agresti. *An introduction to categorical data analysis*, volume 135. Wiley New York, 1996. (Cited on page 33.)

[6] H. Akaike. A new look at the statistical model identification. *IEEE Transactions on Automatic Control*, 19(6):716–723, 1974. (Cited on page 38.)

[7] F. Akiyama. An Example of Software System Debugging. In *Proceedings of the International Federation of Information Processing Societies Congress (IFIP'71)*, pages 353–359, 1971. (Cited on pages 2, 17, and 52.)

[8] D. S. Alberts. The economics of software quality assurance. In *Proceedings of the National Computer Conference and Exposition*, pages 433–442. ACM, 1976. (Cited on page 2.)

[9] D. G. Altman, B. Lausen, W. Sauerbrei, and M. Schumacher. Dangers of Using "optimal" Cutpoints in the Evaluation of Prognostic Factors. *Journal of the National Cancer Institute*, 86(1994):829–835, 1994. (Cited on pages 25, 107, and 163.)

[10] D. G. Altman and P. Royston. What do we mean by validating a prognostic model? *Statistics in Medicine*, 19(4):453–473, 2000. (Cited on page 141.)

[11] G. Antoniol, K. Ayari, M. D. Penta, and F. Khomh. Is it a Bug or an Enhancement? A Text-based Approach to Classify Change Requests. In *Proceedings of the IBM Centre for Advanced Studies Conference (CASCON)*, pages 1–15, 2008. (Cited on pages 43, 52, and 56.)

[12] J. Aranda and G. Venolia. The Secret Life of Bugs: Going Past the Errors and Omissions in Software Repositories. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 298–308, 2009. (Cited on pages 43 and 56.)

[13] A. Arcuri and L. Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 1–10, 2011. (Cited on page 186.)

[14] A. Arcuri and G. Fraser. On parameter tuning in search based software engineering. In *Search Based Software Engineering (SBSE)*, pages 33–47. Springer, 2011. (Cited on page 185.)

[15] A. Arcuri and G. Fraser. Parameter tuning or default values? An empirical investigation in search-based software engineering. *Empirical Software Engineering*, 18(3):594–623, 2013. (Cited on pages 132 and 199.)

[16] E. Arisholm, L. C. Briand, and M. Fuglerud. Data mining techniques for building fault-proneness models in telecom java software. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, pages 215–224, 2007. (Cited on pages 23 and 45.)

[17] E. Arisholm, L. C. Briand, and E. B. Johannessen. A Systematic and Comprehensive Investigation of Methods to Build and Evaluate Fault Prediction Models. *Journal of Systems and Software (JSS)*, 83(1):2–17, 2010. (Cited on pages 25, 63, 107, and 163.)

[18] P. C. Austin and E. W. Steyerberg. Events per variable (EPV) and the relative performance of different strategies for estimating the out-of-sample validity of logistic regression models. *Statistical Methods in Medical Research*, 0(0):1–13, 2014. (Cited on page 141.)

[19] A. Avram. Ensuring product quality at google. `https://www.infoq.com/news/2011/03/Ensuring-Product-Quality-Google`, 2011. (Cited on page 2.)

[20] M. Azzeh, A. B. Nassif, and L. L. Minku. An empirical evaluation of ensemble adjustment methods for analogy-based effort estimation. *Journal of Systems and Software (JSS)*, 103:36–52, 2015. (Cited on pages 24, 46, 138, 186, and 200.)

[21] A. Bachmann, C. Bird, F. Rahman, P. Devanbu, and A. Bernstein. The Missing Links: Bugs and Bug-fix Commits. In *Proceedings of the International Symposium on the Foundations of Software Engineering (FSE)*, pages 97–106, 2010. (Cited on pages 43, 44, 52, 55, 59, and 130.)

[22] V. R. Basili, L. C. Briand, and W. L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering (TSE)*, 22(10):751–761, 1996. (Cited on pages 19, 23, and 45.)

[23] C. Beleites, R. Baumgartner, C. Bowman, R. Somorjai, G. Steiner, R. Salzer, and M. G. Sowa. Variance reduction in estimating classification error using sparse datasets. *Chemometrics and Intelligent Laboratory Systems*, 79(1-2):91–100, 2005. (Cited on page 155.)

[24] J. Bergstra and Y. Bengio. Random search for hyper-parameter optimization. *The Journal of Machine Learning Research (JMLR)*, 13(1):281–305, 2012. (Cited on pages 85 and 97.)

[25] N. Bettenburg and A. E. Hassan. Studying the impact of social interactions on software quality. *Empirical Software Engineering*, 18(2):375–431, 2013. (Cited on page 98.)

[26] N. Bettenburg, M. Nagappan, and A. E. Hassan. Think Locally , Act Globally : Improving Defect and Effort Prediction Models. In *Proceedings of the Working Conference on Mining Software Repositories (MSR)*, pages 60–69, 2012. (Cited on pages 23 and 45.)

[27] S. Bibi, G. Tsoumakas, I. Stamelos, and I. Vlahvas. Software Defect Prediction Using Regression via Classification. In *Proceedings of the International Conference on Computer Systems and Applications (ICCSA)*, pages 330–336, 2006. (Cited on page 96.)

[28] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu. Fair and Balanced? Bias in Bug-Fix Datasets. In *Proceedings of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 121–130, 2009. (Cited on pages 43, 44, 52, 55, 59, and 130.)

[29] C. Bird, B. Murphy, and H. Gall. Don't Touch My Code! Examining the Effects of Ownership on Software Quality. In *Proceedings of the joint meeting of the European Software Engineering Conference and the symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 4–14, 2011. (Cited on pages 20, 23, 63, 64, and 150.)

[30] D. Bowes, T. Hall, and J. Petrić. Different classifiers find different defects although with different level of consistency. In *Proceedings of the International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE)*, page 3, 2015. (Cited on page 131.)

[31] G. E. Box and D. R. Cox. An analysis of transformations. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 211–252, 1964. (Cited on pages 185, 186, and 200.)

[32] U. M. Braga-Neto and E. R. Dougherty. Is cross-validation valid for small-sample microarray classification? *Bioinformatics*, 20(3):374–380, 2004. (Cited on pages 103, 150, 155, 161, 180, and 185.)

[33] L. Breiman. Bias, variance and arcing classifiers. *Statistics*, 1996. (Cited on page 93.)

[34] L. Breiman. Random Forests. *Machine Learning*, 45(1):5–32, 2001. (Cited on pages 65, 67, 161, and 162.)

[35] L. Breiman and P. Spector. Submodel Selection and Evaluation in Regression. The X-Random Case. *International Statistical Institute*, 60(3):291–319, 1992. (Cited on pages 151 and 155.)

[36] L. C. Briand, J. Wüst, and H. Lounis. Replicated case studies for investigating quality factors in object-oriented designs. *Empirical Software Engineering*, 6(1):11–58, 2001. (Cited on pages 23 and 45.)

[37] G. W. Brier. Verification of Forecasets Expressed in Terms of Probability. *Monthly Weather Review*, 78(1):25–27, 1950. (Cited on pages 26 and 164.)

[38] F. J. Buckley and R. Poston. Software quality assurance. *IEEE Transactions on Software Engineering (TSE)*, 10(1):36–41, 1984. (Cited on pages 2 and 15.)

[39] B. Caglayan, B. Turhan, A. Bener, M. Habayeb, A. Miransky, and E. Cialini. Merits of organizational metrics in defect prediction: an industrial replication. In *Proceedings of the International Conference on Software Engineering (ICSE)*, volume 2, pages 89–98, 2015. (Cited on page 3.)

[40] G. Canfora, A. D. Lucia, M. D. Penta, R. Oliveto, A. Panichella, and S. Panichella. Defect prediction as a multiobjective optimization problem. *Software Testing, Verification and Reliability*, 25(4):426–459, 2015. (Cited on page 199.)

[41] A. Canty and B. Ripley. boot: Bootstrap r (s-plus) functions. `http://CRAN.R-project.org/package=boot`, 2014. (Cited on pages 65 and 161.)

[42] M. Cataldo, A. Mockus, J. Roberts, and J. Herbsleb. Software Dependencies, Work Dependencies, and Their Impact on Failures. *IEEE Transactions on Software Engineering (TSE)*, 35(6):864–878, 2009. (Cited on pages 2, 52, and 57.)

[43] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering (TSE)*, 20(6):476–493, 1994. (Cited on page 17.)

[44] J. Cohen. *Statistical Power Analysis for the Behavioral Sciences.* 1988. (Cited on pages 38, 110, 111, and 165.)

[45] J. Cohen. A power primer. *Psychological bulletin*, 112(1):155, 1992. (Cited on pages 111 and 165.)

[46] J. Concato, A. R. Feinstein, and T. R. Holford. The Risk of Determining Risk with Multivariable Models. *Annals of Internal Medicine*, 118(3):201–210, 1993. (Cited on page 141.)

[47] A. Corazza, S. Di Martino, F. Ferrucci, C. Gravino, F. Sarro, and E. Mendes. How effective is tabu search to configure support vector regression for effort estimation? In *Proceedings of the International Conference on Predictive Models in Software Engineering (PROMISE)*, page 4, 2010. (Cited on page 185.)

[48] D. R. Cox. Two Further Applications of a Model for Binary Regression. *Biometrika*, 45(3):562–565, 1958. (Cited on pages 26, 143, 144, 161, 162, and 164.)

[49] H. Cramér. *Mathematical methods of statistics*, volume 9. Princeton university press, 1999. (Cited on page 33.)

[50] M. D'Ambros, M. Lanza, and R. Robbes. An Extensive Comparison of Bug Prediction Approaches. In *Proceedings of the Working Conference on Mining Software Repositories (MSR)*, pages 31–41, 2010. (Cited on pages 2, 52, 100, 101, 142, 158, and 251.)

[51] M. D'Ambros, M. Lanza, and R. Robbes. Evaluating Defect Prediction Approaches: A Benchmark and an Extensive Comparison. *Empirical Software Engineering*, 17(4-5):531–577, 2012. (Cited on pages 10, 17, 23, 24, 46, 100, 101, 138, 142, 158, and 251.)

[52] J. Davis and M. Goadrich. The relationship between precision-recall and roc curves. In *Proceedings of the International Conference on Machine Learning (ICML'06)*, pages 233–240, 2006. (Cited on pages 25, 107, and 163.)

[53] A. C. Davison and P. Hall. On the bias and variability of bootstrap and cross-validation estimates of error rate in discrimination problems. *Biometrika*, 79(2):279–284, 1992. (Cited on page 155.)

