

# **Automated Static Analysis Tools: A Multidimensional View on Software Quality Evolution**

Dissertation  
zur Erlangung des mathematisch-naturwissenschaftlichen Doktorgrades  
“Doctor rerum naturalium”  
der Georg-August-Universität Göttingen

im Promotionsprogramm Computer Science (PCS)  
der Georg-August University School of Science (GAUSS)

vorgelegt von

Alexander Trautsch  
aus Göttingen

Göttingen, Juni 2022

### Betreuungsausschuss

Prof. Dr. Jens Grabowski,  
Institut für Informatik, Georg-August-Universität Göttingen

Prof. Dr. Jürgen Dix,  
TU Claustal

Prof. Dr. Steffen Herbold,  
TU Claustal

### Mitglieder der Prüfungskommission

1. Referent: Prof. Dr. Steffen Herbold,  
TU Claustal

2. Referent: Assistant Prof. Mario Linares-Vásquez, PhD,  
Universidad de los Andes Colombia

3. Referent: Prof. Dr. Jens Grabowski,  
Institut für Informatik, Georg-August-Universität Göttingen

### Weitere Mitglieder der Prüfungskommission

Prof. Dr. Alexander Ecker,  
Georg-August-Universität Göttingen

Prof. Dr. Marcus Baum,  
Georg-August-Universität Göttingen

Prof. Dr. Carsten Damm,  
Georg-August-Universität Göttingen

### Tag der mündlichen Prüfung

7. Juli 2022

## Abstract

Software use is ubiquitous. The quality and the evolution of quality over long periods of time is therefore of notable importance. Software engineering research investigates software quality in multiple areas. One of these areas are predictive models, in which measurements of past changes to the source code or file contents are used to assess the quality of changes, files or even product releases. However, these predictive models have yet to transition from research to practice on a larger scale. In contrast, Automated Static Analysis Tools (ASATs) are used in practice and are also part of several software quality models. ASATs are able to warn developers about parts of the source code that violate best practices or match common defect patterns. One downside of ASATs are false positives, i.e., warnings about parts of the code which are not problematic. Developers have to manually assess the warnings and annotate the code or the ASAT configuration to mitigate this. Within this thesis, we investigate the evolution of software quality with a focus on a general purpose ASAT for Java. Our main objective is to determine if the use of an ASAT can improve software quality, as measured by defects, significantly enough to mitigate additional effort by the developers to use the ASAT. We combine multiple software engineering research techniques and data validation studies to improve the signal-to-noise ratio to increase the validity and stability of our results. We focus on a general purpose ASAT for the Java programming language due to the maturity of the language and the large number of projects available for this language. Both the language and the general purpose ASAT have been available for a long time, which allows us to include longer periods of time for our analyses. We study how the ASAT is applied, how the generated warnings evolve over long time periods, and how it affects the quality of the source code in terms of defects. In addition, we include the perspective of the developers regarding software quality improvement by measuring changes when developers intend to improve the quality of the source code. Our studies yield surprising insights. While our results show that ASATs have a positive impact on software quality, the magnitude of the impact is much smaller than expected. Moreover, we can show that corrective changes are the main driver of complexity in software projects. They introduce more complexity than feature additions or any other type of maintenance. In addition, we find that software quality estimation models benefit more from size and complexity metrics than static analysis warnings of an ASAT. Our study of developer intents to increase software quality mirrors this result.



## Zusammenfassung

Software ist allgegenwärtig. Die Qualität von Software und die Entwicklung der Qualität über lange Zeiträume ist daher von großer Bedeutung. Die Software-Engineering-Forschung untersucht Softwarequalität in mehreren Bereichen, zum Beispiel Vorhersagemodelle für Software Qualität. Dabei werden Messungen vergangener Änderungen am Quellcode oder an Dateinhalten verwendet, um die Qualität von Dateien, Änderungen oder Produktversionen zu bewerten. Allerdings müssen diese prädiktiven Modelle erst noch von der Forschung in die Praxis transferiert werden. Im Gegensatz zu prädiktiven Modellen, werden automatisierte statische Analysetools (ASATs) schon heute in der Praxis eingesetzt und sind auch Teil verschiedener Software-Qualitätsmodelle. ASATs sind in der Lage, Entwickler zu warnen, wenn Teile des Quellcodes gegen Best Practices verstoßen oder gängigen Fehlermustern entsprechen. Ein Nachteil von ASATs sind Warnungen über Teile des Codes die nicht problematisch sind. Die Entwickler müssen daher jede Warnung manuell bewerten. In dieser Arbeit untersuchen wir die Entwicklung der Softwarequalität mit einem Fokus auf einen ASAT für Java. Unser Hauptziel ist es, festzustellen, ob der Einsatz eines ASAT die Softwarequalität, gemessen an Defekten, signifikant genug verbessern kann, um den zusätzlichen Aufwand für den Entwickler zu rechtfertigen. Wir kombinieren mehrere Software-Engineering-Forschungstechniken und Datenvalidierungsstudien, um das Signal-Rausch-Verhältnis unserer Daten für die Analyse zu verbessern. Wir konzentrieren uns auf einen ASAT für die Programmiersprache Java, da sowohl die Sprache als auch der ASAT ausgereift und seit langem verfügbar sind, was es uns ermöglicht, längere Zeiträume in unsere Analysen einzubeziehen. Wir untersuchen wie der ASAT angewendet wird, wie sich die generierten Warnungen über lange Zeiträume entwickeln und wie sich die Qualität des Quellcodes in Bezug auf Fehler entwickelt. Darüber hinaus beziehen wir die Perspektive der Entwickler hinsichtlich der Verbesserung der Softwarequalität ein, indem wir Veränderungen messen, wenn Entwickler beabsichtigen die Qualität des Quellcodes zu verbessern. Unsere Studien liefern überraschende Erkenntnisse. Unsere Ergebnisse zeigen zwar, dass ASATs einen positiven Einfluss auf die Softwarequalität haben, aber die Größenordnung des Einflusses ist viel kleiner als erwartet. Darüber hinaus können wir zeigen, dass Fehlerbehebung der Haupttreiber für die Komplexität von Softwareprojekten ist. Fehler zu beheben führt zu mehr Komplexität als Ergänzungen der Funktionalität oder andere Arten von Wartung. Darüber hinaus stellen wir fest, dass Modelle zur Bewertung von Softwarequalität mehr von Größen- und Komplexitätsmetriken profitieren als von ASAT Warnungen.



# Acknowledgments

I started this journey towards my PhD because I thought it would be fun, and it really was! While learning and working on interesting topics made this fun, it was the people I met that made this journey really enjoyable. I had the privilege to be working and learning together with, and be taught by, very amazing people.

I would like to thank my thesis advisory committee Prof. Dr. Jens Grabowski, Prof. Dr. Jürgen Dix and Prof. Dr. Steffen Herbold for feedback and discussions regarding the thesis. Moreover, I would like to thank Prof. Dr. Marcus Baum, Prof. Dr. Alexander Ecker, Prof. Dr. Carsten Damm and Assistant Prof. Mario Linares-Vásquez, PhD for their time.

I would also like to thank current and former colleagues at the Software Engineering for Distributed Systems research group at the Georg-August-Universität Göttingen as well as the AI Engineering research group at the TU Clausthal. I can not hope to list everyone and everything that had a positive influence on me, nonetheless I want to say thanks in no particular order. Jens for always giving valuable feedback for papers and lots of general advice. Steffen for organizing so much and always having interesting ideas and giving feedback for mine. Patrick for providing positive vibes in the office, as well as teaching the value of feedback and how to give feedback. Ella for throwing the best parties and correcting my English. Johannes for his barbecue and organization skills and for being the best travel companion for business trips. Philip for giving great feedback and asking great questions. Gunnar for making sure the infrastructure runs smoothly and nice conversations about how the infrastructure runs smoothly.

These acknowledgments would not be complete without mentioning Frank Malchow who enabled me to start studying while continuing to work part time. Without this, I would not have been able to even start this journey.

I would also like to thank my brother Fabian for lively discussions and book recommendations, as well as providing me with an adorable and clever niece (thanks to you too, Irina!). Last but not least, I thank my girlfriend Cathrin for always supporting my sometimes erratic working hours. Thank you for brightening up my life. I hope we will be able to go on a longer vacation soon!



# Contents

<b>List of Abbreviations</b>	<b>XI</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Goals . . . . .	3
1.2 Contributions . . . . .	5
1.3 Impact . . . . .	7
1.4 Structure of the thesis . . . . .	10
<b>2 Publications</b>	<b>13</b>
2.1 Overview . . . . .	13
2.2 Summaries . . . . .	15
<b>3 Discussion</b>	<b>21</b>
3.1 ASAT warning evolution . . . . .	21
3.2 Impact on internal software quality . . . . .	22
3.3 Impact on external software quality . . . . .	23
3.4 Internal and external quality from the developers' perspective . . . . .	25
<b>4 Conclusion</b>	<b>27</b>
4.1 Summary . . . . .	27
4.2 Outlook . . . . .	28
<b>Bibliography</b>	<b>31</b>
<b>Appendices</b>	<b>41</b>
A Study of ASAT warning evolution . . . . .	41
B Data validity issues in defect-based analyses . . . . .	99
C Improving predictive models with ASAT information . . . . .	149
D Improving data validity via automatic issue classification . . . . .	163
E An NLP model for software engineering data . . . . .	201
F Analysis of tangled changes . . . . .	225
G Changes in quality increasing commits . . . . .	281
H ASAT warning density in defect inducing changes . . . . .	327



# List of Abbreviations

**LLOC** Logical Lines of Code

**ITS** Issue Tracking System

**VCS** Version Control System

**ASAT** Automated Static Analysis Tool

**CI** Continuous Integration

**NLP** Natural Language Processing

**SZZ** Śliwerski Zimmermann Zeller

# 1 Introduction

Software is used in more and more parts of everyday life, be it home automation via smart home devices, public transport and cars, or everything needed for home office work during a pandemic. The development life cycle of complex software systems is a constant battle between implementing new features and maintaining existing source code [36, 46]. Software quality assurance is an important part of the software development life cycle [39, 59]. Moreover, current software development best practices include many quality focused activities, e.g., testing, code review or retrospectives [28]. Automated Static Analysis Tools (ASATs) can support the developers in all parts of the software development life cycle that are concerned with source code. In a nutshell: ASATs are able to identify possible problems in the source code of a project early on via an automated inspection [76]. The results of this automated inspection can be aggregated over time with file, package, or project granularity. While the term ASAT can encompass all tools that use static analysis, e.g., formal program verification tools or model checkers, we focus on a commonly used subset of static analysis tools also known as linters.

This type of ASATs use predefined rule sets of possible problems to match against existing source code and generate a warning for the developer if a rule is matched. Possible problems that can be found by ASATs include violations of code style conventions, variable naming, as well as common coding mistakes or best practice violations. Some ASATs are security or defect focused, some are focused on style guidelines and some implement a general purpose approach with a mix of rules. ASATs are covered in research from different angles, e.g., custom rule changes [6], use in Continuous Integration (CI)-pipelines [75] or resolution of warnings, [10, 40]. However, ASATs also suffer from false positives [8, 27, 70], i.e., warnings about code without any problems. This led to research investigating if warnings can be selected in a way to ignore false positives [16, 32, 34]. However, there is currently no approach from this research in use. While developers believe that ASATs are improving software quality [9, 40], the fact that they have to inspect warnings and the possibility of false positives also leads to the question if the use of ASATs is worth the additional effort. To measure the effectiveness of ASATs, different avenues of investigation present themselves. The most common approaches involve the impact of ASATs on defects as a relevant and measurable external software quality attribute. Previous work investigates multiple facets of the impact on defects: the correlation between defects and static analysis warnings [50], the number of defects that can

be found with ASATs [15, 62, 71], or the impact of ASAT warnings on models that predict defectiveness [37, 44, 52, 54].

However, a common factor for these studies, finding and linking defects, can introduce multiple data validity problems. Defects indicated by warnings need to be validated in order to remove false positives. Previous research has used manual validation by students [71] or the researchers themselves [15, 62]. Defects used by predictive models are found and linked to changes with some version of the Śliwerski Zimmermann Zeller (SZZ) [58] algorithm. The basic idea behind SZZ is that issues from an Issue Tracking System (ITS) are linked to bug fixing commits from a Version Control System (VCS) which are then traced back to the bug inducing change, i.e., the commit that introduced the faulty code that needed fixing. This introduces more data validity problems due to noisy issue types [4, 24], links to bug fixing and fix inducing changes [12, 55], as well as tangled changes, i.e., a change mixing bug fixing and unrelated changes [23, 25, 42].

Current research suffers from a lack of detailed software evolution information regarding ASATs, as well as a lack of manually validated data to provide empirical evidence on their impact regarding defects. We address this limitation as part of the DFG funded project DEFECTS<sup>1</sup> by extending the existing data collection and validation software ecosystem SmartSHARK [66, 69]. By using the resources available to us via the GWDG<sup>2</sup> HPC System, we are able to perform large-scale studies of software evolution and data validation studies involving multiple researchers. SmartSHARK allows a fine-grained investigation of software quality evolution over a large number of study subjects.

Within this thesis, we combine this data to present a multidimensional view of software quality evolution with a focus on PMD<sup>3</sup>, a general purpose ASAT for Java. We use PMD and ASAT interchangeably because PMD provides a close representation of ASATs as a whole due to its broad rule set. Investigating a general purpose ASAT is especially interesting because its rules contain not only known problems, but also best practices that make source code more maintainable. Its use should have an overall positive impact on software quality evolution. Especially long-term effects, e.g., if a file with low warning density over longer time has fewer defects in its lifetime, are interesting for researchers and practitioners alike. In order to investigate a large amount of data and different domains in a reproducible way, we focus on open source projects. To ensure a consistent project quality and management, we gather open source projects under the umbrella of the Apache Software Foundation. Analogous to [13], we discern between internal and external quality. Internal quality concerns internal factors, e.g., the source code or the development process. External quality

---

<sup>1</sup><https://gepris.dfg.de/gepris/projekt/402774445>

<sup>2</sup><https://www.gwdg.de>

<sup>3</sup><https://pmd.github.io>

concerns external factors, e.g., defects that are observed by the user of the software or efficiency problems. Within this thesis, we focus on defects as the most important external quality factor and ASAT warnings as well as static source code metrics as a measure of internal quality.

In summary, our research covers four parts. First, we study how ASAT warnings are evolving, e.g., if they are constantly removed by the developers or not. Second, we investigate whether our general purpose ASAT has an impact on quality. We not only measure warning density and custom rule configuration, but also investigate defect density over the lifetime of our study subjects. Third, we combine features engineered to capture long-term effects from ASAT warning density with static source code metrics within predictive models. This provides us with an estimation of the effectiveness of defect detection via warning density and a comparison with static source code metrics. Fourth, we investigate quality evolution from the perspective of the developer, i.e., metric value changes when developers intend to improve the quality in their projects. We measure changes and change conditions regarding ASAT warnings and static source code metrics.

For every part, we mitigate data validity problems via manual inspection where applicable. We utilize manual inspection of issue types, commit messages, as well as changes on a line by line basis to mitigate tangled changes. This provides our studies with an improved signal-to-noise ratio, improving validity and stability of our results.

The SmartSHARK ecosystem we extended, as well as the DEFECTS project we conducted enabled a large scale data collection of open source data. The data as well as the tooling used to facilitate the data collection and all analysis scripts are open sourced<sup>4</sup>. Our quantitative empirical software engineering research provides results as part of an evidence-based software engineering approach [33] for researchers and practitioners. We publish implementations and data as part of our replication kits for every case study we conduct. This enables practitioners to benefit from our research, either directly by accessing our data, or indirectly via compilations of data and insights, e.g., [29] or [73].

We believe that our contributions foster future research and provide important insights into different aspects of software engineering and software quality evolution.

## 1.1 Goals

In this thesis, we combine multiple large studies investigating different aspects of software quality evolution with a focus on ASATs. We answer four main research questions that are explored in detail in different publications. The publications that answer the research questions are enabled by large studies which also investigate data validity concerns. Our research questions are:

---

<sup>4</sup><https://github.com/smartshark/>, <https://github.com/atraitsch>

**RQ1** How are ASAT warnings evolving over time?

This research question aims to shed some light into the usage of ASATs in open source software projects. We want to evaluate if ASAT warnings are resolved, how they are distributed and, in a nutshell, if code is improved over time with regard to static analysis warnings.

**RQ2** What is the impact of ASATs on internal software quality?

Instead of examining the trends of ASAT warnings independent of the use of an ASAT in the build process as in our first question, we now include exact build information to determine the inclusion of ASATs. In this question, we investigate the impact of ASAT use on ASAT warning trends, as well as the influence of existing custom rule configurations.

**RQ3** What is the impact of ASATs on external software quality?

In this question, we examine the impact of ASAT warnings on software quality under multiple experimentation strategies common in software engineering research. More specifically, we investigate predictive models that include ASAT warnings and ASAT warning derived features to predict defective changes, i.e., the impact on external quality. In addition, we investigate whether files that contain fewer ASAT warnings also contain fewer defects in a statistical comparison.

**RQ4** What is the developers' perspective on software quality?

To answer this research question, we conduct a confirmatory and an exploratory study. Within the confirmatory study, we examine the differences between changes developers perceive as quality improving and other changes, e.g., feature additions. Within the exploratory study, we investigate the quality of the files with respect to static software metrics and ASAT warnings before a quality improving change is applied by a developer.

Our first two research questions and a part of the third research question is covered by our first study. The longitudinal, retrospective study of ASAT warnings and their effects on software quality analyzes 54 open source projects of the Apache Software Foundation [64] (Appendix A). It includes detailed investigations of trends over time, impact of configuration changes and defect density on a per-year basis in relation to ASAT usage.

The third research question of this thesis is explored within multiple studies. In our first study [64] (Appendix A), we investigate a broad overview of defect density per year and whether ASAT use has an influence on it. In addition, we explore predictive models in a case study [65] (Appendix C) utilizing manually validated data for issue types, collected for a study investigating the impact of SZZ [20] (Appendix B). To complete our results for this research question, we conduct a smaller study

investigating whether files that contain more ASAT warning are also inducing more defects [68] (Appendix H). To increase the validity of the study, we include only manually labeled bug fixing changes from a large tangling study [21] (Appendix F).

Our fourth research question is covered by a study which combines confirmatory and exploratory research while investigating the perspective of the developers themselves on software quality evolution [67] (Appendix G). It uses the same 54 open source projects as the first study. After manually classifying changes into three categories by two researchers, we use this data as ground truth to fine-tune a deep-learning model specialized for Natural Language Processing (NLP) tasks in the software engineering domain [72] (Appendix E). The model then classifies the rest of the available data for all study subjects in this study.

## 1.2 Contributions

This thesis extends the body of knowledge in the field of software quality evolution with a focus on ASATs. To this end, it contributes the following:

- A longitudinal empirical study of the evolution of static analysis warnings [64] (Appendix A). We show how static analysis warnings evolve in our study subjects and how the use of ASATs and their configuration influences the evolution. The study contains data from 54 open source projects between 2001-2017. It provides a broad, long-term, evolutionary perspective on ASAT warnings in open source projects. To the best of our knowledge, this is the first comprehensive, commit level study of ASAT warning evolution. Consequently, this study supplies empirical evidence for researchers and practitioners alike. For practitioners, we provide information about what they can expect when introducing an ASAT as well as evidence for a positive impact of ASATs. For researchers, we show that the correlation between ASAT warnings and Logical Lines of Code (LLOC) as well as long-term trends should be taken in to account for analyses regarding static analysis warnings. This study provides the foundation of our more detailed investigations on the impact of ASATs on defects in open source projects.
- An implementation and a large scale study which investigates the impact of static analysis warnings and static source code metrics on external quality via a ranking of predictive models [65] (Appendix C). The study includes three novel features based on static analysis warnings designed to capture their evolution. In addition, it explores two SZZ [58] labeling approaches in the context of fine-grained just-in-time defect prediction [47]. Moreover, the study explores the cost saving potential of the implemented approach. This study was enabled by our prior investigation of data validity issues [20] (Appendix B) as we use

the manually validated data and insights as part of the published approach. We improve the state-of-the-art of fine-grained just-in-time defect prediction models by showing how additional features influence the prediction performance. We are able to improve the predictive performance significantly by including ASAT-warning-based features and static source code metrics in addition to process metrics that are common in just-in-time defect prediction models. We find that, while static source code metrics improve the models significantly the effect of ASAT-warning-based features is not significant. In addition, we show how the chosen SZZ variant impacts the prediction performance, which is important for future research on just-in-time defect prediction.

- A large scale manual classification study [67] (Appendix G) which provides ground truth data for developer intents to improve software quality. The study also provides a detailed investigation into static source code metric value and ASAT warning changes when a developer intends to improve software quality. The ground truth is used to fine-tune a pre-trained deep-learning model. The study shows that the model provides comparable performance to the state-of-the-art and that it can be used to classify large amounts of data. The results of this study have implications for researchers as well as for practitioners. We can show which measurements are commonly associated with increased quality from the point of view of the developers. Consequently, the data gained from this study can help to improve software quality estimation models as well as ASATs themselves. To the best of our knowledge, this is the first study that investigates ASAT warning and static source code metric changes on commit level granularity on this scale. This study was enabled by an extensive validation of the pre-trained deep-learning model [72] (Appendix E) which emerged as an idea from our investigation of automated issue classification approaches [19] (Appendix D).
- An implementation and a case study that builds upon and extends our earlier work dedicated to investigate whether parts of software that contain more static analysis warnings in comparison to the rest of the project are responsible for more defects found at a later stage [68] (Appendix H). This study gathers more evidence for a common assumption in software engineering: static analysis tools provide a net benefit to software quality. It is a more targeted approach than our first study [64] (Appendix A) and utilizes fine-grained just-in-time defect prediction methods developed for our earlier study on static source code metrics and static analysis warnings in predictive models [65] (Appendix C). As with our previous studies, we measure defects as the most important software quality attribute and relate them to warning density. To increase the data validity we incorporate the insights and data gathered in our large tangling study [22]

(Appendix F). This study empirically shows that there is a measurable, positive impact of ASAT use in open source projects. However, the magnitude of the effect is surprisingly small.

- An approach and an implementation for traversing a git commit graph which yields improved just-in-time defect prediction data. Due to the nature of metric aggregation in a fine-grained just-in-time context, simply ordering the commits by date is not sufficient. Our approach considers different branches of the git commit graph to aggregate and track file renaming. This is used in both studies that utilize fine-grained just-in-time data [65, 68] (Appendix C and Appendix H).
- An approach and an implementation for tracing bug fix commits to bug inducing commits, which improves release level defect prediction data collection. Previous studies on release level defect prediction either used a six months window or affected releases data from the ITS. Both have drawbacks in precision or availability. Our approach provides detailed tracing by considering the commit graph directly. This approach was used to gather detailed release-level defect prediction data for the impact assessment in defect-based analyses [20] (Appendix B).

## 1.3 Impact

The approaches, empirical data, and analyses performed within the time preparing this thesis are published in five scientific journal articles and six peer reviewed conference proceedings. Moreover, we list two more articles that are in the submission process to a scientific journal. In addition, we presented three of the journal articles at the International Conference on Software Engineering (ICSE).

### Journal articles

- A. Trautsch, S. Herbold, J. Grabowski “A Longitudinal Study of Static Analysis Warning Evolution and the Effects of PMD on Software Quality in Apache Open Source Projects” in *Empirical Software Engineering*, 2020. Presented in the journal first track of ICSE 2021.

#### Own Contributions:

I am the lead author of this publication. I designed the case study and performed the analysis of the warning evolution.

- S. Herbold\*, A. Trautsch\*, F. Trautsch\*, B. Ledel “Problems with SZZ and Features: An empirical study of the state of practice of defect prediction data collection” in *Empirical Software Engineering*, 2021

**Own Contributions:**

\* Equally contributing authors. I contributed to the study design and provided a large part of the literature research regarding defect prediction data sets. I provided the implementation for the release-level defect prediction data collection. I provided a large part of the implementation for the manual validation of issue types and issue links. I also took part in the manual issue classification as part of the manual data validations.

- S. Herbold, A. Trautsch, F. Trautsch “On the Feasibility of Automated Prediction of Bug and Non-Bug Issues” in *Empirical Software Engineering*, 2020. Presented in the journal first track of ICSE 2021.

**Own Contributions:**

I provided the initial idea and proposal of the article. I contributed to the implementation of the comparison framework as well as the study design. I contributed to the literature research.

- S. Herbold, A. Trautsch, B. Ledel, A. Aghamohammadi, T. Ahmed Ghaleb, K. Kaur Chahal, T. Bossenmaier, B. Nagaria, P. Makedonski, M. Nili Ahmadabadi, K. Szabados, H. Spieker, M. Madeja, N. Hoy, V. Lenarduzzi, S. Wang, G. Rodriguez Perez, R. Colomo-Palacios, R. Verdecchia, P. Singh, Y. Qin, D. Chakroborti, W. Davis, V. Walunj, H. Wu, D. Marcilio, O. Alam, A. Aldaej, I. Amit, B. Turhan, S. Eismann, A. Wickert, I. Malavolta, M. Sulír, F. Fard, A. Z Henley, S. Kourtzanidis, E. Tüzün, C. Treude, S. Maleki Shamasbi, I. Pashchenko, M. Wyrich, J. C. Davis, A. Serebrenik, E. Albrecht, E. Utku Aktas, D. Strüber, J. Erbel “Large-Scale Manual Validation of Bug Fixing Commits: A Fine-grained Analysis of Tangling” in *Empirical Software Engineering*, 2021. Presented in the journal first track at ICSE 2022.

**Own Contributions:**

I contributed to the experiment design, manual validation as well as implementations for the tools necessary to conduct the study.

- J. von der Mosel, A. Trautsch, S. Herbold “On the validity of pre-trained transformers for natural language processing in the software engineering domain” in *Transactions on Software Engineering*, 2022

**Own Contributions:**

I provided the initial idea for the article. I contributed the task specific fine-tuning of the deep-learning model. I also contributed the raw data for the evaluation of the fine-tuning for binary and multi-class cases. I conducted all fine-tuning experiments to evaluate the feasibility of the model.

### Journal articles in submission

- A. Trautsch, J. Erbel, S. Herbold, J. Grabowski “What really changes when developers intend to improve their source code: A commit-level study of static metric value and static analysis warning changes” submitted to *Empirical Software Engineering*

**Own Contributions:**

I am the lead author of this publication. I designed the experiments, performed half of the manual validation, and most of the writing.

- A. Trautsch, S. Herbold, J. Grabowski “Are automated static analysis tools worth it? An investigation into relative warning density and external software quality” currently in a major revision with *Empirical Software Engineering*

**Own Contributions:**

I am the lead author of this publication. I designed the experiments and performed most of the writing.

### Conference articles

- A. Trautsch “Effects of Automated Static Analysis Tools: A Multidimensional View on Quality Evolution” in *Proceedings of the 41st International Conference on Software Engineering (ICSE): Companion Proceedings*, 2019

**Own Contributions:**

I am the single author of this publication and performed all work myself.

- A. Trautsch, F. Trautsch, S. Herbold, B. Ledel, J. Grabowski “The SmartSHARK Ecosystem for Software Repository Mining” in *Proceedings of the 42nd International Conference on Software Engineering (ICSE): Demos*, 2020

**Own Contributions:**

I provided the implementation of the VisualSHARK a state-of-the art web frontend that facilitates manual validation of data for the SmartSHARK database. I also provided an extension of the ServerSHARK as well as several SmartSHARK plugins.

- S. Herbold, A. Trautsch, B. Ledel “Large-Scale Manual Validation of Bugfixing Changes” in *Proceedings of the 17th International Conference on Mining Software Repositories (MSR)*, 2020

**Own Contributions:**

I provided the extension of the VisualSHARK as well as the extension of the ServerSHARK database as needed for the experiment described in this paper. I contributed to the idea and the experiment design.

- A. Trautsch, S. Herbold, J. Grabowski “Static source code metrics and static analysis warnings for fine-grained just-in-time defect prediction” in *Proceedings of the 36th International Conference on Software Maintenance and Evolution (ICSME)*, 2020. **Awarded with ICSME Distinguished Artifact Award.**

**Own Contributions:**

I am the lead author of this publication. I performed all reported experiments myself. The cost model and the implementation of the rank comparison were contributed by S. Herbold.

- J. Erbel, A. Trautsch, J. Grabowski “Simulating live cloud adaptations prior to a production deployment using a models at runtime approach” in *Proceedings of the 11th International Conference on Simulation and Modeling Methodologies, Technologies and Applications (SIMULTECH)*, 2021. **Awarded with SIMULTECH best poster award.**

**Own Contributions:**

I contributed to the discussion of the simulation environment, the implementation of the replication kit, and the execution of the case study.

- A. Trautsch, S. Herbold “Predicting issue types with seBERT” in *Proceedings of the 1st International Workshop on Natural Language-based Software Engineering (NLBSE)*, 2022.

**Own Contributions:**

I performed the experiment and provided the results for the study.

### Academic services

Over the course of my studies, I performed reviews for the following journals:

- Empirical Software Engineering
- Software Quality Journal
- Information and Software Technology
- Journal of Systems and Software
- Transactions on Neural Networks and Learning Systems

I was also involved in the following conferences:

- Mining challenge Co-Chair for MSR’2022

## 1.4 Structure of the thesis

This cumulative thesis consists of eight publications that are contained in the appendix. Four of these publications answer our research questions, the others provide the

data and foundations needed to answer the research questions. Within this thesis, we provide a high level overview regarding the approaches and results while a detailed description can be found in the publications in the appendix. We discuss all publications within the context of this thesis, describe how the publications build upon each other and provide summaries for all of them. Moreover, we discuss the results of our research questions and how they fit into the context of the related work within this thesis.

The outline of this thesis is as follows.

**Section 2** describes the relationship between our publications (Section 2.1). Moreover, it contains summaries for all publications that either directly answer our research questions or are required as a prerequisite (Section 2.2).

**Section 3** contains the discussion of the results of this thesis and sets them into the context of related work.

**Section 4** concludes the thesis with a summary (Section 4.1) and gives an outlook on future work (Section 4.2).

**Appendix A - Appendix H** contain the publications that are part of this thesis.





which show that test code is different from production code regarding ASAT warnings. It also contains the idea to evaluate a relationship between internal quality in the form of static analysis warnings and external quality in the form of defects.

We expand on this idea by investigating trends of static analysis warnings in a longitudinal study [64] (Appendix A). This is a large scale study of warning density and, as far as we know, the first study with a focus on warning density evolution for a general purpose ASAT. Moreover, it includes a novel investigation of the impact of PMD on warning density and defect density as well as the correlation between custom rules and warning density. We complement previous research by Zampetti et al. [75] by not only investigating rule changes but also measuring the warnings for each commit. Moreover, we complement previous research by Penta et al. [49] and Aloraini et al. [3] by investigating warning density in commit level granularity for a general purpose ASAT.

To further investigate ASAT warnings and warning density effects on defects, we decided to extend a state-of-the-art just-in-time defect prediction approach by Pascarella et al. [47]. In their approach, the authors use just-in-time defect prediction metrics with a file level granularity to predict whether the file in a change introduces a defect. Extending the approach by Pascarella et al. [47] allows us to leverage predictive models from defect prediction research to not only explore how ASAT features improve said models but also how they compare to static source code metric features as well as traditional just-in-time defect prediction features. This yielded our publication about fine-grained just-in-time defect prediction [65] (Appendix C) which also investigates differences between two SZZ [58] variants commonly used to find defect inducing changes. Our publication uses validated issue type data from a previous investigation into validity problems in defect-based data [20] (Appendix B). The inclusion of this data provides this study with unprecedented data validity regarding manually validated issue types. As far as we know, it is also the largest fine-grained defect prediction study as well as the first study which compares two SZZ variants and the impact of different types of features within the context of fine-grained just-in-time defect prediction. Our initial work on data validity [20] (Appendix B) provides the data for this study and led to an exploration of possible automatic classification approaches for issue types as one aspect of data validity in [19] (Appendix D). This work on issue type classification led us to an investigation of more powerful models for NLP that can be used in issue type classification and other use cases in software engineering [72] (Appendix E).

Another study that directly results from our longitudinal study in [64] (Appendix A) investigates what the developers deem as quality improving. In [67] (Appendix G) we investigate differences in software changes that developers describe as quality improving and all other changes. We manually classify 2,533 changes into internal and external quality improvements and use this data as ground truth to fine-tune a pre-trained deep-learning model for NLP, which we introduced in [72] (Appendix E).

The manual classification is, according to the overview provided by AlOmar et al. [2] the largest of this kind. To the best of our knowledge, this study is the first that compares all changes within a study subject by including all available commits with static source code metrics and ASAT warnings.

Our investigation of features for predictive models in [65] (Appendix C) revealed that ASAT-based features do not improve the prediction whether a defect is introduced significantly. However, using feature sets for predictive models and ranking them may be too indirect. Therefore, in our final study contained in this thesis [68] (Appendix H) we investigate whether files which contain more static analysis warnings also induce more defects. To answer this question, we further refine our warning-density-based measurements and include more validated data via our large-scale study of tangled changes [21] (Appendix F) which was registered in [18]. The tangling study in turn, was only made possible by the extension of the SmartSHARK database and extension of the VisualSHARK for manual validation of changes on a line level which is summarized in our SmartSHARK demo in [66].

## 2.2 Summaries

In this section, we summarize each publication included within this thesis.

### A) Study of ASAT warning evolution

A. Trautsch, S. Herbold, J. Grabowski “A Longitudinal Study of Static Analysis Warning Evolution and the Effects of PMD on Software Quality in Apache Open Source Projects” in *Empirical Software Engineering*, 2020

#### Summary

In our longitudinal study, we explore how ASAT warnings are evolving as well as whether and how custom configurations influence this evolution. We also investigate which types of warnings are resolved and how the use of an ASAT correlates with defect density in yearly time steps. To facilitate this study, we use the SmartSHARK ecosystem to extract the history of 54 Java open source projects for up to 17 years. We use the commit graph as well as the build configuration to extract information about the history and active rule configurations for ASATs. Our study shows that the number of ASAT warnings and the number of LLOC are correlated. As the source code grows, so does the number of ASAT warnings. However, we also found a positive trend regarding ASAT warning density, i.e., declining warning density. On average, each study subject resolves 3.5 ASAT warnings per 1000 LLOC per year which means software quality is increasing in our study subjects. The impact of the

build configuration regarding ASAT usage and ASAT rule configuration is negligible. However, we are able to measure a small, positive impact of PMD on defect density.

## **B) Data validity issues in defect-based analyses**

S. Herbold, A. Trautsch, F. Trautsch, B. Ledel “Problems with SZZ and Features: An empirical study of the state of practice of defect prediction data collection” in *Empirical Software Engineering*, 2022

### **Summary**

Defect prediction is a common approach in software engineering research to explore relations between software quality and other aspects of software engineering. However, defect prediction research suffers from data validity problems in multiple data sources. To enable us to investigate more direct relations of ASATs and defects in software we first need to evaluate possible problems with existing or newly created data sets. Within this study, we explore multiple problems with the common usage of defect information as well as features and data sets in the domain of defect prediction. Before we are able to explore the impact, we perform multiple manual data validations which is one of the core contributions of this paper. After manual verification of issue types and issue links to commits, a release-level defect prediction experiment is performed to evaluate the impact of misclassified data on the defect prediction models. The results of our study confirm that data validity is a continuing problem in defect-based analyses. We show that only half of the bug fixing commits, found via the commonly used basic SZZ algorithm, are actually bug fixing. Moreover, we found that in comparison to incorrect defect labels smaller feature sets are not a big threat to validity for defect-based analyses.

## **C) Improving predictive models with ASAT information**

A. Trautsch, S. Herbold, J. Grabowski, “Static source code metrics and static analysis warnings for fine-grained just-in-time defect prediction” in *proceedings of 36th International Conference on Software Maintenance and Evolution (ICSME)*, 2020

### **Summary**

Within this publication, we use the manually validated data from [20] (Appendix B) to explore a second defect prediction method common in software engineering research, just-in-time defect prediction. Because common just-in-time defect prediction only works with changes across files, we adopt a recently introduced specialization called fine-grained just-in-time defect prediction, which utilizes files within changes as this better matches our use case of evaluating ASAT warnings together with other static

source code metrics within files. To the best of our knowledge, this study is the first that includes ASAT warnings and static source code metrics for fine-grained just-in-time defect prediction models. It also introduced three novel features based on ASAT warning density and evaluated two common labeling strategies from the area of defect prediction research regarding their impact on prediction performance. We find that a combination of static source code metrics and ASAT-warning-density-based features perform better than previous approaches, which only used change and process-based metrics. In addition, this combination is also improving the cost saving potential of just-in-time defect prediction models.

#### **D) Improving data validity via automatic issue classification**

S. Herbold, A. Trautsch, F. Trautsch “On the Feasibility of Automated Prediction of Bug and Non-Bug Issues” in *Empirical Software Engineering*, 2022

##### **Summary**

Misclassification of issues in issue tracking systems is a problem [4, 24]. Especially defect-based analyses rely on correctly classified issues as the misclassification of bugs and feature requests add noise to the available data in this type of study. We also confirmed this in [20] (Appendix B). Within this study, we use the data from [20] to create a benchmark data set that is subsequently used to evaluate state-of-the-art approaches of issue type classification in order to investigate whether automated classification is feasible to reduce this data validity problem. In addition, we explore whether manually specified rules improve classification performance and whether a large amount of data, even if it is not validated, can also yield acceptable classification performance. We find that manually specified rules are not able to significantly improve the model performance. Moreover, we find that while data that is not validated can yield performance comparable to existing developer classifications, a focus on minimizing false positives is only achievable with manually validated data.

#### **E) An NLP model for software engineering data**

J. von der Mosel, A. Trautsch, S. Herbold “On the validity of pre-trained transformers for natural language processing in the software engineering domain” in *Transactions on Software Engineering*, 2022

##### **Summary**

After the success of a deep-learning-based NLP approach in our previous investigation of automatic issue type classification [19] (Appendix D) we set out to explore the current state-of-the-art in deep-learning-based NLP: transformers. Instead of only

issue type prediction we extend the evaluation with two more use cases: commit intent classification and sentiment analysis. In this study, we pre-train a transformer model on text from software engineering sources. We perform an evaluation together with general purpose transformer models and one other transformer model pre-trained on a limited subset of software engineering data. We show that models that are pre-trained on software engineering data outperform general purpose models for tasks within the domain of software engineering, e.g., automatic issue type classification or commit intent classification.