[54] K. Dejaeger, W. Verbeke, D. Martens, and B. Baesens. Data Mining Techniques for Software Effort Estimation: A Comparative Study. *IEEE Transactions on Software Engineering (TSE)*, 38(2):375–397, 2012. (Cited on pages 24, 46, and 138.)

[55] S. den Boer, N. F. de Keizer, and E. de Jonge. Performance of prognostic models in critically ill cancer patients - a review. *Critical care*, 9(4):R458–R463, 2005. (Cited on pages 26, 107, and 164.)

[56] P. Domingos and M. Pazzani. On the optimality of the simple bayesian classifier under zero-one loss. *Machine learning*, 29(2-3):103–130, 1997. (Cited on page 161.)

[57] E. Dougherty, C. Sima, J. Hua, B. Hanczar, and U. Braga-Neto. Performance of Error Estimators for Classification. *Bioinformatics*, 5(1):53–67, 2010. (Cited on page 155.)

[58] B. Efron. Estimating the Error Rate of a Prediction Rule: Improvement on Cross-Validation. *Journal of the American Statistical Association*, 78(382):316–331, 1983. (Cited on pages 27, 102, 103, 126, and 153.)

[59] B. Efron. How Biased Is the Apparent Error Rate of a Prediction Rule ? *Journal of the American Statistical Association*, 81(394):461–470, 1986. (Cited on page 152.)

[60] B. Efron and G. Gong. A Leisurely Look at the Bootstrap , the Jackknife , and Cross-Validation. *The American Statistician*, 37(1):36–48, 1983. (Cited on pages 150 and 152.)

[61] B. Efron and R. Tibshirani. Improvements on Cross-Validation: The 632+ Bootstrap Method. *Journal of the American Statistical Association*, 92(438):548–560, 1997. (Cited on page 155.)

[62] B. Efron and R. J. Tibshirani. *An Introduction to the Bootstrap*. Springer US, Boston, MA, 1993. (Cited on pages 27, 34, 102, 126, 150, and 152.)

[63] R. Efron, B. and Tibshirani. *An Introduction to the Bootstrap*. Chapman & Hall, 1993. (Cited on pages 64 and 65.)

[64] M. Fernández-Delgado, E. Cernadas, S. Barro, and D. Amorim. Do we Need Hundreds of Classifiers to Solve Real World Classification Problems? *Journal of Machine Learning Research (JMLR)*, 15(1):3133–3181, 2014. (Cited on page 129.)

[65] R. A. Fisher. The use of multiple measurements in taxonomic problems. *Annals of Eugenics*, 7(2):179–188, 1936. (Cited on page 91.)

[66] J. H. Friedman. Multivariate adaptive regression splines. *The Annals of Statistics*, pages 1–67, 1991. (Cited on page 89.)

[67] M. Friendly. heplots: Visualizing hypothesis tests in multivariate linear models. `http://CRAN.R-project.org/package=heplots`, 2015. (Cited on page 34.)

[68] S. Fritsch and F. Guenther. neuralnet: Training of neural networks. `http://CRAN.R-project.org/package=neuralnet`, 2015. (Cited on page 97.)

[69] W. Fu, T. Menzies, and X. Shen. Tuning for software analytics: Is it really necessary? *Information and Software Technology*, 76:135–146, 2016. (Cited on page 131.)

[70] T. Fukushima, Y. Kamei, S. McIntosh, K. Yamashita, and N. Ubayashi. An Empirical Study of Just-in-Time Defect Prediction using Cross-Project

Models. In *Proceedings of the Working Conference on Mining Software Repositories (MSR)*, pages 172–181, 2014. (Cited on pages 65, 80, 198, 199, and 200.)

[71] J. Gao, W. Fan, J. Jiang, and J. Han. Knowledge transfer via multiple model local structure mapping. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 283–291, 2008. (Cited on page 99.)

[72] K. Gao and T. M. Khoshgoftaar. A comprehensive empirical study of count models for software fault prediction. *IEEE Transactions on Reliability*, 56(2):223–236, 2007. (Cited on pages 47, 138, and 154.)

[73] S. Geisser. Sample Reuse Method The Predictive with Applications. *Journal of the American Statistical Association*, 70(350):320–328, 1975. (Cited on pages 27 and 150.)

[74] B. Ghotra, S. McIntosh, and A. E. Hassan. Revisiting the impact of classification techniques on the performance of defect prediction models. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 789–800, 2015. (Cited on pages 24, 45, 46, 97, 99, 101, 110, 125, 131, 134, 138, 166, 187, and 201.)

[75] G. Gousios, M. Pinzger, and A. V. Deursen. An Exploratory Study of the Pull-Based Software Development Model. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 345–355, 2014. (Cited on pages 65, 80, and 198.)

[76] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering (TSE)*, 26(7):653–661, 2000. (Cited on pages 19 and 20.)

[77] R. Grewal, J. a. Cote, and H. Baumgartner. Multicollinearity and Measurement Error in Structural Equation Models: Implications for Theory Testing. *Marketing Science*, 23(4):519–529, 2004. (Cited on page 34.)

[78] A. Günes Koru and H. Liu. An investigation of the effect of module size on defect prediction using static measures. In *Proceedings of the International Workshop on Predictor Models in Software Engineering (PROMISE)*, pages 1–5, 2005. (Cited on pages 84 and 96.)

[79] L. Guo, Y. Ma, B. Cukic, and H. Singh. Robust prediction of fault-proneness by random forests. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, pages 417–428, 2004. (Cited on page 143.)

[80] T. Gyimothy, R. Ferenc, and I. Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering (TSE)*, 31(10):897–910, 2005. (Cited on page 19.)

[81] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The WEKA data mining software: an update. *SIGKDD explorations newsletter*, 11(1):10–18, 2009. (Cited on pages 24, 96, 97, and 136.)

[82] 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 (TSE)*, 38(6):1276–1304, nov 2012. (Cited on pages 3, 9, 23, 42, 46, 52, 84, 114, 149, and 201.)

[83] M. H. Halstead. *Elements of Software Science (Operating and Programming Systems Series)*. Elsevier Science Inc., 1977. (Cited on page 17.)

[84] J. a. Hanley and B. J. McNeil. The meaning and use of the area under a receiver operating characteristic (ROC) curve. *Radiology*, 143(4):29–36, 1982. (Cited on pages 25, 143, and 163.)

[85] M. Harman. The current state and future of search based software engineering. In *2007 Future of Software Engineering*, pages 342–357. IEEE Computer Society, 2007. (Cited on pages 97 and 132.)

[86] M. Harman, P. McMinn, J. De Souza, and S. Yoo. Search based software engineering: Techniques, taxonomy, tutorial. pages 1–59, 2012. (Cited on pages 85, 97, and 132.)

[87] F. E. Harrell Jr. *Regression Modeling Strategies*. Springer, 1st edition, 2002. (Cited on pages 26, 27, 37, 65, 103, 107, 150, 152, and 164.)

[88] F. E. Harrell Jr. rms: Regression modeling strategies. `http://CRAN.R-project.org/package=rms`, 2015. (Cited on pages 37, 107, and 164.)

[89] F. E. Harrell Jr., K. L. Lee, and D. B. Mark. Tutorial in Biostatistics Multivariable Prognostic Models : Issues in Developing Models, Evaluating Assumptions and Adequacy, and Measuring and Reducing Errors. *Statistics in Medicine*, 15:361–387, 1996. (Cited on pages 26 and 164.)

[90] A. E. Hassan. Predicting Faults Using the Complexity of Code Changes. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 78–88, 2009. (Cited on pages 2, 19, 52, and 62.)

[91] A. E. Hassan and R. C. Holt. The Top Ten List: Dynamic Fault Prediction. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 263–272, 2005. (Cited on pages 19 and 79.)

[92] T. Hastie, R. Tibshirani, and J. Friedman. *The elements of statistical learning*. Springer, 2009. (Cited on pages 27 and 152.)

[93] H. He and E. A. Garcia. Learning from Imbalanced Data. *Transactions on Knowledge and Data Engineering (TKDE)*, 21(9):1263–1284, 2009. (Cited on pages 25, 107, and 163.)

[94] K. Herzig, S. Just, and A. Zeller. It's Not a Bug, It's a Feature: How Misclassification Impacts Bug Prediction. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 392–401, 2013. (Cited on pages 9, 43, 44, 52, 53, 56, 78, and 131.)

[95] K. Herzig and A. Zeller. The Impact of Tangled Code Changes. In *Proceedings of the Working Conference on Mining Software Repositories (MSR)*, pages 121–130, 2013. (Cited on page 59.)

[96] J. Huang and C. X. Ling. Using AUC and accuracy in evaluating learning algorithms. *Transactions on Knowledge and Data Engineering*, 17(3):299–310, 2005. (Cited on page 107.)

[97] A. Isaksson, M. Wallman, H. Göransson, and M. Gustafsson. Cross-validation and bootstrapping are unreliable in small sample classification. *Pattern Recognition Letters*, 29(14):1960–1965, 2008. (Cited on pages 150, 155, 161, 180, and 185.)

[98] E. G. Jelihovschi, J. C. Faria, and I. B. Allaman. *The ScottKnott Clustering Algorithm*. Universidade Estadual de Santa Cruz - UESC, Ilheus, Bahia, Brasil, 2014. (Cited on pages 68, 110, and 166.)

[99] Y. Jia, M. Cohen, and M. Petke. Learning Combinatorial Interaction Test Generation Strategies using Hyperheuristic Search. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 540–550, 2015. (Cited on pages 97 and 132.)

[100] T. Jiang, L. Tan, and S. Kim. Personalized defect prediction. pages 279–289. Ieee, nov 2013. (Cited on pages 23, 45, 99, and 200.)

[101] Y. Jiang, B. Cukic, and Y. Ma. Techniques for evaluating fault prediction models. *Empirical Software Engineering*, 13(5):561–595, 2008. (Cited on pages 96 and 143.)

[102] Y. Jiang, B. Cukic, and T. Menzies. Fault Prediction using Early Lifecycle Data. In *Proceedings of the International Symposium on Software Reliability (ISSRE)*, pages 237–246, 2007. (Cited on page 143.)

[103] Y. Jiang, B. Cukic, and T. Menzies. Can Data Transformation Help in the Detection of Fault-prone Modules? In *Proceedings of the Workshop on Defects in Large Software Systems (DEFECTS)*, pages 16–20, 2008. (Cited on pages 65, 80, 84, 96, 106, 114, 162, 165, and 198.)

[104] Y. Jiang, B. Cukic, T. Menzies, and J. Lin. Incremental development of fault prediction models. *International Journal of Software Engineering and Knowledge Engineering (SEKE)*, 23(10):1399–1425, 2013. (Cited on page 180.)