## F) Analysis of tangled changes

S. Herbold, A. Trautsch, B. Ledel, A. Aghamohammadi, T. Ahmed Ghaleb, K. Kaur Chahal, T. Bossenmaier, B. Nagaria, P. Makedonski, M. Nili Ahmadabadi, K. Szabados, H. Spieker, M. Madeja, N. Hoy, V. Lenarduzzi, S. Wang, G. Rodriguez Perez, R. Colomo-Palacios, R. Verdecchia, P. Singh, Y. Qin, D. Chakroborti, W. Davis, V. Walunj, H. Wu, D. Marcilio, O. Alam, A. Aldaej, I. Amit, B. Turhan, S. Eismann, A. Wickert, I. Malavolta, M. Sulír, F. Fard, A. Z Henley, S. Kourtzanidis, E. Tüzün, C. Treude, S. Maleki Shamasbi, I. Pashchenko, M. Wyrich, J. C. Davis, A. Serebrenik, E. Albrecht, E. Utku Aktas, D. Strüber, J. Erbel “Large-Scale Manual Validation of Bug Fixing Commits: A Fine-grained Analysis of Tangling” in *Empirical Software Engineering*, 2022

## Summary

Tangled changes contain multiple different change types, e.g., a bug fix as well as an addition of a feature or documentation changes. This is a common problem in software engineering research, especially all research areas which use bugs or bug fixing changes. Within this large study, which was one of the first pre-registered studies at the MSR [18], we manually validate bug fixing changes on a line-by-line basis. Each line is classified as belonging to the bug fix or as a tangled change in the form of refactoring, documentation, test, or unrelated change. The whole study is conceptualized as a crowdsourcing experiment for researchers. Qualified researchers participate in the manual labeling and after labeling a predefined amount of changes receive co-authorship. This setup yields a large amount of manually validated data which is also one of the contributions of this study. In our study, we find that while some tangling can be identified by current heuristics, more complex tangling can lead to up to 47% of noise without manual validation.

### **G) Changes in quality increasing commits**

A. Trautsch, J. Erbel, S. Herbold, J. Grabowski “What really changes when developers intend to improve their source code: A commit-level study of static metric value and static analysis warning changes”, preprint, submitted to Empirical Software Engineering

#### **Summary**

Previous work on defect prediction depends on static source code metrics as features to train models that predict probabilities of defects occurring in files. In our own work, we find that static source code metrics can increase predictive power of fine-grained just-in-time defect prediction models [65] (Appendix C). However, recent work investigating program understanding of developers found that common complexity metrics are perceived very differently by developers [48] and do not necessarily impact program understanding [57]. This hints at a different relationship between complexity metrics, developer perception of complexity and defects than indicated in previous work on defect prediction models. Therefore, we investigate which static software metrics and static analysis warnings change in which way when developers intend to increase the quality of the source code. The basis of this publication is a random sample of manually labeled commit messages which the first and second author classified as quality increasing. Both researchers classified the commit message, with the help of predefined guidelines which are validated against results from a different manual classification study. The manual classifications are then used as ground truth to fine-tune a state-of-the-art deep-learning model [72] (Appendix E) that classifies the rest of the data. In contrast to previous studies, we not only look at changes that increase quality but additionally compare against all other changes to see if there is a relevant difference. We are able to show that size and complexity metrics change significantly in quality increasing commits, however, not always in the expected direction, e.g., complexity is not decreased in bug fixing changes.

### **H) ASAT warning density in defect inducing changes**

A. Trautsch, S. Herbold, J. Grabowski “Are automated static analysis tools worth it? An investigation into relative warning density and external software quality”, preprint, in major revision to Empirical Software Engineering

#### **Summary**

In previous work, we performed a coarse-grained investigation of defects and static analysis warnings [64] (Appendix A). This limitation is addressed in this article. We use tools from previous defect prediction research [65] (Appendix C), manually

---

annotated lines that indicate which part of the change is part of the bug fix [21] (Appendix F) and combine it with a current PMD release to extract static analysis warnings and warning-density-based features. Instead of relying on predictive models, we focus on different warning densities between files within changes that introduce bugs and all other changes. We find that bug inducing files do not contain more static analysis warnings than the rest of the project. However, if we measure the differences between bug inducing changes and all other changes we can measure a statistically significant difference on a popular subset of warning rules, but the effect size is surprisingly negligible.

## 3 Discussion

In this chapter, we discuss the results of this thesis in the context of current research and best practices in software development.

### 3.1 ASAT warning evolution

Our first results stem from a longitudinal study [64] (Appendix A) of 54 Java open source projects between 2001-2017. We analyzed static analysis warning trends from PMD via warning density as well as defect density on a per-year basis. The study incorporated different warning aggregations and custom configurations of PMD that the study subjects employed as part of their build system. The trends of ASAT warnings that are analyzed in this study answer **RQ1: How are ASAT warnings evolving over time?**

Our longitudinal study [64] (Appendix A) revealed that static analysis warnings and logical lines of code are correlated. This means that as software grows, so does the number of static analysis warnings. Software growth is inevitable for a system that performs real world activity and needs to continuously adapt to new requirements and circumstances according to Lehman [36]. That the static analysis warnings are also increasing could be a sign of developers not being aware of the warnings or ignoring parts of the warnings as false positives. This increase in static analysis warnings is also found in the replication kit of the study by Marcilio et al. [40] although not reported in the article. Other researchers also found that only few warnings are resolved [32, 38, 40].

While the number of static analysis warnings is increasing, our study also found that the warning density is decreasing. If we think of code with fewer static analysis warnings per lines of code as better quality we can show that code quality increased in our study subjects. If we aggregate all data from our study subjects, we find that each subject in our study removes 3.5 ASAT warnings per kLLOC per year on average. This is a positive result for software development in general as it shows that code quality improves over the period we investigate. However, this result is in contrast to previous work using warning density measurements. Aloraini et al. [3] as well as Penta et al. [49] found that warning density remained constant in their study subjects. Although, they focused on security ASATs while in our study we focus on a general purpose ASAT. This difference could be explained by security warnings being harder to fix for developers. Our results could also be influenced by the improvement

of Java coding style or best practices in our study subjects over the investigated time periods. Not all categories of ASAT warnings are behaving the same in our study. We find that naming, brace, and design warning categories measured as density are decreasing the most. This result is also in line with previous work by Beller et al. [6]. Our work complements the work by Beller et al. by including the warning trends themselves instead of only configuration changes for the different categories. We also expand on the work by Zampetti et al. [75]. Zampetti et al. showed that CI builds break because of coding standard violations. We can show that those types of warnings are also resolved the most.

### 3.2 Impact on internal software quality

Up to now, we only investigated ASAT warning trends independent of the use of the ASAT in our study subjects. In the second part, we include the build information of our study subjects so that we can determine the point in time when an ASAT was included or removed from the project build. This knowledge is used to answer **RQ2: What is the impact of ASATs on internal software quality?**

First of all, most of the projects have a positive trend in warning density in the first year after introducing PMD as an ASAT. The number of projects with a positive warning density trend increases for subsequent years after the ASAT is introduced. This decrease in warning density confirms a common industry best practice which states that only new code is subject to newly introduced ASATs. The reason for this restriction is that the effort for going over the complete code base can be huge. This effect is also reported in large scale ASAT deployments by Google [56] and Facebook [11]. When investigating the reasons for adding or removing PMD from the build configuration within the changes of the study subjects, we found no singular set of reasons. The addition of PMD happened either when re-configuring the build system without a mention or consciously by including the reporting PMD provides within the build configuration and mentioning it by name. Removal of PMD happened consciously, with a mention of PMD and a reason in the commit message three times, although the reasons were different. While half of the projects that are using PMD as part of their build configuration changed their rules at least once, we were not able to measure a correlation between an evolving rule configuration and warning density. A low number of changed configuration files for ASATs was also previously reported by Beller et al. [6]. We confirm this finding and expand on it in our study with warning density measurements. We show that no direct correlation between warning density and configuration changes exists.

If we divide the years of development between PMD and non-PMD years, we can find no difference in warning density between both. This means that we were not able to measure an impact of PMD use in the build configuration and warning density

trends. However, if we use the plain sum of the warnings, we find a statistically significant difference between the groups. The inability to detect a difference for warning density trends may be a result of a flattening warning curve while the LLOC are still increasing. This would mean that the LLOC dominate the warning density term. While this is not a problem if we want to compare study subjects via their defect density it has to be considered by researchers, especially when analyzing long time periods. More targeted analysis, i.e., restricted LLOC, are not problematic, e.g., as reflected in the work by Panichella et al. [45].

### 3.3 Impact on external software quality

A more direct comparison to previous work that investigated bug inducing changes is done in the course of answering **RQ3: What is the impact of ASATs on external software quality?**

Our first study (Appendix A) finds, that years in which PMD is included in the build configuration have lower defect density. In addition to the comparison of the years with PMD and without PMD we built a regularized linear regression model including confounding factors, e.g., project size, number of authors and popularity metrics. While this approach is necessarily limited by the data, it serves to show that there might be a measurable, positive effect of ASATs. This is in line with research investigating ASAT warnings and their correlation with defects [50] as well as the impact of warnings on predictive models for bug inducing changes [37, 52, 53]. In our fine-grained just-in-time defect prediction [65] (Appendix C) study, we conduct a defect prediction experiment to widen our view for this research question. The study uses validated issue type labels from [20] (Appendix B). While this restricts the available number of study subjects, it still is, to the best of our knowledge, the biggest study on fine-grained just-in-time defect prediction. In the study, we examine the results for two variants of SZZ [58] that are commonly used. An ad-hoc SZZ method, as used in, e.g., [7, 35, 47] and an ITS-based method, as used in, e.g., [31, 41, 61]. We introduce new warning-density-based metrics, as well as static source code metrics together with common just-in-time metrics as features for predictive models. We analyze the importance of the features for a regularized linear model and a non-linear random forest model. Moreover, we perform an evaluation of the predictive performance of both models with different sets of features. In addition, we investigate the potential for cost saving when the defect prediction model is applied [17].

Our study shows that warning-density-based metrics are among the ten most important features in both models for the ITS SZZ approach and in one model for the ad-hoc SZZ approach. This highlights that warning-density-based features are able to improve predictive models. To further evaluate this, we perform a just-in-time defect prediction experiment. The experiment showed that the predictive models

which include all available features outperform all other feature sets. Models only using the feature set containing the ASAT warnings were not performing as well in the defect prediction task. However, this performance is in line with investigations by Querel and Rigby [52] as well as the recent study by the same authors [53] who also found that static analysis warnings have an impact, although it is minimal.

One result of this study is that static source code metrics as features performed consistently better than only ASAT-warning-based features for the predictive models. This has implications for further research regarding just-in-time as well as fine-grained just-in-time defect prediction. Researchers as well as practitioners that use just-in-time based approaches to estimate the risk of changes need to be aware that they can improve the performance of their models by including static source code metrics and static analysis warnings in addition to traditional just-in-time metrics.

All of our previous studies focused on ASAT warning evolution, predictive models and the changes in quality increasing commits. In [68] (Appendix H) we study whether files that contain defects also contain more static analysis warnings. To investigate this, we perform an exploratory study on previously defined warning density metrics, which take the state of the project at the time of each change into account. Within the exploratory study, we use manually validated data from our tangling study [21] (Appendix F) with an extended version of our fine-grained just-in-time defect prediction tooling from [65] (Appendix C). While this decreases the number of study subjects, it increases the signal-to-noise ratio. Moreover, this study incorporates a more recent version of PMD which results in about 300 types of warnings to analyze instead of 200 for the previous version.

We find that bug inducing files contain fewer static analysis warnings than the rest of the project at the time of the change. However, this is due to a decreasing warning density which is also visible in other changes, i.e., the files that are changed usually have a reduced warning density in comparison to the rest of the project. This is an effect of gradually declining warning density which, we already saw in our first study (Appendix A). If we compare all bug inducing changes to all other changes, we find that bug inducing changes indeed contain more static analysis warnings than other changes. Although we note that the difference, while statistically significant, is surprisingly small. This hints at the ability for ASAT-warning-based features to improve predictive models which is, together with the small effect size, confirmed by other researchers [53]. Our study also confirmed a common assumption in static analysis usage in software engineering. Just enabling all rules without taking the project and context into account does yield worse performance than falling back on a simpler default set of metrics. We quantitatively show that the popular default set for the Maven-PMD plugin yields a difference in warning density metrics between bug inducing changes and all other changes. We also find that bug inducing changes increase the warning density of the file in comparison to the rest of the system.

### 3.4 Internal and external quality from the developers' perspective

Another important viewpoint on software quality evolution is the perspective from the developers. We explore this further in answering **RQ4: What is the developers' perspective on software quality?**

In our study on the differences between quality increasing changes and other changes [67] (Appendix G) we perform a confirmatory and exploratory study into the differences between changes based on the commit messages the developers write. We first manually classify commits into three maintenance categories after Swanson [60], perfective, corrective and other. This data is used as ground truth to fine-tune a pre-trained deep-learning model for NLP tasks introduced in a previous publication [72] (Appendix E). The model is evaluated first and, as the performance is sufficient, used to classify the rest of the data. This provides us with an unprecedented amount of data for this kind of task.

In the confirmatory part of our study, we confirm previous research on the differences in change size between quality improving and other changes. We confirm previous studies which report that perfective maintenance changes are usually smaller, e.g., [1, 43, 51]. We further confirm that corrective changes are smaller and perfective maintenance removes more source code [43, 51]. The fact that the size of the change is different between the categories is also confirmed and directly used by Hönel et al. [26] which used the size as predictor for the maintenance type.

In addition to the differences in size, we investigated the differences in a set of static source code metrics as well as static analysis warnings. We focus on static source code metrics and static analysis warnings that are part of the ColumbusQM software quality model [5]. We expect these to change during quality increasing maintenance operations due to their use within a software quality model.

For perfective maintenance, the metric values change in the expected direction, e.g., complexity and coupling is reduced. This shows that the developers are in agreement with the quality models regarding the metrics measured. Our results for perfective maintenance also indicate that developers remove complexity in the form of cyclomatic complexity and nesting level of conditional branches. While not a direct contradiction, recent research by Peitek et al. [48] as well as Scalabrino et al. [57] indicate that cyclomatic complexity is not as indicative of code understandability and therefore maintainability as previously assumed. Moreover, developers experience cyclomatic complexity differently [48]. In their study, Peitek et al. [48] showed that in a controlled experiment, developers assigned vastly different complexity ratings to presented code snippets. Our results complement the study by Peitek et al. [48] as we are measuring differences in ongoing open source development in comparison to controlled experiments with code snippets. However, our results indicate that cyclomatic complexity remains important for developers which try to improve the quality of their source code as it is one of the measured metrics whose value changes

the most in developer-perceived quality-increasing commits.

While we expected most of the metric value changes for perfective maintenance, corrective maintenance, i.e., fixing bugs yielded unexpected results. Corrective maintenance does not remove complexity, but instead adds more in most of the cases. While we were not expecting complexity to be reduced for every bug fix, our study compares corrective changes with all other changes which include feature additions and normal day to day work. We expected these metrics to change as they are often part of data sets used in defect prediction research [14, 30, 74]. Our results indicate that researchers from the area of defect prediction can improve their models with these metrics as they change in corrective commits. However, the direction of the metric change, i.e., software getting more complex due to a bug fix is unexpected. Within our study, we quantitatively show that bug fixing is the development activity which adds the most complexity. This result means that practitioners should invest additional time after fixing a bug to clean up, simplify or refactor the fixed code.

The second part of this study investigates which files are the target of either perfective or corrective maintenance. Here, we also see a mix of expected and unexpected results. For perfective maintenance, we would expect large and complex files to be the target. However, this is not the case in our study subjects. On average, the files that are the target of perfective maintenance are already smaller and less complex than the rest. Further study is needed to investigate whether this is a prioritization problem of developer resources or if it is a convenience effect because less complex files are easier to simplify than the complex files. Corrective maintenance activities target files that are large and complex. This is expected and supported by defect prediction research. However, combined with the first part of our study this means that complex files get more complex and simple files get simpler. While the effect sizes are small, the amount of data that we analyze thanks to SmartSHARK and our deep-learning model allows us to uncover these small but significant differences.

Aside from the static source code metrics, static analysis warnings were also changed statistically significantly for perfective and corrective changes, as expected. This complements previous research which indicates that developers think of ASATs as quality improving [8, 9, 53]. While the effect size is small, this is also supported by other researchers [53].

## 4 Conclusion

In this chapter, we conclude the thesis. Section 4.1 summarizes the results and contributions. Section 4.2 provides a short outlook on future work.

### 4.1 Summary

Software quality and the evolution of software quality remain an interesting research topic due to the ubiquitous nature of software. Within this thesis, we adopt and improve several tools that software engineering research provides, to investigate ASATs and software quality evolution. We perform multiple large validation studies to further improve the data we use in our case studies.

We extend the current body of knowledge in software engineering research in multiple ways. We provide a large scale warning-density-based study of ASAT warning evolution of a general purpose ASAT that includes every warning for every commit and defect density [64] (Appendix A). We provide a large and in-depth assessment of data validity considerations for defect prediction research [20] (Appendix B). We conduct a large fine-grained just-in-time case study that includes a comparison of different defect labeling strategies and ranking of feature sets which include static source code metrics and static analysis warnings in addition to traditional just-in-time process metrics [65] (Appendix C). We conduct a large benchmark of issue type classification approaches [19] (Appendix D). We evaluate state-of-the-art deep-learning NLP models in a software engineering context [72] (Appendix E). We conduct a large scale, crowd sourced, manual validation study of tangled changes which yields a large data set of line precision untangled bug fixes [22] (Appendix F). We provide a large study of developer intents that explores what changes when developers intend to improve the quality of their source code [67] (Appendix G). We conduct a large study of warning-density-based metrics and their differences in bug inducing changes when compared to all other changes [68] (Appendix H). All artifacts needed to conduct these studies, data sets as well as analysis scripts are open sourced to provide researchers and practitioners with insights and empirical software engineering data.

From a general software quality evolution standpoint, our fine-grained just-in-time study [65] (Appendix C) revealed that static size and complexity metrics have a higher relevance for defects than warning-density-based metrics. Our case study of developer intents [67] (Appendix G) also reflects this result.

Overall, we have shown that the general purpose ASAT for Java that is used in our case studies has only a limited impact on defects. Our first longitudinal case study [64] (Appendix A) shows this in a coarse-grained way for defect density per year in which an ASAT was used or not used. There, we can see that years in which the ASAT was used had overall lower defect density. The limited detail of the first study was addressed in our final case study [68] (Appendix H). The study reveals that, while we can measure a statistically significant difference in warning-density-based metrics for changes that induce defects and all other changes, the effect size is negligible. This result, together with the low ranks of classifiers which use only static analysis warnings and warning-density-based features in our fine-grained just-in-time defect prediction study [65] (Appendix C), allows us to conclude that the general purpose ASAT for Java has a surprisingly small impact on the presence of defects. We believe, that this result also generalizes to other general purpose ASATs for compiled languages. While ASAT warning density may only have a negligible impact on defects, we found that size and complexity has a more direct impact on defects. However, our study of developer intents [67] (Appendix G) revealed that fixing defects is also the main driver of rising complexity in our case study subjects. This means that, from a practitioner's perspective, simplifications and code clean-up is necessary in most cases after fixing a defect.

The combination of studies included within this thesis provides data validation studies that investigate validation problems and provides solutions in the form of validated data and better approaches. Moreover, we provide an overview of software quality evolution with a focus on ASAT warning density which uses validated data where possible, and provides insights into the evolution of static analysis warnings and how they correlate with defects. Our studies advance empirical software engineering research by providing new empirical data as well as insights for practitioners and researchers.

## 4.2 Outlook

Given the groundwork presented in this thesis, some avenues of further investigation present themselves. The data and approaches presented within this thesis could be adopted to provide information with a different focus. While we used manually validated data to mitigate validity problems to answer our research questions, the same manually validated data could also be used to benchmark approaches for building predictive models with a different focus.

The prediction of false positive ASAT warnings, which could also be combined with a transformer model, is another opportunity for future work. With pre-processing steps analogous to code clone detection techniques, we could train a model on the source code changes contained in the SmartSHARK database to automatically find

common changes. Depending on the preliminary results, we could extend this to automatically find new ASAT rules that could be implemented.

Moreover, due to the large database that accumulated over the course of the DEFECTS project, we can provide benchmark data concerning static analysis and static source code metrics for use as a reference. This data could be used to nudge developers towards resolving unnecessary complexities in their source code by extending ASATs with additional defect data.

Another, more general, avenue of investigation would be to compare our results for Java with other languages, especially interpreted languages, e.g., Python or JavaScript. We assume that ASATs for interpreted languages are more likely to find problems that result in run time errors as the compiler is removed as a corrective step in the development process. Our assumption is, that this would impact the results in favor of ASATs.



## Bibliography

- [1] A. Alali, H. Kagdi, and J. I. Maletic. What's a typical commit? a characterization of open source software repositories. In *2008 16th IEEE International Conference on Program Comprehension*, pages 182–191, 2008. doi: 10.1109/ICPC.2008.24.
- [2] Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, and Ali Ouni. Toward the automatic classification of self-affirmed refactoring. *Journal of Systems and Software*, 171:110821, 2021. ISSN 0164-1212. doi: 10.1016/j.jss.2020.110821. URL <http://www.sciencedirect.com/science/article/pii/S016412122030217X>.
- [3] Bushra Aloraini, Meiyappan Nagappan, Daniel M. German, Shinpei Hayashi, and Yoshiki Higo. An empirical study of security warnings from static application security testing tools. *Journal of Systems and Software*, 158:110427, 2019. ISSN 0164-1212. doi: <https://doi.org/10.1016/j.jss.2019.110427>. URL <http://www.sciencedirect.com/science/article/pii/S0164121219302018>.
- [4] Giuliano Antoniol, Kamel Ayari, Massimiliano Di Penta, Foutse Khomh, and Yann-Gaël Guéhéneuc. Is it a bug or an enhancement?: A text-based approach to classify change requests. In *Proceedings of the 2008 Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds, CASCON '08*, pages 23:304–23:318, New York, NY, USA, 2008. ACM. doi: 10.1145/1463788.1463819. URL <http://doi.acm.org/10.1145/1463788.1463819>.
- [5] T. Bakota, P. Hegedűs, P. Körtvélyesi, R. Ferenc, and T. Gyimóthy. A probabilistic software quality model. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, pages 243–252, Sept 2011. doi: 10.1109/ICSM.2011.6080791.
- [6] M. Beller, R. Bholanath, S. McIntosh, and A. Zaidman. Analyzing the state of static analysis: A large-scale evaluation in open source software. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 470–481, March 2016. doi: 10.1109/SANER.2016.105.
- [7] G. G. Cabral, L. L. Minku, E. Shihab, and S. Mujahid. Class imbalance evolution and verification latency in just-in-time software defect prediction. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 666–676, May 2019. doi: 10.1109/ICSE.2019.00076.

- [8] Maria Christakis and Christian Bird. What developers want and need from program analysis: An empirical study. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016*, pages 332–343, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3845-5. doi: 10.1145/2970276.2970347. URL <http://doi.acm.org/10.1145/2970276.2970347>.
- [9] P. Devanbu, T. Zimmermann, and C. Bird. Belief evidence in empirical software engineering. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 108–119, May 2016. doi: 10.1145/2884781.2884812.
- [10] G. Digkas, M. Lungu, P. Avgeriou, A. Chatzigeorgiou, and A. Ampatzoglou. How do developers fix issues and pay back technical debt in the apache ecosystem? In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 153–163, March 2018. doi: 10.1109/SANER.2018.8330205.
- [11] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O’Hearn. Scaling static analyses at facebook. *Commun. ACM*, 62(8):62–70, July 2019. ISSN 0001-0782. doi: 10.1145/3338112. URL <http://doi.acm.org/10.1145/3338112>.
- [12] Y. Fan, X. Xia, D. Alencar da Costa, D. Lo, A. E. Hassan, and S. Li. The impact of changes mislabeled by szz on just-in-time defect prediction. *IEEE Transactions on Software Engineering*, pages 1–1, 2019. doi: 10.1109/TSE.2019.2929761.
- [13] Norman Fenton and James Bieman. *Software Metrics: A Rigorous and Practical Approach, Third Edition*. CRC Press, Inc., Boca Raton, FL, USA, 3rd edition, 2014. ISBN 1439838224, 9781439838228.
- [14] Rudolf Ferenc, Péter Gyimesi, Gábor Gyimesi, Zoltán Tóth, and Tibor Gyimóthy. An automatically created novel bug dataset and its validation in bug prediction. *Journal of Systems and Software*, 169:110691, 2020. ISSN 0164-1212. doi: <https://doi.org/10.1016/j.jss.2020.110691>. URL <http://www.sciencedirect.com/science/article/pii/S0164121220301436>.
- [15] Andrew Habib and Michael Pradel. How many of all bugs do we find? a study of static bug detectors. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018*, pages 317–328, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5937-5. doi: 10.1145/3238147.3238213. URL <http://doi.acm.org/10.1145/3238147.3238213>.
- [16] S. Heckman and L. Williams. A model building process for identifying actionable static analysis alerts. In *2009 International Conference on Software Testing*

- Verification and Validation*, pages 161–170, April 2009. doi: 10.1109/ICST.2009.45.
- [17] S. Herbold. On the costs and profit of software defect prediction. *IEEE Transactions on Software Engineering*, pages 1–1, 2019.
- [18] Steffen Herbold, Alexander Trautsch, and Benjamin Ledel. Large-Scale Manual Validation of Bugfixing Changes. In *2020 IEEE/ACM 17th International Conference on Mining Software Repositories (MSR 2020)*. ACM, 2020.
- [19] Steffen Herbold, Alexander Trautsch, and Fabian Trautsch. On the Feasibility of Automated Prediction of Bug and Non-Bug Issues. *Empirical Software Engineering*, August 2020. URL <https://arxiv.org/abs/2003.05357>.
- [20] Steffen Herbold, Alexander Trautsch, Fabian Trautsch, and Benjamin Ledel. Problems with szz and features: An empirical study of the state of practice of defect prediction data collection, 2020.
- [21] Steffen Herbold, Alexander Trautsch, Benjamin Ledel, Alireza Aghamohammadi, Taher Ahmed Ghaleb, Kuljit Kaur Chahal, Tim Bossenmaier, Bhaveet Nagaria, Philip Makedonski, Matin Nili Ahmadabadi, Kristof Szabados, Helge Spieker, Matej Madeja, Nathaniel Hoy, Valentina Lenarduzzi, Shangwen Wang, Gema Rodríguez-Pérez, Ricardo Colomo-Palacios, Roberto Verdecchia, Paramvir Singh, Yihao Qin, Debasish Chakroborti, Willard Davis, Vijay Walunj, Hongjun Wu, Diego Marcilio, Omar Alam, Abdullah Aldaej, Idan Amit, Burak Turhan, Simon Eismann, Anna-Katharina Wickert, Ivano Malavolta, Matus Sulir, Fatemeh Fard, Austin Z. Henley, Stratos Kourtzanidis, Eray Tuzun, Christoph Treude, Simin Maleki Shamasbi, Ivan Pashchenko, Marvin Wyrich, James Davis, Alexander Serebrenik, Ella Albrecht, Ethem Utku Aktas, Daniel Strüber, and Johannes Erbel. Large-scale manual validation of bug fixing commits: A fine-grained analysis of tangling, 2021.
- [22] Steffen Herbold, Alexander Trautsch, Benjamin Ledel, Alireza Aghamohammadi, Taher Ahmed Ghaleb, Kuljit Kaur Chahal, Tim Bossenmaier, Bhaveet Nagaria, Philip Makedonski, Matin Nili Ahmadabadi, Kristóf Szabados, Helge Spieker, Matej Madeja, Nathaniel Hoy, Valentina Lenarduzzi, Shangwen Wang, Gema Rodriguez Perez, Ricardo Colomo-Palacios, Roberto Verdecchia, Paramvir Singh, Yihao Qin, Debasish Chakroborti, Willard Davis, Vijay Walunj, Hongjun Wu, Diego Marcilio, Omar Alam, Abdullah Aldaej, Idan Amit, Burak Turhan, Simon Eismann, Anna-Katharina Wickert, Ivano Malavolta, Matúš Sulír, Fatemeh Fard, Austin Z Henley, Stratos Kourtzanidis, Eray Tüzün, Christoph Treude, Simin Maleki Shamasbi, Ivan Pashchenko, Marvin Wyrich, James C. Davis, Alexander Serebrenik, Ella Albrecht, Ethem Utku Aktas, Daniel Strüber, and Johannes

- Erbel. A fine-grained data set and analysis of tangling in bug fixing commits. *Empirical Software Engineering*, October 2021. ISSN 1382-3256.
- [23] Kim Herzig and Andreas Zeller. The impact of tangled code changes. In *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13*, page 121–130. IEEE Press, 2013. ISBN 9781467329361.
- [24] Kim Herzig, Sascha Just, and Andreas Zeller. It's not a bug, it's a feature: How misclassification impacts bug prediction. In *Proceedings of the International Conference on Software Engineering, ICSE '13*, pages 392–401, Piscataway, NJ, USA, 2013. IEEE Press. ISBN 978-1-4673-3076-3. URL <http://dl.acm.org/citation.cfm?id=2486788.2486840>.
- [25] Kim Herzig, Sascha Just, and Andreas Zeller. The impact of tangled code changes on defect prediction models. *Empirical Software Engineering*, pages 1–34, April 2016. URL <https://www.microsoft.com/en-us/research/publication/the-impact-of-tangled-code-changes-on-defect-prediction-models/>.
- [26] S. Hönel, M. Ericsson, W. Löwe, and A. Wingkvist. Importance and aptitude of source code density for commit classification into maintenance activities. In *2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS)*, pages 109–120, 2019. doi: 10.1109/QRS.2019.00027.
- [27] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *Proceedings of the 2013 International Conference on Software Engineering, ICSE '13*, pages 672–681, Piscataway, NJ, USA, 2013. IEEE Press. ISBN 978-1-4673-3076-3. URL <http://dl.acm.org/citation.cfm?id=2486788.2486877>.
- [28] C. Jones. *Software Engineering Best Practices: Lessons from Successful Projects in the Top Companies*. McGraw-Hill Education, 2009. ISBN 9780071621625. URL [https://books.google.de/books?id=CJd\\_\\_8ANvtQC](https://books.google.de/books?id=CJd__8ANvtQC).
- [29] Derek M. Jones. *Evidence-based Software Engineering: based on the publicly available data*. Knowledge Software, Ltd, November 2020. ISBN 978-1-8382913-0-3.
- [30] Marian Jureczko and Lech Madeyski. Towards identifying software project clusters with regard to defect prediction. In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering, PROMISE '10*, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781450304047. doi: 10.1145/1868328.1868342. URL <https://doi.org/10.1145/1868328.1868342>.

- [31] 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*, 39(6):757–773, June 2013. ISSN 2326-3881. doi: 10.1109/TSE.2012.70.
- [32] Sunghun Kim and Michael D. Ernst. Which warnings should i fix first? In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 45–54, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-811-4. doi: 10.1145/1287624.1287633.
- [33] Barbara A. Kitchenham, Tore Dyba, and Magne Jorgensen. Evidence-based software engineering. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 273–281, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2163-0. URL <http://dl.acm.org/citation.cfm?id=998675.999432>.
- [34] Ugur Koc, Parsa Saadatpanah, Jeffrey S. Foster, and Adam A. Porter. Learning a classifier for false positive error reports emitted by static code analysis tools. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 35–42, 2017. ISBN 978-1-4503-5071-6. doi: 10.1145/3088525.3088675.
- [35] Masanari Kondo, Daniel M German, Osamu Mizuno, and Eun-Hye Choi. The impact of context metrics on just-in-time defect prediction. *Empirical Software Engineering*, 25(1):890–939, 2020.
- [36] M. M. Lehman. Laws of software evolution revisited. In *Proceedings of the 5th European Workshop on Software Process Technology*, EWSPPT '96, pages 108–124, Berlin, Heidelberg, 1996. Springer-Verlag. ISBN 3-540-61771-X. URL <http://dl.acm.org/citation.cfm?id=646195.681473>.
- [37] Valentina Lenarduzzi, Francesco Lomio, H. Huttunen, and D. Taibi. Are sonar-qube rules inducing bugs? *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 501–511, 2020.
- [38] K. Liu, D. Kim, T. F. Bissyande, S. Yoo, and Y. Le Traon. Mining fix patterns for findbugs violations. *IEEE Transactions on Software Engineering*, pages 1–1, 2018. ISSN 0098-5589. doi: 10.1109/TSE.2018.2884955.
- [39] J. Ludewig and H. Lichter. *Software Engineering: Grundlagen, Menschen, Prozesse, Techniken*. dpunkt.verlag, 2013. ISBN 9783864912993. URL <https://books.google.de/books?id=OMtNDAQAQBAJ>.

- [40] Diego Marcilio, Rodrigo Bonifácio, Eduardo Monteiro, Edna Canedo, Welder Luz, and Gustavo Pinto. Are static analysis violations really fixed?: A closer look at realistic usage of sonarqube. In *Proceedings of the 27th International Conference on Program Comprehension, ICPC '19*, pages 209–219, Piscataway, NJ, USA, 2019. IEEE Press. doi: 10.1109/ICPC.2019.00040. URL <https://doi.org/10.1109/ICPC.2019.00040>.
- [41] S. McIntosh and Y. Kamei. Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction. *IEEE Transactions on Software Engineering*, 44(5):412–428, May 2018. ISSN 2326-3881. doi: 10.1109/TSE.2017.2693980.
- [42] Chris Mills, Jevgenija Pantiuchina, Esteban Parra, Gabriele Bavota, and Sonia Haiduc. Are bug reports enough for text retrieval-based bug localization? In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 381–392, 2018. doi: 10.1109/ICSME.2018.00046.
- [43] Mockus and Votta. Identifying reasons for software changes using historic databases. In *Proceedings 2000 International Conference on Software Maintenance*, pages 120–130, 2000. doi: 10.1109/ICSM.2000.883028.
- [44] Nachiappan Nagappan and Thomas Ball. Static analysis tools as early indicators of pre-release defect density. In *Proceedings of the 27th International Conference on Software Engineering, ICSE '05*, pages 580–586, New York, NY, USA, 2005. ACM. ISBN 1-58113-963-2. doi: 10.1145/1062455.1062558. URL <http://doi.acm.org/10.1145/1062455.1062558>.
- [45] S. Panichella, V. Arnaoudova, M. Di Penta, and G. Antoniol. Would static analysis tools help developers with code reviews? In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 161–170, March 2015. doi: 10.1109/SANER.2015.7081826.
- [46] David Lorge Parnas. *Software Aging*, page 551–567. Addison-Wesley Longman Publishing Co., Inc., USA, 2001. ISBN 0201703696.
- [47] Luca Pascarella, Fabio Palomba, and Alberto Bacchelli. Fine-grained just-in-time defect prediction. *Journal of Systems and Software*, 150:22 – 36, 2019. ISSN 0164-1212. doi: <https://doi.org/10.1016/j.jss.2018.12.001>. URL <http://www.sciencedirect.com/science/article/pii/S0164121218302656>.
- [48] Norman Peitek, Sven Apel, Chris Parnin, André Brechmann, and Janet Siegmund. Program comprehension and code complexity metrics: An fmri study. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 524–536, 2021. doi: 10.1109/ICSE43902.2021.00056.

- [49] Massimiliano Di Penta, Luigi Cerulo, and Lerina Aversano. The life and death of statically detected vulnerabilities: An empirical study. *Information and Software Technology*, 51(10):1469 – 1484, 2009. ISSN 0950-5849. doi: <https://doi.org/10.1016/j.infsof.2009.04.013>. URL <http://www.sciencedirect.com/science/article/pii/S0950584909000500>. Source Code Analysis and Manipulation, SCAM 2008.
- [50] R. Plosch, H. Gruber, A. Hentschel, G. Pomberger, and S. Schiffer. On the relation between external software quality and static code analysis. In *2008 32nd Annual IEEE Software Engineering Workshop*, pages 169–174, Oct 2008. doi: 10.1109/SEW.2008.17.
- [51] R. Purushothaman and D. E. Perry. Toward understanding the rhetoric of small source code changes. *IEEE Transactions on Software Engineering*, 31(6):511–526, 2005. doi: 10.1109/TSE.2005.74.
- [52] Louis-Philippe Querel and Peter C. Rigby. Warningsguru: Integrating statistical bug models with static analysis to provide timely and specific bug warnings. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2018*, page 892–895, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450355735. doi: 10.1145/3236024.3264599. URL <https://doi.org/10.1145/3236024.3264599>.
- [53] Louis-Philippe Querel and Peter C. Rigby. Warning-introducing commits vs bug-introducing commits: A tool, statistical models, and a preliminary user study. In *29th IEEE/ACM International Conference on Program Comprehension, ICPC 2021, Madrid, Spain, May 20-21, 2021*, pages 433–443. IEEE, 2021. doi: 10.1109/ICPC52881.2021.00051. URL <https://doi.org/10.1109/ICPC52881.2021.00051>.
- [54] Foyzur Rahman, Sameer Khatri, Earl T. Barr, and Premkumar Devanbu. Comparing static bug finders and statistical prediction. In *Proceedings of the 36th International Conference on Software Engineering, ICSE 2014*, pages 424–434, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2756-5. doi: 10.1145/2568225.2568269. URL <http://doi.acm.org/10.1145/2568225.2568269>.
- [55] Giovanni Rosa, Luca Pascarella, Simone Scalabrino, Rosalia Tufano, Gabriele Bavota, Michele Lanza, and Rocco Oliveto. Evaluating SZZ implementations through a developer-informed oracle. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*, pages 436–447, 2021. doi: 10.1109/ICSE43902.2021.00049. URL <https://doi.org/10.1109/ICSE43902.2021.00049>.