[105] Y. Jiang, J. Lin, B. Cukic, and T. Menzies. Variance Analysis in Software Fault Prediction Models. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, pages 99–108, 2009. (Cited on pages 23, 45, 141, 143, 150, and 155.)

[106] J. Jiarpakdee, C. Tantithamthavorn, A. Ihara, and K. Matsumoto. A study of redundant metrics in defect prediction datasets. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, page To Appear, 2016. (Cited on page 106.)

[107] M. Jorgensen and M. Shepperd. A Systematic Review of Software Development Cost Estimation Studies. *IEEE Transactions on Software Engineering (TSE)*, 33(1):33–53, 2007. (Cited on page 97.)

[108] M. Jureczko and L. Madeyski. Towards identifying software project clusters with regard to defect prediction. In *Proceedings of the International Conference on Predictive Models in Software Engineering (PROMISE)*, pages 9:1–9:10, 2010. (Cited on pages 100, 158, 248, 249, and 250.)

[109] Y. Kamei, S. Matsumoto, A. Monden, K.-i. Matsumoto, B. Adams, and A. E. Hassan. Revisiting Common Bug Prediction Findings Using Effort-Aware Models. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 1–10, 2010. (Cited on pages 42, 63, 65, 80, and 198.)

[110] Y. Kamei, A. Monden, S. Morisaki, and K.-i. Matsumoto. A hybrid faulty module prediction using association rule mining and logistic regression analysis. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM)*, page 279, 2008. (Cited on pages 23 and 45.)

[111] Y. Kamei, E. Shihab, B. Adams, a. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi. A Large-Scale Empirical Study of Just-in-Time Quality Assurance. *IEEE Transactions on Software Engineering (TSE)*, 39(6):757–773, 2013. (Cited on pages 17, 62, 199, and 200.)

[112] P. Kampstra. Beanplot: A boxplot alternative for visual comparison of distributions. *Journal of Statistical Software*, 28(1):1–9, 2008. (Cited on pages 71 and 142.)

[113] A. E. Kazdin. The meanings and measurement of clinical significance. 1999. (Cited on page 106.)

[114] J. Keung, E. Kocaguneli, and T. Menzies. Finding conclusion stability for selecting the best effort predictor in software effort estimation. *Automated Software Engineering (ASE)*, 20(4):543–567, 2013. (Cited on page 126.)

[115] T. Khoshgoftaar and N. Seliya. Comparative Assessment of Software Quality Classification Techniques : An Empirical Case Study. *Empirical Software Engineering*, 9:229–257, 2004. (Cited on pages 24, 46, 99, and 138.)

[116] J.-H. Kim. Estimating classification error rate: Repeated cross-validation, repeated hold-out and bootstrap. *Computational Statistics & Data Analysis*, 53(11):3735–3745, 2009. (Cited on pages 155, 161, 180, and 185.)

[117] S. Kim, E. J. W. Jr, and Y. Zhang. Classifying Software Changes: Clean or Buggy? *IEEE Transactions on Software Engineering (TSE)*, 34(2):181–196, 2008. (Cited on pages 199 and 200.)

[118] S. Kim, H. Zhang, R. Wu, and L. Gong. Dealing with Noise in Defect Prediction. In *Proceeding of the International Conference on Software Engineering (ICSE)*, pages 481–490, 2011. (Cited on pages 9, 42, 44, 52, 57, 80, 81, 100, 101, 130, 142, 158, 250, and 251.)

[119] S. Kim, T. Zimmermann, E. J. Whitehead Jr., and A. Zeller. Predicting Faults from Cached History. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 489–498, 2007. (Cited on pages 2 and 52.)

[120] B. A. Kitchenham. Software quality assurance. *Microprocessors and microsystems*, 13(6):373–381, 1989. (Cited on pages 2 and 15.)

[121] E. Kocaguneli and T. Menzies. Software effort models should be assessed via leave-one-out validation. *Journal of Systems and Software (JSS)*, 86(7):1879–1890, 2013. (Cited on pages 151, 155, and 180.)

[122] E. Kocaguneli, T. Menzies, A. B. Bener, and J. W. Keung. Exploiting the essential assumptions of analogy-based effort estimation. *IEEE Transactions on Software Engineering (TSE)*, 38(2):425–438, 2012. (Cited on pages 85 and 97.)

[123] E. Kocaguneli, T. Menzies, and E. Mendes. Transfer learning in effort estimation. *Empirical Software Engineering*, mar 2014. (Cited on pages 199 and 200.)

[124] R. Kohavi. A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI'95)*, pages 1137–1143, 1995. (Cited on pages 155, 161, 173, 180, and 185.)

[125] R. Kohavi and D. H. Wolpert. Bias plus variance decomposition for zero-one loss functions. In *Proceedings of the International Conference on Machine Learning (ICML'96)*, 1996. (Cited on pages 161, 180, and 185.)

[126] O. Kononenko, O. Baysal, L. Guerrouj, Y. Cao, and M. W. Godfrey. Investigating code review quality: Do people and participation matter? In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, pages 111–120. IEEE, 2015. (Cited on page 98.)

[127] A. G. Koru and J. Tian. Defect handling in medium and large open source projects. *IEEE Software*, 21(4):54–61, 2004. (Cited on page 44.)

[128] S. Kotsiantis. Supervised Machine Learning: A Review of Classification Techniques. *Informatica*, 31:249–268, 2007. (Cited on pages 88, 91, and 92.)

[129] M. Kuhn. Building Predictive Models in R Using caret Package. *Journal of Statistical Software*, 28(5), 2008. (Cited on pages 105, 161, and 163.)

[130] M. Kuhn. C50: C5.0 decision trees and rule-based models. `http://CRAN.R-project.org/package=C50`, 2015. (Cited on page 114.)

[131] M. Kuhn. caret: Classification and regression training. `http://CRAN.R-project.org/package=caret`, 2015. (Cited on pages 85, 103, 105, 133, 135, 136, 161, and 163.)

[132] I. H. Laradji, M. Alshayeb, and L. Ghouti. Software defect prediction using ensemble learning on selected features. *Information and Software Technology (IST)*, 58:388–402, 2015. (Cited on page 96.)

[133] S. Lessmann, S. Member, 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. (Cited on pages 10, 24, 25, 45, 46, 65, 80, 96, 107, 110, 125, 138, 143, 163, and 198.)

[134] C. Lewis, Z. Lin, C. Sadowski, X. Zhu, R. Ou, and E. J. W. Jr. Does Bug Prediction Support Human Developers ? Findings From a Google Case Study. In *Proceedings of the International Conference on Software Engineering (ICSE)*. (Cited on page 3.)

[135] A. Liaw and M. Wiener. randomforest: Breiman and cutler's random forests for classification and regression. `http://CRAN.R-project.org/package=randomForest`, 2015. (Cited on pages 97 and 162.)

[136] G. A. Liebchen and M. Shepperd. Data sets and data quality in software engineering. In *Proceedings of the International Workshop on Predictor Models in Software Engineering (PROMISE)*, pages 39–44. ACM, 2008. (Cited on page 43.)

[137] A. Lim, L. Breiman, and A. Cutler. *bigrf: Big Random Forests: Classification and Regression Forests for Large Data Sets*, 2014. (Cited on pages 65, 67, and 97.)

[138] S. Lohar, A. Zisman, M. Keynes, and J. Cleland-huang. Improving Trace Accuracy through Data-Driven Configuration and Composition of Tracing Features. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, pages 378–388, 2013. (Cited on page 132.)

[139] Y. Ma and B. Cukic. Adequate and Precise Evaluation of Quality Models in Software Engineering Studies. In *Proceedings of the International Workshop on Predictor Models in Software Engineering (PROMISE)*, pages 1–9, 2007. (Cited on pages 25, 107, and 163.)

[140] Y. Ma, G. Luo, X. Zeng, and A. Chen. Transfer learning for cross-company software defect prediction. *Information and Software Technology (IST)*, 54(3):248–256, 2012. (Cited on pages 10, 24, 46, 99, 138, 199, and 200.)

[141] C. Mair and M. Shepperd. The consistency of empirical comparisons of regression and analogy-based software project cost prediction. In *Proceedings of the International Symposium on Empirical Software Engineering (ESEM)*, page 10, 2005. (Cited on page 97.)

[142] M. V. Mantyla, F. Khomh, B. Adams, E. Engstrom, and K. Petersen. On rapid releases and software testing. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 20–29, 2013. (Cited on page 2.)

[143] MATLAB. *version 8.5.0 (R2015a)*. The MathWorks Inc., Natick, Massachusetts, 2015. (Cited on pages 96 and 97.)

[144] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering (TSE)*, (4):308–320, 1976. (Cited on page 17.)

[145] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan. The Impact of Code Review Coverage and Code Review Participation on Software Quality. In *Proceedings of the Working Conference on Mining Software Repositories (MSR)*, pages 192–201, 2014. (Cited on pages 2, 23, 52, 57, and 98.)

[146] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan. An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering*, 2015. (Cited on page 98.)

[147] T. Mende. Replication of Defect Prediction Studies: Problems, Pitfalls and Recommendations. In *Proceedings of the International Conference on Predictive Models in Software Engineering (PROMISE)*, pages 1–10, 2010. (Cited on pages 25, 46, 84, 96, 101, 103, 107, 141, 155, and 163.)

[148] T. Mende and R. Koschke. Revisiting the evaluation of defect prediction models. In *Proceedings of the International Conference on Predictive Models in Software Engineering (PROMISE)*, page 7, 2009. (Cited on pages 46, 84, and 96.)

[149] T. Mende and R. Koschke. Effort-Aware Defect Prediction Models. *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*, pages 107–116, mar 2010. (Cited on page 143.)

[150] T. Mende, R. Koschke, and M. Leszak. Evaluating Defect Prediction Models for a Large Evolving Software System. In *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*, pages 247–250, 2009. (Cited on page 96.)

[151] A. Meneely, L. Williams, W. Snipes, and J. Osborne. Predicting failures with developer networks and social network analysis. In *Proceedings of the International Symposium on the Foundations of Software Engineering (FSE)*, page 13, New York, New York, USA, 2008. (Cited on page 23.)

[152] 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. (Cited on pages 10, 47, 99, 138, 142, 154, 162, and 165.)

[153] T. Menzies, C. Pape, R. Krishna, and M. Rees-Jones. The Promise Repository of Empirical Software Engineering Data. `http://openscience.us/repo`, 2015. (Cited on page 142.)