- [56] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspán. Lessons from building static analysis tools at google. *Commun. ACM*, 61(4):58–66, March 2018. ISSN 0001-0782. doi: 10.1145/3188720.
- [57] Simone Scalabrino, Gabriele Bavota, Christopher Vendome, Mario Linares-Vásquez, Denys Poshyvanyk, and Rocco Oliveto. Automatically assessing code understandability. *IEEE Transactions on Software Engineering*, 47(3):595–613, 2021. doi: 10.1109/TSE.2019.2901468.
- [58] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? *SIGSOFT Softw. Eng. Notes*, 30(4):1–5, May 2005. ISSN 0163-5948. doi: 10.1145/1082983.1083147.
- [59] I. Sommerville. *Software Engineering*. International Computer Science Series. Pearson, 2011. ISBN 9780137053469.
- [60] E. Burton Swanson. The dimensions of maintenance. In *Proceedings of the 2nd International Conference on Software Engineering, ICSE '76*, page 492–497, Washington, DC, USA, 1976. IEEE Computer Society Press.
- [61] M. Tan, L. Tan, S. Dara, and C. Mayeux. Online defect prediction for imbalanced data. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 99–108, May 2015. doi: 10.1109/ICSE.2015.139.
- [62] F. Thung, Lucia, D. Lo, L. Jiang, F. Rahman, and P. T. Devanbu. To what extent could we detect field defects? an empirical study of false negatives in static bug finding tools. In *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 50–59, Sep. 2012. doi: 10.1145/2351676.2351685.
- [63] Alexander Trautsch. Effects of Automated Static Analysis Tools: A Multidimensional View on Quality Evolution. In *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings, ICSE '19*, pages 184–185, Piscataway, NJ, USA, 2019. IEEE Press. doi: 10.1109/ICSE-Companion.2019.00075.
- [64] Alexander Trautsch, Steffen Herbold, and Jens Grabowski. A Longitudinal Study of Static Analysis Warning Evolution and the Effects of PMD on Software Quality in Apache Open Source Projects. *Empirical Software Engineering*, 2020. doi: 10.1007/s10664-020-09880-1. URL <https://doi.org/10.1007/s10664-020-09880-1>.
- [65] Alexander Trautsch, Steffen Herbold, and Jens Grabowski. Static source code metrics and static analysis warnings for fine-grained just-in-time defect predic-

- tion. In *36th International Conference on Software Maintenance and Evolution (ICSME 2020)*, 2020.
- [66] Alexander Trautsch, Fabian Trautsch, Steffen Herbold, Benjamin Ledel, and Jens Grabowski. The SmartSHARK Ecosystem for Software Repository Mining. In *42nd International Conference on Software Engineering (ICSE 2020 Demos)*, 2020. URL <https://arxiv.org/abs/2001.01606>.
- [67] Alexander Trautsch, Johannes Erbel, Steffen Herbold, and Jens Grabowski. What really changes when developers intent to improve their source code: A commit-level study of static metric value and static analysis warning changes, 2021. Submitted to *Empirical Software Engineering*.
- [68] Alexander Trautsch, Steffen Herbold, and Jens Grabowski. Are automated static analysis tools worth it? an investigation into relative warning density and external software quality, 2021.
- [69] Fabian Trautsch, Steffen Herbold, Philip Makedonski, and Jens Grabowski. Addressing problems with replicability and validity of repository mining studies through a smart data platform. *Empirical Software Engineering*, August 2017. doi: 10.1007/s10664-017-9537-x.
- [70] Carmine Vassallo, Sebastiano Panichella, Fabio Palomba, Sebastian Proksch, Harald C. Gall, and Andy Zaidman. How developers engage with static analysis tools in different contexts. *Empirical Software Engineering*, 25(2):1419–1457, nov 2019. doi: 10.1007/s10664-019-09750-5.
- [71] A. Vetro, M. Morisio, and M. Torchiano. An empirical validation of findbugs issues related to defects. In *15th Annual Conference on Evaluation Assessment in Software Engineering (EASE 2011)*, pages 144–153, April 2011. doi: 10.1049/ic.2011.0018.
- [72] Julian von der Mosel, Alexander Trautsch, and Steffen Herbold. On the validity of pre-trained transformers for natural language processing in the software engineering domain, 2021. Currently in a major revision with *Transactions on Software Engineering*.
- [73] Greg Wilson and Andy Orham, editors. *Making Software: What Really Works, and Why We Believe It*. O’Reilly Press, 2010. ISBN 978-0596808327.
- [74] S. Yatish, J. Jiarpakdee, P. Thongtanunam, and C. Tantithamthavorn. Mining software defects: Should we consider affected releases? In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 654–665, 2019. doi: 10.1109/ICSE.2019.00075.

- 
- [75] F. Zampetti, S. Scalabrino, R. Oliveto, G. Canfora, and M. Di Penta. How open source projects use static code analysis tools in continuous integration pipelines. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 334–344, May 2017. doi: 10.1109/MSR.2017.2.
- [76] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl, and M. A. Vouk. On the value of static analysis for fault detection in software. *IEEE Transactions on Software Engineering*, 32(4):240–253, April 2006. doi: 10.1109/TSE.2006.38.

## **A Study of ASAT warning evolution**

This section contains a copy of the following publication.

A. Trautsch, S. Herbold, J. Grabowski: A longitudinal study of static analysis warning evolution and the effects of PMD on software quality in Apache open source projects. *Empirical Software Engineering* (2020) 25:5137–5192. Springer Nature

© 2020, The Author(s). Reprinted with permission.

<https://doi.org/10.1007/s10664-020-09880-1>



# A longitudinal study of static analysis warning evolution and the effects of PMD on software quality in Apache open source projects

Alexander Trautsch<sup>1</sup> · Steffen Herbold<sup>1</sup> · Jens Grabowski<sup>1</sup>

Published online: 21 September 2020  
© The Author(s) 2020

## Abstract

Automated static analysis tools (ASATs) have become a major part of the software development workflow. Acting on the generated warnings, i.e., changing the code indicated in the warning, should be part of, at latest, the code review phase. Despite this being a best practice in software development, there is still a lack of empirical research regarding the usage of ASATs in the wild. In this work, we want to study ASAT warning trends in software via the example of PMD as an ASAT and its usage in open source projects. We analyzed the commit history of 54 projects (with 112,266 commits in total), taking into account 193 PMD rules and 61 PMD releases. We investigate trends of ASAT warnings over up to 17 years for the selected study subjects regarding changes of warning types, short and long term impact of ASAT use, and changes in warning severities. We found that large global changes in ASAT warnings are mostly due to coding style changes regarding braces and naming conventions. We also found that, surprisingly, the influence of the presence of PMD in the build process of the project on warning removal trends for the number of warnings per lines of code is small and not statistically significant. Regardless, if we consider defect density as a proxy for external quality, we see a positive effect if PMD is present in the build configuration of our study subjects.

**Keywords** Static code analysis · Quality evolution · Software metrics · Software quality

## 1 Introduction

Automated static analysis tools (ASATs) support software developers with warnings and information regarding common coding mistakes, design anti-patterns like code smells (Fowler 1999), or code style violations. ASATs work directly on the source code or bytecode without executing the program. They are using abstract models of the source code, e.g., the Abstract Syntax Tree (AST) or the control flow graph to match the provided source

---

Communicated by: Meiyappan Nagappan

✉ Alexander Trautsch  
alexander.trautsch@cs.uni-goettingen.de

<sup>1</sup> Institute of Computer Science, University of Goettingen, Göttingen, Germany

code against a set of rules defined in the ASAT. If a part of the source code violates a pre-defined rule, a warning is generated. These rules can be customized by the project using the ASAT to fit their needs by removing rules deemed unnecessary. ASAT reports usually contain a type of warning, a short description, and the file and line number of the source code that triggered the warning. Developers can then inspect the line specified in the warning and decide if a change is necessary.

The defects that can be found by static analysis include varying severities. Java String comparisons with “==” instead of using the equals() method, would compare the object reference instead of the object contents. The severity rating for this type of warning is critical as it can lead to undesired behavior in the program. Naming convention warnings, e.g., not using camel case for class names on the other hand have a minor severity.

ASATs are able to uncover problems with significant real world impact. The Apple Goto Fail defect <sup>1</sup> for example could have been detected by static analysis utilizing the control flow graph. This importance regarding software quality is further demonstrated by the inclusion of ASATs in software quality models, e.g., Quamoco (Wagner et al. 2012) and ColumbusQM (Bakota et al. 2011). Zheng et al. (2006) found that the number of ASAT warnings can be used to effectively identify problematic modules. Moreover, developers also believe that static analysis improves quality as reported by a survey of Microsoft employees by Devanbu et al. (2016).

ASATs can be integrated as part of general static analysis via IDE plugins where the developer can see the warnings almost instantly. Usually IDE plugins are able to access a central configuration for rules that generate warnings. A central rule configuration is essential for project specific rules and exclusions of rules and directories. Integrating ASATs in the software development process as part of the buildfile of the project has the advantage of providing a central point of configuration which can also be accessed by IDE plugins. It also enables the developer to view generated reports prior to, or after the compilation as part of the build process. Moreover, the inclusion into the buildfile also allows Continuous Integration (CI) Systems to generate reports automatically. The reports can then be used to plot trends for general quality management or provide assistance in code reviews. Published industry reports share some findings regarding static analysis infrastructure and warning removal. Google (Sadowski et al. 2018) and Facebook (Distefano et al. 2019) both found that just presenting developers with a large list of findings rarely motivates them to fix all reported warnings. However, reporting the warnings as early as possible, or at latest at code review time, improves the adoption and subsequent removal of static analysis warnings. One of the lessons that Facebook and Google learned, is that static analysis warnings are not removed in bulk, but as part of a continuous process when code is added after the static analysis is configured or old code is changed.

ASAT warnings are able to indicate software quality because of how the rules that trigger the warnings are designed. The ASAT developers designed these rules not only to remove obvious bugs but also to express what is important for high quality source code via the designed rules. Therefore, a lot of the existing rules are based on best practices, common coding mistakes and coding style recommendations. Best practices and coding styles are also subject to evolution as the user base of a programming language evolves, new tooling is created and also as a programming language itself gets new features. For example, the Java code written today is different than the Java code written 10 years ago. These differences and, more importantly, the evolution of the usage of best practices are an interesting

---

<sup>1</sup><https://www.imperialviolet.org/2014/02/22/applebug.html>, last accessed: 2018-11-19

research topic, e.g., language feature evolution (Malloy and Power 2019), and design pattern evolution (Aversano et al. 2007).

The topics covered in research with regards to ASATs are concerned with configuration changes (Beller et al. 2016), CI-pipelines (Zampetti et al. 2017), finding reported defects (Habib and Pradel 2018; Thung et al. 2012; Vetro et al. 2011) or warning resolution times (Marcilio et al. 2019; Digkas et al. 2018). Vassallo et al. (2019) provide a thorough investigation of developer usage of ASATs in different developing contexts. One of the problems identified for ASAT usage is the number of false positives (Johnson et al. 2013; Christakis and Bird 2016; Vassallo et al. 2019) for which warning prioritization (Kim and Ernst 2007a; 2007b) was proposed, sometimes as actionable warning identification (Heckman and Williams 2009), see also the systematic literature review by Heckman and Williams (2011). Developers perceive ASATs as quality improving (Marcilio et al. 2019; Devanbu et al. 2016) although the percentage of resolved ASAT warnings vary, e.g., 0.5% (Liu et al. 2018), 8.77% (Marcilio et al. 2019), 6%-9% (Kim and Ernst 2007b) and 36.3% (Digkas et al. 2018).

What is still missing, is a longitudinal, more general overview of the evolution of ASAT warnings over the years of development which includes complete measurement of ASAT warnings over the complete development history. This would improve our understanding of exactly how the warnings evolve, e.g., how ASAT tools are used and the impact on the overall numbers of warnings over the project evolution. Moreover, a direct linking between static analysis warnings and removal of the implicated code in the process of bug fixing may be limiting the insights that can be gained from investigating ASAT usage. Most ASATs are also detecting problems due to spacing, braces, readability, and best practices which are not directly causing a defect. Therefore, the influence of ASATs on defects or software quality as a whole may be more indirect. To the best of our knowledge, only the work by Plosch et al. (2008) directly investigates this so far (there is also the work by Rahman et al. (2014) however it is not directly investigating correlations). Their work shows a positive correlation between ASAT warnings and defects, although their empirical study is limited to one project.

In this article, we investigate the usage of one ASAT in Java open source projects of the Apache Software Foundation in the context of software evolution. We determine the trends of removal of code with ASAT warnings over the projects lifetime. We are interested in the evolution of ASAT warnings on a project and on a global level, i.e., ASAT warnings for all projects combined. We examine general trends independent of developer interaction, i.e., is the state of software generally improving with regards to ASAT warnings. We also investigate which types of ASAT warnings have positive and negative trends to infer which types of coding standards or best practices are important to developers. To this end, we are not only interested in the absolute numbers of ASAT warnings but put them in relation to the project size. Moreover, we investigate the impact of including an ASAT in the build process on ASAT warning trends regarding their resolution. Additionally, we approximate the impact of including an ASAT in the build process on external quality via defect density (Fenton and Bieman 2014) by including defect information.

Our longitudinal, retrospective case study results in the following contributions of this work:

- An analysis of evolutionary trends of ASAT warnings in 54 open source projects from 2001-2017.
- An assessment of the effects of ASAT usage in open source projects on warning trends and software quality via defect density.
- An extension of prior work by providing a broader, long-term, evolutionary perspective with regards to ASAT warnings in open source projects.

The subjects of our case study are Java open source projects under the umbrella of the Apache Software Foundation. We observe ASAT warning trends via PMD<sup>2</sup> and defects via the Issue Tracking System (ITS) of the respective projects under study. In accordance with evidence based software engineering as introduced by Kitchenham et al. (2004) we provide our data and analysis for researchers and practitioners regarding the evolution of warnings and the impact of PMD on software quality.

The main findings of our study are the following.

- While the number of ASAT warnings is continuously increasing, the density of warnings per line of code is decreasing.
- Most ASAT warning changes are related to style changes.
- The presence of PMD in the build file coincides with reduced defect density.

The remainder of this work is structured as follows. In Section 2, we discuss prior work related to this study. After that, in Section 3, we present a short overview of static analysis in software development and discuss challenges in mining software repository data. In Section 4, we define our research questions, describe the selection criteria for our study subjects and explain our methodology in detail. In Section 5, we present the results. In Section 6, we discuss the results and relate them to current research. In Section 7, we evaluate the threats to validity to our study. Section 8 provides a short conclusion and provides an outlook on future work based on the data and methods described this article.

## 2 Related Work

In this section, we present the related work on empirical studies of ASATs and put them into relation to our work. Beller et al. (2016) investigated the usage of ASATs in open source projects. They focused on the prevalence of the usage of ASATs for different programming languages, how they are configured, and how the configuration evolves. In our work we are also investigating the evolution of the configuration. In contrast to Beller et al., we also run an ASAT on our study subjects for each commit. This enables us to analyze when ASAT warnings are resolved or introduced and the kind and number of warnings. We are using the projects buildfiles to extract whether PMD or other ASATs are used at the time of the commit and if custom rulesets were deployed. Thus, we expand on the previous work by not only investigating the changes in the configuration but also if the ASAT was used to remove any warnings at all. The drawback of this detailed view on ASAT warnings is that we have to narrow the focus on one programming language and one ASAT.

Kim and Ernst (2007b) utilized commit histories of ASAT warnings. They investigated the possibility of leveraging the removal times to prioritize the warnings. Instead of prioritizing ASAT warnings for removal, we are interested in removals on a global scale, by taking a longer history of the projects into account to get a broader view on the evolution of the projects under study with regard to ASAT warnings.

Liu et al. (2018) performed a large scale study using FindBugs<sup>3</sup> via SonarQube<sup>4</sup> where they investigated ASAT warnings over time. They created an approach to identify fix-patterns that are then applied to unfixed warnings. Similar to our own work, Liu et al. have run an ASAT on the project source code retroactively. In comparison to Liu et al., we include

---

<sup>2</sup><https://pmd.github.io/>, last accessed: 2019-04-11

<sup>3</sup><http://findbugs.sourceforge.net/>, last accessed: 2019-04-10

<sup>4</sup><https://www.sonarqube.org/>, last accessed: 2019-04-10

the build system and custom rules in our analysis. Thus, we can be sure that when we investigate removal of warnings that the developers could have seen the warnings. Instead of FindBugs via SonarQube, we focus on PMD which reports a different set of warnings due to PMDs usage of source code instead of byte code.

Digkas et al. (2018) also utilized SonarQube to detect ASAT warnings and their removal. They focused on the technical debt metaphor (Kruchten et al. 2012) and the resolution time that SonarQube assigns to each detected ASAT warning. The authors took snapshots of their projects every two weeks to run the ASAT and store the warnings. In our study, we are not concerned with technical debt. Instead, we want to give a bigger, longitudinal overview over the evolution of the project regarding ASAT warnings. Instead of using snapshots, we ran PMD retroactively on every commit to extract data, although due to run time constraints, this results in a smaller number of projects in our study. Nevertheless, due to utilizing PMD our data covers a longer period of time.

Marcilio et al. (2019) take a closer look at developer usage of ASATs through SonarQube. They investigated the time to fix for different types of issues with a focus on active developer engagement to specifically solve the reported ASAT warnings. In our study, we are not only concerned with resolution times. Instead, we are primarily interested in general trends regarding ASATs to infer information about the evolution of software quality in our candidate projects.

Plosch et al. (2008) utilized data collected by (Zimmermann et al. 2007) and correlated source code quality metrics and defects with warnings found by different static analysis tools. They used three releases of eclipse and presented correlations for different size, complexity and object oriented source code metrics. In contrast to Plösch et al. we are not concerned only with releases, we collected static analysis warnings for every commit of our candidate projects. In addition, we consider multiple projects instead of one. Although we are only able to provide data for one static analysis tool, we are able to provide more detailed defect information and on a larger scale. This should also cover effects of readability and maintainability changes due to ASAT usage.

Querel and Rigby (2018) build additional static analysis on top of CommitGuru (Rosen et al. 2015). Initial results show that the additional information that static analysis warnings provide can improve statistical bug prediction models. In our study, we investigate the evolution of ASAT warnings. Our own investigation into the impact of static analysis warnings on quality complements the initial results by Querel and Rigby (2018).

Static analysis software is often used in a dedicated security context. Penta et al. (2009) analyze security related ASAT warnings for three open source projects along their history. The authors performed an empirical study using three open source projects and three ASATs. Aloraini et al. (2019) also analyze security related ASAT warnings. The authors collect two snapshots, one at 2012 and one at 2017 for 116 open source projects. Both works come to the conclusion that the warning density of the security related warnings stays constant throughout their analysis time span. In contrast to our study both focused exclusively on security related ASATs.

### 3 Background

In this section, we introduce important topics regarding this study. First, we give a short description of the challenges regarding mining software repository histories and how they apply to this study. Then, we briefly discuss static analysis tools for Java and our choice of ASAT as well as software quality evaluation.

### 3.1 Mining Software Repository Histories

Working with old software revisions has its challenges. For projects which use the Java programming language some of these are:

- The build system may have been switched completely, e.g., from Ant to Maven.
- The project has no pinned version for the libraries it needs to be built successfully. This means, it may be impossible to build an older version because of incompatibilities with required libraries or missing versions of libraries (Tufano et al. 2017).
- The main source directory may have been moved, e.g., from `src/java` to `src/main/java` as is the case for most Java projects with a longer history.

In this study, we follow two different paths of inquiry, the first is only concerned with general trends regarding ASAT warnings. Hence, we do not need to consider build systems and libraries. However, even in that simplest approach we ignore test code as we only want to inspect production code. As no direct information via the build system is available we utilize regular expressions to exclude non-production code.

The second path of inquiry provides a more detailed view and also takes the build system into account. This is necessary as we extract ASAT usage via the build system configuration files. We therefore restrict the build system to Apache Maven as it is used by the majority of our candidate projects and allows extraction of this information. Including build information provides us with the ability to restrict the production code via the source directories specified in the configuration. The restriction to production code not only excludes test code but also additional tooling and examples. Apache Maven allows a tree like build configuration, i.e., a root configuration shared by the project and all its modules. As the build configuration can also contain custom rules and ASAT configurations, we have to consider all parts of the configuration tree. The root of the tree, usually parent POMs, can be included via Maven central and the leaves, usually modules that are part of the project, can be included via the filesystem.

In addition to the build system, we restrict ourselves to an ASAT that does not need compiled source code because of preliminary tests which found similar problems as Tufano et al. (2017). Tufano et al. found that 38% of commits in their data could not be compiled anymore. Nevertheless, even without the need to compile to bytecode, we are also experiencing some of the problems Tufano et al. found. As we want to extract custom rule definitions for PMD we need to consider build configuration files that may not exist anymore, e.g., missing parent POMs. To mitigate this problem, we manually rename some artifacts so that they can be found on maven central and incorporated into our extraction process, usually this only consists of removing `-SNAPSHOT` from the package name but in 3 cases we need to change the name of the package, e.g., from `commons` to `commons-parent`.

The extraction first tries to build the effective POM via Maven, i.e., including every module and configuration as well as explicit default values. If this fails it changes the `pom.xml` by removing the `-SNAPSHOT`, or, if the combination from group, artifact and version is in our rename list, it performs the artifact rename. After that the effective POM is built again. In case that the error persists it is logged and the existing state of the custom rules is not changed. The remaining errors consist of Maven configuration mismatches, in most cases a module references a parent with a wrong version because the parent `pom.xml` has been upgraded but the modules still reference the old version.

### 3.2 Static Analysis Tools for Java

Beller et al. (2016) noted, that most static analysis tools are in use for languages which are not compiled, because the compilation process includes certain static checks. Nevertheless, even Java and also C have some static analysis tools that can be utilized by practitioners to warn about potential problems in the source code.

As we are focusing on Java there are a few well known, open source static analysis tools for Java. One of the most prevalent is Checkstyle<sup>5</sup> which works directly on the source code and is mostly concerned with checking the code against certain predefined coding style guidelines. Another one is FindBugs which works on compiled Java bytecode to find bugs and common coding mistakes, e.g., a clone() method that may return null. SonarQube, a cloud based tool, has the ability to use the already described static analysis tools and also defines its own rules, e.g., cognitive complexity for a method is too high. It relates the ASAT warnings to a resolution time via a formula depending on the warning and programming language.

In this work we are focusing on PMD which works on the Java source code and finds coding style problems, e.g., an if without braces, but also common coding mistakes, e.g., comparison of two String objects using “==” instead of the equals() method. PMD provides a broad set of rules from a wide range of categories. Moreover, PMD is available since 2002 and therefore has been in use for a long time. This results in more data for our analysis and in a mature ASAT for us to use. The detailed documentation and changelog allow us to keep track of which ASAT warnings were available at a certain point in time.

### 3.3 Software Quality Evaluation

Software quality is notoriously hard to measure (Kitchenham and Pfleeger 1996). Since the beginning of investigating software quality it seemed clear that software quality consists of a combination of factors. The first models for software quality introduced by Boehm et al. (1976) and McCall et al. (1977) also mirror this combination of quality factors. Multiple quality factors are still in use throughout the subsequent ISO standards 9126 and 25010 and later quality models, e.g., Wagner et al. (2012) and Bakota et al. (2011). Fenton and Bieman (2014) as well as the ISO standards discern between internal and external quality. Internal quality factors concern the source code, e.g., cyclomatic complexity (McCabe 1976) or the process, e.g., the developers. External quality factors are on the customer facing side, e.g., defects, efficiency. Internal quality factors influence external quality factors, the problem is to evaluate which internal factors influence which external factors in which way.

Let us assume that software quality is a combination of multiple factors, e.g., maintainability or efficiency. If we want to automatically evaluate software quality, we need to find the concrete measurements that capture the corresponding factor. We then also need to know how to combine the measurements or factors together for the best approximation of software quality. Instead of using metric measurements as approximations, we can instead use ASATs based on their rules. Some ASATs not only include warnings about possible defects but also directly maintainability related warnings, e.g., default should always come last in a switch, exception handling code should not be empty or class names in Java should be in CamelCase. The rules that trigger the ASAT warnings are based on real world experiences and best practices of the developers, therefore, we expect that they are important in

<sup>5</sup><http://checkstyle.sourceforge.net/>, last accessed: 2019-04-10

an overall evaluation of software quality. Although this means that any ASAT considered for general software quality evaluation should support a broad set of rules.

If we consider the ASATs introduced in the previous Section 3.2 we find that PMD and SonarQube fit that definition best. While FindBugs and Checkstyle are both very established software products they fit different profiles. FindBugs focuses on possible defects and Checkstyle focuses on validating style rule conformance. While SonarQube would be a good fit, it does not exist for as long as PMD, FindBugs and Checkstyle. This limits the ability to observe actual usage of the ASAT in historical data. PMD on the other hand has both a long history of use and a broad set of rules. Therefore, we assume that PMD is a good approximation of internal software quality.

As an approximation for external software quality we utilize defect density (Fenton and Bieman 2014), i.e., the number of defects in relation to the size of the project. With both of these approximations, we can investigate internal and external software quality over the history of our study subjects.

## 4 Case Study Design

The *goal* of the case study is to investigate evolution of ASAT warnings and to examine the impact of PMD in the short and long term on ASAT warning trends as well as its impact on external software quality via defect density. In this Section, we formulate the research questions we aim to answer, explain the selection of subjects of the case study, and describe the methodology for the data collection, and the analysis procedures.

### 4.1 Research Questions

To structure our investigations, we define the following research questions which we separate into two main questions. The first main research question is only concerned with evolution of ASAT warnings over the full lifetime of the project: **How are ASAT warnings evolving over time?** (*RQ1*). We divide this research question into two sub-questions:

- *RQ1.1*: Is the number of ASAT warnings generally declining over time?
- *RQ1.2*: Which warning types have declined or increased the most over time?

Investigating these questions should shed some light on the general evolution of our study subjects regarding ASAT warnings. More specifically, we want to answer the question if “code gets better over time” with regard to ASAT warnings and also if there are differences between the different types of ASAT warnings. Differences between types of ASAT warnings may point to changing Java programming practices or changes in perceived importance, e.g., more camel case name violations for class names at the beginning of 2001 than at the end of 2017. The trend of resolved warnings by type should indicate which warning types are perceived as the most important by the developers that are active in our study subjects. ASAT warnings is a generic term, we specifically investigate ASAT warnings generated by PMD.

The second research question is focused on the impact of ASAT usage on the warning trends and on external software quality: **What is the impact of using PMD?** (*RQ2*). We divide this research question into five sub-questions:

- *RQ2.1*: What is the short term impact of PMD on the number of ASAT warnings?
- *RQ2.2*: What is the long term impact of PMD on the number of ASAT warnings?

**Table 1** Project selection criteria

Criteria	Criteria category
at least one year issue tracker activity (from 1.1.2018 backwards)	Project maturity
at least one year development (from 1.1.2018 backwards)	Project maturity
at least 1000 Commits	Project maturity
no incubator	Project maturity
commit activity since 1.1.2018	Up-to-dateness
issue activity since 1.1.2018	Up-to-dateness
at least 100 Files	Size
uses Maven	Scope
no Android	Scope

- *RQ2.3*: Does the active usage of custom rules for PMD correlate with higher ASAT warning removal?
- *RQ2.4*: Is there a difference in ASAT warning removal trends whether PMD is included in the build process or not?
- *RQ2.5*: Is there a difference in defect density whether PMD is included in the build process or not?

Our second set of research questions focuses on the ASAT usage according to the buildfile of the study subjects. Therefore, we focus only on the project development lifetimes where we can determine that an ASAT is used as part of the build process. Moreover, we consider only source directories configured in the build system. This allows us to exclude examples and tooling. We are also taking time and available rules for PMD into account, e.g., which rules are active in the configuration file and which were available at the time of the commit. This enables us to analyze the impact only for the rules that the developers were able to see and therefore address consciously. Moreover, we investigate the impact of PMD on external software quality via defect density by including information from the issue tracking system of our study subjects.

## 4.2 Subject Selection

Our study subjects are part of a convenience sample of open source projects under the leadership of the Apache Software Foundation<sup>6</sup> but nevertheless we applied some restrictions on our selection of study subjects. The base list of projects consists of every Java project of the Apache Software Foundation. We then apply the restrictions and start mining the remaining projects. The final list of study subjects is a sample of the projects that pass the criteria. The complete data for all cannot be used due to the computational effort required to calculate the ASAT warnings for all commits.

We focus on Java projects but exclude Android projects because of the different structure of the source code of the applications. We also restrict the build system to Maven as we utilize the buildfiles to extract the source directory and ASAT configurations for *RQ2*. Moreover, Maven provides the tooling necessary to combine multiple buildfiles of all sources per project.

<sup>6</sup><https://www.apache.org>, last accessed: 2019-11-10

We only include active, recent projects that are not currently in incubator status within the Apache Software Foundation, i.e., fully integrated into the Apache Software Foundation. All projects are actively using an issue tracking system as part of their development process. Our study subjects consist of libraries and applications with a variety of domains, e.g., math libraries, pdf processing, http processing, machine learning, a web application framework, and a wiki system. Moreover, our study subjects contain a diverse set of project sizes. The size ranges from small projects such as commons-rdf to larger projects such as Jena and Archiva.

The rest of our selection criteria are focused around project size, infrastructure, project maturity, and up-to-dateness of the project. All applied criteria are given in Table 1.

### 4.3 Methodology RQ1

In this section, we explain our approach to extract the required data and to calculate the required metrics to answer our research questions. An overview of the approach for data extraction is given in Fig. 1.

#### 4.3.1 Select Commit Path

To select the commits we are interested in, we build a Directed Acyclic Graph (DAG) from all commits in the repository and their parent-child relationships. After the graph construction, we extract a single path of commits for the project. This is depicted in the first part of Fig. 1. Commits are denoted as circles with a number referring to their order of introduction into the codebase. We extract a single path from the latest master branch commit to the oldest reachable orphan commit. We need to select a path this way because we can not just select the master branch as the information on which branch a commit is created is not stored in Git (Bird et al. 2009). Moreover, we select a single path because if work is done in parallel on two or more branches of the project and we order the commits by date we get jumps in the data as we would have a sequence of commits that represent different states of the codebase at the same time.

The latest master branch commit is extracted via the “origin/head” reference of Git which points to the default branch of the repository. The default branch is usually named master, although in some Apache projects the default branch is called trunk as the projects were converted or are mirrored from Subversion. Orphan commits do not have parents. This is usually the initial commit of the repository. It can also happen that a repository has multiple orphan commits, which also can be merged back into the current development branch. By choosing the oldest orphan commit, we extract the first initial commit. Then, we use the

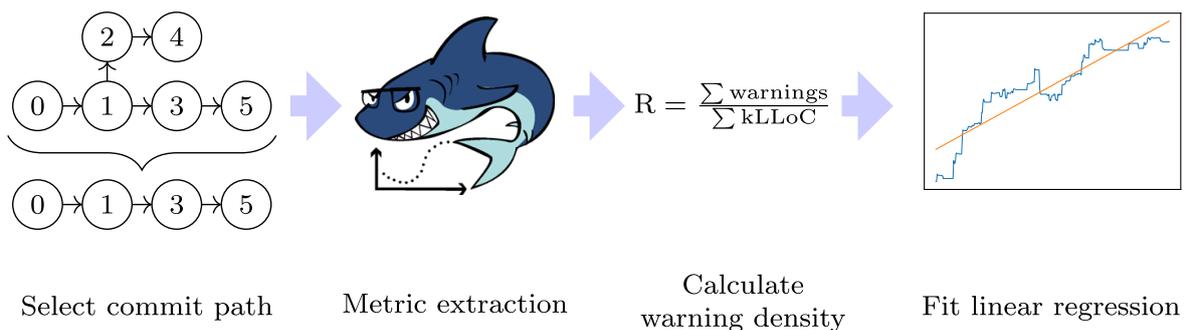


Fig. 1 Methodology RQ1

**Table 2** Regular expressions for excluding non-production code

File type	Regular expression
Test	(^ \)/ (test tests test_long_running testing  legacy-tests  testdata test-framework derbyTesting unitTests  java\ stubs  test-lib src\ it src-lib-test src-test tests-src  test-cactus  test-data test-deprecated src_unitTests test-tools  gateway-test-release-utils gateway-test-ldap  nifi-mock)\ /
Documentation	(^ \)/ (doc docs example examples sample samples demo  tutorial helloworld userguide showcase SafeDemo)\ /
Other	(^ \)/ (_site auxiliary-builds gen-java external  nifi-external)\ /

graph representation to find the shortest path between these two commits via Dijkstra's shortest path first algorithm (Dijkstra 1959). The end result of this step is the shortest path between the oldest orphan commit and the newest commit on the default branch of the project.

### 4.3.2 Metric Extraction

The second step in Fig. 1 depicts the extraction of ASAT warnings and Software metrics. For both we are using OpenStaticAnalyzer<sup>7</sup> as part of a plugin<sup>8</sup> for the SmartSHARK infrastructure (Trautsch et al. 2017). SmartSHARK in conjunction with a HPC-Cluster provided us with the means to extract this information for each file in each commit of our candidate projects. OpenStaticAnalyzer is an open sourced version of the commercial tool SourceMeter (FrontEndART 2019) which has been used in multiple studies, e.g., Faragó et al. (2015), Szóke et al. (2014), and Ferenc et al. (2014) and, more recently (Ferenc et al. 2020). It works by constructing an Abstract Semantic Graph (ASG) from the source code which is then used to calculate static source code metrics. As it is included in SmartSHARK we perform a validation step after each mining step which verifies if the metrics are collected for each source code file. In addition to the size, complexity and coupling metrics OpenStaticAnalyzer also provides us with ASAT warnings by PMD. OpenStaticAnalyzer applies 193 rules from which the warnings are generated including line number, type and severity rating. The source code metrics are provided at package, file, class, method and attribute level. The resulting data from the mining step includes ASAT warnings from PMD and source code metrics for each file in each commit of our candidate projects. As we primarily want to investigate program code we exclude non-production code by path. We use the regular expression shown in Table 2 to filter non-production code. The regular expressions were created based on manual inspection of the directory structure of the projects we use in our study.

<sup>7</sup><https://github.com/sed-inf-u-szeged/OpenStaticAnalyzer/>, last accessed: 2019-11-10

<sup>8</sup><https://www.github.com/smartshark/mecoshark/>, last accessed: 2019-11-10

Furthermore, we only compare full years of continuous development in our analysis, thus we remove incomplete years: we remove the first year and everything after 31.12.2017 because we started collecting the data in 2018. This ensures that we only have trends over the complete development history of the project but also for each single year of development which provides a more detailed view in addition to a full view of the projects lifetime.

### 4.3.3 Calculate Warning Density

The absolute number of ASAT warnings is correlated with the amount of source code in the project. Increasing the code size seems to increase the number of warnings. Even in projects using PMD, this is expected as we also study warnings which the developers could not have seen before. Either because the ASAT did not support them at the time of the commit or the rules that trigger the warnings are not active. Most of the biggest additions and removals of ASAT warnings are due to the addition and removal of files in the repository. The measure for size of the source code we are using is Logical Lines of Code (LLOC) in steps of one thousand (kLLOC). By using LLOC instead of just Lines of Code (LOC) we discard blank lines and comments. LLOC provides a more realistic estimation of the project size.

Table 3 shows the correlation between the sum of ASAT warnings and the sum of kLLOC per commit in all commits available in our data. We are using two non-parametric correlation metrics, Kendall's  $\tau$  (Kendall 1955), which uses concordance of pairs, i.e., if  $x_i > x_j$  and  $y_i > y_j$  and Spearman's  $\rho$  (Spearman 1904) which uses a rank transformation to measure the monotonicity between two sets of values instead of concordance of pairs of observations.

We can see in Table 3 that there is a positive correlation between kLLOC and the number of ASAT warnings, i.e., as kLLOC increases so does the number of ASAT warnings. As we want to analyze ASAT warning trends with minimum interference of functionality being added or deleted we decided to use warning density instead of the absolute number of ASAT warnings. Warning density is the ratio of the ASAT warnings and product size.

$$\text{Warning density} = \frac{\text{Number of ASAT warnings}}{\text{Product size}} \quad (1)$$

As product size we chose kLLOC, the warning density is calculated per commit. The advantage of this measurement is that we still see when code with less warnings is added or removed. This also accounts for the effect of developers only scrutinizing new code being added as the new code would then contain less warnings than the existing and show up in our data as a declining trend of ASAT warnings.

Nevertheless, we keep the sum of all warnings for completeness which means we have two aggregations of warning data for our next step:

*S* (sum): The sum of all ASAT warnings per commit which are also available on basis of warning type and severity rating.

*R* (warning density): The ratio meaning the warning density per commit which is also available on a basis of warning type and severity rating.

**Table 3** Correlation between the number of ASAT warnings and kLLOC

Method	Value	P-value
Kendall's $\tau$	0.57509	0.0
Spearman's $\rho$	0.71654	0.0

#### 4.3.4 Fit Linear Regression

Fitting a regression line results in a trend line that we can use to determine if ASAT warnings are generally increasing or decreasing in a more appropriate way as just using a delta between the last and first data points. This method, while still being simple and comprehensible, utilizes all available information, e.g., if the project contained high numbers of warnings for most of its lifetime and only at the end of the extracted data resolved most of them. We fit multiple linear regression lines to our data:

- all years per project, for a long-term trend,
- per year per project, for a short-term trend.

Moreover, we additionally fit regression lines for each group of filtered ASAT warnings we introduce in Section 4.4.2 for our second main research question. The linear regression lines provide broad overall trends and specific trends for the ASAT warnings to answer our research questions.

After the fitting of the regression lines, we utilize the coefficient of the linear regressions as the slope. As we have only one variable, this is the same as calculating the slope for each line by applying the point slope formula (2) where  $y$  are the values of the fitted regression line and  $x$  is the day of the commit.

$$\text{slope} = \frac{y_n - y_1}{x_m - x_1} \quad (2)$$

The slope provides us with a single number representing the trend which we use for further analyses. Moreover, this enables the merging of results for projects with different lifetimes in order to create a global overview of a trend.

In order to restrict the calculated trends to meaningful values we use an F-Score which is calculated via a correlation between our regression line and the measured value. First, we calculate the correlation:

$$\text{corr} = \frac{(X_i - \bar{X}) \cdot (y_j - \bar{y})}{\sigma(X) \cdot \sigma(y)} \quad (3)$$

Where  $X_i$  is the  $i$ -th day of our commits,  $y_j$  is the  $j$ -th value of the regression line and  $\bar{X}$ ,  $\bar{y}$  is the mean of the number of days of commits and mean of the regression values respectively.  $\sigma(X)$ ,  $\sigma(y)$  denotes the standard deviation of  $X$  and  $y$ . The correlation is then converted to an F-Score and a p-value.

$$F = \frac{\text{corr}^2}{(1 - \text{corr}^2) \cdot (|y| - 2)} \quad (4)$$

The p-value conversion is achieved via the survival function of the F-distribution.

To restrict noise introduced by bad regression fits for trends, we include only slope values in our analysis where the F-Score is above 1 and the p-value for the F-Score is lower than 0.05. As the F-Score describes a relationship between the regression values and the time, we chose this performance metric instead of others related to linear regression such as  $R^2$ .

#### 4.4 Methodology RQ2

To answer our second research question, we need to include knowledge about the inclusion of ASATs in the build process of the projects. As previously mentioned, we focus on Maven as the build system. To extract the additional information, we extend our approach shown in Fig. 1 with the additional steps depicted in Fig. 2. In a nutshell, we filter out commits

where Maven was not used, create new sets of rules depending on custom rules included in the available Maven build configuration, and include defect density as external software quality measure.

#### 4.4.1 Parse buildfiles

We traverse the path of commits previously selected to determine where Maven was introduced to the project and all commits where its configuration file was changed. Maven projects can contain multiple buildfiles, modules and configuration residing in parent buildfiles. In order to account for these features, we utilize a Maven feature that combines all this information including fetching the parent buildfiles from the Maven repository. For each commit where one or multiple Maven files were changed in the target repository, we execute Maven to automatically resolve potential project modules defined in the main Maven configuration file, potential parent configurations, and set all settings explicitly taking default values and overrides into account. We also extract source and test directories from the configuration which allows us to restrict our analysis to program source code and to exclude tests that reside in non-standard directories. This information is further refined to extract which ASATs are currently active, i.e., we detect if PMD, Checkstyle, and FindBugs are configured. If PMD is configured, we extract the configuration including all additional custom configuration files.