[154] T. Menzies and M. Shepperd. Special issue on repeatable results in software engineering prediction. *Empirical Software Engineering*, 17(1-2):1–17, 2012. (Cited on pages 6, 42, 47, 84, 97, 138, 154, and 155.)

[155] D. Meyer, E. Dimitriadou, K. Hornik, A. Weingessel, F. Leisch, C.-C. Chang, and C.-C. Lin. e1071: Misc Functions of the Department of Statistics (e1071), TU Wien. `http://CRAN.R-project.org/package=e1071`, 2014. (Cited on page 161.)

[156] D. Meyer, A. Zeileis, K. Hornik, F. Gerber, and M. Friendly. vcd: Visualizing categorical data. `http://CRAN.R-project.org/package=vcd`, 2015. (Cited on page 33.)

[157] D. Michie, D. J. Spiegelhalter, and C. C. Taylor. Machine learning, neural and statistical classification. 1994. (Cited on page 23.)

[158] M. E. Miller, S. L. Hui, and W. M. Tierney. Validation techniques for logistic regression models. *Statistics in medicine*, 10(8):1213–1226, 1991. (Cited on pages 26 and 164.)

[159] N. Mittas and L. Angelis. Ranking and Clustering Software Cost Estimation Models through a Multiple Comparisons Algorithm. *IEEE Transactions on Software Engineering (TSE)*, 39(4):537–551, 2013. (Cited on pages 10, 24, 38, 42, 46, 47, 84, 97, 110, 138, 154, 186, and 200.)

[160] A. Mockus. Missing Data in Software Engineering. In *Guide to Advanced Empirical Software Engineering*, pages 185–200. 2008. (Cited on pages 43 and 52.)

[161] A. Mockus. Organizational Volatility and its Effects on Software Defects. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, pages 117–127, 2010. (Cited on pages 2, 52, and 57.)

[162] A. Mockus and J. D. Herbsleb. Expertise browser: a quantitative approach to identifying expertise. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 503–512. ACM, 2002. (Cited on page 20.)

[163] A. Mockus and D. M. Weiss. Predicting Risk of Software Changes. *Bell Labs Technical Journal*, 5(6):169–180, 2000. (Cited on pages 2, 3, 23, 25, 52, 63, and 163.)

[164] A. M. Molinaro, R. Simon, and R. M. Pfeiffer. Prediction error estimation: a comparison of resampling methods. *Bioinformatics*, 21(15):3301–3307, Aug. 2005. (Cited on pages 150 and 155.)

[165] D. S. Moore. *The basic practice of statistics*, volume 2. WH Freeman New York, 2007. (Cited on page 186.)

[166] R. Morales, S. McIntosh, and F. Khomh. Do code review practices impact design quality? a case study of the qt, vtk, and itk projects. In *Proceedings of the International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 171–180, 2015. (Cited on page 98.)

[167] R. Moser, W. Pedrycz, and G. Succi. A Comparative Analysis of the Efficiency of Change Metrics and Static Code Attributes for Defect Prediction. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 181–190, 2008. (Cited on pages 2, 52, and 63.)

[168] I. Myrtveit and E. Stensrud. Validity and reliability of evaluation procedures in comparative studies of effort prediction models. *Empirical Software Engineering*, 17(1-2):23–33, 2011. (Cited on page 155.)

[169] I. Myrtveit, E. Stensrud, and M. Shepperd. Reliability and Validity in Comparative Studies of Software Prediction Models. *IEEE Transactions*

on Software Engineering (TSE), 31(5):380–391, 2005. (Cited on pages 10, 24, 46, 47, 97, 138, 139, 154, and 155.)

[170] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. *Proceedings International Conference on Software Engineering (ICSE)*, 2005. (Cited on pages 19, 25, and 163.)

[171] N. Nagappan and T. Ball. Using software dependencies and churn metrics to predict field failures: An empirical case study. In *First International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 364–373, 2007. (Cited on pages 3 and 19.)

[172] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 452–461, 2006. (Cited on page 3.)

[173] N. Nagappan, B. Murphy, and V. Basili. The influence of organizational structure on software quality: an empirical case study. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 521–530. ACM, 2008. (Cited on page 20.)

[174] N. Nagappan, B. Murphy, and V. R. Basili. The Influence of Organizational Structure on Software Quality: An Empirical Case Study. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 521–530, 2008. (Cited on pages 63 and 150.)

[175] N. Nagappan, A. Zeller, T. Zimmermann, K. Herzig, and B. Murphy. Change Bursts as Defect Predictors. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, pages 309–318, 2010. (Cited on pages 2, 3, and 52.)

[176] J. Nam and S. Kim. Heterogeneous Defect Prediction. In *Proceedings of the European Software Engineering Conference and the Foundations of*

Software Engineering (ESEC/FSE), pages 508–519, 2015. (Cited on pages 199 and 200.)

[177] P. Nemenyi. *Distribution-free multiple comparisons*. PhD thesis, Princeton University, 1963. (Cited on page 110.)

[178] A. T. Nguyen, T. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. Multi-layered Approach for Recovering Links between Bug Reports and Fixes. In *Proceedings of the International Symposium on the Foundations of Software Engineering (FSE)*, pages 63:1–63:11, 2012. (Cited on pages 43, 44, 52, and 55.)

[179] T. H. Nguyen, B. Adams, and A. E. Hassan. A Case Study of Bias in Bug-Fix Datasets. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, pages 259–268, 2010. (Cited on pages 55, 56, and 130.)

[180] T. H. D. Nguyen, B. Adams, and A. E. Hassan. Studying the impact of dependency network measures on software quality. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, 2010. (Cited on page 23.)

[181] J. W. Osborne. Improving your data transformations: Applying the box-cox transformation. *Practical Assessment, Research & Evaluation*, 15(12):1–9, 2010. (Cited on pages 186 and 200.)

[182] T. Ostrand, E. Weyuker, and R. Bell. Predicting the location and number of faults in large software systems. *IEEE Transactions on Software Engineering (TSE)*, 31(4):340–355, apr 2005. (Cited on page 3.)

[183] A. Panichella, B. Dit, R. Oliveto, M. D. Penta, D. Poshyvanyk, and A. D. Lucia. How to Effectively Use Topic Models for Software Engineering Tasks

? An Approach Based on Genetic Algorithms. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 522–531, 2013. (Cited on page 132.)

[184] A. Panichella, R. Oliveto, and A. De Lucia. Cross-project defect prediction models: L'union fait la force. In *Proceedings of the International Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR/WCRE)*, pages 164–173, 2014. (Cited on pages 45 and 131.)

[185] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al. Scikit-learn: Machine learning in python. *The Journal of Machine Learning Research (JMLR)*, 12:2825–2830, 2011. (Cited on pages 24, 96, and 136.)

[186] P. Peduzzi, J. Concato, E. Kemper, T. R. Holford, and A. R. Feinstein. A Simulation Study of the Number of Events per Variable in Logistic Regression Analysis. *Journal of Clinical Epidemiology*, 49(12):1373–1379, 1996. (Cited on pages xvi, 102, 141, 142, 143, 185, and 199.)

[187] F. Peters, T. Menzies, L. Gong, and H. Zhang. Balancing Privacy and Utility in Cross-Company Defect Prediction. *IEEE Transactions on Software Engineering (TSE)*, 39(8):1054–1068, 2013. (Cited on pages 24, 46, and 138.)

[188] K. L. Priddy and P. E. Keller. *Artificial Neural Networks: An Introduction*, volume 68. SPIE Press, 2005. (Cited on page 152.)

[189] B. R. Clifford. Tests of Hypotheses for Unbalanced Factorial Designs Under Various Regression/Coding Method Combinations. *Educational and Psychological Measurement*, (38):621–631, 1978. (Cited on page 34.)

[190] R Core Team. R: A language and environment for statistical computing. http://www.R-project.org/, 2013. (Cited on pages 24, 34, 96, 143, and 162.)

[191] D. Radjenovic, M. Hericko, R. Torkar, and A. Zivkovic. Software Fault Prediction Metrics: A Systematic Literature Review. *Information and Software Technology (IST)*, 55(8):1397–1418, 2013. (Cited on pages 52 and 201.)

[192] F. Rahman and P. Devanbu. How, and Why, Process Metrics Are Better. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 432–441, 2013. (Cited on pages 25, 42, 56, 59, 63, 64, 79, 80, 107, 142, 163, and 180.)

[193] F. Rahman, I. Herraiz, D. Posnett, and P. Devanbu. Sample Size vs. Bias in Defect Prediction. In *Proceedings of the joint meeting of the European Software Engineering Conference and the symposium on the Foundations of Software Engineering (FSE)*, pages 147–157, 2013. (Cited on pages 44, 52, 130, and 180.)

[194] F. Rahman, D. Posnett, and P. Devanbu. Recalling the "Imprecision" of Cross-Project Defect Prediction. In *Proceedings of the International Symposium on the Foundations of Software Engineering (FSE)*, pages 61:1–61:11, 2012. (Cited on page 180.)

[195] F. Rahman, D. Posnett, A. Hindle, E. Barr, and P. Devanbu. BugCache for Inspections: Hit or Miss? In *Proceedings of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 322–331, 2011. (Cited on pages 2 and 52.)

[196] C. Ramirez, M. Nagappan, and M. Mirakhorli. Studying the impact of evolution in r libraries on software engineering research. In *1st International*

Workshop on Software Analytics (SWAN), pages 29–30, 2015. (Cited on page 201.)

[197] L. M. Rea and R. A. Parker. *Designing and conducting survey research: A comprehensive guide*. John Wiley & Sons, 2014. (Cited on page 33.)

[198] T. C. Redman. The impact of poor data quality on the typical enterprise. *Commun. ACM*, 41(2):79–82, Feb. 1998. (Cited on page 43.)

[199] J. T. E. Richardson. Eta squared and partial eta squared as measures of effect size in educational research. *Educational Research Review*, 6(2):135–147, 2011. (Cited on pages 34 and 38.)

[200] B. Ripley. nnet: Feed-forward neural networks and multinomial log-linear models. `http://CRAN.R-project.org/package=nnet`, 2015. (Cited on page 97.)

[201] K. Rufibach. Use of Brier score to assess binary predictions. *Journal of Clinical Epidemiology*, 63(8):938–939, 2010. (Cited on pages 26 and 164.)

[202] W. S. Sarle. The varclus procedure. In *SAS/STAT User's Guide. SAS Institute, Inc, 4th edition*, 1990. (Cited on page 106.)

[203] A. J. Scott and M. Knott. A Cluster Analysis Method for Grouping Means in the Analysis of Variance. *Biometrics*, 30(3):507–512, 1974. (Cited on pages 67, 141, 165, and 197.)

[204] C. Seiffert, T. M. Khoshgoftaar, and J. Van Hulse. Improving software-quality predictions with data sampling and boosting. *IEEE Transactions on Systems, Man, and Cybernetics Part A:Systems and Humans*, 39(6):1283–1294, 2009. (Cited on pages 23 and 45.)

[205] C. Seiffert, T. M. Khoshgoftaar, J. Van Hulse, and A. Folleco. An Empirical Study of the Classification Performance of Learners on Imbalanced and

Noisy Software Quality Data. *Information Sciences*, 259:571–595, 2014. (Cited on page 57.)

[206] G. Seni and J. F. Elder. *Ensemble Methods in Data Mining: Improving Accuracy Through Combining Predictions*, volume 2. Jan. 2010. (Cited on pages 24, 46, and 138.)

[207] M. Shepperd, D. Bowes, and T. Hall. Researcher Bias: The Use of Machine Learning in Software Defect Prediction. *IEEE Transactions on Software Engineering (TSE)*, 40(6):603–616, 2014. (Cited on pages 3, 5, 8, 32, 34, 38, 39, 97, and 201.)

[208] M. Shepperd and G. Kadoda. Comparing software prediction techniques using simulation. *IEEE Transactions on Software Engineering (TSE)*, 27(11):1014–1022, 2001. (Cited on pages 10, 138, and 155.)

[209] M. Shepperd, Q. Song, Z. Sun, and C. Mair. Data quality: Some comments on the NASA software defect datasets. *IEEE Transactions on Software Engineering (TSE)*, 39(9):1208–1215, 2013. (Cited on pages 100, 101, 134, 142, 158, 187, 247, and 248.)

[210] E. Shihab. *An Exploration of Challenges Limiting Pragmatic Software Defect Prediction*. PhD thesis, Queen's University, 2012. (Cited on pages 23, 25, 52, 149, and 201.)

[211] E. Shihab, A. E. Hassan, B. Adams, and Z. M. Jiang. An Industrial Study on the Risk of Software Changes. In *Proceedings of the International Symposium on the Foundations of Software Engineering (FSE)*, page 62, 2012. (Cited on pages 2, 3, 107, and 163.)

[212] E. Shihab, Z. M. Jiang, W. M. Ibrahim, B. Adams, and A. 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 International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2010. (Cited on pages 2, 52, and 63.)

[213] E. Shihab, Y. Kamei, B. Adams, and A. E. Hassan. High-Impact Defects : A Study of Breakage and Surprise Defects. pages 300–310, 2011. (Cited on page 17.)

[214] E. Shihab, Y. Kamei, B. Adams, and A. E. Hassan. High-Impact Defects: A Study of Breakage and Surprise Defects. In *Proceedings of the joint meeting of the European Software Engineering Conference and the symposium on the Foundations of Software Engineering (ESEC/FSE'11)*, pages 300–310, 2011. (Cited on pages 2, 25, 52, 57, 98, and 163.)

[215] J. Shimagaki, Y. Kamei, S. McIntosh, A. E. Hassan, and N. Ubayashi. A study of the quality-impacting practices of modern code review at sony mobile. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 212–221. ACM, 2016. (Cited on page 3.)

[216] J. Siegmund, N. Siegmund, and S. Apel. Views on internal and external validity in empirical software engineering. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 1276–1304, 2015. (Cited on page 40.)

[217] M. R. Sikonja. Improving Random Forests. In *Proceedings of the European Conference on Machine Learning (ECML)*, pages 359–370, 2004. (Cited on page 65.)

[218] Y. Singh and A. S. Chauhan. Neural networks in data mining. *Journal of Theoretical and Applied Information Technology*, 5(6):36–42, 2009. (Cited on page 90.)

[219] L. Song, L. L. Minku, and X. Yao. The impact of parameter tuning on software effort estimation using learning machines. In *Proceedings of the International Conference on Predictive Models in Software Engineering (PROMISE)*, page 9, 2013. (Cited on page 185.)

[220] L. Song, L. L. Minku, and X. Yao. The Impact of Parameter Tuning on Software Effort Estimation Using Learning Machines. In *Proceedings of the International Conference on Predictive Models in Software Engineering (PROMISE)*, pages 1–10, 2013. (Cited on pages 98, 114, and 132.)

[221] Q. Song, Z. Jia, M. Shepperd, S. Ying, and J. Liu. A General Software Defect-Proneness Prediction Framework. *IEEE Transactions on Software Engineering (TSE)*, 37(3):356–370, 2011. (Cited on pages 45, 99, 101, 143, and 156.)

[222] E. W. Steyerberg. *Clinical prediction models: a practical approach to development, validation, and updating*. Springer Science & Business Media, 2008. (Cited on pages 26, 27, 107, 150, 152, and 164.)

[223] E. W. Steyerberg, M. J. Eijkemans, F. E. Harrell Jr., and J. D. Habbema. Prognostic modelling with logistic regression analysis: a comparison of selection and estimation methods in small data sets. *Statistics in Medicine*, 19:1059–1079, 2000. (Cited on pages 26, 141, and 164.)

[224] E. W. Steyerberg, A. J. Vickers, N. R. Cook, T. Gerds, N. Obuchowski, M. J. Pencina, and M. W. Kattan. Assessing the performance of prediction models: a framework for some traditional and novel measures. *Epidemiology*, 21(1):128–138, 2010. (Cited on pages 26, 107, and 164.)

[225] M. Stone. Cross-Validatory Choice and Assessment of Statistical Predictions. *Journal of the Royal Statistical Society*, 36(2):111–147, 1974. (Cited on pages 27 and 150.)

[226] M. D. Syer, M. Nagappan, B. Adams, and A. E. Hassan. Replicating and re-evaluating the theory of relative defect-proneness. *IEEE Transactions on Software Engineering (TSE)*, 41(2):176–197, 2015. (Cited on page 19.)

[227] M. Tan, L. Tan, S. Dara, and C. Mayeux. Online Defect Prediction for Imbalanced Data. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 99–108, 2015. (Cited on pages 3, 25, 99, 107, 163, and 200.)

[228] C. Tantithamthavorn. ScottKnottESD: An R package of The Scott-Knott Effect Size Difference (ESD) Test. `https://cran.r-project.org/web/packages/ScottKnottESD/index.html`, 2016. (Cited on page 165.)

[229] C. Tantithamthavorn. Towards a Better Understanding of the Impact of Experimental Components on Defect Prediction Modelling. In *Companion Proceedings of the International Conference on Software Engineering (ICSE)*, pages 867—870, 2016. (Cited on pages 1, 41, 97, 130, 134, 138, 184, 185, 187, and 201.)

[230] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, A. Ihara, and K. Matsumoto. The Impact of Mislabelling on the Performance and Interpretation of Defect Prediction Models. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 812–823, 2015. (Cited on pages 51, 97, 98, 130, 131, 134, 142, 187, and 201.)

[231] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto. An Empirical Comparison of Model Validation Techniques for Defect Prediction Models. *IEEE Transactions on Software Engineering (TSE)*, 2016. (Cited on pages 102, 110, 131, and 137.)

[232] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto. Automated Parameter Optimization of Classification Techniques for Defect

Prediction Models. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 321–322, 2016. (Cited on pages 24, 46, 83, 138, and 185.)

[233] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto. Comments on "Researcher Bias: The Use of Machine Learning in Software Defect Prediction". *IEEE Transactions on Software Engineering (TSE)*, 2016. (Cited on pages 31, 101, 102, 103, 106, 126, and 156.)

[234] A. Tarvo. Using Statistical Models to Predict Software Regressions. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, 2008. (Cited on page 150.)

[235] G. Tassey. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology, RTI Project*, 7007(011), 2002. (Cited on page 2.)

[236] M. Tenenhaus, V. E. Vinzi, Y.-M. Chatelin, and C. Lauro. Pls path modeling. *Computational statistics & data analysis*, 48(1):159–205, 2005. (Cited on page 89.)

[237] S. W. Thomas, M. Nagappan, D. Blostein, and A. E. Hassan. The Impact of Classifier Configuration and Classifier Combination on Bug Localization. *IEEE Transactions on Software Engineering (TSE)*, 39(10):1427–1443, 2013. (Cited on page 132.)

[238] P. Thongtanunam, S. McIntosh, A. E. Hassan, and H. Iida. Revisiting code ownership and its relationship with software quality in the scope of modern code review. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 1039–1050, 2016. (Cited on pages 20 and 98.)

[239] Y. Tian, N. Ali, D. Lo, and A. E. Hassan. On the unreliability of bug severity data. *Empirical Software Engineering*, pages 1–26, 2015. (Cited on page 43.)

[240] M. Torchiano. effsize: Efficient effect size computation. `http://CRAN.R-project.org/package=effsize`, 2015. (Cited on pages 110 and 166.)

[241] A. Tosun. Ensemble of Software Defect Predictors: A Case Study. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 318–320, 2008. (Cited on pages 24, 46, and 138.)

[242] A. Tosun and A. Bener. Reducing false alarms in software defect prediction by decision threshold optimization. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 477–480, 2009. (Cited on pages 84, 96, and 114.)

[243] Y.-K. Tu, M. Kellett, V. Clerehugh, and M. S. Gilthorpe. Problems of correlations between explanatory variables in multiple regression analyses in the dental literature. *British dental journal*, 199(7):457–461, 2005. (Cited on page 34.)

[244] B. Turhan. On the dataset shift problem in software engineering prediction models. *Empirical Software Engineering*, 17(1-2):62–74, 2011. (Cited on page 162.)

[245] B. Turhan, G. Kocak, and A. Bener. Data mining source code for locating software bugs: A case study in telecommunication industry. *Expert Systems with Applications*, 36(6):9986–9990, 2009. (Cited on pages 24, 46, and 138.)

[246] B. Turhan, A. Tosun Misirli, and A. Bener. Empirical evaluation of the effects of mixed project data on learning defect predictors. *Information*

and Software Technology (IST), 55(6):1101–1118, 2013. (Cited on pages 10, 42, 47, 138, and 154.)

[247] H. Wang, T. M. Khoshgoftaar, and A. Napolitano. A Comparative Study of Ensemble Feature Selection Techniques for Software Defect Prediction. In *International Conference on Machine Learning and Applications (ICMLA)*, pages 135–140, 2010. (Cited on pages 24, 46, and 138.)

[248] S. Wang and X. Yao. Using Class Imbalance Learning for Software Defect Prediction. *IEEE Transactions on Reliability*, 62(2):434–443, 2013. (Cited on page 132.)