Custom configurations for PMD can consist of multiple files with rules and categories of rules. We parse every custom ruleset file and extract rule categories and single rules. Single rules are used as is, whereas the rule categories are expanded to the single rules they contain according to the current PMD documentation on all rulesets<sup>9</sup>. This ensures that we have an accurate representation of the warnings that were actively presented to the developers.

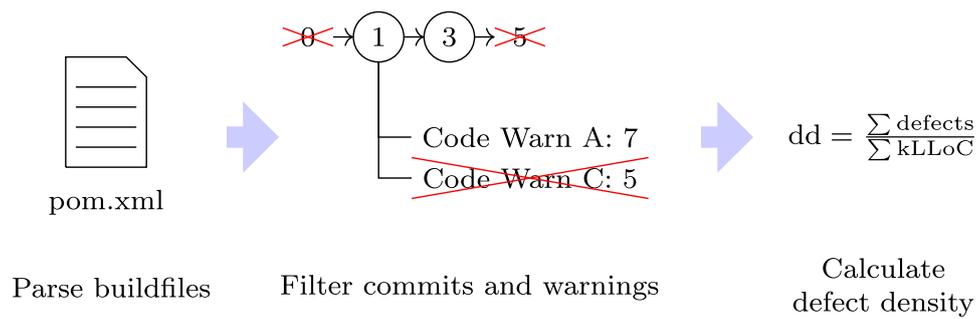
#### 4.4.2 Filter Commits and Warnings

We remove all commits where no Maven buildfile was present. This is true for commits where the build system is not Maven but, e.g., Ant or Gradle. To make our comparisons viable, we restrict our data to Maven and remove commits until a Maven buildfile is introduced. The project selection performed in the first step ensures that we only have projects where the Maven buildfile was present in the latest commit of our data. Thus, we do not have to remove commits due to the project switching its buildsystem from Maven to Gradle. After that, we create subsets of warnings in our data by filtering certain warnings.

*t* (time-corrected) The first subset consists of time corrected warnings. This includes only warnings that were available at the time of the commit where we collected the warning. To be able to utilize this information, we included a mapping for each detected rule to the PMD version that introduced the warning together with the release date of that version. Then we filter out the rules that could not have been reported because at the time of the commit the rule was not available in PMD yet. This is only possible because of the very thorough documentation of rules from PMD and the detailed changelog that stretches back to the first version. We include this subset because of the length of the project histories considered. As some of our data goes back to 2007<sup>10</sup>, we need to take the ASAT warnings into account that were possible to gain from PMD at that point in time.

<sup>9</sup>[https://pmd.github.io/latest/pmd\\_rules\\_java.html#additional-rulesets](https://pmd.github.io/latest/pmd_rules_java.html#additional-rulesets)

<sup>10</sup>Earliest date for which a Maven buildfile exists in our data.



**Fig. 2** Methodology extension for RQ2

*d* (default): The second subset represents the default configuration of the maven-pmd-plugin. The default rules are taken from the most recent configuration<sup>11</sup> and filtered to include only detectable rules. This results in a set of 45 rules.

*e* (effective): For the third subset we want to include as much detail as possible. To achieve this, we calculate the currently active ASAT rules for each commit. These effective active rules take all custom rules and rule excludes into account. If no custom rules are defined we are using the default rules according to the documentation of the maven-pmd-plugin<sup>11</sup>, same as for the subset *d*. This subset contains all information that a developer on the project under investigation can acquire by utilizing the buildfile.

*o* (without overlapping): The final subset removes rules overlapping with other ASATs used in the projects. This is achieved by filtering rules which overlap with rules supported by current versions of Checkstyle and FindBugs. This subset enables us to increase the precision of our impact measurement for the effects of using PMD on ASAT warning trends. This avoids skewed results for study subjects which are in the non PMD group but utilize FindBugs or Checkstyle which contain rules that are also present in PMD. A complete list of overlapping rules can be found in the Appendix.

All of these subsets of warnings can be combined to provide us with a set of rules for the analyses, e.g., the warning density of the default rules with time-correction or the warning density of the effective rules with time-correction without overlapping rules.

#### 4.4.3 Calculate Defect Density

This step utilizes the SmartSHARK infrastructure which we already used for the metrics collection to incorporate information from the ITS into our data. The extracted information is condensed to a metric per development year, the “de facto standard measure of software quality” (Fenton and Bieman 2014), defect density, which is a ratio of the number of known defects and the size of the product.

$$\text{Defect density} = \frac{\text{Number of known defects}}{\text{Product size}} \quad (5)$$

In this study we use the mean kLLoC per year as the product size and the number of created bug reports per year as the number of known defects. This provides us with a metric per year which we can then utilize to measure the impact of PMD usage on external software quality (Fenton and Bieman 2014). As we have projects in our dataset which switched ITS

<sup>11</sup><https://maven.apache.org/plugins/maven-pmd-plugin/examples/usingRuleSets.html>, last accessed: 2019-03-18

and as we need full development years we discard the first year for which we have defects in our data.

#### 4.5 Analysis Procedure

In our case study, we investigate different questions which require a different analysis procedures. For *RQ1.1*, we aggregate the plain sum of ASAT warnings per commit over the projects development and the warning density as defined in Section 4.3.3. Then, we fit regression lines and calculate the F-Score as well as the slope of the regression to get the general trend.

In the case of *RQ1.2* we do the same, but for completeness we additionally calculate the delta of the last and the first commit of the data as well as the number of remaining warnings per kLLOC. In order to aggregate data of all projects we calculate the mean and median of the data.

The short and long term impact of PMD on the number of ASAT warnings in *RQ2.1* and *RQ2.2* is measured via the number of projects for *RQ2.1* and the median of slopes of the trend line for all years following PMD introduction per project for *RQ2.2*. The slopes are calculated via the warning density but without overlapping rules from FindBugs and Checkstyle.

For *RQ2.3*, we sum the number of rule changes per year for projects using PMD and correlate them via Kendall's  $\tau$  and Spearman's  $\rho$  to the warning density trends of the rules used ( $R+e+t$ ).

To answer the research questions *RQ2.4* and *RQ2.5*, we measure the difference between two samples. We first investigate the distribution of our data via the Shapiro-Wilk test (Wilk and Shapiro 1965) for normality and Levene's test (Levene 1960) for variance homogeneity. As these tests revealed that the data is non-normal with a homogeneous variance, we decided to use the Mann-Whitney-U test (Mann and Whitney 1947). Although the Mann-Whitney-U test is a ranked test we still talk about differences in median for the sake of simplicity. In both research questions, we measure the difference between the years of PMD usage and the years where PMD was not used. Partial use in a year is excluded from the analysis. We chose a significance level of  $\alpha = 0.05$ , after Bonferroni (Abdi 2007) correction for 24 statistical tests, we reject the  $H_0$  hypothesis at  $p < 0.002$ . The difference in median between both samples is not significant every time for  $p < 0.002$ . Therefore, only for the last comparison of *RQ2.4* and for *RQ2.5* we also calculate effect size and confidence interval.

To calculate the effect size of the Mann-Whitney-U test, we utilize the fact that for sample sizes  $> 8$  the U test statistic is approximately normally distributed (Mann and Whitney 1947). We first perform a  $z$ -standardization (Kreyszig 2000). We are assuming that our sample's mean and standard deviation are a good approximation of the populations mean and standard deviation. After we calculate  $z$  we can calculate the effect size  $r$ . A value of  $r < 0.3$  is considered a small effect,  $0.3 \leq r \leq 0.5$  is a medium effect and  $0.5 < r$  is a strong effect (Cohen 1988). For the confidence interval we follow (Campbell and Gardner 1988) who use the  $K$ -th smallest to the  $K$ -th largest difference between two samples as the interval. The confidence interval then consists of the  $K$ -th difference and the  $\max(n, m) - K$ -th difference between both samples.

#### 4.6 Replication Kit

All extracted data can be found online (Trautsch et al. 2020). The code for creation of the tables and figures used in this paper as well as a dynamic view of warning density, LLoC and warning sum for each project is included.

## 5 Case Study Results

In this section we present the results of our study. This section is split into two parts, one for each of our main research questions.

### 5.1 RQ1: How are ASAT warnings evolving over time?

Our first research question considers ASAT warning evolution over the complete lifetime of each project. We do not consider PMD usage in the build process, custom rulesets or the availability of warnings in PMD at the time of the commit in this section. We use all PMD rules that are available. Table 4 shows the trend of ASAT warnings for every project and year as well as the approximate change per year over all years. Furthermore, the table includes the trend over the complete lifetime of the project with two base values: the sum of ASAT warnings  $S$  and the warning density  $R$ . The trends for single years are calculated based on warning density. The arrows indicate the trend of the ASAT warnings. A downwards arrow indicate a positive trend, i.e., the warning density declines, an upwards arrow negative trend, i.e., the warning density increases. If our criteria for the regression fits are not met, i.e., the F-Score is below 1 and the corresponding p-value is above 0.05 a straight rightway arrow is used. Hence, the straight rightway arrow indicates that there was no significant change.

#### 5.1.1 RQ1.1: Is the number of ASAT warnings generally declining over time?

Table 4 shows, that if we consider the complete lifetime of the project warning density ( $R$ ) increases in only 8 of 54 projects. The majority of our study subjects improve with regard to warning density. If we only consider the sum ( $S$ ) the picture is not as clear, here we have more negative trends, i.e., the number of ASAT warnings increase. This is expected as the number of ASAT warnings usually increases with addition of new code and both are positively correlated as mentioned previously. This shows that if we consider warning density to be a code quality measure, that the code quality steadily increases in most projects. We also include the value of the slope of the trend for warning density ( $R$  p.a.) in the table, which indicates a change of warning density in years over the complete lifetime and on average over all projects. We exclude projects where the slope does not met our criteria for F-Score. The value in column  $R$  p.a. quantifies the average change in warning density per year, e.g., commons-math removes on average 5 ASAT warnings per 1000 Logical Lines of Code per year. When we consider the mean of all projects we see that on average 3.5 ASAT warnings per 1000 LLoC are removed per year.

*RQ1.1 Summary:* The number of ASAT warnings from PMD are not generally declining. However, if we consider warning density then most are declining. Out of 54 projects, 43 show declining trends, 10 are showing an increasing trend of warning density and 1 is showing minimal changes. On average, each project removes 3.5 ASAT warnings per 1000 LLoC per year. This result shows that, on average, code quality is improving over time with regard to static analysis warnings.

#### 5.1.2 RQ1.2: Which warning types have declined or increased the most over time?

To answer the next research question, we consider how groups of rules have changed in their evolution over the projects lifetime. In this case, we not only report the slope of the trend but also report the delta of the first warning density measurement per project and the last

**Table 4** Trends for the number of ASAT warnings over time including sum (S), warning density (R) and warning density change per year (R p.a.)

Project	2001	2002	2003	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013	2014	2015	2016	2017	S	R	R p.a.		
archiva						↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	2.0638	
calcite						↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	1.4294
cayenne		↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	-
commons-beel		↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	-5.6761
commons-beanutils		↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	-2.4991
commons-codec		↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	-1.1347
commons-collections		↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	-2.2388
commons-compress		↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	-0.8231
commons-configuration		↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	-1.9519
commons-dbcp		↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	-5.9619
commons-digester		↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	-7.9792
commons-imaging		↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	-15.7747
commons-io			↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	-0.6867
commons-jcs			↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	-3.0907
commons-jexl			↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	-9.5675
commons-lang			↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	-1.8343
commons-math			↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	-5.4131
commons-net			↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	-4.6747
commons-rdf																						9.0116
commons-scxml												↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	-1.9494
commons-validator			↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	-3.7802
commons-vfs			↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	-0.5570
eagle												↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	-22.7229
falcon												↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	-8.9550

**Table 4** (continued)

Project	2001	2002	2003	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013	2014	2015	2016	2017	S	R	R p.a.
flume																				-0.1371
giraph																				-7.8048
gora																				2.2756
helix																				0.5769
htpcomponents-client																				-2.0253
htpcomponents-core																				-1.1508
jena																				-5.4301
jspwiki																				-3.9080
knox																				-2.9519
kylin																				-0.2880
lens																				-8.4280
mahout																				-4.0993
manifoldcf																				-0.9399
mina-sshd																				-8.2677
nifi																				1.8912
opennlp																				0.1831
parquet-mr																				-5.3441
pdfbox																				-3.1469
phoenix																				1.0603
ranger																				-3.1562
roller																				-1.9939
santuario-java																				-6.9039
storm																				-4.1547

**Table 4** (continued)

Project	2001	2002	2003	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013	2014	2015	2016	2017	S	R	R p.a.	
streams													↗	↗	↗	↗	↗	↗	↗	-12.3819	
struts							↗				↗	↗	↗	↗	↗	↗	↗	↗	↗	↗	1.2610
systemml												↗	↗	↗		↗	↗	↗	↗	↗	-4.4465
tez														↗	↑	↗	↑	↗	↗	↗	-0.3926
tika												↗	↗	↗	↗	↗	↗	↗	↗	↗	2.5588
wss4j																	↑	↗	↗	↗	-8.9378
zeppelin																					-4.4358
mean																					-3.5035

measurement. This provides us with a delta of the absolute number of ASAT warnings per kLLOC per warning group and severity. Due to different project lifetimes we measure these numbers per project and then average the values to end up with a number that encompasses all of our data.

Table 5 contains all rule groups and severities in our data provided by PMD. It contains the slope of the trend, the average change of warning density per project over the complete lifetime of the project and the remaining warnings per kLLOC. We can see that, e.g., on average a project removed 7 naming rule warnings per kLLOC over its complete lifetime and still has 7.5 warnings per kLLOC left. When considering the trend, we can see that each project, on average, removes 0.42 naming rule warnings per kLLOC per year. Moreover, we see that each project on average resolves 34.03 warnings per kLLOC regardless of its type or severity over its complete lifetime and still has 58.06 warnings per kLLOC left.

These changes in the number of occurrences of rules by type can hint at potential changes in coding standards in the the years between the beginning of 2002 and end of 2017. Most prominently brace, design and naming rules, which consist of best practices regarding code blocks and naming conventions, e.g., an *if* should be followed by braces even if it is followed only by a single instruction and class names should be in camel case. Design rules contain best practices regarding overall code structure, e.g., avoiding deeply nested if statements and simplify boolean returns. We can see that the trend for specialized rules like Java and

**Table 5** Mean warning density change per year (MR p.a.), median (MEDR p.a.) and delta for ASAT groups and severities

ASAT group / severity	MR p.a.	MEDR p.a.	Delta	Remaining
minor	-2.5897	-1.7125	25.74	42.95
major	-0.8852	-0.6832	7.228	12.2
critical	-0.1456	-0.1244	1.061	2.909
brace rules	-0.7105	-0.1426	6.265	4.793
design rules	-0.5788	-0.5187	5.507	16.36
java logging rules	-0.4755	-0.0902	3.094	1.968
jakarta commons logging rules	-0.4643	-0.0595	1.72	3.053
naming rules	-0.4221	-0.2667	7.041	7.526
type resolution rules	-0.3442	-0.1602	2.631	3.02
controversial rules	-0.2548	-0.1134	1.414	6.651
optimization rules	-0.2485	-0.1107	2.961	3.178
basic rules	-0.1301	-0.0613	1.63	1.677
unnecessary and unused code rules	-0.0705	-0.0355	0.7491	0.6812
string and stringbuffer rules	-0.0632	-0.0198	0.2866	3.263
strict exception rules	-0.0562	-0.0125	0.2469	3.626
security code guideline rules	-0.0401	0.0028	0.02258	0.7211
junit rules	-0.0144	-0.0034	0.01282	0.004718
javabean rules	-0.0079	-0.0030	0.3478	0.1154
finalizer rules	-0.0006	-0.0003	-0.001112	0.01842
j2ee rules	-0.0003	-0.0003	0.08508	0.1346
clone implementation rules	0.0002	-0.0028	0.06261	0.08835
import statement rules	0.1722	-0.0046	-0.04623	1.182

jakarta logging rules is steeper than naming rules, although when we consider the delta it is clear that naming rules are removed far more by number. Moreover, when we consider the median (MEDR p.a.) instead of the mean trend (MR p.a.) we can see that brace, design, and naming rules also have high median trends. A complete list of the rules and their groups as well as their severities is given in the Appendix.

The two groups of warnings that are increasing by delta, although only slightly, are finalizer and import statement rules. By mean trend only import statement and clone implementation rule violations are increasing slightly. Finalizer rules are concerned with the correct implementation of `finalize()` which is called by the garbage collector of Java. Import statement rules contain rules regarding duplicate imports, unused imports and unnecessary imports, e.g., `java.lang` or imports of classes from the same package. Clone implementation rules are focused on checking implementations of `clone()` methods.

Regarding the severity of the warnings we see that minor severity warnings are resolved the most, major severity warnings second most and critical warnings last. When we calculate the percentages of reduction in warning density by severity we see that minor and major are reduced by about 37% each while critical by about 27%. This may indicate that developers do not necessarily try to remove all critical warnings. However, this could also be an indication of critical severity warnings being more prone to false positives.

The types of warning that declined the most may hint at developer preference or possibly easy resolution of reported warnings. The declining of naming, brace and design rules may also be a consequence of changing coding standards or, more generally, a maturation of Java software coding style. The results may also hint at some rules which are ignored by developers. The density of import statement rules is increasing. This may indicate that this type of rule is more often ignored by developers.

*RQ1.2 Summary:* The warning density of Naming, brace and design rules have declined the most. Finalizer and import statement rules are the only ASAT warning types that increase.

## 5.2 RQ2: What is the impact of using PMD?

This part of the study discards every commit up until the point in time Maven was introduced as a build system. Although we shorten the project history that is available for analysis, keeping only commits with a Maven buildfile allows us to be certain that we detect the ASAT inclusion via the Maven configuration. Moreover, this allows us to read custom ruleset definitions and source directories. Utilizing the source directories from the Maven configuration narrows the scope for the files to code only files. We effectively discard tests and tooling which are not part of the build process. Our aim is to be as detailed as possible and counting only the rules that were available at the point in time of the commit. We also include only files that were part of the analysis if the projects developers had run the ASAT via the buildfile.

### 5.2.1 RQ2.1: What is the short term impact of PMD on the number of ASAT warnings?

Table 6 shows the trends of ASAT warnings for full years of development. The color indicates if PMD was used for all commits that year: green indicates PMD was used for the complete year, red indicates no use of PMD for the complete year, black indicates partial usage due to introduction or removal of PMD from the buildfile during that year. In seven

of the 54 projects listed in Table 6, PMD was removed at least once. We inspected every case to investigate the reasons for the removal.

**Table 6** Trends for warning density without overlapping rules ( $R+o$ ), green indicates use of PMD in buildfile, red indicates absence of PMD, black indicates partial use of PMD over the year

Project	06	07	08	09	10	11	12	13	14	15	16	17
archiva	↘	↗	↘	↘	↗	↗	→	↘	↘	↘	↗	↗
calcite								↗	↗	↗	↘	↗
cayenne							↗	↗	↗	↗	↗	↗
commons-bcel		↗	→	↗	↘	↘	↗	↗	→	↗	↘	↘
commons-beanutils		↗	↗	↗	↗	↗	↗	↘	↘	↗	→	↗
commons-codec			→	↘	↗	↗	↗	↘	↘	↘	↘	↘
commons-collections		↗	↗	↘	→	↘	↗	↘	↗	↗	↘	↘
commons-compress		↗	↗	→	↗	↘	↘	↗	↗	↗	↘	↘
commons-configuration			↘	↘	↘	↘	↘	↘	↘	↘	↗	↗
commons-dbcp			↘	↘	↘	↘	↗	↗	→	↘	↘	↘
commons-digester		↘	↗	↗	↘	↘	↗	↘	↗	↗	↗	↗
commons-imaging			↘	↗	↘	↘	→	↘	↗	↗	↗	↗
commons-io		↗	↗	↗	↗	↘	↗	↘	↘	↘	↗	↘
commons-jcs			↗	↗		→	↘	↗	↘	↘	↗	↗
commons-jexl			↘	↘	↗	↘	↗	↗	↗	↘	↗	↘
commons-lang		↗	↘	→	↘	↘	↘	↘	↘	↘	↘	↘
commons-math			↘	→	→	→	↘	↘	↘	↘	↗	↘
commons-net			↘	↘	↘	↘	↘	↘	↘	↘	↘	↘
commons-rdf										↘	→	↗
commons-scxml		↗	↘	↘	↘	↘	↘	↘	↘	↘	↘	↗
commons-validator			→	↗	↗	↗	↘	↘	→	↘	↘	↗
commons-vfs		↘	↗	↘	↘	↗	→	↗	↘	↘	↘	↗
eagle											↘	↘
falcon							↗	↘	↘	↘	↗	↗
flume							↗	↗	↘	↘	↘	↗
giraph					↘		↘	↗	↘	↘	↘	↘
gora							↗	↗	↘	↘	↗	↘
helix							↗	↗	↘	↘	↘	↘
httpcomponents-client			↗	↗	↗	↘	↘	↘	↘	↗	↘	↗
httpcomponents-core		↘	↘	↘	↗	↗	↗	↘	↗	↘	↘	↘
jena								↘	↘	↘	↗	↗
jspwiki									↗	↘	↘	↘

**Table 6** (continued)

Project	06	07	08	09	10	11	12	13	14	15	16	17
knox								↘	↘	↗	↘	↘
kylin										↗	↗	↘
lens										↘	↗	↘
mahout				↗	↘	↘	↘	↗	↗	↗	↗	↗
manifoldcf						↘	↗	↗	↗	↗		↘
mina-sshd				↘	↗	↘	↘	↘	↘	↘	↘	↘
nifi											↗	↗
opennlp						↗	↗	↗	↗	↘	↘	↘
parquet-mr								↘	↘	↘	↘	↗
pdfbox				↘	↗	↗	↗	↘	↘	↘	↗	↘
phoenix										↗	↗	↘
ranger										↘	↗	↘
roller								↘	↗	↘		
santuario-java					↘	↘	↘	↘	↗	↗	↗	↗
storm									↗	↗	↗	↗
streams								↘	↘	↗	↘	↗
struts		↘	↗	↗	↘	↘	↘	↘	↘	↘	↘	↘
systemml											↗	↗
tez									↘	↘	↗	↗
tika			↘	↘	↗	↗	↗	↘	↘	↗	↘	↘
wss4j			↘	↘	↘	↘	↗	↘	↘	↗	↗	↘
zeppelin									↗	↗	↘	↘

Archiva removed PMD in 2012 when they moved reporting to a parent pom which did not include PMD anymore<sup>12</sup>. Neither the commit message, nor the project documentation mention whether this removal is accidental or not.

Commons-bcel briefly introduced and then removed PMD in 2008. The removal does not mention PMD or reports of the build system. This brief introduction happened at the same time as the move from Ant to Maven as a build system. This indicates that the developers were testing features of Maven. In 2014 the project included PMD again in its buildfile. The trend of warnings is declining nonetheless.

Commons-compress removed and re-introduced PMD in short order while configuring the build system in multiple commits. PMD is mentioned explicitly in the commit it was re-added.

Commons-dbcj removed PMD in 2014 but that year still shows a declining trend. The commit message states that the removal was due to switching to FindBugs. Although the year is not part of this study, PMD was re-added to the build system in 2018.

Commons-math changed the ASAT configuration in 2009 - 2011 so that there were at least some commits without an active PMD configuration. Therefore, these are colored black in Table 6. Those years also had a declining trend. The commit message indicate that in 2009 the reporting section of which PMD is part of was dropped due to a release and later added again. In 2010 PMD was dropped due to compatibility problems and enabled again in 2011.

<sup>12</sup><https://repo1.maven.org/maven2/org/apache/archiva/archiva-parent/9/archiva-parent-9.pom>

Commons-validator had some commits in 2008 with PMD enabled. The removal was a conscious decision as it is mentioned in the commit message although the reason is missing. PMD was added again in 2014.

Tika removed PMD in 2009 and never re-introduced it. The commit message mentions removing obsolete reporting from the parent pom. This indicates that the developers did not act on reported PMD warnings, either because they ignored them completely or because they found that there were too many false positives.

We are only considering projects where we can determine the time when PMD was introduced. If it was either introduced together with Maven as a build tool or was introduced before Maven we do not consider it here. We consider projects which introduced PMD and at least used it for a full year afterwards, i.e., a black arrow followed by a green arrow in Table 6. The short term impact as estimated by the trend of warning density for 15 projects where PMD was used at least once is declining in 9 projects while 6 projects have an increase in warning trend. While we expected to see a drop in ASAT warnings after introduction of PMD, this is only the case in 9 of the 15 projects we consider here. An explanation for this result could be that the developers introduce the ASAT but do not immediately scrutinize enough code to make a difference in the short term.

*RQ2.1 Summary:* The short term impact of PMD on warning density trends is positive in 9 of 15 projects. Not every project is immediately resolving enough warnings after introducing PMD to affect the trend of the year.

### 5.2.2 RQ2.2: What is the long term impact of PMD on the number of ASAT warnings?

Table 7 shows the number of ASAT warnings over the projects lifetime from the point in time where Maven was used as a build system, i.e. the point in time where we are sure that we can capture the effective rules of the ASAT.  $S$  is the plain sum of the number of warnings,  $R$  is the warning density.  $R+t$  is the warning density with time-correction where we only count the warnings that PMD supported at the time of the commit.  $R+d+t$  is the warning density with only the rules counted that the Maven PMD plugin has enabled by default with time-correction.  $R+e+t$  is the warning density with only the rules counted that are definitely enabled via the parsed Maven configuration file, i.e., the most exact and only available in projects where we have the PMD plugin enabled in the Maven configuration file. Figure 3 visualizes this information using the project Commons-lang as an example. The red line represents the number of rules considered, the blue line is the number of ASAT warnings which is a sum ( $S$ ) in the first subplot and warning density ( $R$ ) in all following subplots. The orange line is the regression line. The number of rules is constant if no time-correction is applied. Figure 3 also shows that for the effective ruleset, we only count from the point of inclusion of the ASAT. Otherwise we would skew the data in this case. We should also mention that jumps in effective rules can be due to inclusion of new rules by the developers or by inclusion of new rules for PMD due to group expansion, i.e., the project configures all rules for category A, PMD adds new rules for category A at that point in time which results in rising number of rules considered.

A first interesting result is that if we look at the warning density, there is a downward or neutral trend for all but 13 projects. This means independent of the presence of PMD in the buildfile the overall quality of the code per kLLOC with regards to ASAT warnings improves in most projects. This could be for example through changes in coding style coinciding with some ASAT rules, e.g., no if statement without curly braces. The number of projects

**Table 7** Trends for the number of ASAT warnings ( $S$ ), warning density ( $R$ ), time-corrected ( $R+t$ ), default rules ( $R+d+t$ ) and effective rules ( $R+e+t$ ) over all years after Maven introduction

Project	$S$	$R$	$R+t$	$R+d+t$	$R+e+t$
archiva	↗	↗	↗	↗	↘
calcite	↗	↗	↗	↗	→
cayenne	↗	↗	↗	↗	↗
commons-bcel	↗	↘	↗	↗	↘
commons-beanutils	↗	↘	↗	↗	→
commons-codec	↗	↗	↗	↘	↘
commons-collections	↗	↘	↗	↘	↘
commons-compress	↗	↗	↗	↘	↘
commons-configuration	↗	↘	↘	↘	→
commons-dbc	↗	↘	↗	→	→
commons-digester	↗	↘	↗	↘	↘
commons-imaging	↘	↘	↘	→	↗
commons-io	↗	↘	↗	↗	→
commons-jcs	↗	↘	↗	↘	↘
commons-jexl	↗	↘	↗	↘	↘
commons-lang	↗	↘	↗	↘	↘
commons-math	↗	↘	↗	↘	↗
commons-net	↗	↘	↗	↗	→
commons-rdf	↗	↗	↗	↗	↗
commons-scxml	↗	↘	↗	↗	→
commons-validator	↗	↘	↗	↘	↗
commons-vfs	↗	→	↗	↘	↘
eagle	↗	↘	↘	↗	→
falcon	↗	↘	↘	↗	→
flume	↗	↘	↗	→	→
giraph	↗	↘	→	↗	→
gora	↗	↗	↗	↘	→
helix	↗	↗	↗	→	→
httpcomponents-client	↗	↘	↗	↘	→
httpcomponents-core	↗	↗	↗	↗	→
jena	↗	↘	↘	↘	↘
jspwiki	↗	↘	↗	↘	→
knox	↗	↘	↘	↗	→
kylin	↗	↘	↘	↗	→
lens	↗	↘	↘	↗	→
mahout	↗	↘	↗	↘	↘
manifoldcf	↗	↘	↗	↗	→
mina-sshd	↗	↘	↘	↗	→

**Table 7** (continued)

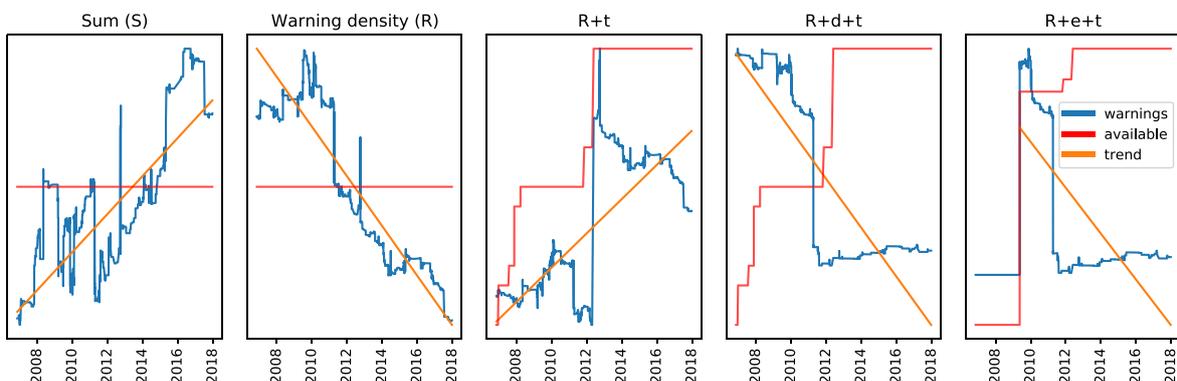
Project	<i>S</i>	<i>R</i>	<i>R+t</i>	<i>R+d+t</i>	<i>R+e+t</i>
pdfbox	↗	↘	↘	↘	→
phoenix	↗	↗	↗	↗	→
ranger	↗	↘	↘	↘	↗
roller	↗	↘	↗	↘	→
santuario-java	↗	↘	↘	↘	↘
storm	↗	↗	↗	↗	↗
streams	↗	↘	↘	↗	→
struts	↗	↘	↗	↗	→
systemml	↗	↘	↘	↗	→
tez	↗	↘	↗	↗	→
tika	↗	↗	↗	↗	↘
wss4j	↗	↘	↗	↘	↘
zeppelin	↗	↘	↘	↗	→

is higher than in the previous Section 5.1 where we considered the complete lifetime of the project. In Section 5.1, we observed a rising trend of warning density in only 10 projects.

If we only consider the effective rules (*R+e+t*) the picture is not that clear, which means even though the developers have the ability to look at the reports containing these warnings the overall quality per LLoC does not always improve. This could be due to perceived or real false positives of the reporting ASAT which are ignored by the developers.

To answer *RQ2.2* we refer to Table 6 again and note the green trends following the introduction of PMD in black. If we add up the slopes of the subsequent years after the introduction of the ASAT, we can estimate the long term impact. We notice that we have more positive years than negative years in our data following the introduction of an ASAT. Positive years are identified by a decreasing warning density whereas negative years are identified by a increasing warning density. On a more quantitative note we can sum the slopes of years following the introduction of the ASAT which we report in Table 8. We are not listing mina-sshd even though it uses PMD because it was only introduced in 2017 which is the last year of our data, therefore it is excluded from the long term impact analysis.

Table 8 shows the median change in warning density per year, e.g., commons-lang decreases the number of warnings per kLLoC by 1.7 per year, which is almost the same than its overall decrease over all years (1.8) which can be seen in Table 4. We can also see



**Fig. 3** Example of ASAT warning trends (Commons-lang)

that the average change per project is 2.3 which is less than the mean over all projects over all years reported in Table 4 which was 3.5. Nevertheless the projects predominantly show a negative median which indicates a positive trend in the number of ASAT warnings, i.e., warnings decrease. Only 5 projects, commons-rdf, commons-beanutils, commons-validator, cayenne and commons-imaging have a positive median, i.e., a majority of negative trends of warning density after PMD was introduced. The long term impact as estimated by the trend of warning density is positive in 19 of 24 projects. On average each project removed 2.3 warnings per kLLOC each year after PMD was introduced in the buildfile. Thus we can further conclude that the long term impact of PMD on warning density is better by trend alone than the observed short term impact. Although, its impact is weaker than the overall trend of warning density which encompasses the years where PMD was not present in the buildfile. This may be an effect of changing of coding style as the rules that changed the most are related to naming and style (see Section 5.1.2).

*RQ2.2 Summary:* The long term impact of PMD on warning density trends is positive in 19 of 24 projects. The majority of projects reduce their warning density in the years following PMD introduction.

**Table 8** Warning density without overlapping rules median change per year (MEDR+o p.a.) after PMD introduction

Project	MEDR+o p.a.
archiva	-2.2656
cayenne	0.75534
commons-bcel	-15.828
commons-beanutils	0.61942
commons-codec	-0.38184
commons-collections	-0.65821
commons-compress	-1.5768
commons-dbc	-1.7514
commons-digester	-0.18101
commons-imaging	0.62411
commons-jcs	-2.3598
commons-jexl	-1.4328
commons-lang	-1.7042
commons-math	-0.96496
commons-rdf	2.8955
commons-validator	1.4614
commons-vfs	-0.19471
jena	-0.9757
mahout	-0.51027
range	-5.524
santuario-java	-1.5697
stor	-6.735
tika	-14.176
wss4j	-2.5572
Mean	-2.2913

### 5.2.3 RQ2.3: Does the active usage of custom rules for PMD correlate with higher ASAT warning removal?

We first extract all changes to the buildfile and specifically to the custom rules as shown in Table 9. It shows the number of rules changed over the project lifetime and the number of commits where the build file or a configuration file was changed. To give a perspective regarding the number of commits we additionally included the mean and median for rule and build changes.

The sum of rule changes is the sum of all deltas of rule changes, this includes additions and removals of rules. As we also count the default rules when the ASAT is introduced there is a minimum of 45 rules that are changed if PMD is introduced and there are no custom rules right from the start. If custom rules are added or removed later this number increases.

The true relation to the trends can be seen in Table 10, it shows the correlation between the number of rule changes to the general trend of ASAT warnings of the project over each year where the ASAT was used. The results for this research question could be seen as inevitable because we do not have a lot of rule changes. Nevertheless, we find that 12 of 25 projects have at least performed some changes to their PMD rulesets.

While this may sound discouraging to developers, we note that while the correlation is negligible it retains a negative sign for both correlation measures. This means that while rule changes increase the warning density decreases. However, other factors are probably also important for developers which profit from a well maintained rule set, e.g., acceptance of the ASAT by other developers.

*RQ2.3 Summary:* The impact of rule changes, i.e., an active, evolving ASAT configuration on the ASAT warning trends is negligible. Nevertheless, 12 of 25 projects changed their rules at least once.

### 5.2.4 RQ2.4: Is there a difference in ASAT warning removal trends whether PMD is included in the build process or not?

We first have to split our data into two groups, one group contains all years from all projects where PMD was used as indicated by its inclusion in the buildfile, and the other contains all the other years. We then investigate our two samples for differences. We want to know if the two groups have different warning trends and if this difference is statistically significant. We now describe the groups of warnings considered here. The  $R+t$  contains the time corrected warning density which contains all possible warnings that were available to the developers at the time of the commit, i.e., a warning added in 2017 would not be included in the warning density of a commit in 2016.  $R+d+t$  contains only the warning density of only the default rules that are enabled by PMD, they are also time corrected.  $R+e+t$  contains the time corrected effective rules, this contains the default rules except in cases where developers added custom rule sets. If custom rule sets are found, they are used exclusively.  $R+e+t+o$  contain the time corrected effective rules without overlapping rules. We subtract PMD warnings which are also reported by other ASATs if the project uses them, we consider FindBugs and Checkstyle. This removes possible influence in the no PMD group.

The Tables 11, 12, 13, 14 show a difference in the trends of the ASAT warnings between non PMD usage and PMD usage but it is not statistically significant. Table 15 completes the reporting for the Mann-Whitney-U test, it includes sample sizes and the median of the samples. The sample sizes are changing because we remove incomplete years of ASAT usage, overlapping rules and we also remove insignificant trends as described in Section 4.3.

**Table 9** Rule and build changes for each project

Project	Rule changes			Build changes		
	Sum	Mean	Median	Sum	Mean	Median
archiva	45	0.01	0.00	145	0.02	0.00
calcite	0	0.00	0.00	90	0.05	0.00
cayenne	130	0.03	0.00	80	0.02	0.00
commons-bcel	87	0.06	0.00	21	0.02	0.00
commons-beanutils	45	0.04	0.00	28	0.03	0.00
commons-codec	94	0.05	0.00	35	0.02	0.00
commons-collections	45	0.01	0.00	25	0.01	0.00
commons-compress	133	0.05	0.00	65	0.03	0.00
commons-configuration	0	0.00	0.00	50	0.02	0.00
commons-dbc	45	0.03	0.00	42	0.02	0.00
commons-digester	45	0.04	0.00	52	0.04	0.00
commons-imaging	100	0.10	0.00	37	0.04	0.00
commons-io	0	0.00	0.00	53	0.03	0.00
commons-jcs	45	0.03	0.00	65	0.05	0.00
commons-jexl	44	0.03	0.00	96	0.07	0.00
commons-lang	45	0.01	0.00	73	0.02	0.00
commons-math	62	0.01	0.00	52	0.01	0.00
commons-net	0	0.00	0.00	15	0.01	0.00
commons-rdf	45	0.17	0.00	22	0.08	0.00
commons-scxml	0	0.00	0.00	31	0.04	0.00
commons-validator	45	0.03	0.00	25	0.02	0.00
commons-vfs	45	0.02	0.00	61	0.03	0.00
eagle	0	0.00	0.00	48	0.06	0.00
falcon	0	0.00	0.00	57	0.03	0.00
flume	0	0.00	0.00	40	0.03	0.00
giraph	0	0.00	0.00	27	0.03	0.00
gora	0	0.00	0.00	42	0.09	0.00
helix	0	0.00	0.00	164	0.09	0.00
httpcomponents-client	0	0.00	0.00	78	0.03	0.00
httpcomponents-core	0	0.00	0.00	90	0.03	0.00
jena	45	0.02	0.00	111	0.04	0.00
jspwiki	0	0.00	0.00	38	0.00	0.00
knox	0	0.00	0.00	96	0.07	0.00
kylin	0	0.00	0.00	57	0.02	0.00
lens	0	0.00	0.00	49	0.06	0.00
mahout	127	0.04	0.00	261	0.08	0.00
manifoldcf	0	0.00	0.00	64	0.03	0.00
mina-sshd	78	0.04	0.00	123	0.07	0.00
nifi	0	0.00	0.00	133	0.05	0.00
opennlp	0	0.00	0.00	124	0.07	0.00
parquet-mr	0	0.00	0.00	60	0.10	0.00
pdfbox	0	0.00	0.00	61	0.01	0.00
phoenix	0	0.00	0.00	49	0.02	0.00

**Table 9** (continued)

Project	Rule changes			Build changes		
	Sum	Mean	Median	Sum	Mean	Median
ranger	43	0.02	0.00	70	0.03	0.00
roller	0	0.00	0.00	10	0.00	0.00
santuario-java	193	0.07	0.00	79	0.03	0.00
storm	45	0.19	0.00	17	0.07	0.00
streams	0	0.00	0.00	76	0.20	0.00
struts	0	0.00	0.00	155	0.04	0.00
systemml	0	0.00	0.00	68	0.01	0.00
tez	0	0.00	0.00	43	0.02	0.00
tika	45	0.02	0.00	98	0.03	0.00
wss4j	50	0.02	0.00	96	0.04	0.00
zeppelin	0	0.00	0.00	77	0.03	0.00

The reason we found no significant difference could be that the changes resulting from ASAT usage in the buildfile are too small when considering the general number of changes developers apply due to normal code maintenance work. To remove potential influences from overlapping rules of PMD with FindBugs and Checkstyle, we removed them prior to the test in Table 14. This did not change the results significantly. Overall, we can see that the results are not significant, even as we get more detailed, i.e., from just all rules to only the effective rules with removed overlapping rules from other ASATs.

*RQ2.4 Summary:* The presence of PMD in the build process has no significant effect on the warning removal trends of warning density.

These results are surprising but looking at our data we can see that there is an effect of PMD usage, just not in a general code quality sense by utilizing the warning density. If we look at the raw sum of ASAT warnings which we have seen to increase in almost all projects, we can detect an effect. In Fig. 4, we can see that for most projects the slope of the trend of warning density per year is near 0 whereas for the non PMD using years it is higher.

A possible explanation for this data is, that projects which utilize PMD scrutinize most of the new code, which results in a rising trend of ASAT warnings but only slightly due to some left over warnings or ignored false positive warnings. In contrast, for projects which do not utilize PMD the trend of the sum of ASAT warnings is rising more steeply.

As shown in Table 16 in case we utilize the sum of all ASAT warnings, the difference is significant, albeit small.

**Table 10** Correlation between number of rule changes and warning density

Method	Value	P-value
Kendall's $\tau$	-0.17089	0.037058
Spearman's $\rho$	-0.20872	0.042373

**Table 11** Warning density, time corrected rules ( $R+t$ ) significance test prerequisites and results

Test	Sample	Test Statistic	P-value
Shapiro-Wilk	No PMD	0.43633	1.1615e-20
Shapiro-Wilk	PMD	0.17532	6.5589e-31
Levene	Both	0.52831	0.46778
Mann-Whitney-U	Both	1.6344e+04	0.61671

**Table 12** Warning density, only default time corrected rules ( $R+d+t$ ) significance test prerequisites and results

Test	Sample	Test Statistic	P-value
Shapiro-Wilk	No PMD	0.13952	9.8708e-25
Shapiro-Wilk	PMD	0.12195	1.2032e-31
Levene	Both	0.34585	0.55683
Mann-Whitney-U	Both	1.4763e+04	0.099212

**Table 13** Warning density, effective time corrected rules ( $R+e+t$ ) significance test prerequisites and results

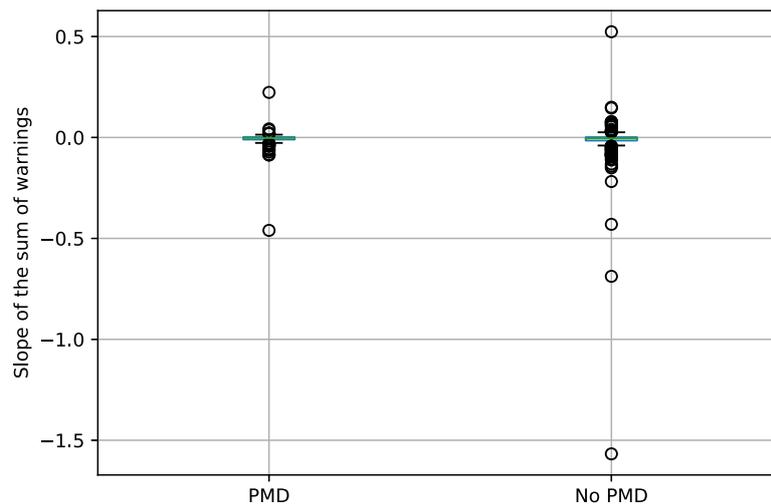
Test	Sample	Test Statistic	P-value
Shapiro-Wilk	No PMD	0.19915	5.1652e-24
Shapiro-Wilk	PMD	0.12195	1.2032e-31
Levene	Both	0.0050551	0.94336
Mann-Whitney-U	Both	1.5212e+04	0.20141

**Table 14** Warning density, effective time corrected rules without overlap ( $R+e+t+o$ ) significance test prerequisites and results

Test	Sample	Test Statistic	P-value
Shapiro-Wilk	No PMD	0.13609	5.0062e-24
Shapiro-Wilk	PMD	0.64262	2.8805e-20
Levene	Both	0.4575	0.49928
Mann-Whitney-U	Both	1.2067e+04	0.22387

**Table 15** Mann-Whitney-U reporting

Rules	Samples	Size	Median
Sum ( $S$ )	No PMD	241	0.13086
	PMD	136	0.02793
$R+t$	No PMD	236	-0.00093
	PMD	136	-0.00050
$R+d+t$	No PMD	236	-0.00003
	PMD	136	-0.00009
$R+e+t$	No PMD	236	-0.00003
	PMD	136	-0.00002
$R+e+t+o$	No PMD	200	-0.00004
	PMD	127	-0.00002



**Fig. 4** Slope of the sum ( $S$ ) of all ASAT warnings for PMD and non PMD years over all projects

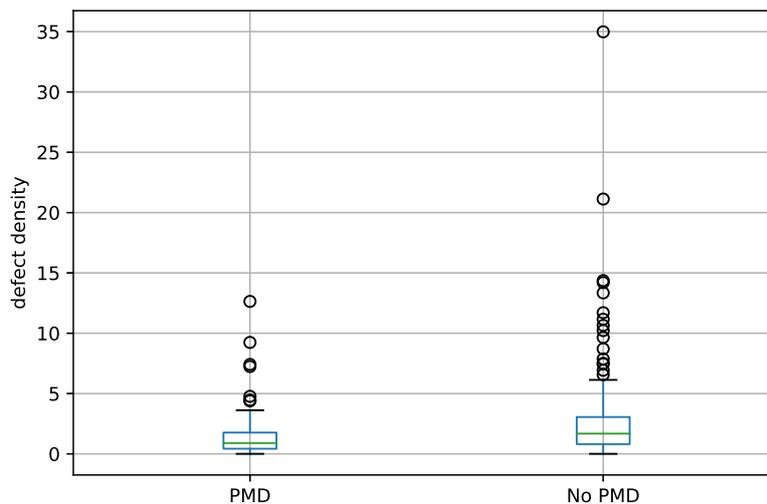
### 5.2.5 RQ2.5: Is there a difference in defect density whether PMD is included in the build process or not?

We extract issues for all projects from the ITS created in a certain year and then calculate the defect density as described in Section 4.4.3. We then build two groups again for years of development where PMD was used and compare it to the second group of years of development where PMD was not used. Instead of the slope of the ASAT warning trends, we now compare the defect densities of the two groups. We only include years in which PMD was included in the buildfile for every commit or for none to mitigate problems of partial use. The defect density contains only issues marked as a bug by the developers, so we discard improvements, documentation changes. Moreover, the issue type used is the one at the end of the data collection. If an issue was misclassified and the classification was changed at some point, the changed classification is the one we use. This also removes duplicate bug reports from the data, if the developers marked the duplicate bug report as duplicate or invalid as is customary in that case.

Figure 5 shows the defect densities of the two groups, we can see that PMD using years have a slight advantage of less defect density. Table 17 contains the significance test for whether there is a difference between the two groups and its prerequisites. We can see that years in which PMD is present in the buildfile show a statistically significant difference of defect densities. To complete the reporting of the Mann-Whitney-U test in Table 17, the sample sizes are 132 and 249 for years where PMD was used and years where PMD was not used. The respective median values are 0.89500 and 1.67743. Resulting in a difference in median of 0.78243 between both samples. The 95% confidence interval of the difference in median is (0.36038, 0.86744).

**Table 16** Sum ( $S$ ) of all ASAT warnings significance test prerequisites and results

Test	Sample	Test Statistic	P-value
Shapiro-Wilk	No PMD	0.3517	5.7853e-22
Shapiro-Wilk	PMD	0.5162	2.9655e-25
Levene	Both	0.97435	0.32423
Mann-Whitney-U	Both	1.2919e+04	0.00032041
Effect size	Both	0.17583	-



**Fig. 5** Defect density for PMD and non PMD years over all projects

*RQ2.5 Summary:* Years in which PMD is present in the build process have a lower defect density (about 0.78 less defects per 1000 LLoC in median). The difference is statistically significant, albeit the effect size is small.

This result is very coarse grained. We consider defect density per year which can only hint at a correlation instead of a direct causal relation. However, the ASAT we consider in this study contains a broad set of rules some of which also pertain to more generic maintainability and readability best practices. This may have a more indirect or long term effect on the quality, which is why we decided to include *RQ2.5* in this way. However, to further validate this result and include confounding factors we build a regularized linear regression model which includes these factors to see if PMD usage still is of importance. To this end we enhanced the available data with additional features per project per year. We include the number of commits, the number of distinct developers, the year, the number of commits in which PMD was used / not used and the project name as a number. As a popularity proxy we include the Github information from that project, namely stars and forks. We train the linear regression model with this data and give the resulting coefficients in Table 18.

We can see that the regularization of the model removes the project number, and the year as well as the number of commits where PMD was used (although, we note the negative sign). The number of forks, commits, authors, and stars are more important and not removed by the regularization. The most important feature is the number of commits in which PMD was not used which indicates that it is an important factor when determining defect density. We also note that except for the number of commits in which PMD was used we retain positive signs on the coefficients of the model. The interpretation is that these factors have

**Table 17** Defect densities of PMD and non PMD years significance test

Test	Sample	Test Statistic	P-value
Shapiro-Wilk	No PMD	0.56878	1.9853e-24
Shapiro-Wilk	PMD	0.6299	9.5835e-17
Levene	Both	7.6372	0.0059968
Mann-Whitney-U	Both	1.1054e+04	7.2559e-08
Effect size	Both	0.26943	-

a detrimental effect on defect density, i.e., as #stars or #commits without PMD increases, so does defect density.

## 6 Discussion

The sum of ASAT warnings is increasing in most of our projects. As the number of ASAT warnings is correlated with the logical lines of code as shown in Table 3 this is not surprising. The rising size of the projects is in line with the rules of software evolution (Lehman 1996) which claim that E-Type software<sup>13</sup> continues to increase in size. There is no theory for explaining the continued growth of ASAT warnings but it could be interpreted as an indication that some warnings are ignored by developers because they may be deemed unnecessary or false positives (warnings in code without problems). An increasing number of ASAT warnings is also supported by the raw data provided by (Marcilio et al. 2019) in their replication kit. Although Marcilio et al. investigate real usage of Sonarcube by developers and primarily the resolution times of ASAT warnings, they provide the dates where ASAT warnings are opened and closed. After transforming their data to show the sum of open ASAT warnings for given days it shows rising sums of ASAT warnings for almost all of the projects. Furthermore, current research found that only a small fraction of ASAT warnings are fixed by developers (Liu et al. 2018; Marcilio et al. 2019; Kim and Ernst 2007b).

Table 5 provide us with additional interesting insights. First of all, code quality, if measured by warning density, is increasing. Second, the different types of ASAT warnings evolve differently. The order of the ASAT removal trends provided in Table 5 shows which types of ASAT warnings developers removed most in our candidate projects from 2001-2017. This provides us with a hint of what issues developers deemed most important in that timeframe. As this first part of our study is independent of ASAT usage, we can not quantify the influence of PMD or other ASATs, i.e., Checkstyle or FindBugs. Nevertheless, the results show that a certain importance is assigned to code readability and maintainability by the developers. This result is in line with research by (Beller et al. 2016) who found that the majority of actively enabled and disabled rules are maintainability-related. Beller et al. studied configuration changes. Our work expands on the work of Beller et al. and confirms that not only were the rules more often changed for maintainability related warnings, they were also globally resolved the most. This finding is also supported by (Zampetti et al. 2017) who analyzed CI build logs for ASAT warnings. They found that most builds break because of coding standard violations. However, checking adherence to coding standards via CI quality gates is an industry practice, which is probably a contributing factor. The most frequently fixed warnings found by (Marcilio et al. 2019) also contain rules regarding naming conventions and coding style. The only groups of ASAT warnings for which we found more introductions than removals in our trend analysis were import and clone rules. If we measure by warning density delta between the first and last commit of our analysis period, the only increasing rules are import and finalizer rules. However, as the number of rules here is small and the delta of the change is also small we can not draw any conclusions from this result.

As previously explained we think of code containing less ASAT warnings per line as higher quality code. In this case study we found that the warning density is decreasing, i.e.,

---

<sup>13</sup>Program that performs real world activity, needs to continuously adapt to new requirements and circumstances.

**Table 18** Linear regression coefficients of the defect density model

Name	Coefficient
Project number	0.000000
Year	-0.000000
#commits with PMD	-0.000000
#forks	0.049105
#commits	0.064325
#authors	0.097168
#stars	0.106038
#commits without PMD	0.217620

the overall quality is increasing. This is a positive result for the studied open source projects and may also be a positive result for software development in general.

However, we could not measure a significant difference between the trend of warning density in years of development between PMD usage and no PMD usage. Although, if we do not consider warning density but the raw sum of warnings there is indeed a measurable, significant difference. This could be a result of a flattening trend of ASAT warning removal after some time which means the LLoC then becomes the dominating factor of the warning density equation. This can be seen as evidence of industry best practices like utilizing static analysis tools only on new code as reported by Google (Sadowski et al. 2018) and Facebook (Dis-tefano et al. 2019). Further evidence of this best practice is shown in the results of short and long term impact of PMD. We found that shortly after the introduction of PMD only 9 of 15 projects show decreasing warning density whereas 19 of 24 projects show decreasing warning density in the years following PMD introduction. To the best of our knowledge this would be the first publication to empirically find this effect in open source projects.

As a result of its behavior with regards to the project source code over longer time frames, warning density should be handled with care by researchers including effort aware models using ASAT warnings. A more targeted warning density as used by Panichella et al. (2015) on the other hand is less problematic. Panichella et al. used warning density targeted on code reviews not on the whole project, therefore avoiding the problem of increasing LLoC on warning density. Nevertheless, if we want to rank projects by the number of warnings per kLLoC the warning density approach is still viable.

In our ruleset analysis, we found that the number of rule changes does not correlate with the trend of ASAT warnings. Additionally, we found that only a limited number of rule changes in the study subjects are performed. This is also supported by Beller et al. (2016) who found that most configuration files never change. We are now able to expand on the work of Beller et al. and show that there is no direct correlation between ruleset changes and the observable trend of ASAT warnings.

For practitioners, the most interesting result of this study is that defect density, which we use as a proxy for external software quality, is lower when PMD is included in the build-file. This is also in line with related research from Plosch et al. (2008) who found a positive correlation between defects and the number of ASAT warnings by PMD. Although in our case we are not talking about correlations but differences in reported defects. The reported difference in defect density may not necessarily be a result of using PMD and removing its reported warnings but could also be an effect of the developers keeping the codebase healthy, which results in the usage of static analysis tools and subsequently to lower defect densities. Nevertheless, this may serve as a further indication that ASAT warnings and static analysis has a positive impact on software quality evolution. Initial results by Querel and

Rigby (2018) show that including static analysis warnings can improve bug prediction models. This is an indication that our results for *RQ2.5* may also hold in a more direct way, i.e., improving direct bug prediction instead of defect density prediction.

Our investigation of PMD removal in our study subjects revealed multiple cases where PMD was removed but the removal not explicitly mentioned, e.g., build system reconfigurations, switching parent POMs. Moreover, some study subjects do not change the PMD default rules. It seems that although some developers advocate static analysis tools like PMD there is no strategy encompassing documentation, continuous integration or integration of project specific rules for local IDEs. Thus, we recommend adopting a strategy concerning static analysis tools which includes documenting the tools and reasons for inclusion, which rules are enabled for all tools and how the code is checked in different contexts, e.g., continuous integration, code review or local development.

Our study revealed a general decrease in warning density. This may be a result of our chosen ASAT as it supports a wide range of rules. Other researchers that focused on security related ASATs come to a different conclusion regarding warning density. Penta et al. (2009) found that warning density stays roughly constant in their study. More recently, Aloraini et al. (2019) also found similar constant warning density for security related ASATs. This may indicate that security warnings are harder to find for developers or require specialized knowledge that fewer developers have. In our study, we found that a lot of brace and naming rule related warnings were addressed in our study subjects. This effect may also have been due to changing or adopting coding standards for Java and may contribute to our finding of declining warning density. However, they were not the only contributing rules to the decline. Our data shows almost every type of rule contributes to the trend of declining warning density.

## 7 Threats to Validity

In this section, we discuss the threats to validity we identified for our work. To structure this section we discuss four basic types of validity separately, as suggested by Wohlin et al. (2000).

### 7.1 Construct Validity

Construct validity is concerned with the relation between theory and observation. In our retrospective case study the main source for this threat is due to the observations, i.e., measurements over the course of the change history of our study subjects. Static analysis warning evolution in test code may be different than in production code, to mitigate this source of noise in our measurements we excluded all non-production code for the measurements. To validate our exclusion filter we randomly sampled 1% of the commits of each study subject and the first author manually inspected the changed files for misclassified production files. Out of 3322 production files 3 were misclassified. Out of 1614 non-production files none were misclassified.

While some of our data is newer we decided on a common cut off date to simplify the data. To evaluate the impact this has on our results we performed the analysis with some additional data we have available, i.e. projects for which we have data from 2019. As we only use full years of development we would have to cut off 2019 and would be left with one additional year for some projects. We ran the analysis pipeline again and instead of a hard cut off date we just remove the last year. This leaves us with an additional year for 14 projects. The differences are small, the correlation between LLoC and warning density increases by 0.015. The mean change per year changes by 0.03, the mean change per year

after PMD introduction changes by 0.2, we add an additional project here, mina-sshd, which introduced PMD in 2018. The basic trends and statements are the same with the additional year. That being said, we have not included the additional year in our final analysis. We believe that a fixed cutoff date that is the same for all projects is more appropriate. Otherwise, the description of the analysis would get more complex unnecessarily for almost no benefit.

Changes in projects with release branches, e.g., commons-math, may be applied to the release branch as well as the master branch. We mitigate this threat of duplicate measurements by only utilizing a single path through the commit graph. Considering *RQ2*, the time corrected rules rely on the release date extracted from the PMD changelog. This would in effect mean that as soon as a new PMD version is released the study subjects would be able to see the new rules. This may not always be a realistic scenario due to delayed updates of, e.g., maven-pmd-plugin. However the data that is available to us does not allow to mitigate this. The extraction of the effective rules was not possible for every commit due to problems with the Maven buildfile, e.g., XML errors or unavailable parent POMs. The buildfile parsing failed for 1361 commits, out of these, 39 errors are due to XML and maven parse errors, 26 errors were due to missing pom.xml files (can happen when the repository is moved but the new folder is not added), 74 due to missing child pom.xml (this happens when the project consists of multiple pom.xml for different modules and the parent references a non existing child) and 1012 errors due to missing parent pom.xml. The last is due to either missing parent pom.xml in the Maven repository or due to a module within the same project not finding its local parent pom.xml. This happens often when a change increases the version of the local parent pom.xml but does not change the referenced parent version in the other modules.

As there is no way to mitigate this automatically, the rules are assumed to be unchanged for these commits and are changed when the buildfile can be parsed again if there was a change. To mitigate effects of overlapping static analysis tools, we checked the rules we utilize against the current rulesets of Checkstyle and FindBugs to mark overlapping rules so that we can remove them from the analysis in years and projects where these ASATs are used. The design of our case study and the chosen statistical tests may influence the results. We include an extensive description of the analysis method and how we preprocess our data prior to the description of the statistical tests we use, and the reasons we chose them. The statistical tests we utilize in this work depend on their implementation. To mitigate this threat we only rely on well-known and used Python packages scikit-learn (Pedregosa et al. 2011), scipy (Jones et al. 2001) and NetworkX (Hagberg et al. 2008).

## 7.2 Internal Validity

Internal validity is threatened by external influences that we did not, or are not able to consider when trying to infer cause-effect relationships. An external factor we are not able to consider is the usage of tools that are not bound to the Version Control System (VCS), e.g., IDE plugins and cloud services without configurations in the VCS. This has no impact on questions regarding general trends as in *RQ1* because for this kind of question only the “end result”, the code that is available in the VCS, is important. However, for *RQ2* this may interfere with our ability to infer a causal relationship between PMD usage and defect density or warning density. As the external use of tooling without traces in the VCS is not something that we can include in our available data, we restrict our questions and conclusions to PMD usage via buildfiles and not general usage as in IDE plugins or related tooling. We are not able to mitigate this effect with our available data and, therefore, note this here as a limitation to our internal validity.

### 7.3 External Validity

External validity is concerned with the generalizability of the conclusions we draw in this study. As we cannot include every Java project, we depend on our sampling of the existing Java projects. We restricted ourselves to a convenience sample of Java projects managed by the Apache Software Foundation. Nevertheless, our study subjects consist of a diverse set of projects used in different domains to reduce this threat due to the chosen projects.

Furthermore we observe only one ASAT, namely PMD. This restriction is necessary because we cannot rely on all commits in projects being able to compile (Tufano et al. 2017). Other ASATs, e.g., FindBugs need bytecode files which can be problematic if the project is not being able to compile due to missing dependencies. Although this is a limitation of our study, PMD includes a wide range of rules. They range from coding style rules to very specific rules concerned with BigInteger usage in Java.

### 7.4 Conclusion Validity

Threats to conclusion validity include everything which hinders our ability to draw the correct conclusion about relations between our observed measurements. For the complete first research question, we are just counting our collected data. Thus, there should be no threat to conclusion validity. We partially plotted and manually verified data to validate that our extraction works as expected. In *RQ2.4* we are comparing the differences in ranks of two samples, i.e., non-PMD and PMD warning trends via a hypothesis test. The employed hypothesis test, as all hypothesis tests, cannot directly tell us if our assumption is true. We are employing the test under the assumption that there should be a difference and find only a small one. This does not necessarily mean that the difference is really small. The cause for the difference could also be an effect we do not know about. We tried to mitigate this threat by removing overlapping rules of other ASATs which did not yield significant different results. For *RQ2.5*, we created two groups for PMD and non PMD using development years, we show that defect density is slightly smaller in years where PMD was used. Although this is what the data shows it could also be a secondary effect not visible to us, e.g., the projects using PMD have a smaller defect density overall due to being more stable feature wise. For both comparisons of samples we checked the prerequisites for the used statistical test. To correct for the number of statistical tests we employed Bonferroni correction (Abdi 2007).

## 8 Conclusion and Future work

In this work we investigated PMD usage in open source projects in the context of software evolution. We extracted detailed software repository data over multiple years containing static analysis warnings reported, and extracted additional source code metrics. In order to determine if our study subjects remove static analysis warnings, we calculated trends of warning density (the number of ASAT warnings per kLLOC) by fitting a linear regression onto cleaned and preprocessed data. To the best of our knowledge, this is the first longitudinal, commit level study of the evolution of ASAT warnings. Our work complements existing work, which investigated ASAT warnings per warning, by providing a broader, global overview of resolution trends and effects.

To answer our first main research question regarding the evolution of ASAT warnings over time we performed a retrospective case study on a convenience sample of 54 open source projects. We first investigated the evolution of ASAT warnings without taking ASAT

inclusion in buildfiles into account. We found that the general quality of code with regards to ASAT warnings is improving, i.e., warning density is declining. We also found indications of changing coding conventions in our data as the most decreasing types of ASAT warnings were consisting of naming and brace rules. Moreover, we found that on average every project removes 3.5 ASAT warnings per thousand LLoC per year.

To answer our second main research question regarding the impact of using PMD on the trend of ASAT warnings we leveraged our evolution data to provide answers to the short and long term effects. We found that the short term effects were diverse while the long term effects were positive in the majority of our study subjects. After that, we split the data into years of development where PMD was included in the build process and years where it was not included. This was done multiple times with different sets of rules. We compared both populations and performed a statistical test on both samples. The test yielded a surprisingly small difference, i.e., there is no statistical significant difference of using PMD via the build process with regards to the warning density.

We then performed the comparison not on the warning density but on the overall sum of ASAT warnings per commit which is mostly increasing due to its correlation with to the size of the projects and possibly false positive warnings. We found that the difference between years where PMD was used and years where it was not used was significant and that the slopes of ASAT warning trends for years where PMD was used were near zero in most cases. This could be an indication that best practices that were reported by Google (Sadowski et al. 2018) and Facebook (Distefano et al. 2019), i.e., only new code is scrutinized during static analysis are also utilized in open source projects.

To measure the impact of PMD on software quality, we measured defect density as a proxy metric for external software quality. We compared defect density of samples of development years where PMD was used and where PMD was not used. We found a statistically significant difference of defect density between years where PMD was used and years where it was not used. This result shows that for years in which PMD was included in the build process the study subjects had a smaller defect density than in years where PMD was not used. Future work in this topic, aside from increasing the number of projects in our dataset, could encompass inspecting code changes that increase quality as perceived by developers. When we determine changes which the developers perceive as quality increasing, we could measure how many ASAT warnings are removed or introduced in these changes. This would provide a more developer centric viewpoint to complement the defect density view on software quality investigated in this publication.

A subset of data available to us contains manually validated information (Herbold et al. 2019). This information consists of links between commits and bugs and types of bug reports as well as an improved SZZ (Śliwerski et al. 2005) variant to create links between bug fixes and their inducing changes. It would be interesting to measure impact of bug fixing changes on the number of ASAT warnings. This could shed light on how many ASAT warnings may be part of a bug. In our opinion, this would be more within the scope of FindBugs, as PMD contains more general rules. As far as we know there is no study that investigated PMD with validated issue types and improved links to inducing changes.

**Acknowledgments** This work was partly funded by the German Research Foundation (DFG) through the project DEFECTS, grant 402774445. We also want to thank the GWDG Göttingen<sup>14</sup>, as without the usage of their HPC-Cluster the data collection would have taken decades.

---

<sup>14</sup><https://www.gwdg.de>

**Funding** Open Access funding provided by Projekt DEAL.

**Open Access** This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

## Appendix A : Study subjects

Project	Timeframe	#Files	#Commits
archiva	2006-2017	750	7170
calcite	2013-2017	1626	1543
cayenne	2008-2017	3608	3655
commons-bcel	2002-2017	489	1285
commons-beanutils	2002-2017	257	1028
commons-codec	2004-2017	126	1562
commons-collections	2002-2017	531	2885
commons-compress	2004-2017	335	2197
commons-configuration	2004-2017	457	2639
commons-dbcpl	2002-2017	105	1578
commons-digester	2002-2017	315	1143
commons-imaging	2008-2017	491	981
commons-io	2003-2017	234	1908
commons-jcs	2003-2017	559	1288
commons-jexl	2003-2017	149	1207
commons-lang	2003-2017	323	4394
commons-math	2004-2017	1374	5603
commons-net	2003-2017	272	1570
commons-rdf	2015-2017	165	221
commons-scxml	2006-2017	176	760
commons-validator	2003-2017	149	1233
commons-vfs	2003-2017	382	1921
eagle	2016-2017	1801	725
falcon	2012-2017	850	1669
flume	2012-2017	646	973
giraph	2010-2017	1569	876
gora	2011-2017	440	417
helix	2012-2017	823	952
httpcomponents-client	2006-2017	660	2799
httpcomponents-core	2006-2017	747	2592
jena	2013-2017	5669	2120
jspwiki	2001-2017	529	6829
knox	2013-2017	1031	967
kylin	2015-2017	1384	2145
lens	2014-2017	846	763
mahout	2009-2017	1220	3065
manifoldcf	2011-2017	1283	1569

mina-sshd	2009-2017	931	1217
nifi	2015-2017	3993	2165
opennlp	2011-2017	949	1703
parquet-mr	2013-2017	685	501
pdfbox	2009-2017	1192	6512
phoenix	2015-2017	1731	1613
ranger	2015-2017	944	1651
roller	2006-2015	610	2257
santuario-java	2002-2017	660	2583
storm	2012-2017	1897	172
streams	2013-2017	528	313
struts	2007-2017	1953	2628
systemml	2012-2017	1628	3917
tez	2014-2017	1089	1737
tika	2008-2017	988	2724
wss4j	2005-2017	720	2156
zeppelin	2014-2017	563	2185

## Appendix B : PMD rules with groups and severities

Group	Severity	Rule
Basic Rules	Major	Avoid Branching Statement As Last In Loop
Basic Rules	Critical	Avoid Decimal Literals In Big Decimal Constructor
Basic Rules	Major	Avoid Multiple Unary Operators
Basic Rules	Critical	Avoid Thread Group
Basic Rules	Major	Avoid Using Hard Coded IP
Basic Rules	Critical	Avoid Using Octal Values
Basic Rules	Minor	Big Integer Instantiation
Basic Rules	Minor	Boolean Instantiation
Basic Rules	Critical	Broken Null Check
Basic Rules	Critical	Check Result Set
Basic Rules	Critical	Check Skip Result
Basic Rules	Critical	Class Cast Exception With To Array
Basic Rules	Minor	Collapsible If Statements
Basic Rules	Critical	Dont Call Thread Run
Basic Rules	Critical	Dont Use Float Type For Loop Indices
Basic Rules	Critical	Double Checked Locking
Basic Rules	Critical	Empty Catch Block
Basic Rules	Minor	Empty Finally Block
Basic Rules	Major	Empty If Stmt
Basic Rules	Minor	Empty Statement Block
Basic Rules	Minor	Empty Statement Not In Loop
Basic Rules	Minor	Empty Static Initializer
Basic Rules	Major	Empty Switch Statements
Basic Rules	Major	Empty Synchronized Block

Basic Rules	Major	Empty Try Block
Basic Rules	Critical	Empty While Stmt
Basic Rules	Minor	Extends Object
Basic Rules	Minor	For Loop Should Be While Loop
Basic Rules	Critical	Jumbled Incrementer
Basic Rules	Critical	Misplaced Null Check
Basic Rules	Critical	Override Both Equals And Hashcode
Basic Rules	Critical	Return From Finally Block
Basic Rules	Major	Unconditional If Statement
Basic Rules	Minor	Unnecessary Conversion Temporary
Basic Rules	Critical	Unused Null Check In Equals
Basic Rules	Critical	Useless Operation On Immutable
Basic Rules	Minor	Useless Overriding Method
Brace Rules	Minor	For Loops Must Use Braces
Brace Rules	Minor	If Else Stmts Must Use Braces
Brace Rules	Minor	If Stmts Must Use Braces
Brace Rules	Minor	While Loops Must Use Braces
Clone Implementation Rules	Major	Clone Throws Clone Not Supported Exception
Clone Implementation Rules	Critical	Proper Clone Implementation
Controversial Rules	Minor	Assignment In Operand
Controversial Rules	Major	Avoid Accessibility Alteration
Controversial Rules	Minor	Avoid Prefixing Method Parameters
Controversial Rules	Major	Avoid Using Native Code
Controversial Rules	Minor	Default Package
Controversial Rules	Major	Do Not Call Garbage Collection Explicitly
Controversial Rules	Major	Dont Import Sun
Controversial Rules	Minor	One Declaration Per Line
Controversial Rules	Major	Suspicious Octal Escape
Controversial Rules	Minor	Unnecessary Constructor
Design Rules	Minor	Abstract Class Without Abstract Method
Design Rules	Minor	Abstract Class Without Any Method
Design Rules	Critical	Assignment To Non Final Static
Design Rules	Minor	Avoid Constants Interface
Design Rules	Major	Avoid Instanceof Checks In Catch Clause
Design Rules	Minor	Avoid Protected Field In Final Class
Design Rules	Minor	Avoid Protected Method In Final Class Not Extending
Design Rules	Minor	Avoid Reassigning Parameters
Design Rules	Minor	Avoid Synchronized At Method Level
Design Rules	Critical	Bad Comparison
Design Rules	Minor	Class With Only Private Constructors Should Be Final
Design Rules	Critical	Close Resource
Design Rules	Critical	Constructor Calls Overridable Method
Design Rules	Minor	Default Label Not Last In Switch Stmt
Design Rules	Major	Empty Method In Abstract Class Should Be Abstract

Design Rules	Critical	Equals Null
Design Rules	Minor	Field Declarations Should Be At Start Of Class
Design Rules	Minor	Final Field Could Be Static
Design Rules	Major	Idempotent Operations
Design Rules	Minor	Immutable Field
Design Rules	Major	Instantiation To Get Class
Design Rules	Minor	Logic Inversion
Design Rules	Critical	Missing Break In Switch
Design Rules	Minor	Missing Static Method In Non Instantiatable Class
Design Rules	Critical	Non Case Label In Switch Statement
Design Rules	Critical	Non Static Initializer
Design Rules	Critical	Non Thread Safe Singleton
Design Rules	Major	Optimizable To Array Call
Design Rules	Critical	Position Literals First In Case Insensitive Comparisons
Design Rules	Critical	Position Literals First In Comparisons
Design Rules	Major	Preserve Stack Trace
Design Rules	Major	Return Empty Array Rather Than Null
Design Rules	Minor	Simple Date Format Needs Locale
Design Rules	Minor	Simplify Boolean Expressions
Design Rules	Minor	Simplify Boolean Returns
Design Rules	Minor	Simplify Conditional
Design Rules	Major	Singular Field
Design Rules	Major	Switch Stmtns Should Have Default
Design Rules	Minor	Too Few Branches For A Switch Statement
Design Rules	Minor	Uncommented Empty Constructor
Design Rules	Minor	Uncommented Empty Method
Design Rules	Minor	Unnecessary Local Before Return
Design Rules	Critical	Unsynchronized Static Date Formatter
Design Rules	Major	Use Collection Is Empty
Design Rules	Critical	Use Locale With Case Conversions
Design Rules	Critical	Use Notify All Instead Of Notify
Design Rules	Minor	Use Varargs
Finalizer Rules	Major	Avoid Calling Finalize
Finalizer Rules	Minor	Empty Finalizer
Finalizer Rules	Critical	Finalize Does Not Call Super Finalize
Finalizer Rules	Minor	Finalize Only Calls Super Finalize
Finalizer Rules	Critical	Finalize Overloaded
Finalizer Rules	Critical	Finalize Should Be Protected
Import Statement Rules	Minor	Dont Import Java Lang
Import Statement Rules	Minor	Duplicate Imports
Import Statement Rules	Minor	Import From Same Package
Import Statement Rules	Major	Too Many Static Imports
Import Statement Rules	Minor	Unnecessary Fully Qualified Name
J2EE Rules	Critical	Do Not Call System Exit
J2EE Rules	Major	Local Home Naming Convention

J2EE Rules	Major	Local Interface Session Naming Convention
J2EE Rules	Major	MDBAnd Session Bean Naming Convention
J2EE Rules	Major	Remote Interface Naming Convention
J2EE Rules	Major	Remote Session Interface Naming Convention
J2EE Rules	Critical	Static EJBField Should Be Final
JUnit Rules	Minor	JUnit Assertions Should Include Message
JUnit Rules	Critical	JUnit Spelling
JUnit Rules	Critical	JUnit Static Suite
JUnit Rules	Minor	JUnit Test Contains Too Many Asserts
JUnit Rules	Major	JUnit Tests Should Include Assert
JUnit Rules	Minor	Simplify Boolean Assertion
JUnit Rules	Minor	Test Class Without Test Cases
JUnit Rules	Minor	Unnecessary Boolean Assertion
JUnit Rules	Major	Use Assert Equals Instead Of Assert True
JUnit Rules	Minor	Use Assert Null Instead Of Assert True
JUnit Rules	Minor	Use Assert Same Instead Of Assert True
JUnit Rules	Minor	Use Assert True Instead Of Assert Equals
Jakarta Commons Logging Rules	Major	Guard Debug Logging
Jakarta Commons Logging Rules	Minor	Guard Log Statement
Jakarta Commons Logging Rules	Minor	Proper Logger
Jakarta Commons Logging Rules	Major	Use Correct Exception Logging
Java Logging Rules	Major	Avoid Print Stack Trace
Java Logging Rules	Minor	Guard Log Statement Java Util
Java Logging Rules	Minor	Logger Is Not Static Final
Java Logging Rules	Major	More Than One Logger
Java Logging Rules	Major	System Println
JavaBean Rules	Major	Missing Serial Version UID
Naming Rules	Minor	Avoid Dollar Signs
Naming Rules	Minor	Avoid Field Name Matching Method Name
Naming Rules	Minor	Avoid Field Name Matching Type Name
Naming Rules	Minor	Boolean Get Method Name
Naming Rules	Minor	Class Naming Conventions
Naming Rules	Minor	Generics Naming
Naming Rules	Minor	Method Naming Conventions
Naming Rules	Minor	Method With Same Name As Enclosing Class
Naming Rules	Minor	No Package
Naming Rules	Minor	Package Case
Naming Rules	Minor	Short Class Name
Naming Rules	Minor	Short Method Name
Naming Rules	Minor	Suspicious Constant Field Name
Naming Rules	Critical	Suspicious Equals Method Name
Naming Rules	Critical	Suspicious Hashcode Method Name

---

Naming Rules	Minor	Variable Naming Conventions
Optimization Rules	Minor	Add Empty String
Optimization Rules	Major	Avoid Array Loops
Optimization Rules	Minor	Redundant Field Initializer
Optimization Rules	Major	Unnecessary Wrapper Object Creation
Optimization Rules	Minor	Use Array List Instead Of Vector
Optimization Rules	Major	Use Arrays As List
Optimization Rules	Major	Use String Buffer For String Appends
Security Code Guideline Rules	Major	Array Is Stored Directly
Security Code Guideline Rules	Major	Method Returns Internal Array
Strict Exception Rules	Major	Avoid Catching Generic Exception
Strict Exception Rules	Critical	Avoid Catching NPE
Strict Exception Rules	Major	Avoid Catching Throwable
Strict Exception Rules	Major	Avoid Losing Exception Information
Strict Exception Rules	Minor	Avoid Rethrowing Exception
Strict Exception Rules	Minor	Avoid Throwing New Instance Of Same Exception
Strict Exception Rules	Critical	Avoid Throwing Null Pointer Exception
Strict Exception Rules	Major	Avoid Throwing Raw Exception Types
Strict Exception Rules	Critical	Do Not Extend Java Lang Error
Strict Exception Rules	Critical	Do Not Throw Exception In Finally
Strict Exception Rules	Major	Exception As Flow Control
String and StringBuffer Rules	Major	Avoid Duplicate Literals
String and StringBuffer Rules	Minor	Avoid String Buffer Field
String and StringBuffer Rules	Minor	Consecutive Appends Should Reuse
String and StringBuffer Rules	Minor	Consecutive Literal Appends
String and StringBuffer Rules	Minor	Inefficient String Buffering
String and StringBuffer Rules	Critical	String Buffer Instantiation With Char
String and StringBuffer Rules	Minor	String Instantiation
String and StringBuffer Rules	Minor	String To String
String and StringBuffer Rules	Minor	Unnecessary Case Change
String and StringBuffer Rules	Critical	Use Equals To Compare Strings
Type Resolution Rules	Major	Clone Method Must Implement Cloneable
Type Resolution Rules	Major	Loose Coupling
Type Resolution Rules	Major	Signature Declare Throws Exception
Type Resolution Rules	Minor	Unused Imports
Unnecessary and Unused Code Rules	Major	Unused Local Variable
Unnecessary and Unused Code Rules	Major	Unused Private Field
Unnecessary and Unused Code Rules	Major	Unused Private Method