[249] T. Wang and W.-h. Li. Naive Bayes Software Defect Prediction Model. *International Conference on Computational Intelligence and Software Engineering (CISE)*, (2006):1–4, 2010. (Cited on page 143.)

[250] E. J. Weyuker, T. J. Ostrand, and R. M. Bell. Do too many cooks spoil the broth? Using the number of developers to enhance defect prediction models. *Empirical Software Engineering*, 13(5):539–559, 2008. (Cited on pages 63 and 96.)

[251] I. H. Witten and E. Frank. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2005. (Cited on page 23.)

[252] D. H. Wolpert. Stacked generalization. *Neural networks*, 5(2):241–259, 1992. (Cited on pages 24, 46, and 138.)

[253] R. Wu, H. Zhang, S. Kim, and S. C. Cheung. ReLink: Recovering Links between Bugs and Changes. In *Proceedings of the joint meeting of the European Software Engineering Conference and the symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 15–25, 2011. (Cited on pages 44, 55, 101, 130, and 142.)

[254] Z.-H. Zhou. Ensemble learning. In *Encyclopedia of Biometrics*, pages 270–273. Springer, 2009. (Cited on page 93.)

[255] T. Zimmermann and N. Nagappan. Predicting defects using network analysis on dependency graphs. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 531–540, 2008. (Cited on pages 3, 19, and 150.)

[256] T. Zimmermann and N. Nagappan. Predicting defects with program dependencies. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 435–438, 2009. (Cited on page 3.)

[257] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy. Cross-project defect prediction. In *Proceedings of the European Software Engineering Conference and the symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 91–100, 2009. (Cited on pages 3, 10, 19, 24, 46, 138, and 199.)

[258] T. Zimmermann, R. Premraj, and A. Zeller. Predicting Defects for Eclipse. In *Proceedings of the International Workshop on Predictor Models in Software Engineering (PROMISE)*, pages 9–20, 2007. (Cited on pages 2, 17, 19, 25, 52, 100, 101, 142, 150, 158, 163, and 250.)

APPENDIX A

Replication Package for Chapter 3

A.1. Download dataset and scripts

- Download misc.R from http://sailhome.cs.queensu.ca/replication/researcher_bias_comments/misc.R
- Download rawdata.csv from http://sailhome.cs.queensu.ca/replication/researcher_bias_comments/rawdata.csv, which is a copy version of https://codefeedback.cs.herts.ac.uk/mlbias/rawdata.csv

A.2. Load miscellaneous functions

```
source("misc.R")
```

A.3. The Presence of Collinearity

```
pair <- combn(c("ResearcherGroup","MetricFamily","DatasetFamily","ClassifierFamily"),2)
results <- NULL
```

```
for(i in 1:ncol(pair)){
    Xsq <- assocstats(table(data[,pair[1,i]],data[,pair[2,i]]  ))

    results <- rbind(results, c(
        "Pair"=paste0(pair[,i],collapse=" = ") ,
        "Cramer's V"=sprintf("%.2f%s",
                             Xsq$cramer,
                             ifelse(Xsq$chisq_tests[2,3] <= 0.05,
   "***" , "-")
        ),
        "Magnitude"=Cramer(Xsq$cramer)
    )
    )
}
```

A.3.1. Table 3.1

```
results[rev(order(results[,"Cramer's V"])),]
```

```
##      Pair                                    Cramer's V Magnitude
## [1,] "ResearcherGroup = MetricFamily"     "0.65***"  "Strong"
## [2,] "ResearcherGroup = DatasetFamily"    "0.56***"  "Rel.strong"
## [3,] "MetricFamily = DatasetFamily"       "0.55***"  "Rel.strong"
## [4,] "ResearcherGroup = ClassifierFamily" "0.54***"  "Rel.strong"
## [5,] "DatasetFamily = ClassifierFamily"   "0.34***"  "Moderate"
## [6,] "MetricFamily = ClassifierFamily"    "0.21***"  "Moderate"
```

A.4. The Interference of Collinearity

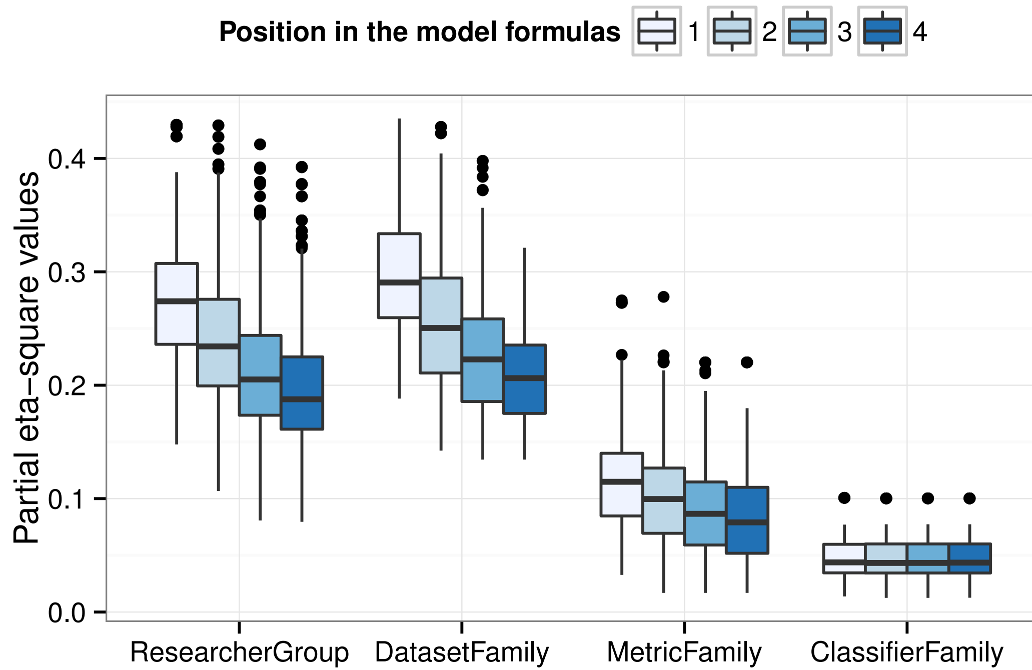
```
bootstrap_analysis <- function(data, indep, N){

    results <- NULL
    for(i in 1:N){
```

```
5         # generate a bootstramp sample
6         indices <- sample(nrow(data),nrow(data),replace=T)
7
8         # permute the ordering of explanatory variables
9         for(ordering in permn(1:length(indep)) ){
10            predictor <- indep[ordering]
11
12            # train a linear regression model using the bootstrap
    sample
13            f <- formula(paste("MCC ~ " , paste(predictor,collapse="
     * ")))
14            m <- lm(f, data=data[indices,])
15
16            # compute the partial eta-square statistic
17            eta_results <- etasq(aov(m),partial=T)[,"Partial eta
    ^2"][1:length(indep)]
18            names(eta_results) <- predictor
19
20            ranking <- 1:4
21            names(ranking) <- indep[ordering]
22
23            results <- rbind(results, data.frame(
24                variable=indep,
25                ranking=ranking[indep],
26                value=eta_results[indep])
27            )
28        }
29    }
30    rownames(results) <- NULL
31    return(data.frame(results))
32 }
33
34 set.seed(1234)
35 indep <- c("ResearcherGroup","DatasetFamily","MetricFamily","
    ClassifierFamily")
```

```
36 results <- bootstrap_analysis(data,indep,1000)
37 saveRDS(results, file = "bootstrap-analysis.rds")
```

A.4.1. Figure 3.1

```
1 results <- readRDS("bootstrap-analysis.rds")
2 results$variable <- factor(results$variable,
3                            levels=c("ResearcherGroup","DatasetFamily
    ",
4                                     "MetricFamily","ClassifierFamily
    "))
5 results$ranking <- factor(results$ranking)
6
7 ggplot( results ,aes(x=variable,y=value,fill=ranking )) +
    geom_boxplot() + theme_bw() +
8    xlab("") + ylab("Partial eta-square values") +
9    scale_fill_brewer(palette="Blues") + theme(legend.position="top
    ") +
10   guides(fill=guide_legend(title="Position in the model formulas")
    )
```



A.5. Mitigating the Collinearity

A.5.1. Select the Eclipse dataset family

```
1 eclipse_data <- data[data$DatasetFamily == "ECLIP",]
2 eclipse_data$DatasetFamily <- NULL
```

A.5.2. Define two models

```
indep_1 <- c("ResearcherGroup","ClassifierFamily")
indep_2 <- c("MetricFamily","ClassifierFamily")
m1 <- lm(formula(paste("MCC ~" , paste(rev(indep_1),collapse=" * "))
    ), data=eclipse_data)
m2 <- lm(formula(paste("MCC ~" , paste(rev(indep_2),collapse=" * "))
    ), data=eclipse_data)
```

A.5.3. Analyze the ResearcherGroup Model

Redundancy analysis

```
redun(formula(paste("~ " , paste(indep_1,collapse=" + "))), data=
    eclipse_data, nk=0)
```

```
##
## Redundancy Analysis
##
## redun(formula = formula(paste("~ ", paste(indep_1,
##     collapse = " + "))), data = eclipse_data, nk = 0)
##
## n: 126   p: 2    nk: 0
##
## Number of NAs:    0
##
## Transformation of target variables forced to be linear
##
## R-squared cutoff: 0.9    Type: ordinary
##
## R^2 with which each variable can be predicted from
##     all other variables:
```

```
##
##  ResearcherGroup ClassifierFamily
##            0.785            0.785
##
## No redundant variables
```

AIC computation

```
AIC(m1);
```

```
## [1] -77.20829
```

Adjusted R2 computation

```
summary(m1)$adj.r.squared
```

```
## [1] 0.1932176
```

Partial eta-square values computation

```
etasq(aov(m1))
```

```
##                                  Partial eta^2
## ClassifierFamily                    0.12237562
## ResearcherGroup                     0.12679423
## ClassifierFamily:ResearcherGroup    0.02177965
## Residuals                                   NA
```

A.5.4. Analyze the MetricFamily Model

Redundancy analysis

```
redun(formula(paste("~ " , paste(indep_2,collapse=" + "))), data=
    eclipse_data, nk=0)
```

```
##
## Redundancy Analysis
##
## redun(formula = formula(paste("~ ", paste(indep_2,
##     collapse = " + "))), data = eclipse_data, nk = 0)
##
## n: 126   p: 2    nk: 0
##
## Number of NAs:    0
##
## Transformation of target variables forced to be linear
##
## R-squared cutoff: 0.9    Type: ordinary
##
## R^2 with which each variable can be predicted from
##     all other variables:
##
##     MetricFamily ClassifierFamily
##            0.279            0.279
##
## No redundant variables
```

AIC computation

```
AIC(m2);
```

```
## [1] -104.9324
```

Adjusted R2 computation

```
summary(m2)$adj.r.squared
```

```
## [1] 0.3612852
```

Partial eta-square values computation

```r
etasq(aov(m2))
```

```
##                                   Partial eta^2
## ClassifierFamily                      0.1128156
## MetricFamily                          0.2352727
## ClassifierFamily:MetricFamily         0.1625307
## Residuals                                    NA
```

APPENDIX B

A List of 101 Defect Datasets.

B.1. Install and load necessary R packages

```
1 install.packages("devtools")
2 devtools::install_github("klainfo/DefectData")
3 library(DefectData)
```

B.2. A list of 101 datasets

```
1 listData
```

ID	Dataset	Corpus	#Ratio	#Modules	#Defective	#Metrics	EPV
1	JM1	Shepperd *et al.* [209]	21.49	7782	1672	21	79.62
2	KC3	Shepperd *et al.* [209]	18.56	194	36	39	0.92
3	MC1	Shepperd *et al.* [209]	2.31	1988	46	38	1.21
4	MC2	Shepperd *et al.* [209]	35.20	125	44	39	1.13
5	MW1	Shepperd *et al.* [209]	10.67	253	27	37	0.73

Continued on next page

Table B.1 – continued from previous page

ID	Dataset	Corpus	#Ratio	#Modules	#Defective	#Metrics	EPV
6	PC1	Shepperd *et al.* [209]	8.65	705	61	37	1.65
7	PC2	Shepperd *et al.* [209]	2.15	745	16	36	0.44
8	PC3	Shepperd *et al.* [209]	12.44	1077	134	37	3.62
9	PC4	Shepperd *et al.* [209]	13.75	1287	177	37	4.78
10	PC5	Shepperd *et al.* [209]	27.53	1711	471	38	12.39
11	ar1	Shepperd *et al.* [209]	7.44	121	9	29	0.31
12	ar3	Shepperd *et al.* [209]	12.70	63	8	29	0.28
13	ar4	Shepperd *et al.* [209]	18.69	107	20	29	0.69
14	ar5	Shepperd *et al.* [209]	22.22	36	8	29	0.28
15	ar6	Shepperd *et al.* [209]	14.85	101	15	29	0.52
16	cm1	Shepperd *et al.* [209]	12.84	327	42	37	1.14
17	kc2	Shepperd *et al.* [209]	20.50	522	107	21	5.10
18	ant-1.3	Jureczko *et al.* [108]	16	125	20	20	1
19	ant-1.4	Jureczko *et al.* [108]	22.47	178	40	20	2
20	ant-1.5	Jureczko *et al.* [108]	10.92	293	32	20	1.60
21	ant-1.6	Jureczko *et al.* [108]	26.21	351	92	20	4.60
22	ant-1.7	Jureczko *et al.* [108]	22.28	745	166	20	8.30
23	arc	Jureczko *et al.* [108]	11.54	234	27	20	1.35
24	berek	Jureczko *et al.* [108]	37.21	43	16	20	0.80
25	camel-1.0	Jureczko *et al.* [108]	3.83	339	13	20	0.65
26	camel-1.2	Jureczko *et al.* [108]	35.53	608	216	20	10.80
27	camel-1.4	Jureczko *et al.* [108]	16.63	872	145	20	7.25
28	camel-1.6	Jureczko *et al.* [108]	19.48	965	188	20	9.40
29	ckjm	Jureczko *et al.* [108]	50	10	5	20	0.25
30	e-learning	Jureczko *et al.* [108]	7.81	64	5	20	0.25
31	forrest-0.6	Jureczko *et al.* [108]	16.67	6	1	20	0.05
32	forrest-0.7	Jureczko *et al.* [108]	17.24	29	5	20	0.25
33	forrest-0.8	Jureczko *et al.* [108]	6.25	32	2	20	0.10
34	intercafe	Jureczko *et al.* [108]	14.81	27	4	20	0.20
35	ivy-1.1	Jureczko *et al.* [108]	56.76	111	63	20	3.15

Continued on next page

Table B.1 – continued from previous page

ID	Dataset	Corpus	#Ratio	#Modules	#Defective	#Metrics	EPV
36	ivy-1.4	Jureczko *et al.* [108]	6.64	241	16	20	0.80
37	ivy-2.0	Jureczko *et al.* [108]	11.36	352	40	20	2
38	jedit-3.2	Jureczko *et al.* [108]	33.09	272	90	20	4.50
39	jedit-4.0	Jureczko *et al.* [108]	24.51	306	75	20	3.75
40	jedit-4.1	Jureczko *et al.* [108]	25.32	312	79	20	3.95
41	jedit-4.2	Jureczko *et al.* [108]	13.08	367	48	20	2.40
42	jedit-4.3	Jureczko *et al.* [108]	2.24	492	11	20	0.55
43	kalkulator	Jureczko *et al.* [108]	22.22	27	6	20	0.30
44	log4j-1.0	Jureczko *et al.* [108]	25.19	135	34	20	1.70
45	log4j-1.1	Jureczko *et al.* [108]	33.94	109	37	20	1.85
46	log4j-1.2	Jureczko *et al.* [108]	92.20	205	189	20	9.45
47	lucene-2.0	Jureczko *et al.* [108]	46.67	195	91	20	4.55
48	lucene-2.2	Jureczko *et al.* [108]	58.30	247	144	20	7.20
49	lucene-2.4	Jureczko *et al.* [108]	59.71	340	203	20	10.15
50	nieruchomosci	Jureczko *et al.* [108]	37.04	27	10	20	0.50
51	pbeans1	Jureczko *et al.* [108]	76.92	26	20	20	1
52	pbeans2	Jureczko *et al.* [108]	19.61	51	10	20	0.50
53	pdftranslator	Jureczko *et al.* [108]	45.45	33	15	20	0.75
54	poi-1.5	Jureczko *et al.* [108]	59.49	237	141	20	7.05
55	poi-2.0	Jureczko *et al.* [108]	11.78	314	37	20	1.85
56	poi-2.5	Jureczko *et al.* [108]	64.42	385	248	20	12.40
57	poi-3.0	Jureczko *et al.* [108]	63.57	442	281	20	14.05
58	prop-1	Jureczko *et al.* [108]	14.82	18471	2738	20	136.90
59	prop-2	Jureczko *et al.* [108]	10.56	23014	2431	20	121.55
60	prop-3	Jureczko *et al.* [108]	11.49	10274	1180	20	59
61	prop-4	Jureczko *et al.* [108]	9.64	8718	840	20	42
62	prop-5	Jureczko *et al.* [108]	15.25	8516	1299	20	64.95
63	prop-6	Jureczko *et al.* [108]	10	660	66	20	3.30
64	redaktor	Jureczko *et al.* [108]	15.34	176	27	20	1.35
65	serapion	Jureczko *et al.* [108]	20	45	9	20	0.45

Continued on next page

Table B.1 – continued from previous page

ID	Dataset	Corpus	#Ratio	#Modules	#Defective	#Metrics	EPV
66	skarbonka	Jureczko *et al.* [108]	20	45	9	20	0.45
67	sklebagd	Jureczko *et al.* [108]	60	20	12	20	0.60
68	synapse-1.0	Jureczko *et al.* [108]	10.19	157	16	20	0.80
69	synapse-1.1	Jureczko *et al.* [108]	27.03	222	60	20	3
70	synapse-1.2	Jureczko *et al.* [108]	33.59	256	86	20	4.30
71	systemdata	Jureczko *et al.* [108]	13.85	65	9	20	0.45
72	szybkafucha	Jureczko *et al.* [108]	56	25	14	20	0.70
73	termoproject	Jureczko *et al.* [108]	30.95	42	13	20	0.65
74	tomcat	Jureczko *et al.* [108]	8.97	858	77	20	3.85
75	velocity-1.4	Jureczko *et al.* [108]	75	196	147	20	7.35
76	velocity-1.5	Jureczko *et al.* [108]	66.36	214	142	20	7.10
77	velocity-1.6	Jureczko *et al.* [108]	34.06	229	78	20	3.90
78	workflow	Jureczko *et al.* [108]	51.28	39	20	20	1
79	wspomaganiepi	Jureczko *et al.* [108]	66.67	18	12	20	0.60
80	xalan-2.4	Jureczko *et al.* [108]	15.21	723	110	20	5.50
81	xalan-2.5	Jureczko *et al.* [108]	48.19	803	387	20	19.35
82	xalan-2.6	Jureczko *et al.* [108]	46.44	885	411	20	20.55
83	xalan-2.7	Jureczko *et al.* [108]	98.79	909	898	20	44.90
84	xerces-1.2	Jureczko *et al.* [108]	16.14	440	71	20	3.55
85	xerces-1.3	Jureczko *et al.* [108]	15.23	453	69	20	3.45
86	xerces-1.4	Jureczko *et al.* [108]	74.32	588	437	20	21.85
87	xerces-init	Jureczko *et al.* [108]	47.53	162	77	20	3.85
88	zuzel	Jureczko *et al.* [108]	44.83	29	13	20	0.65
89	eclipse-2.0	Zimmermann *et al.* [258]	14.49	6729	975	32	30.47
90	eclipse-2.1	Zimmermann *et al.* [258]	10.83	7888	854	32	26.69
91	eclipse-3.0	Zimmermann *et al.* [258]	14.80	10593	1568	32	49
92	Apache	Kim *et al.* [118]	50.52	194	98	26	3.77
93	Safe	Kim *et al.* [118]	39.29	56	22	26	0.85
94	Zxing	Kim *et al.* [118]	29.57	399	118	26	4.54
95	eclipse34_debug	Kim *et al.* [118]	24.69	1065	263	17	15.47

Continued on next page

Table B.1 – continued from previous page

ID	Dataset	Corpus	#Ratio	#Modules	#Defective	#Metrics	EPV
96	eclipse34_swt	Kim *et al.* [118]	43.97	1485	653	17	38.41
97	equinox	D'Ambros *et al.* [50, 51]	39.81	324	129	15	8.60
98	jdt	D'Ambros *et al.* [50, 51]	20.66	997	206	15	13.73
99	lucene	D'Ambros *et al.* [50, 51]	9.26	691	64	15	4.27
100	mylyn	D'Ambros *et al.* [50, 51]	13.16	1862	245	15	16.33
101	pde	D'Ambros *et al.* [50, 51]	13.96	1497	209	15	13.93

Listing B.1: A List of 101 Defect Datasets.