---

### Appendix C: PMD rules with overlapping ASATs

Rule	Overlapping
Avoid Branching Statement As Last In Loop	
Avoid Decimal Literals In Big Decimal Constructor	FindBugs (DMI_BIGDECIMAL_CONSTRUCTED_FROM_DOUBLE)
Avoid Multiple Unary Operators	
Avoid Thread Group	
Avoid Using Hard Coded IP	
Avoid Using Octal Values	
Big Integer Instantiation	
Boolean Instantiation	Checkstyle (ExplicitInitialization) FindBugs (DM_BOOLEAN_CTOR)
Broken Null Check	
Check Result Set	
Check Skip Result	
Class Cast Exception With To Array	
Collapsible If Statements	
Dont Call Thread Run	FindBugs (RU_INVOKE_RUN)
Dont Use Float Type For Loop Indices	
Double Checked Locking	
Empty Catch Block	Checkstyle (EmptyCatchBlock)
Empty Finally Block	
Empty If Stmt	
Empty Statement Block	
Empty Statement Not In Loop	
Empty Static Initializer	
Empty Switch Statements	
Empty Synchronized Block	
Empty Try Block	
Empty While Stmt	
Extends Object	
For Loop Should Be While Loop	

Jumbled Incrementer	
Misplaced Null Check	
Override Both Equals And Hashcode	Checkstyle (EqualsHashCode) FindBugs (HE_EQUALS_NO_HASHCODE, HE_HASHCODE_NO_EQUALS)
Return From Finally Block	
Unconditional If Statement	
Unnecessary Conversion Temporary	
Unused Null Check In Equals	
Useless Operation On Immutable	
Useless Overriding Method	
For Loops Must Use Braces	Checkstyle (NeedBraces)
If Else Stmt Must Use Braces	Checkstyle (NeedBraces)
If Stmt Must Use Braces	Checkstyle (NeedBraces)
While Loops Must Use Braces	Checkstyle (NeedBraces)
Clone Throws Clone Not Supported Exception	
Proper Clone Implementation	Checkstyle (SuperClone) FindBugs (CN_IDIOM_SUPER_CALL)
Assignment In Operand	Checkstyle (InnerAssignment)
Avoid Accessibility Alteration	
Avoid Prefixing Method Parameters	
Avoid Using Native Code	
Default Package	
Do Not Call Garbage Collection Explicitly	
Dont Import Sun	
One Declaration Per Line	
Suspicious Octal Escape	
Unnecessary Constructor	
Abstract Class Without Abstract Method	
Abstract Class Without Any Method	
Assignment To Non Final Static	Checkstyle (IllegalImport)

Avoid Constants Interface	Checkstyle (InterfacelsType)
Avoid Instanceof Checks In Catch Clause	
Avoid Protected Field In Final Class	
Avoid Protected Method In Final Class Not Extending	FindBugs (DLS_DEAD_LOCAL_STORE, DLS_DEAD_LOCAL_STORE_OF_NULL)
Avoid Reassigning Parameters	Checkstyle (ParameterAssignment)
Avoid Synchronized At Method Level	
Bad Comparison	
Class With Only Private Constructors Should Be Final	Checkstyle (FinalClass)
Close Resource	FindBugs (ODR_OPEN_DATABASE_RESOURCE, ODR_OPEN_DATABASE_RESOURCE_EXCEPTION_PATH, OS_OPEN_STREAM, OS_OPEN_STREAM_EXCEPTION_PATH, OBL_UNSATISFIED_OBLIGATION, OBL_UNSATISFIED_OBLIGATION_EXCEPTION_EDGE)
Constructor Calls Overridable Method	
Default Label Not Last In Switch Stmt	Checkstyle (DefaultComesLast)
Empty Method In Abstract Class Should Be Abstract	
Equals Null	FindBugs (EC_NULL_ARG)
Field Declarations Should Be At Start Of Class	
Final Field Could Be Static	
Idempotent Operations	
Immutable Field	
Instantiation To Get Class	
Logic Inversion	
Missing Break In Switch	
Missing Static Method In Non Instantiatable Class	FindBugs (SF_SWITCH_FALLTHROUGH)
Non Case Label In Switch Statement	
Non Static Initializer	
Non Thread Safe Singleton	
Optimizable To Array Call	
Position Literals First In Case Insensitive Comparisons	

Position Literals First In Comparisons	Checkstyle (EqualsAvoidNull)
Preserve Stack Trace	
Return Empty Array Rather Than Null	FindBugs (PZLA_PREFER_ZERO_LENGTH_ARRAYS)
Simple Date Format Needs Locale	
Simplify Boolean Expressions	Checkstyle (SimplifyBooleanExpression)
Simplify Boolean Returns	Checkstyle (SimplifyBooleanReturn)
Simplify Conditional	
Singular Field	
Switch Stmt's Should Have Default	Checkstyle (MissingSwitchDefault) FindBugs (SF_SWITCH_NO_DEFAULT)
Too Few Branches For ASwitch Statement	
Uncommented Empty Constructor	
Uncommented Empty Method	
Unnecessary Local Before Return	
Unsynchronized Static Date Formatter	FindBugs (STCAL_STATIC_SIMPLE_DATE_FORMAT_INSTANCE, STCAL_STATIC_SIMPLE_DATE_FORMAT_INSTANCE)
Use Collection Is Empty	
Use Locale With Case Conversions	
Use Notify All Instead Of Notify	FindBugs (NO_NOTIFY_NOT_NOTIFYALL)
Use Varargs	
Avoid Calling Finalize	FindBugs (FLEXPLICIT_INVOCATION)
Empty Finalizer	FindBugs (FLEEMPTY)
Finalize Does Not Call Super Finalize	Checkstyle (SuperFinalize) FindBugs (FLMISSING_SUPER_CALL)
Finalize Only Calls Super Finalize	
Finalize Overloaded	
Finalize Should Be Protected	FindBugs (FL_PUBLIC_SHOULD_BE_PROTECTED)
Dont Import Java Lang	Checkstyle (RedundantImport)
Duplicate Imports	Checkstyle (RedundantImport)
Import From Same Package	Checkstyle (RedundantImport)
Too Many Static Imports	
Unnecessary Fully Qualified Name	

Do Not Call System Exit	
Local Home Naming Convention	
Local Interface Session Naming Convention	
MDBAnd Session Bean Naming Convention	
Remote Interface Naming Convention	
Remote Session Interface Naming Convention	
Static EJBField Should Be Final	
JUnit Assertions Should Include Message	
JUnit Spelling	
JUnit Static Suite	FindBugs (IJU_BAD_SUITE_METHOD, IJU_SUITE_NOT_STATIC)
JUnit Test Contains Too Many Asserts	
JUnit Tests Should Include Assert	
Simplify Boolean Assertion	
Test Class Without Test Cases	FindBugs (IJU_NO_TESTS)
Unnecessary Boolean Assertion	
Use Assert Equals Instead Of Assert True	
Use Assert Null Instead Of Assert True	
Use Assert Same Instead Of Assert True	
Use Assert True Instead Of Assert Equals	
Guard Debug Logging	
Guard Log Statement	
Proper Logger	
Use Correct Exception Logging	
Avoid Print Stack Trace	
Guard Log Statement Java Util	
Logger Is Not Static Final	
More Than One Logger	
System Println	
Missing Serial Version UID	FindBugs (SE_NO_SERIALVERSIONID)
Avoid Dollar Signs	

Avoid Field Name Matching Method Name	
Avoid Field Name Matching Type Name	
Boolean Get Method Name	
Class Naming Conventions	Checkstyle (TypeName) FindBugs (NM_CLASS_NAMING_CONVENTION)
Generics Naming	Checkstyle (ClassTypeParameterName, InterfaceTypeParameterName, MethodTypeParameterName)
Method Naming Conventions	Checkstyle (MethodName) FindBugs (NM_METHOD_NAMING_CONVENTION)
Method With Same Name As Enclosing Class	Checkstyle (MethodName) FindBugs (NM_METHOD_CONSTRUCTOR_CONFUSION)
No Package	Checkstyle (PackageDeclaration)
Package Case	
Short Class Name	
Short Method Name	
Suspicious Constant Field Name	FindBugs (NM_BAD_EQUAL)
Suspicious Equals Method Name	FindBugs (NM_LCASE_HASHCODE)
Suspicious Hashcode Method Name	Checkstyle (StaticVariableName, ParameterName, ParameterName, ParameterName, ParameterName, ConstantName) FindBugs (NM_FIELD_NAMING_CONVENTION)
Variable Naming Conventions	
Add Empty String	
Avoid Array Loops	
Redundant Field Initializer	
Unnecessary Wrapper Object Creation	
Use Array List Instead Of Vector	
Use Arrays As List	
Use String Buffer For String Appends	
Array Is Stored Directly	
Method Returns Internal Array	
Avoid Catching Generic Exception	Checkstyle (IllegalCatch) FindBugs (REC_CATCH_EXCEPTION)
Avoid Catching NPE	
Avoid Catching Throwable	Checkstyle (IllegalCatch)
Avoid Losing Exception Information	

Avoid Rethrowing Exception	
Avoid Throwing New Instance Of Same Exception	
Avoid Throwing Null Pointer Exception	
Avoid Throwing Raw Exception Types	Checkstyle (IllegalThrows)
Do Not Extend Java Lang Error	
Do Not Throw Exception In Finally	
Exception As Flow Control	
Avoid Duplicate Literals	FindBugs (HSC_HUGE_SHARED_STRING_CONSTANT)
Avoid String Buffer Field	
Consecutive Appends Should Reuse	
Consecutive Literal Appends	
Inefficient String Buffering	
String Buffer Instantiation With Char	
String Instantiation	
String To String	
Unnecessary Case Change	
Use Equals To Compare Strings	Checkstyle (StringLiteralEquality) FindBugs (ES_COMPARING_PARAMETER_STRING_WITH_EQ, ES_COMPARING_PARAMETER_STRING_WITH_EQ)
Clone Method Must Implement Cloneable	FindBugs (CN_IMPLEMENTATIONS_CLONE_BUT_NOT_CLONEABLE)
Loose Coupling	Checkstyle (IllegalType)
Signature Declare Throws Exception	Checkstyle (UnusedImports)
Unused Imports	Checkstyle (FinalLocalVariable) FindBugs (DLS_DEAD_LOCAL_STORE, DLS_DEAD_LOCAL_STORE_OF_NULL)
Unused Local Variable	FindBugs (UWF_UNWRITTEN_PUBLIC_OR_PROTECTED_FIELD, UWF_UNWRITTEN_PUBLIC_OR_PROTECTED_FIELD, UWF_UNWRITTEN_PUBLIC_OR_PROTECTED_FIELD, UWF_UNWRITTEN_PUBLIC_OR_PROTECTED_FIELD)
Unused Private Field	UUF_UNUSED_FIELD, URF_UNREAD_FIELD, UWF_UNWRITTEN_FIELD)
Unused Private Method	FindBugs (UPM_UNCALLED_PRIVATE_METHOD)

## References

- Abdi H (2007) Bonferroni and Sidak corrections for multiple comparisons. In: Encyclopedia of Measurement and Statistics, Sage, Thousand Oaks, CA, pp 103–107
- Aloraini B, Nagappan M, German DM, Hayashi S, Higo Y (2019) An empirical study of security warnings from static application security testing tools. *J Syst Softw* 158:110427. <https://doi.org/10.1016/j.jss.2019.110427>. <http://www.sciencedirect.com/science/article/pii/S0164121219302018>
- Aversano L, Canfora G, Cerulo L, Del Grosso C, Di Penta M (2007) An empirical study on the evolution of design patterns. In: Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, ACM, New York, NY, USA, ESEC-FSE '07, pp 385–394. <http://doi.acm.org/10.1145/1287624.1287680>
- Bakota T, Hegedűs P, Körtvélyesi P, Ferenc R, Gyimóthy T (2011) A probabilistic software quality model. In: 2011 27th IEEE International Conference on Software Maintenance (ICSM), pp 243–252
- Beller M, Bholanath R, McIntosh S, Zaidman A (2016) Analyzing the state of static analysis: A large-scale evaluation in open source software. In: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), 1, pp 470–481
- Bird C, Rigby PC, Barr ET, Hamilton DJ, German DM, Devanbu P (2009) The promises and perils of mining git. In: 2009 6th IEEE International Working Conference on Mining Software Repositories, pp 1–10
- Boehm BW, Brown JR, Lipow M (1976) Quantitative evaluation of software quality. In: Proceedings of the 2Nd International Conference on Software Engineering, IEEE Computer Society Press, Los Alamitos, CA, USA, ICSE '76, pp 592–605. <http://dl.acm.org/citation.cfm?id=800253.807736>
- Campbell MJ, Gardner MJ (1988) Statistics in medicine: Calculating confidence intervals for some non-parametric analyses. *BMJ* 296(6634):1454–1456. <https://doi.org/10.1136/bmj.296.6634.1454>. <https://www.bmj.com/content/296/6634/1454.full.pdf>, <https://www.bmj.com/content/296/6634/1454>
- Christakis M, Bird C (2016) What developers want and need from program analysis: An empirical study. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ACM, New York, NY, USA, ASE 2016, pp 332–343. <http://doi.acm.org/10.1145/2970276.2970347>
- Cohen J (1988) Statistical power analysis for the behavioral sciences, L. Erlbaum Associates
- Devanbu P, Zimmermann T, Bird C (2016) Belief evidence in empirical software engineering. In: 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), pp 108–119
- Digkas G, Lungu M, Avgeriou P, Chatzigeorgiou A, Ampatzoglou A (2018) How do developers fix issues and pay back technical debt in the apache ecosystem? In: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp 153–163
- Dijkstra EW (1959) A note on two problems in connexion with graphs. *Numer Math* 1(1):269–271. <https://doi.org/10.1007/BF01386390>
- Distefano D, Fähndrich M, Logozzo F, O'Hearn PW (2019) Scaling static analyses at facebook. *Commun. ACM* 62(8):62–70. <https://doi.org/10.1145/3338112>
- Faragó C, Hegedűs P, Ferenc R (2015) Code ownership: Impact on maintainability. In: Gervasi O, Murgante B, Misra S, Gavrilova ML, Rocha AMAC, Torre C, Tanar D, Apduhan BO (eds) Computational Science and Its Applications – ICCSA 2015, Springer International Publishing, Cham, pp 3–19
- Fenton N, Bieman J (2014) Software metrics: A rigorous and practical approach, 3rd edn. CRC Press, Inc., Boca Raton, FL, USA
- Ferenc R, Hegedűs P, Gyimóthy T (2014) Software product quality models. In: Mens T, Serebrenik A, Cleve A (eds) Evolving Software Systems, Springer Berlin Heidelberg, Berlin, Heidelberg, pp 65–100. [https://doi.org/10.1007/978-3-642-45398-4\\_3](https://doi.org/10.1007/978-3-642-45398-4_3)
- Ferenc R, Tóth Z, Ladányi G, Siket I, Gyimóthy T (2020) A public unified bug dataset for java and its assessment regarding metrics and bug prediction. *Softw Qual J*. <https://doi.org/10.1007/s11219-020-09515-0>
- Fowler M (1999) Refactoring: Improving the design of existing code. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA
- FrontEndART (2019) Sourcemeeter. <https://www.sourcemeeter.com/>
- Habib A, Pradel M (2018) How many of all bugs do we find? a study of static bug detectors. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ACM, New York, NY, USA, ASE 2018, pp 317–328. <http://doi.acm.org/10.1145/3238147.3238213>
- Hagberg AA, Schult DA, Swart PJ (2008) Exploring network structure, dynamics, and function using NetworkX. In: Proceedings of the 7th Python in Science Conference (SciPy2008), Pasadena, CA USA, pp 11–15
- Heckman S, Williams L (2009) A model building process for identifying actionable static analysis alerts. In: 2009 International Conference on Software Testing Verification and Validation, pp 161–170

- Heckman S, Williams L (2011) A systematic literature review of actionable alert identification techniques for automated static code analysis. *Inf. Softw. Technol.* 53(4):363–387. <https://doi.org/10.1016/j.infsof.2010.12.007>. <http://dx.doi.org/10.1016/j.infsof.2010.12.007>
- Herbold S, Trautsch A, Trautsch F (2019) Issues with szz: An empirical assessment of the state of practice of defect prediction data collection. arxiv:1911.08938, Article is currently in submission to IEEE Transactions on Software Engineering
- Johnson B, Song Y, Murphy-Hill E, Bowdidge R (2013) Why don't software developers use static analysis tools to find bugs? In: Proceedings of the 2013 International Conference on Software Engineering, IEEE Press, Piscataway, NJ, USA, ICSE '13, pp 672–681. <http://dl.acm.org/citation.cfm?id=2486788.2486877>
- Jones E, Oliphant T, Peterson P et al (2001) SciPy: Open source scientific tools for Python. <http://www.scipy.org/>, [Online; accessed 17.09.2018]
- Kendall MG (1955) Rank correlation methods. Charles Griffin & Co. Ltd
- Kim S, Ernst MD (2007) Prioritizing warning categories by analyzing software history. In: Proceedings of the Fourth International Workshop on Mining Software Repositories, IEEE Computer Society, Washington, DC, USA, MSR '07, pp 27–, <https://doi.org/10.1109/MSR.2007.26>
- Kim S, Ernst MD (2007) Which warnings should i fix first? In: Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, ACM, New York, NY, USA, ESEC-FSE '07, pp 45–54
- Kitchenham B, Pfleeger SL (1996) Software quality: the elusive target [special issues section]. *IEEE Softw* 13(1):12–21. <https://doi.org/10.1109/52.476281>
- Kitchenham BA, Dyba T, Jorgensen M (2004) Evidence-based software engineering. In: Proceedings of the 26th International Conference on Software Engineering, IEEE Computer Society, Washington, DC, USA, ICSE '04, pp 273–281. <http://dl.acm.org/citation.cfm?id=998675.999432>
- Kreyszig E (2000) Advanced engineering mathematics: Maple computer guide, 8th edn. John Wiley & Sons, Inc., New York, NY, USA
- Kruchten P, Nord RL, Ozkaya I (2012) Technical debt: From metaphor to theory and practice. *IEEE Softw* 29(6):18–21. <https://doi.org/10.1109/MS.2012.167>
- Lehman MM (1996) Laws of software evolution revisited. In: Proceedings of the 5th European Workshop on Software Process Technology, Springer-Verlag, Berlin, Heidelberg, EWSPT '96, pp 108–124. <http://dl.acm.org/citation.cfm?id=646195.681473>
- Levene H (1960) Robust tests for equality of variances. In: Contributions to probability and statistics, Stanford Univ. Press, Stanford, Calif., pp 278–292
- Liu K, Kim D, Bissyande TF, Yoo S, Le Traon Y (2018) Mining fix patterns for findbugs violations. *IEEE Trans Softw Eng*, p 1–1. <https://doi.org/10.1109/TSE.2018.2884955>
- Malloy BA, Power JF (2019) An empirical analysis of the transition from python 2 to python 3. *Empir Softw Eng* 24(2):751–778. <https://doi.org/10.1007/s10664-018-9637-2>
- Mann HB, Whitney DR (1947) On a test of whether one of two random variables is stochastically larger than the other. *Ann Math Stat* 18(1):50–60
- Marcilio D, Bonifácio R, Monteiro E, Canedo E, Luz W, Pinto G (2019) Are static analysis violations really fixed?: A closer look at realistic usage of sonarqube. In: Proceedings of the 27th International Conference on Program Comprehension, IEEE Press, Piscataway, NJ, USA, ICPC '19, pp 209–219. <https://doi.org/10.1109/ICPC.2019.00040>
- McCabe TJ (1976) A complexity measure. *IEEE Trans. Softw. Eng.* 2(4):308–320. <https://doi.org/10.1109/TSE.1976.233837>
- McCall JA, Richards PK, Walters GF (1977) Factors in software quality: concept and definitions of software quality. Rome Air Development Center, Air Force Systems Command, Griffiss Air Force Base, New York, (13)
- Panichella S, Arnaoudova V, Di Penta M, Antoniol G (2015) Would static analysis tools help developers with code reviews? In: 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER), pp 161–170
- Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O, Blondel M, Prettenhofer P, Weiss R, Dubourg V, Vanderplas J, Passos A, Cournapeau D, Brucher M, Perrot M, Duchesnay E (2011) Scikit-learn: Machine learning in Python. *J Mach Learn Res* 12:2825–2830
- Penta MD, Cerulo L, Aversano L (2009) The life and death of statically detected vulnerabilities: An empirical study. *Inf Softw Technol* 51(10):1469–1484. <https://doi.org/10.1016/j.infsof.2009.04.013>. <http://www.sciencedirect.com/science/article/pii/S0950584909000500>, Source Code Analysis and Manipulation, SCAM 2008

- Plosch R, Gruber H, Hentschel A, Pomberger G, Schiffer S (2008) On the relation between external software quality and static code analysis. In: 2008 32nd Annual IEEE Software Engineering Workshop, pp 169–174
- Querel L-P, Rigby PC (2018) Warningsguru: Integrating statistical bug models with static analysis to provide timely and specific bug warnings. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Association for Computing Machinery, New York, NY, USA, ESEC/FSE 2018, pp 892–895. <https://doi.org/10.1145/3236024.3264599>
- Rahman F, Khatri S, Barr ET, Devanbu P (2014) Comparing static bug finders and statistical prediction. In: Proceedings of the 36th International Conference on Software Engineering, ACM, New York, NY, USA, ICSE 2014, pp 424–434. <http://doi.acm.org/10.1145/2568225.2568269>
- Rosen C, Grawi B, Shihab E (2015) Commit guru: Analytics and risk prediction of software commits. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, Association for Computing Machinery, New York, NY, USA, ESEC/FSE 2015, pp 966–969. <https://doi.org/10.1145/2786805.2803183>
- Sadowski C, Aftandilian E, Eagle A, Miller-Cushon L, Jaspan C (2018) Lessons from building static analysis tools at google. *Commun. ACM* 61(4):58–66. <https://doi.org/10.1145/3188720>
- Śliwerski J, Zimmermann T, Zeller A (2005) When do changes induce fixes? *SIGSOFT Softw. Eng. Notes* 30(4):1–5. <https://doi.org/10.1145/1082983.1083147>
- Spearman C (1904) The proof and measurement of association between two things. *Am J Psychol* 15:88–103
- Szóke G, Antal G, Nagy C, Ferenc R, Gyimóthy T (2014) Bulk fixing coding issues and its effects on software quality: Is it worth refactoring? In: 2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation, pp 95–104
- Thung F, Lucia, Lo D, Jiang L, Rahman F, Devanbu PT (2012) To what extent could we detect field defects? an empirical study of false negatives in static bug finding tools. In: 2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, pp 50–59
- Trautsch A, Herbold S, Grabowski J (2020) A longitudinal study of static analysis warning evolution and the effects of pmd on software quality in apache open source projects - online appendix and replication kit. <http://www.user.informatik.uni-goettingen.de/~{trautsch2/emse2019>
- Trautsch F, Herbold S, Makedonski P, Grabowski J (2017) Addressing problems with replicability and validity of repository mining studies through a smart data platform. *Empir Softw Eng* 23:1036–1083. <https://doi.org/10.1007/s10664-017-9537-x>
- Tufano M, Palomba F, Bavota G, Penta MD, Oliveto R, Lucia AD, Poshyvanyk D (2017) There and back again: Can you compile that snapshot? *J Soft Evol Process* 29(4):e1838. <http://dblp.uni-trier.de/db/journals/smr/smr29.html#TufanoPBPOLP17>
- Vassallo C, Panichella S, Palomba F, Proksch S, Gall HC, Zaidman A (2019) How developers engage with static analysis tools in different contexts. *Empir Softw Eng*, pp 1419–1457. <https://doi.org/10.1007/s10664-019-09750-5>
- Vetro A, Morisio M, Torchiano M (2011) An empirical validation of findbugs issues related to defects. In: 15th Annual Conference on Evaluation Assessment in Software Engineering (EASE 2011), pp 144–153
- Wagner S, Lochmann K, Heinemann L, Kläs M, Trendowicz A, Plösch R, Seidl A, Goeb A, Streit J (2012) The quamoco product quality modelling and assessment approach. In: Proceedings of the 34th International Conference on Software Engineering, IEEE Press, Piscataway, NJ, USA, ICSE '12, pp 1133–1142. <http://dl.acm.org/citation.cfm?id=2337223.2337372>
- Wilk MB, Shapiro SS (1965) An analysis of variance test for normality (complete samples)†. *Biometrika* 52(3-4):591–611. <https://doi.org/10.1093/biomet/52.3-4.591>
- Wohlin C, Runeson P, Höst M, Ohlsson MC, Regnell B, Wesslén A (2000) Experimentation in software engineering: An introduction. Kluwer Academic Publishers, Norwell, MA, USA
- Zampetti F, Scalabrino S, Oliveto R, Canfora G, Penta MD (2017) How open source projects use static code analysis tools in continuous integration pipelines. In: 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), pp 334–344
- Zheng J, Williams L, Nagappan N, Snipes W, Hudepohl JP, Vouk MA (2006) On the value of static analysis for fault detection in software. *IEEE Trans Softw Eng* 32(4):240–253. <https://doi.org/10.1109/TSE.2006.38>
- Zimmermann T, Premraj R, Zeller A (2007) Predicting defects for eclipse. In: Proceedings of the Third International Workshop on Predictor Models in Software Engineering, IEEE Computer Society, Washington, DC, USA, PROMISE '07, pp 9–. <http://dx.doi.org/10.1109/PROMISE.2007.10>



## **B Data validity issues in defect-based analyses**

This section contains a copy of the following publication.

S. Herbold, A. Trautsch, F. Trautsch, B. Ledel: Problems with SZZ and Features: An empirical study of the state of practice of defect prediction data collection. *Empirical Software Engineering* (2022) 27,42. Springer Nature

© 2022, The Author(s). Reprinted with permission.

<https://doi.org/10.1007/s10664-021-10092-4>



# Problems with SZZ and features: An empirical study of the state of practice of defect prediction data collection

Steffen Herbold<sup>1</sup> · Alexander Trautsch<sup>2</sup> · Fabian Trautsch<sup>2</sup> · Benjamin Ledel<sup>1</sup>

Accepted: 10 November 2021 / Published online: 18 January 2022  
© The Author(s) 2021

## Abstract

**Context** The SZZ algorithm is the de facto standard for labeling bug fixing commits and finding inducing changes for defect prediction data. Recent research uncovered potential problems in different parts of the SZZ algorithm. Most defect prediction data sets provide only static code metrics as features, while research indicates that other features are also important.

**Objective** We provide an empirical analysis of the defect labels created with the SZZ algorithm and the impact of commonly used features on results.

**Method** We used a combination of manual validation and adopted or improved heuristics for the collection of defect data. We conducted an empirical study on 398 releases of 38 Apache projects.

**Results** We found that only half of the bug fixing commits determined by SZZ are actually bug fixing. If a six-month time frame is used in combination with SZZ to determine which bugs affect a release, one file is incorrectly labeled as defective for every file that is correctly labeled as defective. In addition, two defective files are missed. We also explored the impact of the relatively small set of features that are available in most defect prediction data sets, as there are multiple publications that indicate that, e.g., churn related features are important for defect prediction. We found that the difference of using more features is not significant.

**Conclusion** Problems with inaccurate defect labels are a severe threat to the validity of the state of the art of defect prediction. Small feature sets seem to be a less severe threat.

**Keywords** SZZ · Bug fix labeling · Bug inducing changes · Defect prediction data · Data set

---

Communicated by : Andrea De Lucia.

---

Steffen Herbold, Alexander Trautsch, and Fabian Trautsch contributed equally to this work.

---

✉ Steffen Herbold  
steffen.herbold@tu-clausthal.de

Extended author information available on the last page of the article

## 1 Introduction

Defect prediction is an active direction of software engineering research with hundreds of publications. The systematic literature review by Hall et al. (2012) already found 208 studies on defect prediction published between 2000 and 2010, many more have been published since then. Many of these studies were enabled by the sharing of data, highlighted by the early efforts from the PROMISE repository (Menzies et al. 2015), which is nowadays known as Seacraft (Menzies et al. 2017). Only few publications on defect prediction collect new data. Instead, most researchers rely on well-known data sets, e.g., the NASA data (NASA 2004), the SOFTLAB data (Turhan et al. 2009), or the data about Java projects from Jureczko and Madeyski (2010) often referred to as PROMISE. A recent literature review on cross-project defect prediction highlights that these and other data sets have become the de facto standard for defect prediction research (Hosseini et al. 2017). While sharing and re-using data is a good thing in general, heavy re-use may also lead to problems with the external validity of results (Trautsch et al. 2017). The biggest problem outlined by Trautsch et al. (2017) is that if there are problems that affect the validity of shared data, these problems lead to threats to the validity of all research that re-used this data. Unfortunately, there is evidence that shared defect prediction data is affected by two problems: 1) problems with the defect labels; and 2) limitations regarding the features used by researchers.

The first problem is related to the defect labels, that were determined by different publications that consider different aspects of the defect labeling process. The focus of these publications is mostly on the SZZ algorithm (Śliwerski et al. 2005), which was applied by most of the currently used data sets (see Section 2).<sup>1</sup> Research revealed several problems with SZZ, e.g., due to ignoring the affected version field of issue reports (Da Costa et al. 2017) or the identification of irrelevant changes (Mills et al. 2018). The use of a six-month time frame for the assignment of defects to releases has also recently been identified as a problem (Yatish et al. 2019). Moreover, SZZ relies on the correct labeling of issues as bug by the developers in the issue tracking system. However, research shows that about 33% of bug reports are mislabeled and are actually improvements or other issues, like outdated documentation (Herzig et al. 2013). Additionally, SZZ was designed for version control systems that used a mostly linear development process on a main development branch. Due to the success of Git, this is often not the case anymore and there are many new challenges that need to be considered (Bird et al. 2009b). For example, prior research found that data, which takes branches into account, leads to better results (Kovalenko et al. 2018).

The second problem with the re-use of the existing data is the limited feature space that researchers use to create defect prediction models. If a data set does not contain certain features, it is unlikely that they are added by other researchers, even if research indicates that these features may be useful. For example, multiple publications indicate that features based on code changes potentially outperform static metrics as features (Moser et al. 2008; D'Ambros et al. 2012). Regardless, researchers mostly rely on data sets that only consist of

---

<sup>1</sup> Please note that we use the term SZZ for both phases of SZZ, i.e., the issue linking and the identification of inducing changes. While we are aware that parts of the research community restrict the term SZZ to the identification of inducing changes, i.e., only the second phase of the original SZZ algorithm, this has not been the case in the literature on defect prediction data. Here, the opposite is the case and many data sets use the term SZZ and only apply the first phase of SZZ (see Section 2).

static features (Hosseini et al. 2017). Thus, many publications are using a potentially inferior set of features, which could alter their results.

Thus, we know from the related work about many separate problems with defect prediction data, especially with respect to the way software artifacts are labeled as defective, but also due to a potential lack of relevant features. However, each of the prior publications on this topic focuses on a single problem with defect prediction data. What is missing is a view on the impact of the problems if they are not considered in isolation, but together. Within this article, we close this gap and consider the following research question.

**RQ:** *What is the overall impact of the numerous problems with defect labeling and the usually small set of features on defect prediction data and the results of prediction?*

We provide insights into the quality problems that existing data may have, with a focus on the defect labeling. To this aim, we performed an in-depth analysis of the weaknesses of existing defect labeling strategies with a focus on SZZ. The original SZZ is still commonly used for the labeling defect prediction data (see Section 2) and, hence, the *state of practice*. We also compare and discuss our results with respect to newer SZZ variants that are *state of the art* and discuss how the results change. We analyze all aspects of the defect labeling process, i.e., the links between commits and issues, the impact of mislabeled issues, the identification of affected files and the inducing changes, as well as the assignment of defects to releases. Additionally, we use the large sample of data we generate to assess the impact of the lack of features on the performance of defect prediction results.

The primary contribution of our article are the following findings with respect to the current state of practice for the generation of defect prediction data.

- About one quarter of the links to defects detected by SZZ is wrong, both due to missed links as well as false positive links.
- We confirm the results by Herzig et al. (2013) and found that for every issue that is correctly labeled as a bug, there are 0.74 mislabeled bug issues.
- Due to the combination of wrong links and mislabeled issues, only about half of the commits SZZ identifies are actually bug fixing and SZZ misses about one fifth of all bug fixing commits.
- The assignment of defects to releases based on a six months time frame, as well as based on the affected versions field of issue tracking systems is unreliable. With SZZ and a six months time frame, we found that for every file, that is correctly labeled as defective, there are roughly two files that are incorrectly labeled as defective, and two files that are incorrectly labeled as non-defective. Moreover, the quality of the data in the affected version field is questionable for mining purposes without prior manual validation, due to many missing, incomplete, or wrong values.
- The difference of using many features of different types over using only static features of the source code is not statistically significant when we consider the cost saving potential of classifiers for release-level defect prediction. Moreover, we found that mislabels during training do not have a significant impact, but mislabels in test data can significantly alter evaluation results.

As a secondary contribution, we provide all data we generated through manual validation and repository mining to study the state of practice of defect prediction data generation. As a result, we provide a new defect prediction data set with 4198 features, including change

metrics (Moser et al. 2008; Hassan 2009; D'Ambros et al. 2012), and different aggregation strategies (Zhang et al. 2017). The data set contains defect data for 398 releases of 38 projects from the Apache ecosystem.

The remainder of this paper is structured as follows. We discuss the state of practice for the collection of defect prediction data with respect to existing data sets in Section 2, followed by an analysis of problems with the established data collection methods reported in the state of the art in Section 3. Afterwards, we discuss our suggested improvements to the state of practice in Section 4.1. Section 5 presents the results of our empirical study on defect labeling and the impact of feature sets. We discuss our results in Section 6 and address the threats to the validity of our work in Section 7. Finally, we conclude the article in Section 8.

## 2 Existing data sets

This article is about the state of practice regarding the collection of defect prediction data. Therefore, the related work are articles that collected defect prediction data. Articles that only discuss specific aspects of this data collection are instead discussed together with the problems of the currently established ways for the collection of defect prediction data in Section 3. We discuss the prior defect prediction data sets with respect to the following criteria:

- the number of distinct projects, as well as the number of releases;
- the level of abstraction of the data set, e.g., modules, files, or classes;
- the features that are provided; and
- the defect labeling strategy and the labels that are provided.

Overall, we are aware of fifteen publicly available data sets for defect prediction as of August 2019. The NASA (NASA 2004), ECLIPSE (Zimmermann et al. 2007), SOFT-LAB (Turhan et al. 2009), PROMISE (Jureczko and Madeyski 2010), RELINK (Wu et al. 2011), AEEEM (D'Ambros et al. 2012), NETGENE (Herzig et al. 2013), MJ12A (Madeyski and Jureczko 2015), SHIPPEY (Shippey et al. 2016), GITHUB (Tóth et al. 2016), UNIFIED (Ferenc et al. 2018, 2020b), RNALYTICA (Yatish et al. 2019) contain release-level data and the JIT (Kamei et al. 2013), AUDI (Altinger et al. 2015), MT (McIntosh and Kamei 2018), and FJIT (Pascarella et al. 2019) just-in-time data. The BUGHUNTER (Ferenc et al. 2020a) contains release-level data for all bugfixing commits, i.e., is neither a traditional release level data set, nor a just-in-time data set. Table 1 gives an overview about the data sets. The data contains mostly open source projects (OSS), but also proprietary projects (PROP). The PROMISE data is actually a mix of 15 open source projects with 48 releases, 6 proprietary projects with 27 releases, and 17 student projects with 17 releases.

These data sets can be distinguished between data for release-level defect prediction and just-in-time defect prediction. There are two major differences between the release-level and just-in-time data sets. 1) Release-level data contains features for all software artifacts for a certain revision (usually a release) of a project, while just-in-time data contains data for every commit of a project, possibly restricted to the main development branch. 2) The release-level data consists mostly of features that measure the source code directly, e.g., its size, structure, or coupling. Just-in-time data consists mostly of features that measure the

**Table 1** Overview of existing public data sets for defect prediction research. The data sets in the first compartment are for release-level defect prediction, the data sets in the second compartment are for just-in-time defect prediction. The BUGHUNTER data set contains release-level data for all bug fixing commits

Dataset	#Projects / #Releases	Type	Language	Granularity	Features	Defect Linking	Label Type	Year
NASA	13 / 13	PROP	C, C++, Java	Module	SIZE, COM	NA	Binary	2003
ECLIPSE	1 / 3	OSS	Java	File, Package	SIZE, COM, CHURN	SZZ	Counts	2007
SOFTLAB	5 / 5	PROP	C, C++	Module	SIZE, COM	NA	Binary	2009
PROMISE	37 / 92	Mixed	Java	Class	SIZE, COM, OO	REGEX	Counts	2010
RELINK	3 / 3	OSS	Java	Class	SIZE, COM	GOLDEN	Binary	2011
AEEEM	5 / 5	OSS	Java	Class	SIZE, COM, CHURN	SZZ	Counts	2012
NETGENE	4 / 4	OSS	Java	File	SIZE, COM, GEN	SZZ	Counts	2013
MJ12A	18 / 70	Mixed	Java	Class	SIZE, COM, OO, CHURN	NA	NA	2015
SHIPPEY	23 / 69	OSS	Java	Class, Method	SIZE, COM	SZZ	Counts	2016
GITHUB	15 / 15	OSS	Java	Class, File	SIZE, COM, OO, DOC, CLONE	SZZ	Binary	2016
UNIFIED	37 / 71	OSS	Java	Class, File	SIZE, COM, OO, DOC	NA	NA	2018
RNALYTICA	9 / 32	OSS	Java	File	SIZE, COM, OO, CHURN	Affected Version	Counts	2019
JIT	6 / -	OSS	Java, Perl, C++, Ruby	Commit	CHURN, DEV	SZZ	Binary	2013
AUDI	3 / -	Prop.	C	Commit	SIZE, COM, CHURN	SZZ	Binary	2015
MT	2 / -	OSS	Python, C++	Commit	CHURN, REVIEW	SZZ	Binary	2017
FJIT	10 / -	OSS	Java, JavaScript, Perl, C	Commit+File	CHURN, DEV	SZZ	Binary	2019
BUGHUNTER	15 / -	OSS	Java	File, Class, Method	SIZE, COM, OO, CLONE	SZZ	Binary	2020

changes, e.g., the number of lines that are changed. The second difference has a major consequence regarding the programming languages that are considered: the collection of data about the source code structure requires language-specific tooling for the (static) analysis of source code, while the collection of data about code changes and ownership can be done directly using the version control system. This is reflected directly in the languages of the projects: while most release-level data sets are only for one specific programming language, two out of three just-in-time data sets are for a diverse set of languages.

We note that there is a strong focus on Java in the release-level data sets. Although we have no scientific evidence for this, we believe that the reason for this is likely the good tool support for the static analysis of Java. Moreover, we note that the features for the release level data sets are mostly static product metrics of the types that measure the size (SIZE), code complexity (COM), or aspects of object orientation (OO), e.g., using Chidamber and Kemerer's metrics (Chidamber and Kemerer 1994). The GITHUB, UNIFIED, and BUGHUNTER data sets also contain other features based on static product metrics, i.e., regarding the code documentation (DOC) and code clones (CLONE). The notable exceptions are the AEEEM, MJ12A, and RNALYTICA data, which also contain features based on code changes, e.g., added and deleted lines (CHURN). The AEEEM data even contains features that measure the entropy of code changes as proposed by Hassan (2009) and D'Ambros et al. (2012). The ECLIPSE, AEEEM, and MJ12A data also contain metrics regarding prior bug fixes. The MJ12A data is an extension of a subset of the PROMISE data with CHURN metrics. The UNIFIED data is a special case: this data set is actually a combination of the PROMISE (only OSS projects), AEEEM, ECLIPSE, and GITHUB data. All data is conserved as is in the data set and augmented with additional metric data to create the UNIFIED data set (Ferenc et al. 2018, 2020a). Most data sets for release-level defect prediction are using the file, class, respectively module level. These are all very similar, as they encompass the complete contents of a single file in most cases, with the exception of anonymous and inner classes, that can lead to differences. The notable exceptions are the SHIPPEY and BUGHUNTER data, which also contain method-level data.

For the just-in-time defect prediction data, we note that JIT, MT, and FJIT are independent of the programming language, which is a big advantage for generalizing the defect prediction approach. These data sets are using metrics that can directly be inferred from the version control system. MT also uses metrics about code REVIEW. An important difference between the data sets is that the JIT and MT data only contain the information which commits induced a defect, while FJIT contains information which changes to files (hereafter referred to as file actions) induced a defect. The AUDI data set is not comparable to the other two data sets: the data is about source code that was not written by developers, but instead generated from Simulink models developed by engineers. Moreover, the data contains not only information that is collected from the version control system, but also static product metrics about SIZE and COM.

The identification of defective artifacts is a major aspect of defect prediction data that can greatly affect the quality of the data. For example, Yatish et al. (2019) recently found that labeling based on the affected releases leads to large differences in comparison to labeling based on keywords and a six-month time window as proposed by Fischer et al. (2003) as it was used by Zimmermann et al. (2007). The SZZ algorithm first identifies bug fixing commits and then the corresponding inducing changes. However, the identification of inducing changes is, to the best of our knowledge, so far ignored by all release-level data sets. Instead, only bug fixing commits are identified using SZZ and then the six-month time window as proposed by Fischer et al. (2003) is used by Zimmermann et al. (2007). This approach was used for the collections of the ECLIPSE, AEEEM, NETGENE, SHIPPEY

and GITHUB data sets. The BUGHUNTER data uses the links on GitHub between commits and issues to determine bug fixing commits. To determine which methods, classes, and files contribute to the bug fix, they use a SZZ variant similar to the work by Williams and Spacco (2008) that determines ranges of lines based on diffs and match these ranges to the code artifacts. The authors manually validated this approach and find that this works well in over 99% of the cases. We note that the manual validation only covered if changed source code was matched correctly, and not if the changes contributed to the bug fix.

The just-in-time defect prediction data sets JIT, AUDI, MT, and FJIT data<sup>2</sup> use SZZ including the identification of inducing commits, usually with the addition to ignore white-space and comment only changes. The RELINK data was created as a case study for an issue linking approach. The authors created manually validated issue links and used these links for the identification of bug fixing commits. All files that were implicated in any bug fix are considered as defective, without using a time window. The PROMISE and MJ12A data identify bug fixing commits using regular expressions applied to commit messages and a six-month time window. For the NASA and SOFTLAB data, no information on how the defects are linked to source code is given. Since the UNIFIED data is an aggregation of other data sets that reuses defect labels, there is no defect identification approach. The RNALYTICA data uses an approach for the identification of defects based on the affected version field of the bug tracking system. The authors establish links between commits and issues based on references from issues to the commits in which they were addressed. They then use the hunks changed in these commits to identify which files were changed. The authors then use the affected version field of the issue tracker to assign the defect for the file to releases.

We note that there may be additional data sets, that we did not discuss above. For example, we excluded the data used by Zhang et al. (2014) based on the census data by Mockus (2009). The reason for this exclusion is that this data is, to the best of our knowledge, not publicly available anymore, because the links in both papers do not work anymore. Regardless, these data sets would not add anything regarding the methodology for collecting defect prediction data, as the approach is almost exactly the same as for PROMISE and MJ12A: the defect identification is based on commit messages and a six month time window, the metrics are SIZE and COM, and in case of Zhang et al. (2014) also CHURN. Data like Defects4J is out of scope of our discussion of defect prediction data. While Defects4J contains detailed information about a subset of defects from a project, the data neither contains features, nor examples for changes without defects. As a consequence, such data would only consist of positive labels and not be suited for any defect prediction experiment.

### 3 Problems with existing data sets

The sharing of public data sets in defect prediction is a success story that enabled defect prediction researchers to conduct many experiments with the data. Moreover, the use of the same data by different authors enables comparisons between approaches through meta studies, as was, e.g., done by Hall et al. (2012) who exploited that many papers are based on the NASA data. However, there are a number of potential problems that researchers

---

<sup>2</sup> According to the replication kit, the FJIT data may use only a subset of SZZ for identification of bug fixing commits, i.e., a pure keyword based approach.

found regarding the data collection procedures used for the creation of the defect prediction data sets. Within this section, we summarize problems regarding algorithms for defect labeling and the features available in current data sets.

### 3.1 Defect labeling

Defect labels are the key component of any defect prediction data set. These labels mark artifacts as defective, e.g., files in a release or in a commit. These labels are the dependent variable that defect prediction models try to predict based on the independent variables, i.e., the features. In the existing defect prediction data, labels are either binary or defect counts. Noisy defect labels may negatively affect the training of defect prediction models or make the evaluation of results unreliable. Especially the impact of the noise on the evaluation of results is problematic. For example, if a defect labeling approach marks too many instances as defective, i.e., produces false positives, the commonly used measures *recall*, *precision* and *F-measure* are not trustworthy anymore, because values may change if the distribution between defective and non-defective instances changes. Consider an example with 100 software artifacts, 25 artifacts are actually defective, but the defect labeling algorithm introduces noise and labels 50 artifacts as defective. A trivial model that predicts everything as defective will overestimate the *precision* as 0.5 instead of the actual 0.25, which would also affect the *F-measure* which would be 0.7 instead of 0.4.

As we discussed above, the de facto standard for labeling defective instances is the use of the SZZ algorithm (Śliwerski et al. 2005) in the variant used by Zimmermann et al. (2007). The SZZ algorithm works in two steps. First, bug fixing commits are identified. The SZZ algorithm tries to find a matching issue, based on the numbers found in the commit messages. In case any number is found, the algorithm tries to find an issue for the project that has the same number. If an issue is found, semantic checks for the following properties are performed (Śliwerski et al. 2005):

- The issue was resolved as FIXED at least once.
- The author of the commit is assigned to the issue.
- The title or description of the issue is contained in the commit message.
- One or more files that are changed by the commit are attached to the issue.

A commit is identified as bug fixing if there is at least one linked issue, that passes at least two of the above semantic checks. In case only one semantic check is passed, the commit is labeled as bug fixing, if the commit message contains a term like “bug” or “fix”, or it is clear that a number in the commit is a link to an issue, e.g., because the number starts with “Bug #111”, or the commit contains only a list of numbers.

Once a commit is identified as bug fixing, the second part of the SZZ algorithm is the identification of the inducing changes. SZZ identifies the last changes to each line that was touched as part of a bug fixing commit as candidate for an inducing change. All candidates, that took place before the reporting date of the issue are immediately considered as bug inducing changes. Changes that took place after the reporting date of the issue are suspect, because they were performed after the bug was already in the software. However, because of the chance of bad fixes or partial bug fixes, the changes are not automatically discarded. If the suspects are part of a bug fixing commit (partial fix) or the commit contains changes that are inducing for a different bug (weak suspect), they are considered as inducing. The remaining suspects are considered to be hard suspects and not inducing for the bug fix.

The identification of the bug inducing commits is only used for the just-in-time data. For the release level data sets, defects are assigned to releases based on the reporting date: all bugs that were reported in the first six months after the release are assigned to a release (Zimmermann et al. 2007).

While our focus is on defect prediction data, a study by Rodríguez-Pérez et al. (2018) further highlights the relevance of SZZ beyond defect prediction data. The study highlights that the original SZZ is commonly used, i.e., 38% of the identified literature. Another 40% of the identified literature refers to some adoption of SZZ, without specifying what this adoption is. Only 14% of the publications that use SZZ specifically mentioned that they use a variant that ignores cosmetic changes, like documentation changes.

In recent years quality problems with the labels produced by SZZ came into focus. The impact of using all changes in a bug fixing commit as foundation for the defect labeling was investigated in detail by Mills et al. (2018). The authors manually validated which files that were modified in a bug fixing commit were actually part of the bug fix and found that about 64% of file changes made in bug fixing commits are not part of the bug fixes, but other changes. They found that mistakenly identified files are due to code that is only added and not modified or deleted (46.58% of all cases), changes are performed on test code (30.90%), refactorings (8.73%), and changes of comments (8.49%). SZZ already ignores pure additions for the identification of inducing changes, because there is no prior commit, where the code was last changed. To the best of our knowledge, none of the SZZ implementations used to create the defect prediction data sets ignores test code, refactorings, or comments. This means that based on the estimation of Mills et al. (2018), about 34% of files in bug fixing commits are false positives, i.e., incorrectly identified as defective.

Another potential source of false positives of SZZ are commits that are mistakenly identified as bug fixes. With SZZ, there are two main sources for this problem: the first is due to the strategy for the identification of links between commits and issues, that works based on numbers. If a core developer of a project fixes a bug with an issue number like one, 256 or other frequently occurring numbers, every commit by this developer that contains this number will be identified as a bug fixing commit. While Bird et al. (2009a) found that this problem can be mitigated through additional filters, e.g., based on the date of the commit and the issue resolution, this is not part of the standard SZZ algorithm. There are also approaches that try to recover links between commits and issues, that are not explicit, e.g., ReLink (Wu et al. 2011). Such links also cannot be captured by SZZ. The second source of false positive for bug fixing commits are the issues themselves. According to Herzig et al. (2013), about 33% of issues that are reported as a bug in the issue tracking system are actually requests for new features, bad documentation, or simply result in refactorings. They found that due to this 39% of the identified defective files were actually not defective. A smaller study on this topic was conducted by Antoniol et al. (2008), who came on a smaller sample to a similar result.<sup>3</sup> To the best of our knowledge, only the NETGENE data is based on manually validated issues. That both happen in practice can, e.g., be seen with the issue NUTCH-1.<sup>4</sup> This test issue created by a core developer was not for any real bug in the software. However, all commits by this developer for the Apache Nutch project, where the

<sup>3</sup> Unfortunately, Antoniol et al. (2008) did not specify their criteria for the differentiation between bugs and enhancements and only stated that bugs are corrective changes. Due to the uncertainty of this term, we only compare our results to Herzig et al. (2013) in the following, as they provide clear guidelines for the identification of different issue types.

<sup>4</sup> <https://issues.apache.org/jira/browse/NUTCH-1>

message contains the number one will be mistakenly identified as bug fixing. Moreover, Herzig et al. (2013) note that these mislabels are mostly irrelevant for developers, but very important for data miners:

*“The question of whether an issue is a bug or not is a hard one. And the definition of a bug not only differs between users and developers but also between developers themselves. In principle, if an issue bothers the user, the developer should fix it, whether or not he considers it to be a bug or not. From this perspective, the issue report category does not matter. But a data miner building a defect prediction model must distinguish between bugs and non-bugs. Otherwise, the prediction model would predict changes, not defects.”* Herzig et al. (2013)

Another aspect related to this is to differentiate between *intrinsic* and *extrinsic* bugs, a notion recently introduced by Rodríguez-Pérez et al. (2020). This notion extends the concept that not all bugs have inducing changes (Rodríguez-Pérez et al. 2018). Intrinsic bugs are “introduced by one or more specific changes to the source code” (Rodríguez-Pérez et al. 2020) and extrinsic bugs were introduced, e.g., “from external requirements or changes to the requirements” (Rodríguez-Pérez et al. 2020). In the sense of Herzig et al. (2013) and our study, we only consider intrinsic bugs, because issues due external dependencies and changing requirements are not bugs, but feature requests for improvements. Thus, extrinsic bugs are not within the scope of defect prediction. Regardless, it is valid for the developers to label extrinsic bugs as bugs within the issue tracker. Thus, both the work by Herzig et al. (2013) and Rodríguez-Pérez et al. (2020) indicate that there are likely many issues mislabeled as bug in issue trackers, at least from the point of view of defect prediction that is interested in intrinsic bugs.

The assignment of the identification of the inducing file actions has also come under scrutiny. Da Costa et al. (2017) investigated how well SZZ identifies inducing commits and found that SZZ implementations perform better, if they ignore changes that only modify whitespaces. This is in line with the findings by Mills et al. (2018). Moreover, they suggest that using the affected versions field of issue tracking systems can improve the validity of SZZ results. Developers can use this field to mark versions of a software that are affected by a specific defect. Regardless, all data sets we discussed in Section 2 use a basic SZZ variant that does not ignore whitespace changes or use the affected version field. However, Da Costa et al. (2017) do not suggest how the affected version should be integrated into the SZZ algorithm. Rodríguez-Pérez et al. (2020) went one step further with their investigation of the bug inducing changes of SZZ and manually validated the inducing changes of 86 bugs for two projects.<sup>5</sup> They found that about 70% of the correct inducing changes can be found.<sup>6</sup> However, they also find that of all inducing changes identified by SZZ only 24% are actually true positives. However, they only consider the first change that is related to the introduction of the bug as inducing. Additional changes that are made later, which may have also contributed to the bug are also identified as false positive. Regardless, these findings indicate that SZZ may overestimate the number of bug inducing commits.

Yatish et al. (2019) directly used the affected version field to assign defects to releases. Theoretically, this could lead to a perfect assignment of defects to releases. However, this depends on the maintenance of this field in the issue tracker by the developer. In practice, the value of this field is usually set by the reporter of an issue as the version of the software

<sup>5</sup> These are the intrinsic bugs from their article.

<sup>6</sup> 76% for the Nova project, 63% for the Elasticsearch project

that the reporter currently uses. An analysis if this defect was already in the software in earlier versions is often not performed and, consequently, the field is not updated. An example for this is the issue CAY-1657.<sup>7</sup> The affected version of that issue is 3.1M3. However, as part of the description, the author writes “*I am sure this affects ALL versions of Cayenne, but my testing is done on 3.1 M3/M4*”. That this is not an isolated problem is also suggested by further anecdotal evidence, e.g., on the Apache Spark mailing list.<sup>8</sup> Additionally, the affected version field is often not used at all. For example, Da Costa et al. (2017) report that only 1,268 of the 32,033 linked bug reports contained data in the affected versions field, i.e., less than 4% of the issues. Thus, this approach is likely to lead to many false negatives, i.e., mistakenly not assigned defects to releases that are affected, because the field is not maintained properly.

Regarding the six month time frame that was proposed by Zimmermann et al. (2007) for the release assignment with SZZ, we could not find an empirical basis for this in the literature. Yatish et al. (2019) already broke with this rule and demonstrated that there are both defects within the six-month time that were introduced after the release, leading to false positives, as well as defects that were fixed more than six months after the release, leading to false negatives. Thus, the work by Yatish et al. (2019) provides a strong indication that the complete history of a project after the releases should be considered for assigning defects to releases. We are not aware of other studies that evaluate the impact of the six month time frame.

Finally, the mislabels in data we discuss above may be acceptable, if they do not affect the prediction models. The literature indicates that there is a threat to the validity of the results due to this kind of noise in the data. According to Herzig et al. (2013), the mislabeled issues translate into bad estimation of defect proneness of files, because 16% to 40% of the top 10% of files with most defects change due to the correction of issue types. Additionally, Tantithamthavorn et al. (2015) found that while the models trained on noisy data may achieve similar *precision* as models trained without noise, their *recall* is typically reduced by 32%-44%, suggesting a degradation in prediction performance due to the noise. We note that the studies by Herzig et al. (2013) and Tantithamthavorn et al. (2015) both only consider mislabels due to wrong issue types. The other potential reasons for mislabels are not considered.

### 3.2 Incomplete feature sets

The features (or independent variables) are at the core of every learning algorithm: they are the information that is available to make a decision or they can be correlated with the outcome. If good features are missing in defect prediction data, this can lead to a loss in performance. This loss in performance can vary between algorithms used for training prediction models. This leads to a troubling question: if researchers find a difference between defect prediction approaches on data that does not contain all important features, would the difference still be there if all relevant features are available? As a consequence, conclusions regarding the performance of defect prediction algorithms based on data that does not contain features that were demonstrated to be valuable have a severe problem with the external validity of the findings. We note that the curators of data sets are not to fault for

<sup>7</sup> <https://issues.apache.org/jira/browse/CAY-1657>

<sup>8</sup> <https://hdl.handle.net/21.11101/0000-0007-E183-6>

features that are not within their data sets. Most data sets are collected based on what is currently known in the state of the art and contain the currently relevant features. However, this problem can easily manifest if the data is used for years and, in the meantime, the state of the art progresses and suggests that important features may be missing from older data sets. The defect prediction literature suggest that there are at least several such kinds of such features: CHURN related features, as well as different variants of aggregated features.

CHURN related features are based on the findings that defects are more likely in often changed parts of a software, especially in case a prior change already removed defects (Rahman et al. 2011). Moreover, such features may also include information about past defects (BUG), e.g., the number of prior defects that were already corrected in a file. Publications that include CHURN features consistently find that CHURN features are among the most important predictors for defects. This already started with the pioneering work by Ostrand et al. (2005) at Bell labs and was later confirmed, e.g., by Moser et al. (2008) and D'Ambros et al. (2012) proposed to use the concepts of entropy and linear decay to further improve the impact of CHURN features, which was also confirmed by D'Ambros et al. (2012). Another strong indicator for the importance of CHURN related features is that they are the main features in the just-in-time defect prediction data sets in combination with code ownership. Regardless, the only release-level data sets that contain CHURN features are ECLIPSE, AEEEM, MJ12A, and RNALYTICA.

However, there is another problem that is known related to CHURN features, i.e., those that take the history into account. The current data sets collect this data only based on the main development branch of a repository. However, due to the advent of Git as version control system, features are often developed on branches. These feature branches are merged through a single merge commit into the main development branch. If only the main branch is considered, information about the history of the development is ignored. Only the RNALYTICA data may consider branches for the CHURN feature, as the authors state that “one must focus on the development activities of interest that correspond to the release branch” (Yatish et al. 2019). However, Yatish et al. (2019) do not discuss further how branches are handled, which is why we are not sure if and how different branches affect the collection of the CHURN metrics or if the branches are used for the assignment of defects to releases. To the best of our knowledge, the other current defect prediction data sets only use the main development branch. Kovalenko et al. (2018) evaluated the impact of using feature branches as part of the data collection on results of various software mining approaches, including defect prediction. They found that the performance of defect prediction may improve slightly, if data from branches is included in the mining process. In particular, they found that the results are never worse. Thus, while this problem probably does not have the same impact as the general lack of CHURN features, this could still lead to underestimating the performance of defect prediction approaches.

There are other types of features, which are ignored by current defect prediction data altogether. Zhang et al. (2017) found that how measurements from lower level software artifacts are aggregated into metrics for higher level artifacts impacts the performance of defect prediction models. A popular example of such a metric is the Weighted Method per Class (WMC) metric from the Chidamber and Kemerer's metrics for object-oriented software (Chidamber and Kemerer 1994). This metric measures the complexity of a class by summation of the complexity of the methods. However, Zhang et al. (2017) found that aggregation through a single marker, like summation, can actually lead to inferior defect prediction models. Instead, they recommend to use different aggregation strategies to provide multiple aggregations such as summation, median, and the standard deviation and later use feature reduction techniques to remove redundancies. We note that while Zhang

et al. (2017) found that using all aggregation schemes leads to the best results overall, they also observed that using only summation is a close second, i.e., the advantage of using multiple summation schemes may be negligible. Regardless, to the best of our knowledge, none of the current publicly available data sets support this kind of analysis.

Plosch et al. (2008) analyzed the correlation between warnings produced by the static analysis tools FindBugs<sup>9</sup> and PMD.<sup>10</sup> They found that the warnings have a stronger correlation with bugs than OO and SIZE metrics. In contrast, Rahman et al. (2014) found that features from static analysis tools do not improve the performance of defect prediction models. Due to the contradictory results, we believe that more data is required, e.g., through additional studies that evaluate the impact of such features. Bird et al. (2011) found that code ownership is correlated with defects and may be used to improve defect prediction models. Palomba et al. (2019) found that it may be useful to derive code smells from static metrics, to improve defect prediction models. We note that this data need not be actually part of the data sets, but can be derived as long as static source code metrics like number of methods are available. Bowes et al. (2016) found that dynamically collected features from mutation testing can also improve defect prediction models. A notable difference between the mutation features by Bowes et al. (2016) and all other features we discuss is that they require that the software can be compiled and executed. Moreover, Thongtanunam et al. (2016) found a correlation between code review activity and defects, however, they did not study if this correlation can improve defect prediction models. Similarly, Spadini et al. (2018) found that smelly test cases are correlated with defects, but have also not studied if this correlation can improve prediction models.

## 4 Improving defect prediction data collection

We now outline how we believe that defect prediction data can be improved to avoid the problems discussed in the literature. Table 2 summarizes the key problems we discussed in Section 3 as well as the solutions we suggest within this section. In the following, we describe the improvements in detail. Wherever possible, we re-use or adapt existing solutions from the literature and rather propose to combine the separate findings and solutions.

### 4.1 Improving defect labeling

In principle, we believe that the SZZ algorithm (Śliwerski et al. 2005) provides a very good foundation for the labeling of defects. Thus, we do not propose a radically new algorithm, but rather modify the SZZ algorithm to work well together with the Jira issue tracking system, as well as take the problems from the state of the art into account.

As we described in Section 3.1, the SZZ algorithm may suffer from misidentified defect links due to often occurring numbers like 1, that are not related to an issue number. We note that SZZ was designed with the Bugzilla issue tracker in mind. Here, issues identifiers are just a single number, i.e., there is no good resolution for this. This is different in Jira, where the issue identifiers have the structure <PROJECTID>-<NUMBER>. Thus, we modified the identification of linked issues to take this structure into account to define a

<sup>9</sup> <http://findbugs.sourceforge.net/>

<sup>10</sup> <https://pmd.github.io/>

new linking approach we call JL<sup>11</sup>. JL is conceptually identical to SZZ, the difference is the focus on Jira instead of Bugzilla.

JL exploits the semantics of the string descriptor of Jira, i.e., we search for the complete identifier in commit messages, and not just any number. The drawback of this is that spelling problems in the project identifier would mean that we miss issue links. To account for this, we manually check all strings that are a combination of a string followed by an integer and supply a list with all wrong spellings, such that they can be corrected by the linking algorithm. While this requires manual effort, this can be done in a matter of minutes. The problem with JL is that it also captures links to commits, where an issue is only mentioned, but not actually addressed. Moreover, if the numbers are alone, i.e., not part of a Jira identifier, JL may miss links. SZZ can detect such links, because the algorithm works only on numbers. Thus, to account for links that SZZ detects but JL misses, we semi-automatically analyze all messages of commits that contain a link determined by JL or SZZ. The goal of this additional step is to create a validated set of links from commits to issues. For many commits, this is not a problem. In case we determined only one link from a commit to an issue, and this link is established because the exact name of the issue occurs at the start of the commit message, we assume that this commit addresses the mentioned issue. An example for such a commit message from the ant-ivy project is “*IVY-1391 - IvyPublish fails when using extend tags with no explicit location attribute*”. An expert must inspect all remaining commit messages for which a link was detected regarding two criteria: 1) are the links correct, i.e., are the issues actually mentioned by the commit message and 2) which issues were actually addressed by the commit and which issues were only mentioned. Only correct links that were actually addressed in the commit are then validated by the expert. We refer to the combination of JL with the validated data as JLM<sup>12</sup>.

For both JL and JLM we use rules similar to SZZ (Śliwerski et al. 2005) to determine if a commit is bug fixing:

- a bug fixing commit must have a validated link to an issue that is validated as BUG; and
- the linked issues must have been in a closed or resolved state at any point in its lifetime.

The major assumption behind JLM is that the labeling of issues as bugs in the issue tracker is correct. Since we know from the results by Herzig et al. (2013) that this is often not the case, we propose that the type of issues should be manually validated. Taking pattern from Herzig et al. (2013), we used the following five categories.

- BUG for null pointer exceptions, runtime or memory issues caused by defects, or semantic changes to the code to perform corrective maintenance task. This is the same as the BUG category from Herzig et al. (2013).
- IMPROVEMENT for feature requests or the non-corrective improvement of existing features. This bundles the categories RFE (Feature Request), IMPR (Improvement Request), REFAC (Refactoring Request) from Herzig et al. (2013).
- TEST for issues that only require changes to the software tests. This category was not used by Herzig et al. (2013).
- DOC for requested changes to the documentation of the software. This is the same as the DOC category from Herzig et al. (2013).

---

<sup>11</sup> Short for Jira Links

<sup>12</sup> Short for Jira Links Manual

**Table 2** Summary of problems with defect prediction data creation and the improvements we consider

Problem	Improvement	Identifier
Wrong links to bugs	Exploit Jira identifier pattern	JL
	Manually validate correctness of links	JLM
Wrong issue types	Manually validate issue types	JLMIV
Too many inducing changes	Exclude non-production code, whitespace and documentation changes	JLMIV+
	Ignore refactorings	JLMIV+R
6-month time-window	Use affected versions field	JLMIV+RAV
	Use commit graph	IND-JLMIV+R
Lack of features	Provide a large set of features	-

The problems are described in detail in Section 3, the improvements are described in Section 4. The identifiers refer to the SZZ variant that implements the improvement

- OTHER for all other issues, e.g., questions or brainstormings. This is the same as the OTHER category from Herzig et al. (2013).

Our reasons for the differences between our work and Herzig et al. (2013) are mainly due to the efficiency, i.e., a faster manual validation process. The different types of IMPROVEMENT are often very hard to distinguish based on the description of an issue, while they all lead to improvements of the software. We kept the DOC and OTHER and added TEST because these are clearly distinguishable from the other issue types. For maximal efficiency, one could also use a simple binary classification, i.e., BUG, and not BUG. For our research, we decided against this to facilitate research using this data regarding the automated correction of issue types.

We propose that the manual validation should be done in two steps. First, all linked issues of type BUG should be independently labeled by two experts. The experts have access to the description and comments of the issue, as well as the source code that was changed as part of the commits that were linked to this issue. If both experts agree, we assume their assessment is correct. In case of disagreement, the issues should be presented to a panel of at least two experts, one of which did not participate in the initial labeling. The experts then decide the issue type based on the blinded labels determined by the two experts, the issue description and comments, and the source code changes. This validation should be based on the principle “innocent unless proven guilty”, i.e., in case there is doubt whether the issue is a BUG or not, the experts should not modify the label, i.e., always label such issues as BUG.

While other issues of type other than BUG could also be manually analyzed, the work by Herzig et al. (2013) showed that bugs are almost always correctly labeled as BUG. Thus, we suggest to restrict the manual labeling to issues of type BUG, due to the time intensive nature of the manual labeling step. In the following, we use JLMIV<sup>13</sup> to refer to bug fix labeling that accounts for the manual validation of issue types.

The results by Da Costa et al. (2017) and Mills et al. (2018) indicate problems with the way SZZ determines bug inducing changes. Both studies highlight that changes that only

<sup>13</sup> Jira Links Manual Issues Validated

affect the whitespaces or modify comments should be ignored during the identification of bug inducing changes. To address this concern, we use a regular expression approach to identify changes that only modify comments or whitespaces and ignore them, similar to what Kim et al. (2006) proposed. Mills et al. (2018) also found that changes to tests are also inadvertently covered during the search for bug inducing changes. Since bugs can, by definition, only be in production code, these changes are all false positives. We extend this notion to changes to examples or tutorials, which may contain code files. These are also not production code, but documentation of the project. We are not aware of an SZZ variant that takes this into account. Additionally, we ignore changes that can be explained by refactorings similar to the work by Neto et al. (2018, 2019). Based on the results by Mills et al. (2018), these modifications should be able to account for about 94.6% of the false positive bug inducing changes. We refer to this improvement of the detection of inducing changes as JLMIV+R.

Da Costa et al. (2017) also noted that SZZ should take the affected version field of issue tracking systems into account, because this gives further information about the time when the software was defective. In their study, Da Costa et al. (2017) mark all changes after the release of the earliest affected version as incorrect. Yatish et al. (2019) assign bug fixes directly to releases based on the affected version field and do not use SZZ at all. In comparison to Yatish et al. (2019) and Da Costa et al. (2017), we do not consider the affected version label as ground truth, due to the reasons we discussed in Section 3.1. Because we assume that the affected version field is likely missing completely or at least missing older releases that were also affected by a bug, we believe that the approaches by Da Costa et al. (2017) and Yatish et al. (2019) are too strict for real world data and would lead to false negatives, i.e., not assigning bugs to affected releases because the affected versions field is incomplete. A less strict variant that takes the affected version field into account would be to integrate the affected version in the strategy to determine inducing changes of the SZZ algorithm, based on how the bug reporting date in the issue tracker is used. SZZ assumes that changes after the reporting date of a bug are suspect, but may still be inducing for the bug fix, e.g., because they are bad fixes or partial fixes. The same logic can be applied to changes that happen after the release of an affected version. Thus, our proposal to utilize the affected version field to enhance the SZZ algorithm is to extend the notion of suspect changes and use the minimal date of the release of all affected versions and the reporting date of the bug as the boundary for suspects. We refer to this approach as JLMIV+RAV.

For release level-data, there is another problem to consider, i.e., how we decide which releases were affected by which bugs and label files within releases accordingly. We already discussed that the six month timeframe has no empirical foundation and leads to mislabels, as demonstrated by Yatish et al. (2019). However, since we believe that the affected version field is unreliable (see sections 3.1 and 5.6), we propose a different approach than Yatish et al. (2019). We propose an approach that is directly based on the bug inducing changes. If the bug inducing changes are determined correctly, this means that the bug was in the software, when the last non-suspect bug inducing change took place. Suspect changes have to be excluded here, because there is confirmation in the issue tracking system that the bug already affected the system, when this change took place. Similarly, we can determine when the bug was fixed as the last bug fixing commit for the bug. Consequently, a bug affects a release, if all non-suspect inducing changes took place before the release, i.e., there is a path in the commit graph from all inducing changes to the release, and at least one bug fixing commit took place after the release. We refer to this approach as IND-JLMIV. This approach is in line with the theory of how bugs are introduced by Rodríguez-Pérez et al. (2020), which is based on the idea that an intrinsic bug is within the

software once the changes that lead to the bug are performed. Assuming that all inducing changes are required to produce the bug, the last inducing change is the point in time where the bug was introduced into the software.

## 4.2 Reducing the lack of features

The remedy for the lack of features is straight forward: collect data for more features based on prior work from the state of the art. This requires no change in the general approach for data collection, i.e., to compute features from repositories mostly using off-the-shelf static analysis tools. The only required change is that more tools are used to collect these features to obtain a broader feature set.

## 5 Empirical study

Within this section, we describe the results of an empirical study we conducted. The goal of this study was two-fold. On the one hand, we want to determine the impact of the problems we discussed in Section 3. On the other hand, we want to determine if our proposed improvements can effectively resolve these problems. Figure 1 outlines our experiment. First, we go step by step through the defect labeling. Then, we use the resulting defect prediction data to evaluate the effect of mislabels and the lack of features on the prediction performance of a defect prediction model.

For all of these analyses we only presented aggregated results for all projects we analyzed. The detailed tables with all data, including all code required for an exact replication of our work as well as the collected release-level and just-in-time defect prediction data sets, are part of the supplemental material.<sup>14</sup>

### 5.1 Data

We conducted our study on projects from the Apache Software Foundation.<sup>15</sup> Apache projects must have reached a certain level of maturity in order to be considered as a top-level project. Especially the use of Jira as issue tracking system is highly recommended and followed by most Apache projects. Additionally, Bissyandé et al. (2013) found that “Apache developers are meticulous in their efforts to insert bug references in the change logs of the commits”. This is an important property for the projects under consideration because it removes the need for link recovery and we can safely assume that links from commits to issues are available in the data. Yatish et al. (2019) also used a convenience sample of Apache projects for the same reason. A similar reasoning was also used for studies other than defect prediction, e.g., by Di Penta et al. (2020). To further ensure the maturity of projects, we used the criteria listed in Table 3. In addition, we had a soft criterion that we focused on projects with less than 10.000 commits on the main development branch. The reason for this is the very high demand on the resources for the collection of the metric

<sup>14</sup> <https://doi.org/10.5281/zenodo.5675024>

<sup>15</sup> <http://www.apache.org>

data for every commit in the Git repository.<sup>16</sup> Please note that the total number of commits can still be larger than 10.000 commits, because of we collect the data for all branches.

Table 4 lists the 38 projects for which we collected data, including the versions of the 398 releases for which we collected release-level data. The releases were determined using the project homepages. For each release, we looked up the commit of the release in the Git repository. For most releases, there was a related tag in the Git repository. If this was not the case, we manually analyzed the commit history to determine the release commit, using the information we found on the project homepage, as well as related tags and branches.

We collected all data using the SmartSHARK (Trautsch et al. 2017, 2020) platform. The advantage of this approach is that we aggregate all collected data in a single MongoDB database, including the results of our manual validations. Details regarding the collection of this data can be found in Appendix 1.

The data for just-in-time defect prediction is provided through the MongoDB. Additionally, we prepared release-level defect prediction data. As features we collected static code metrics, clone metrics, PMD warnings, AST node counts, the number of the different kinds of changes (Fluri et al. 2007) and refactorings (Silva and Valente 2017) from the last six months, and churn metrics proposed by Moser et al. (2008), Hassan (2009), and D'Ambros et al. (2012), and all thirteen aggregations that were proposed by Zhang et al. (2017) for the software metrics are not on the file level, i.e., class, method, interface, enum, attribute and annotation metrics. The data set contains a total of 4198 features. We decided not to add features with code smells (Palomba et al. 2019) to the data set, because these can be calculated indirectly from the available source code metrics and definition of smells like god class may change over time. Additionally, features based on mutation testing are not available, because retrospective execution of tests is, unfortunately, often not possible (Tufano et al. 2017). We also have not added features regarding test smells (Spadini et al. 2018) and review activity (Thongtanunam et al. 2016), because current results only show correlation with defects, but have not yet shown that these features may actually improve defect prediction models.

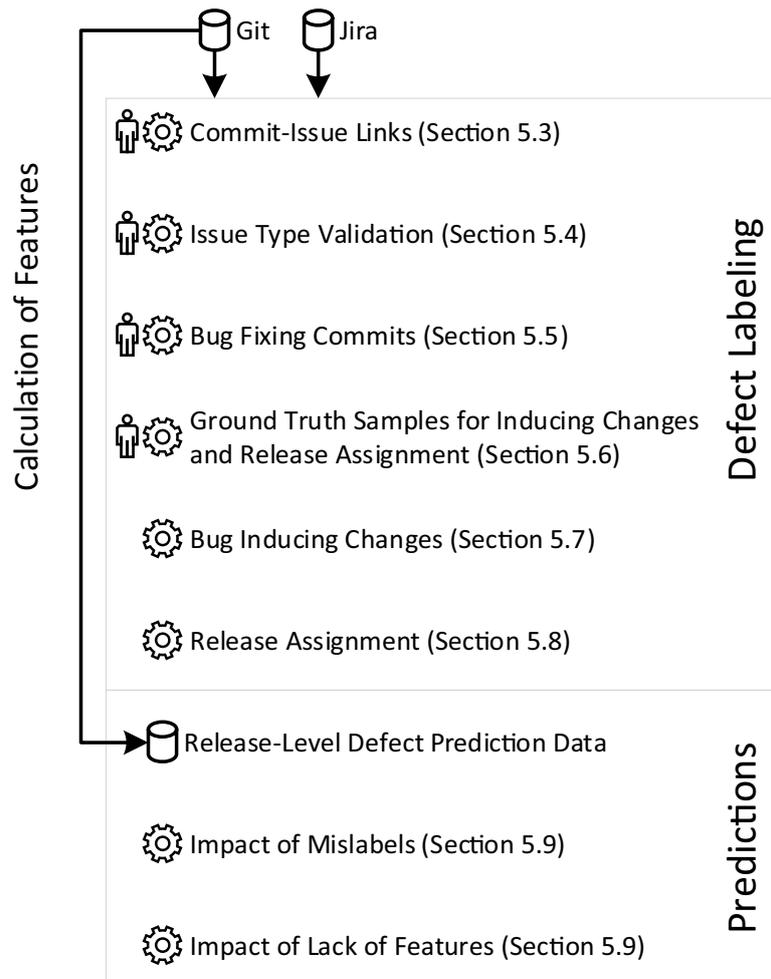
The defect labels are assigned using the JLMIV+IND approach. For each file, we stores the number of bugs that were fixed in this file. Moreover, the data contains a matrix that has as columns the issues and as rows the files. This matrix contains the value one, if the issue affected a file. This allows a fine-grained analysis which issues affected which file in a release and also which issues affected multiple files and is required for the analysis of costs (Herbold 2019). The column names of this matrix contain the identifier of the issue, the severity of the issue, and the date of the last bug fixing commit for the issue. This meta data allows for later filtering, e.g., to exclude trivial bugs or to ensure that there is no data leakage, e.g., to exclude bugs that were not yet fixed at the time of a release for which a prediction model is trained.