APPENDIX C

An example R script for Caret parameter optimization

C.1. Install necessary R packages

```
1 install.packages("caret")
2 install.packages("devtools")
3 devtools::install_github("klainfo/DefectData")
```

C.2. Caret parameter optimization

```
1 library(DefectData)
2 library(caret)
3
4 results <- NULL
5
6 grid.param <-  expand.grid(trials=c(1,10,20,30,40),model=c("tree","
    rules"),winnow=c(TRUE,FALSE))
7 default.param <-  expand.grid(trials=c(1),model=c("rules"),winnow=c(
    FALSE))
```

```
8
9  for(system in listData[listData$EPV > 10 & listData$DefectiveRatio <
       50,]$system){
10     Data <- loadData(system)
11     data <- Data$data
12     dep <- Data$dep
13     indep <- Data$indep
14
15     transformLog  <- function(y){ y <- log1p(y)}
16     indep.log <- apply(data[,indep], 2, function(x) { !(min(x) < 0)
      })
17     indep.log <- names(indep.log[which(indep.log == TRUE)])
18     data[,indep.log] <- data.frame(apply(data[,indep.log], 2,
      transformLog))
19     data[,dep] <- as.factor(ifelse(data[,dep] == "TRUE","T","F"))
20
21     ctrl <- trainControl(method = "boot", number = 100, classProbs =
       TRUE, summaryFunction=twoClassSummary)
22
23     set.seed(1234)
24     optimize <- caret::train(data[,indep], data[,dep],
25                              method = "C5.0",
26                              trControl = ctrl,
27                              tuneGrid = grid.param,
28                              metric = "ROC")
29
30     set.seed(1234)
31     default <- caret::train(data[,indep], data[,dep],
32                              method = "C5.0",
33                              trControl = ctrl,
34                              tuneGrid = default.param,
35                              metric = "ROC")
36
37     results <- rbind(results, c(optimize=max(optimize$results$ROC),
      default=max(default$results$ROC)))
```

```
38 }
39
40 results <- data.frame(results)
41 results$system <- listData[listData$EPV > 10 &
       listData$DefectiveRatio < 50,]$system
42 saveRDS(results, file="C5.0.rds")
```

Listing C.1: An example R script for grid-search parameter optimization.

C.3. Results

```r
results <- readRDS(file="C5.0.rds")
results$improvement <- results$optimize-results$default
results
```

```
##     optimize   default          system improvement
## 1  0.6812443 0.5276690             JM1  0.15357536
## 2  0.7718680 0.4831502             PC5  0.28871785
## 3  0.6817029 0.4994888       camel-1.2  0.18221415
## 4  0.8179451 0.4536583          prop-1  0.36428682
## 5  0.8545482 0.4823125          prop-2  0.37223573
## 6  0.7467892 0.4786966          prop-3  0.26809259
## 7  0.7624442 0.4425753          prop-4  0.31986892
## 8  0.7533058 0.4630926          prop-5  0.29021322
## 9  0.7626922 0.6653037       xalan-2.5  0.09738852
## 10 0.8443650 0.7655026       xalan-2.6  0.07886231
## 11 0.8401523 0.4243956     eclipse-2.0  0.41575663
## 12 0.7895530 0.4386999     eclipse-2.1  0.35085306
## 13 0.8010375 0.5310208     eclipse-3.0  0.27001672
## 14 0.8002733 0.5397900 eclipse34_debug  0.26048335
## 15 0.9685388 0.9149021   eclipse34_swt  0.05363675
## 16 0.8139885 0.5447301             jdt  0.26925838
## 17 0.7917020 0.4136083           mylyn  0.37809369
## 18 0.7260917 0.4600913             pde  0.26600047
```

APPENDIX D

An R implementation of the generic variable importance computation

D.1. Install necessary R packages

```
1 install.packages("caret")
2 install.packages("devtools")
3 devtools::install_github("klainfo/ScottKnottESD")
4 devtools::install_github("klainfo/DefectData")
```

```
1 GenericVarImp <- function(data, model, dep, indep, shuffletimes =
    25){
2     if(shuffletimes < 10){throw("#Repetition should be higher than
    10.")}
3     set.seed(1234)
4     featuresMeanScores <- c()
5     for(predictorName in indep){
6         featureScores <- c()
7         shuffledData <- data[,indep]
8         for (iter in 1:shuffletimes) {
9             shuffledData[,predictorName] <- sample(shuffledData[,
```

```
    predictorName], length(shuffledData[,predictorName]))
10              predictions <- ifelse(predict(object=model, shuffledData
    [,indep], type='prob')[[2]] > 0.5, 'T', 'F')
11              featureScores <- c(featureScores, mean(data[,dep] !=
    predictions) )
12          }
13          featuresMeanScores <- cbind(featuresMeanScores,
    featureScores)
14      }
15      colnames(featuresMeanScores) <- indep
16      return(data.frame(featuresMeanScores))
17 }
```

Listing D.1: An R implementation of the generic variable importance function.

```
1 library(DefectData)
2 library(caret)
3 library(ScottKnottESD)
4 
5 results <- NULL
6 
7 Data <- loadData("eclipse-2.0")
8 data <- Data$data
9 dep <- Data$dep
10 indep <- Data$indep
11 
12 transformLog  <- function(y){ y <- log1p(y)}
13 indep.log <- apply(data[,indep], 2, function(x) { !(min(x) < 0)})
14 indep.log <- names(indep.log[which(indep.log == TRUE)])
15 data[,indep.log] <- data.frame(apply(data[,indep.log], 2,
    transformLog))
16 data[,dep] <- as.factor(ifelse(data[,dep] == "TRUE","T","F"))
17 
18 ctrl <- trainControl(method = "boot", number = 100, classProbs =
    TRUE, summaryFunction=twoClassSummary)
```

```
19
20 m <- train(data[,indep], data[,dep],
21                   method = "C5.0",
22                   trControl = ctrl,
23                   tuneLength = 3,
24                   metric = "ROC")
25
26 varimp <- GenericVarImp(data, m, dep, indep, shuffletimes = 25)
27
28 print(SK.ESD(varimp)$group)
```

Listing D.2: An example usage of the generic variable importance function.

```
##     pre  NBD_sum FOUT_max   VG_avg FOUT_sum
##       1        2        2        3        4
## PAR_avg     TLOC  PAR_max  NBD_avg   VG_sum
##       5        5        6        7        8
## NBD_max   VG_max MLOC_sum FOUT_avg MLOC_max
##       8        9        9        9        9
## PAR_sum  NOM_avg MLOC_avg  NSF_max  NSM_max
##      10       11       11       12       12
##     ACD  NOM_max  NOM_sum  NSM_avg  NOF_avg
##      12       13       13       13       13
## NOF_max  NSF_avg      NOI  NOF_sum  NSM_sum
##      13       13       14       14       14
##     NOT  NSF_sum
##      14       14
```

APPENDIX E

An R Implementation of the Scott-Knott ESD Test

```
1  function SK.ESD(data, long = FALSE)
2  {
3      library(ScottKnott)
4      library(reshape)
5      library(effsize)
6      if (long) {
7          tmp <- do.call(cbind, split(data, data$variable))
8          tmp <- tmp[, grep("value", names(tmp))]
9          names(tmp) <- gsub(".value", "", names(tmp))
10         data <- tmp
11     }
12
13     # Normallity Correction
14     data <- data.frame(data)
15     transformLog <- function(y) {y <- log1p(y)}
16     data <- data.frame(apply(data, 2, transformLog))
17
18     av <- aov(value ~ variable, data = melt(data))
```

```
19    sk <- SK(av, which = "variable", dispersion = "s", sig.level =
      0.05)
20    sk$original <- sk$groups
21    names(sk$original) <- rownames(sk$m.inf)
22    ranking <- sk$groups
23    names(ranking) <- rownames(sk$m.inf)
24    sk$diagnosis <- NULL
25    keys <- names(ranking)
26
27    # Effect Size Correction
28    for (k in seq(2, length(keys))) {
29        eff <- unlist(cohen.d(data[, keys[k]], data[, keys[k -
30            1]])[c("magnitude", "estimate")])
31        sk$diagnosis <- rbind(sk$diagnosis, c(sprintf("[%d] %s (%.3f
      )",
32            ranking[k - 1], keys[k - 1], mean(data[, keys[k -
33                1]])), sprintf("[%d] %s (%.3f)", ranking[k],
34            keys[k], mean(data[, keys[k]])), eff))
35        if (eff["magnitude"] == "negligible" && ranking[k] !=
36            ranking[k - 1]) {
37            ranking[seq(k, length(keys))] = ranking[seq(k, length(
      keys))] -
38                1
39        }
40    }
41    sk$groups <- ranking
42    sk$reverse <- max(ranking) - ranking + 1
43    sk$diagnosis <- data.frame(sk$diagnosis)
44    return(sk)
45 }
```

Listing E.1: An R implementation of the Scott-Knott ESD test.

APPENDIX F

An example R script for out-of-sample bootstrap validation

F.1. Install necessary R packages

```
install.packages("pROC")
install.packages("devtools")
devtools::install_github("klainfo/DefectData")
```

```
library(DefectData)
library(pROC)
Data <- loadData("eclipse-2.0")
data <- Data$data
dep <- Data$dep
indep <- Data$indep

# Normality Correction
transformLog  <- function(y){ y <- log1p(y)}
data[,indep] <- data.frame(apply(data[,indep], 2, transformLog))
data[,dep] <- as.factor(ifelse(data[,dep] == "TRUE","T","F"))

```

```
13 performance <- NULL
14 for(i in seq(1,100)){
15     # Generate a bootstrap sample with replacement
16     indices <- sample(nrow(data),replace=TRUE)
17 
18     # Generate training dataset using a bootstrap sample
19     training <- data[indices,]
20 
21     # Generate testing dataset (i.e., instances that
22     # are not included in the bootstrap sample)
23     testing <- data[-unique(indices),]
24 
25     # Generate model formula
26     f <- as.formula(paste0(dep, " ~ ", paste0(indep,collapse = "+"))
    )
27 
28     # Fit a prediction model using a logistic regression model
29     m <- glm(f, data=training, family="binomial")
30 
31     # Extract probabilities using the testing dataset
32     prob <- predict(m, testing, type="response")
33 
34     # Compute AUC performance
35     performance <- c(performance, auc(testing[,dep],prob))
36 }
```

Listing F.1: An example R script for out-of-sample bootstrap validation.

```
1 # Report the average AUC performance
2 mean(performance)
```

```
[1] 0.8319609
```