## 5.2 Evaluation criteria

For the evaluation of the different aspects of defect labeling, we determine baselines and then determine how well other approaches perform with respect to the baseline. The concrete baselines are discussed at the beginning of sections 5.3, 5.4, 5.5, 5.7, and 5.8.

<sup>16</sup> While this criterion is irrelevant for the evaluation of the defect labeling, we selected the projects also with the goal to provide a new defect prediction data set.

**Fig. 1** Overview of the Experiment. The human figure indicates that we used manual validation



**Table 3** Criteria for the inclusion of projects

Criterion	Rational
Uses Git	Most projects either already use a Git repository, or provide a Git mirror of a SVN repository.
Java as main language	Our static analysis only works for Java code.
Uses Jira	The Jira of the Apache Software Foundation is the main resource for tracking issues of most Apache projects.
At least two years old	Project as a sufficient development history.
Not in incubator stage	Project has been fully accepted by the Apache Software Foundation.
> 100 Issues in Jira	Project is mature and actively uses Jira
> 1000 Commits	Project has a sufficient development history.
> 100 Files	Project should have a reasonable size.
Activity since 2018-01-01	Project is still active in both Jira and Git.

We evaluate two aspects: how many artifacts determined by the baseline are correctly identified and how many additional artifacts are identified. Artifacts are, for example, links from commits to issues, commits, or files. This approach is similar to the concept of true positives and false positives. For example, if a baseline determines  $n$  artifacts as defective, and an approach for comparison  $A$  identifies  $n_{tp}$  of these artifacts as defective

**Table 4** Apache projects and releases used for the empirical study

Project	Commits	Bugs	Releases
ant-ivy	3,189	535	1.4.1, 2.0.0, 2.1.0, 2.2.0, 2.3.0, 2.4.0
archiva	10,262	542	1.0, 1.1, 1.2, 1.3, 2.0.0, 2.1.0, 2.2.0
calcite	2,926	842	1.0.0, 1.1.0, 1.2.0, 1.3.0, 1.4.0, 1.5.0, 1.6.0, 1.7.0, 1.8.0, 1.9.0, 1.10.0, 1.11.0, 1.12.0, 1.13.0, 1.14.0, 1.15.0, 1.15.0
cayenne	6,619	530	3.0.0, 3.1.0
commons-bcel	1,429	53	5.0, 5.1, 5.2, 6.0, 6.1, 6.2
commons-beanutils	1,341	76	1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7.0, 1.8.0, 1.9.0
commons-codec	1,838	64	1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 1.10, 1.11
commons-collections	3,380	115	1.0, 2.0, 2.1, 3.0, 3.1, 3.2, 3.3, 4.0, 4.1
commons-compress	2,755	172	1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 1.10, 1.11, 1.12, 1.13, 1.14, 1.15, 1.16
commons-configuration	3,717	188	1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 1.10, 2.0, 2.1, 2.2
commons-dbcp	2,205	127	1.0, 1.1, 1.2, 1.3, 1.4, 2.0, 2.1, 2.2.0, 2.3.0, 2.4.0, 2.5.0
commons-digester	2,525	26	1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 2.0, 2.1, 3.0, 3.1, 3.2
commons-io	2,262	131	1.0, 1.1, 1.2, 1.3, 1.4, 2.0, 2.1, 2.2, 2.3, 2.4, 2.5
commons-jcs	1,622	80	1.0, 1.1, 1.3, 2.0, 2.1, 2.2
commons-jexl	3,276	84	1.0, 1.1, 2.0, 2.1, 3.0, 3.1
commons-lang	5,792	318	1.0, 2.0, 2.1, 2.2, 2.3, 2.4, 2.5, 2.6, 3.0, 3.1, 3.2, 3.3, 3.4, 3.5, 3.6, 3.7
commons-math	7,222	415	1.0, 1.1, 1.2, 2.0, 2.1, 2.2, 3.0, 3.1, 3.2, 3.3, 3.4, 3.5, 3.6
commons-net	2,270	176	1.0.0, 1.1.0, 1.2.0, 1.3.0, 1.4.0, 2.0, 2.1, 2.2, 3.0, 3.1, 3.2, 3.3, 3.4, 3.5, 3.6
commons-scxml	1,216	70	0.5, 0.6, 0.7, 0.8, 0.9
commons-validator	3,416	73	1.0, 1.1.0, 1.2.0, 1.3.0, 1.4.0, 1.5.0, 1.6.0
commons-vfs	2,212	156	1.0, 2.0, 2.1, 2.2
deltaspike	2,311	302	0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 1.0.0, 1.1.0, 1.2.0, 1.3.0, 1.4.0, 1.5.0, 1.6.0, 1.7.0, 1.8.0
eagle	1,119	225	0.3.0, 0.4.0, 0.5.0
giraph	1,121	337	0.1.0, 1.0.0, 1.1.0
gora	1,329	113	0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8

**Table 4** (continued)

Project	Commits	Bugs	Releases
jspwiki	8,809	274	1.4.0, 1.5.0, 1.6.0, 1.7.0, 1.8.0, 2.0.36, 2.2.19, 2.4.56, 2.6.0, 2.8.0, 2.9.0, 2.10.0
knox	2,069	568	0.3.0, 0.4.0, 0.5.0, 0.6.0, 0.7.0, 0.8.0, 0.9.0, 0.10.0, 0.11.0, 0.12.0, 0.13.0, 0.14.0, 1.0.0
kylín	12,975	732	0.6.1, 0.7.1, 1.0, 1.1, 1.2, 1.3, 1.5.0, 1.6.0, 2.0.0, 2.1.0, 2.2.0
lens	2,418	397	2.6.0, 2.7.0
mahout	4,167	513	0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 0.10.0, 0.11.0, 0.12.0, 0.13.0
manifoldcf	5,936	633	0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 1.10, 2.0, 2.1, 2.2, 2.3, 2.4, 2.5, 2.6, 2.7, 2.8, 2.9, 2.10
nutch	3,532	643	0.7, 0.8, 0.9, 1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 1.10, 1.11, 1.12, 1.13, 1.14, 2.0, 2.1, 2.2, 2.3
opennlp	2,685	219	1.6.0, 1.7.0, 1.8.0
parquet-mr	2,249	1413	1.0.0, 1.1.0, 1.2.0, 1.3.0, 1.4.0, 1.5.0, 1.6.0, 1.7.0, 1.8.0, 1.9.0
santuario-java	3,376	83	<i>1.0.0, 1.2, 1.4.5, 1.5.9, 2.0.0, 2.1.0</i>
systemml	6,196	452	0.9, 0.10, 0.11, 0.12, 0.13, 0.14, 0.15, 1.0.0, 1.1.0, 1.2.0
tika	4,933	605	0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 0.10, 1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 1.10, 1.11, 1.12, 1.13, 1.14, 1.15, 1.16, 1.17
wss4j	3,734	241	1.5.0, 1.6.0, 2.0.0, 2.1.0, 2.2.0

We did not assign any bugs to releases in italics as part of our empirical study

as well, but also  $n_{fp}$  additional artifacts, we say that A identifies  $\frac{n_{tp}}{n}$  artifacts correctly as true positives and  $\frac{n_{fn}}{n}$  additional false positive artifacts. For these comparisons, we report the median and the Median Absolute Difference (MAD) which is defined as  $MAD = 1.4826 \cdot \text{median}(|x_i - \text{median}(X)|)$  for a sample  $X = \{x_1, \dots, x_m\}$  (Rousseeuw and Croux 1993). The median and the MAD have the advantage over the mean value and the standard deviation that they are robust in case of non-normal distributions, which is the case for most data in our empirical study. The value 1.4826 is a scaling factor for MAD that makes the values of MAD similar to the standard deviation of normally distributed data (Rousseeuw and Croux 1993).

We use two independent researchers for the labeling of issue types of bug issues. We report Cohen's  $\kappa$  (Cohen 1960) to estimate the reliability of the consensus, which is defined as

$$\kappa = \frac{p_0 - p_e}{1 - p_e} \quad (1)$$

where  $p_0$  is the observed agreement between the two raters and  $p_e$  the probability of random agreements.  $p_e$  depends on the number of categories  $k$  and is defined as

$$p_e = \frac{1}{N^2} \sum_{i=1}^k n_i^1 \cdot n_i^2 \quad (2)$$

where  $N$  is number of issues and  $n_i^1$ , resp.  $n_i^2$  is the number of times researcher 1, resp. 2 labeled an issue as category  $i$ . We use the table from Landis and Koch (1977) for the interpretation of  $\kappa$  (see Table 5).

We also evaluate defect prediction models as part of our empirical study, to evaluate the impact of mislabels and feature sets. To evaluate the practical relevance of differences between the models, we evaluate how training on noisy data or fewer features impacts the cost effectiveness of defect prediction models based on the cost model by Herbold (2019). The advantage of using the cost model by Herbold instead of commonly used metrics like recall, precision, or F-measure is that the values of these metrics are not reliable measures with respect to the cost saving potential of defect prediction models Herbold (2019). Moreover, Yao and Shepperd (2020) found that many commonly used measures are not able to reliably determine if models are better than random predictions and suggest to use Matthews Correlation Coefficient. However, since the relationship between MCC and costs is also unclear, which is why we decided to directly evaluate costs.

The cost model estimates boundaries on the ratio between costs of quality assurance and costs of defects. Defect predictions can save costs for a project based on the ratio  $C$  between the costs for quality assurance and the cost of defects. The cost model estimates lower and upper boundaries for  $C$  that must be fulfilled to allow the defect prediction model to save costs. For cost ratios  $C$  less than the lower boundary, it would be better to not perform any additional quality assurance, because the quality assurance would be more expensive than the cost of the defects. The lower boundary increases with more predictions of files as defective (regardless whether the predictions are correct) and decreases with more defects being predicted. Thus, the lower boundary penalizes false positive predictions and rewards true positive predictions. For cost ratios  $C$  greater than the upper boundary, it would be better to apply additional quality assurance everywhere, because the costs for the defects would be higher than the investment for the quality assurance. The upper boundary increases with fewer predictions of files as defects (regardless whether the predictions are

correct) and decreases with more defects missed by the predictions. Thus, the upper boundary penalizes false negatives and rewards true negatives. An additional caveat of the cost model is that it accounts for the  $m$ -to- $n$  relationship between files and defects, i.e., that each file may be affected by multiple defects and the each defect may affect multiple files. As a result, the cost model does not use the confusion matrix for the estimation of true predictions of defects, but rather the bug-issue matrix to evaluate if all files that are affected by a defect are predicted by the defect prediction model. We use the size in logical lines of code as proxy for the effort for the quality assurance of the artifacts. Moreover, we assume that quality assurance is perfect, i.e., that all predicted defects are actually found by subsequent quality assurance measures. Following Herbold (2019), we can thus compute the boundaries on  $C$  as

$$\frac{\sum_{s \in S: h(s)=1} size(s)}{|D_{PRED}|} < C < \frac{\sum_{s \in S: h(s)=0} size(s)}{|D_{MISS}|} \quad (3)$$

$S$  is the set of files for which defects are predicted,  $h$  is the defect prediction model,  $D$  is the set of defects all  $d \in \mathcal{P}(S)$ <sup>17</sup>,  $D_{PRED} = \{d \in D : \forall s \in d \mid h(s) = 1\}$  is the set of predicted defects, and  $D_{MISS} = \{d \in D : \exists s \in d \mid h(s) = 0\}$  is the set of missed defects.

We use Demsar's guidelines (Demšar 2006) for the comparison of classifiers to evaluate if differences of the lower boundary and upper boundary are significant. The cost boundaries are not normally distributed and can even be infinite.<sup>18</sup> Therefore, we use the Friedman test (Friedman 1940) with the post-hoc Nemenyi test (Nemenyi 1963) to evaluate significant differences. In case differences are significant, we use Cliff's  $\delta$  (Cliff 1993) to estimate the effect sizes. According to Romano et al. (2006), the effect is negligible if  $\delta < 0.147$ , small if  $0.147 \leq \delta < 0.33$ , medium if  $0.33 \leq \delta < 0.474$  and large if  $\delta \geq 0.474$ . Since there is no guarantee that the defect prediction model can actually save costs, i.e., that the lower boundary of the model is less than the upper boundary, we also report the percentage of releases for which the defect prediction cannot save costs because the lower boundary is greater or equal to the upper boundary.

### 5.3 Identification of issue links

The first step of defect labeling is the identification of links between commits and issues. We restricted the analysis only to links to issues of type BUG that were closed and fixed at least once in their lifetime. Thus, we restrict this analysis to the links to issues that are relevant for defect labels.

Our JLM approach that is based on a semi-automated validation of the links found by SZZ and JL identified 18,721 correct links from commits to issues. 5,311 of these links were manually validated by the first author of this article, the remaining 13,410 links were validated by our heuristic, i.e., had a link to a single issue directly at the beginning. We sampled 1000 of the links that were validated by our heuristic to evaluate the correctness of the heuristic. The heuristic was correct in all cases. Moreover, we randomly sampled 1000 commits from the all commits for which neither SZZ nor JL found a link to a Jira issue. We found no links to Jira that we failed to identify. However, we found 40 links to

<sup>17</sup> In the cost model, a defect is defined by the set of files that it affects.

<sup>18</sup> For example, the upper bound is infinite if no defects are missed and a single file is not predicted as defective.

**Table 5** Interpretation of Cohen's  $\kappa$  according to Landis and Koch (1977)

$\kappa$	Interpretation
<0	Poor agreement
0.01 – 0.20	Slight agreement
0.21 – 0.40	Fair agreement
0.41 – 0.60	Moderate agreement
0.61 – 0.80	Substantial agreement
0.81 – 1.00	Almost perfect agreement

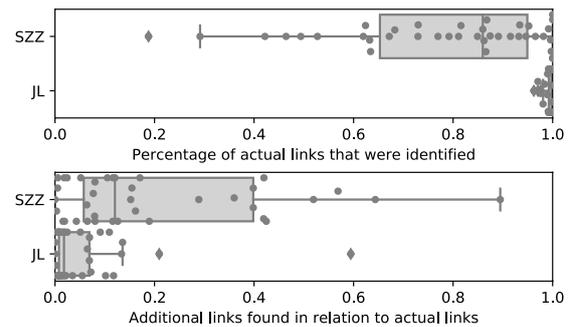
the Bugzilla<sup>19</sup> of the Apache Software Foundation. Since the data is not available anymore, we could not validate, if these issues are bugs or improvements. Therefore, about 4.0% of the commits for which we found no link may also be bug fixing, but cannot be determined as such anymore because the issue data is missing. Otherwise, we found no errors in our data. Thus, while we believe that there may still be missed or invalid links in the data, the amount of data that is affected would be very small. Therefore, we can consider JLM as ground truth for the links between commits and Jira issues. We note that these findings are in line the empirical study by Bissyandé et al. (2013) on issue links in Apache projects.

We evaluate the performance of SZZ and JL with respect to the ground truth data determined with JLM. Figure 2 summarizes the results. JL finds almost all correct links with a median of 99.7% (MAD=0.4%). In the worst case, JL still identifies 96.2% of all links. However, JL finds a median of 1.7% (MAD=1.9%) additional links that are wrong. In the worst case, JL identifies 59.4% additional links. This happened for the commons-jcs project and was due to the very frequent usage of version numbers within commit messages. Further investigation revealed that the usage of version numbers was the main reason for the false positive links by JL for all projects.

SZZ finds a median of 85.4% (MAD=19.6%) of the correct links to issues. We note that the results of SZZ strongly vary, in the worst case SZZ only finds 18.8% of the correct links. This happened for the commons-collections project. When we evaluated this, we found that for commons-collections, many issues were never assigned to a developer in the Jira. This breaks the semantic check of SZZ for the equality of the assignee in the issue tracker and the author of the commit. Further investigation revealed that this semantic check did not hold for all missed links by SZZ. Moreover, SZZ identifies a median of 12.3% (MAD=15.5%) additional links that are wrong. We note that while the median is relatively low, there is a long tail of projects with many additional links. There are even two outliers which are not shown in Figure 2. The outliers are for parquet-mr (430.4% additional links) and cayenne (124.3% additional links). In both cases, the broken links are due to links to pull requests on Github, which have the pattern #<NUMBER>. Since SZZ cannot distinguish between different issue tracking systems, all these numbers are checked against the Jira of the projects and lead to additional links. Further investigation revealed that links to pull requests were the most frequent reason for additional links in general. Commonly used numbers were also problematic, but not as frequent.

<sup>19</sup> <https://bz.apache.org>

**Fig. 2** Results of the validation of the link correctness. For the additional links with SZZ, outliers were cut of which were located at 124.3% and at 430.4%



## 5.4 Issue type validation

The second part of the validation of the quality of the issue data is a partial conceptual replication (Shull et al. 2008) of the results by Herzig et al. (2013). Based on the data for five projects by Herzig et al. (2013), we expect that between 27.4% and 42.9% of BUG issues are mislabeled with 99.5% confidence and that between 0.4% and 3.5% of issues of other types than BUG should actually be bugs. The first and third author of this article manually labeled the types for all linked issues of type BUG, regardless of whether the link was established by SZZ, JL, or JLM. The inter-rater reliability between the first and the third author was substantial ( $\kappa = 0.62$ ) with an agreement on 77% of the issues. All three authors determined the correct label as committee for the remaining 23% of the issues. This way, we manually validated the issue type for all 11,295 issues that were linked by commits and labelled as BUG in the issue tracker.

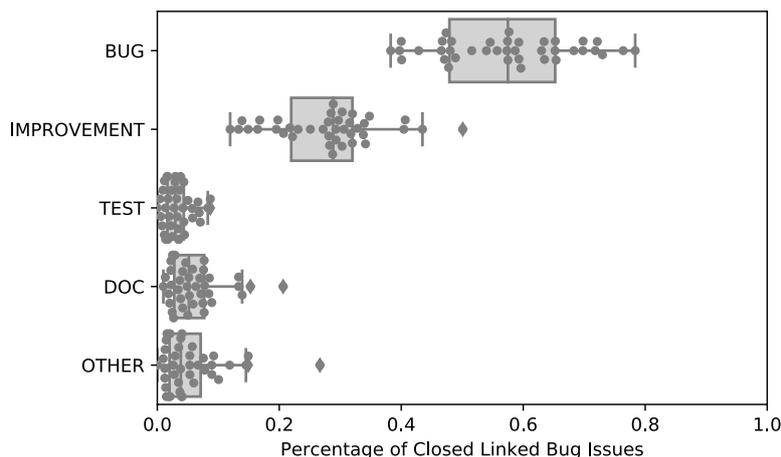
Figure 3 summarizes the results of the evaluation. Overall, we found that a median of 42.3% (MAD=13.3%) of linked BUG issues are mislabeled: 29.0% (MAD=6.4%) are actually IMPROVEMENT, 5.2% (MAD=3.6%) are DOC, 3.0% (MAD=2.0%) are TEST, and 3.8% (MAD=3.1%) are OTHER.<sup>20</sup> Thus, our results replicate the findings by Herzig et al. (2013) regarding the misclassification of BUG issues, even though we are close to the upper bound of the confidence interval. Another way to read these numbers is that for every correctly labeled BUG issue, there are a median of  $\frac{42.3\%}{57.7\%} = 0.73$  incorrectly labeled issues. Figure 3 demonstrates that all projects are affected by this kind of noise in the data, i.e., even in the best case about one fifth of the bugs are mislabeled.

We also randomly sampled 50 issues that are not of type BUG for each project, i.e., a total of 1833 issues.<sup>21</sup> These issues were manually validated by the first and second author of this article and the first three authors resolved conflicts as committee. A mean of 1.3% of these issues were actually bugs. Thus, our results also replicate these findings by Herzig et al. (2013). Overall, there are 11980 resolved linked issues that are not of type bug in our data. Thus, we expect that we miss about  $1.3\% \cdot 11980 = 155.7$  bugs in our data. Since we have found 6367 validated bugs, we are missing about 2.4% of the bugs in the data, i.e., the impact on our results is small.

<sup>20</sup> The sum of the median values for for IMPROVEMENT, DOC, TEST, and OTHER is 41.0%, i.e., less than the median of not being a BUG, which is 42.5%. This is possible because the median is not linear.

<sup>21</sup> The projects commons-bcel, commons-digester, commons-jcs, and commons-validator had fewer than 50 issues that were linked by commits, which is way we do not have  $38 \cdot 50 = 1900$  issues.

**Fig. 3** Results of the manual validation of linked bug issues



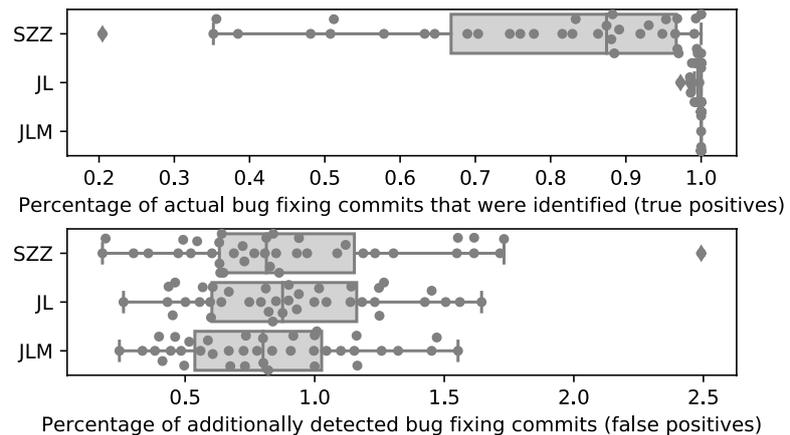
### 5.5 Bug fixing commits

Neither the broken links, nor the wrong issue types have direct impact on defect prediction. The impact on defect prediction research only manifests, if there are false positives or false negatives for the labeling of bug fixing commits based on this data. To validate how the bug fixing commits change, we compare SZZ, JL, JLM, and JLMIV with each other, using JLMIV as baseline. JLMIV constitutes our ground truth for this evaluation, because it is based on the validated links and the validated issues. JLMIV labels a median of 5.6% (MAD=3.3%) of the total commits of a project as bug fixing. Figure 4 summarizes the actual bug fixes that are detected correctly (true positives) and the wrongly detected bug fixing commits (false positives). The results for the true positives mirror the results of the detection between the commits and issues. SZZ identifies a median of 86.9% (MAD=18.2%) of all bug fixing commits, JL is almost perfect with a median of 100% (MAD=0.4%). JLM identifies all bug fixes identified by JLMIV, because JLMIV uses the same links as JLM and only reduces the bug fixing commits, because fewer issues are considered as BUG.

Regarding the false positives, our results are mostly influenced by the issue type validation. SZZ finds a median of 81.1% (MAD=40.0%) false positive bug fixing commits, JL finds a median of 86.3% (MAD=40.4%), and JLM finds a median of 78.9% (MAD=39.3%). While these numbers are very high, they are expected given that there are a median of 0.73 wrong BUG issues for every correct bug issue. There are also additional false positives for SZZ and JL due to additional issue links that are wrong. We note that SZZ has a lower median than JL. This is counter intuitive, because SZZ should have more false positives, because SZZ has more additional issue links than JL. However, this is offset by the correct links that SZZ missed. These not only cause SZZ to miss true positives, but also to miss false positives due to wrong issue labels.

### 5.6 Ground truth for inducing changes and affected releases

We were able to establish ground truth data for all our empirical data so far. Unfortunately, this was not possible for us for the bug inducing changes and the assignment of bugs to releases, including the quality of the data in the affected versions field of Jira. When we considered if we could achieve this through manual validation, we came to the conclusion that this is impossible on this scale for a group of researchers (Section 6.4). Even

**Fig. 4** Results of the validation of the link correctness

on a smaller scale, we could often not be sure if our assessment is correct, because we lack understanding of the details of the source code within the projects. We could also not re-use an established approach from the literature. Mills et al. (2018) determined similar ground truth data, but only for the file actions addressed in bug fixing commits, not their inducing counterparts. The projects that are part of our empirical study that were also investigated by Mills et al. (2018) are mahout and tika. Mills et al. (2018) sampled 34 issues for mahout and 22 issues for tika, 23 of these issues are validated bug fixes in our data. We cross-checked our data with these 23 issues to determine if all files that were found to be part of the bug fix would be detected as such by our approach. This is the case, if there is an inducing change for the file. We found that we correctly detect which file actions were actually part of the fix for 21 of the issues. For the issue TIK-1110, we fail to identify one file, that Mills et al. (2018) say is part of the fix. However, this is the deletion of a file, because there are no further references to it. Since the deletion does not cause any change to the logic of the code, this is not a problem of our data. The cases where we miss changes are for TIK-1070 and TIK-961. In both cases, there is a pure addition of source code. Since pure additions do not have inducing changes, we cannot identify an inducing commit and would, therefore, miss the related file. Overall, our approach was correct for 21 of 23 issues, i.e., 91% of the file actions.

For the bug inducing changes, only Rodríguez-Pérez et al. (2020) created manually validated data regarding bug inducing changes. However, they target different projects than our work. To provide an indication about the inducing changes in our work, we wanted to rely on developers for this task. Thus, we looked for information in our data, which we could use to accurately determine inducing changes. We then compared the results to the work by Rodríguez-Pérez et al. (2020).

Moreover, we require an approach different from the suggested framework for evaluating SZZ variants suggested by Da Costa et al. (2017). In their work, they suggest to use indicators for potential mislabels such as the disagreement with the affected versions field, the number of subsequent bug fixes, or the age of the bug, to identify potential mislabels. However, such anomalies only indicate that something may be wrong, but are not definitive indicators of mislabels, i.e., ground truth. Consequently, their results cannot be directly compared to our work.

We were able to extract knowledge about inducing changes from our data by exploiting the false positive links of our JL approach. One reason for false positives is that an issue is referenced in a commit message as the cause of a bug. Thus, we scanned all bug fixing commits identified by JLMIV for false positive issue links. We manually checked the commit messages and found twenty commits which clearly state that one issue was the cause of

**Table 6** Ground truth data for inducing changes and release assignments

Fixed Issue	Inducing Issue	Time to Fix	Deviation JLMIV+R	Affected Versions Field	Actually Affected Releases	6M	AV	IND
IVY-882	IVY-857	45 days	0 days	2.0-RC1	-	✓	(✓)	✓
CALCITE-2253	CALCITE-2206	5 days	0 days	-	-		✓	✓
CALCITE-1215	CALCITE-1212	0 days	0 days	-	-		✓	✓
CALCITE-822	CALCITE-783	17 days	0 days	-	-		✓	✓
KYLIN-3223	KYLIN-3239	1 days	0 days	2.2.0	-			✓
NUTCH-683	NUTCH-676	20 days	0 days	1.0.0	-	✓	(✓)	✓
PARQUET-214	PARQUET-139	54 days	-29 days	1.6.0	-	✓	(✓)	✓
TIKA-599	TIKA-528	147 days	0 days	0.9	0.8, 0.9	✓		✓
TIKA-2483	TIKA-2311	196 days	0 days	1.16	1.15, 1.16	✓		✓
SYSTEMML-1126	SYSTEMML-584	323 days	-7 days	-	0.10, 0.11, 0.12, 0.13			✓
SYSTEMML-2162	SYSTEMML-1919	163 days	0 days	-	1.0.0			✓
SYSTEMML-2275	SYSTEMML-2217	323 days	0 days	-	-		✓	✓
LENS-538	LENS-486	0 days	0 days	2.2	-	✓	(✓)	✓
KNOX-1134	KNOX-1119	2 days	0 days	-	-			✓
VALIDATOR-376	VALIDATOR-273	474 days	-174 days	1.4.1	(1.4.1)	✓	✓	✓
GIRAPH-918	GIRAPH-908	2 days	0 days	-	-	✓		✓
GIRAPH-832	GIRAPH-792	1 days	0 days	-	-	✓		✓
GIRAPH-88	GIRAPH-11	0 days	0 days	0.1.0	-	✓	(✓)	✓
GIRAPH-34	GIRAPH-27	4 days	-1 days	0.1.0, 1.0.0	-	✓	(✓)	✓
EAGLE-573	EAGLE-569	0 days	0 days	0.5.0	-	✓		✓

The affected version for VALIDATOR-376 is in braces because it is minor version, which we omitted otherwise

another issue and that also indicate that this is not just re-opening of the same issue again. For example, we found the following commit message: “FIX: ChainResolverTest failures (IVY-882): The problem was due to the changes introduced in IVY-857. [...]”<sup>22</sup> Thus, we can say that IVY-857 is the *inducing issue* of IVY-882. It follows that the bug was in the software, when the work on the inducing issue was finished. Thus, we looked up the commits in our data, which were also linked to the inducing issue. In case there were multiple commits for the work on the inducing issue, we manually validated which of the commits on the inducing issue was the latest change that was related to the work on the fixed issue and marked this commit as the inducing commit. Then, we manually validated that these were indeed the commits that introduced the bugs. In all but one case, this was the latest commit on the inducing issue. The exception is the work on TIKA-2483,<sup>23</sup> where the inducing change was not in the latest change,<sup>24</sup> but one of the prior changes.<sup>25</sup> We note that a similar approach was suggested in parallel work by Rosa et al. (2021), who also looked for fixing commits where the message indicates when the bug was introduced. In their study, they found 129 such commits that referenced issues within a sample of 19,603,736 commits.

Table 6 lists the data we retrieved. We first note that ten of the issues we found only existed for less than five days in the projects. This is expected, because this data is not an unbiased sample from all bug fixing commits, but rather a sample of commits where developers identified a concrete issue as the reason. In these cases, a developer immediately noticed and fixed the problem and referenced the prior work. The other ten issues lived longer, one issue even existed for more than one year. We use this data to evaluate three aspects: 1) the accuracy of the detection of the latest inducing change with JLMIV+R; 2) the quality of the data in the affected versions field of the issue tracking system; and 3) the correctness of the assignment of the bugs to releases based on a six month timeframe (6M), the affected versions field (AV), and the inducing changes determined by JLMIV+ (IND).

JLMIV+R finds the correct latest commit regarding the inducing file action for sixteen of the twenty issues. This is in line with the expectation from Rodríguez-Pérez et al. (2020) who also determined a precision of about 70%. In four cases, JLMIV+R finds a commit that is newer than the actual inducing change. This is an expected weakness of JLMIV+R and of strategies that use the blame mechanism to find the most recent change in a version control system in general. This happens if there is a change on the defective source code between the inducing change and the bug fixing change.

Regarding the affected versions field, we note that there are nine cases, in which the field was not used in the issue tracker. In seven of those cases, this is correct, as the bugs never affected a release, i.e., they were introduced and fixed between releases. In the other two cases, we validated that the bugs affected multiple releases of the software. Of the eleven cases, where the affected version field is defined, only one entry is completely correct (VALIDATOR-376). For two issues, the affected version fields contain a partially correct entry (TIKA-599 and TIK-2483). In both of these cases, only the latest release is mentioned, the older releases which are also affected are ignored. The remaining eight entries of the affected version field are wrong, i.e., they list releases which are not affected by the bugs. In six cases, the bugs were fixed prior to the release (IVY-882, NUTCH-683,

<sup>22</sup> <https://github.com/apache/ant-ivy/commit/6d6d34>

<sup>23</sup> <https://github.com/apache/tika/commit/06486c>

<sup>24</sup> <https://github.com/apache/tika/commit/6930ff>

<sup>25</sup> <https://github.com/apache/tika/commit/3aab15>

PARQUET-214, LENS-538, GIRAPH-88, GIRAPH-34), in the other two cases the bugs were only induced after the release (KYLIN-3223, EAGLE-573). In the first six cases, the developers assigned the version number of the release that is currently a work in progress, in the last two cases they assigned the version number of the latest release.

The problems with the affected version field are more severe than we expected. Overall, the bugs in this sample affect 10 releases, the affected version field only mentions three releases correctly. We expected that we would find this kind of error in the data of the affected version field, even though we expected fewer differences. Our analysis revealed that the affected versions field may also contain affected versions that are wrong, which we did not expect. The case where the version of the work in progress release is used, is relatively harmless for defect prediction: since the bug fixing commit is before the release, the commit will not be considered during the labeling of the release and the wrong value of the field will, consequently, be ignored. Similarly, our proposed improvement for the detection of inducing changes JLMIV+RAV would just use the date of the reporting of the bug and, therefore, also not be affected. The second case, where the latest release is entered, even though the bug was never in that release, is problematic. This leads to an additional assignment to a release and also breaks JLMIV+RAV as the actually inducing changes are after the date of this wrongly mentioned release and would, therefore, be flagged as suspect.

The last aspect is the assignment to releases. Assignment based on bug fixes six months after the release is correct for twelve issues, assignment based on affected versions is correct for eleven issues, and assignment based on the inducing changes for all twenty issues. The correctness of the six month time frame depends on two factors: the time to fix and the activity of the project. In case the time to fix was more than six months, the correct release was missed. In case the project was very active, e.g., with multiple releases within the last six months, the bug would be assigned to a release in which it was not yet introduced into the software. With the exception of VALIDATOR-273, the release assignment based on the affected version field is correct if no release was affected and the affected versions either contained a not yet released version or was empty. The assignment based on inducing changes is correct, even in the four cases where a wrong change is identified as inducing. In one of these cases, there is only a small deviation of less than one week between the actual time to fix and the determined inducing change. In case of PARQUET-214 and VALIDATOR-376 the inducing change is relatively far off and it is pure chance due to the project activity that the assignment is correct.

## 5.7 Bug inducing changes

For just-in-time data, the identification of bug fixing commits is only the precursor for finding the inducing changes, which are then the target of the prediction. Moreover, we described how the inducing change can be used for assigning defects to releases in Section 4.1. To this aim, we compare four approaches for the identification of bug inducing changes: 1) the standard SZZ algorithm 2) JLMIV, i.e., our improved linking with the issue validation, but standard SZZ to determine inducing changes; 3) JLMIV+R, i.e., the improvement to ignore changes to non-java files, whitespace only changes, documentation changes, and refactorings; 4) JLMIV+AV that further extends JLMIV by using the affected versions field; and 5) SZZ-RA, i.e., a state of the art variant of SZZ based on the work by Neto et al. (2018, 2019). Our variant of SZZ-RA differs from the original work only in details: we added a filter that ignores test and documentation files, and use the links between commits and issues based on the Jira issue pattern.

We do not have ground truth data for this analysis, as discussed in Section 5.6. Instead, we use JLMIV+R as a baseline and analyze the differences between SZZ, JLMIV, JLMIV+AV and SZZ-RA with respect to JLMIV+R. The reason for this is two-fold. First, the inducing changes of JLMIV+R are a subset of JLMIV that only reduces the inducing changes, e.g., due to whitespace changes. Thus, in case of deviations, the change identified by JLMIV is always a false positive. Second, JLMIV+R is based on our ground truth for bug fixing commits. Since SZZ uses the same inducing strategy as JLMIV, but is based on the inferior SZZ labels for bug fixing commits, all deviations of SZZ from JLMIV+ are also mislabels. Similarly, all deviations from SZZ-RA from JLMIV+R are mislabels, because of wrong bugfixing commits. Regarding JLMIV+AV, we cannot state whether deviations from JLMIV+ are correct or not: this depends on the affected version field. In case the affected version field contains valid data, JLMIV+AV is likely to be correct, because the identification of suspect changes is improved. In case of invalid data, JLMIV+R is likely to be correct, because the inducing changes would be wrongly flagged as suspect by JLMIV+AV.

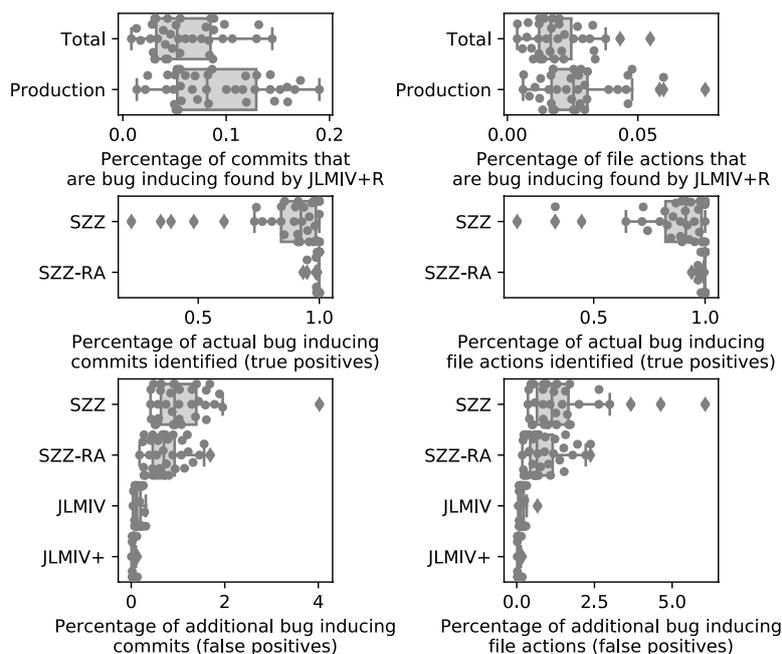
Figure 5 summarizes the results for the inducing strategies. JLMIV+R finds that a median of 5.2% (MAD=3.4%) of the commits are bug inducing. If we only consider the 78.296 commits in which at least one Java production file<sup>26</sup> was changed, the percentage of bug inducing commits has a median of 8.1% (MAD=5.0%). When we consider this on the level of changes to files, as is done by Pascarella et al. (2019), we find that a median of 2.5% (MAD=1.2%) of all changes to Java production files are inducing for a bug.

SZZ identifies a median of 92.4% (MAD=10.7%) of the correct bug inducing commits and a median of 92.7% (MAD=53.6%) false positive bug inducing commits. SZZ identifies a median of 91.3% (MAD=11.9%) of the correct bug inducing file actions and a median of 113.3% (MAD=80.6%) of false positive inducing file actions of Java production files. These values are similar to the results for the bug fixing commits, i.e., mislabels due to the inducing strategy are hidden due to the large number of mislabeled bug fixes. The evaluation of JLMIV gives a better insight into the inducing strategy, because there is no noise due to mislabeled bug fixing commits. JLMIV identifies a median of 12.1% (MAD=6.2%) false positive bug inducing commits and a median of 13.3% (MAD=6.7%) false positive inducing file actions for java production files. This reduction is in line with the expectations due to the results from Mills et al. (2018), which found that 8.7% of false positives for the bug fixing actions are due to changes to comments and whitespace only changes and 8.49% false positives are due to refactorings.

With respect to JLMIV+AV, we find a median of 4.8% (MAD=5.2%) commits and a median of 4.9% (MAD=4.2%) of file actions are detected less than with JLMIV+R. Based on our limited data regarding the correctness of the affected versions field, we would expect that roughly half of these file actions are actually false positives (Section 5.6), i.e., are incorrectly detected by JLMIV+ and constitute noise. Thus, the potential impact of the affected versions field is relatively small with an expected reduction of false positive inducing changes by about 2.4% of the total amount of inducing file actions. Regardless, we cannot recommend to use JLMIV+AV without first validating the data in the affected version field, because the potential benefit due to fewer false positives are offset by an equally large loss due to false negatives.

<sup>26</sup> Java files excluding tests and documentation.

**Fig. 5** Results for the identification of bug inducing changes

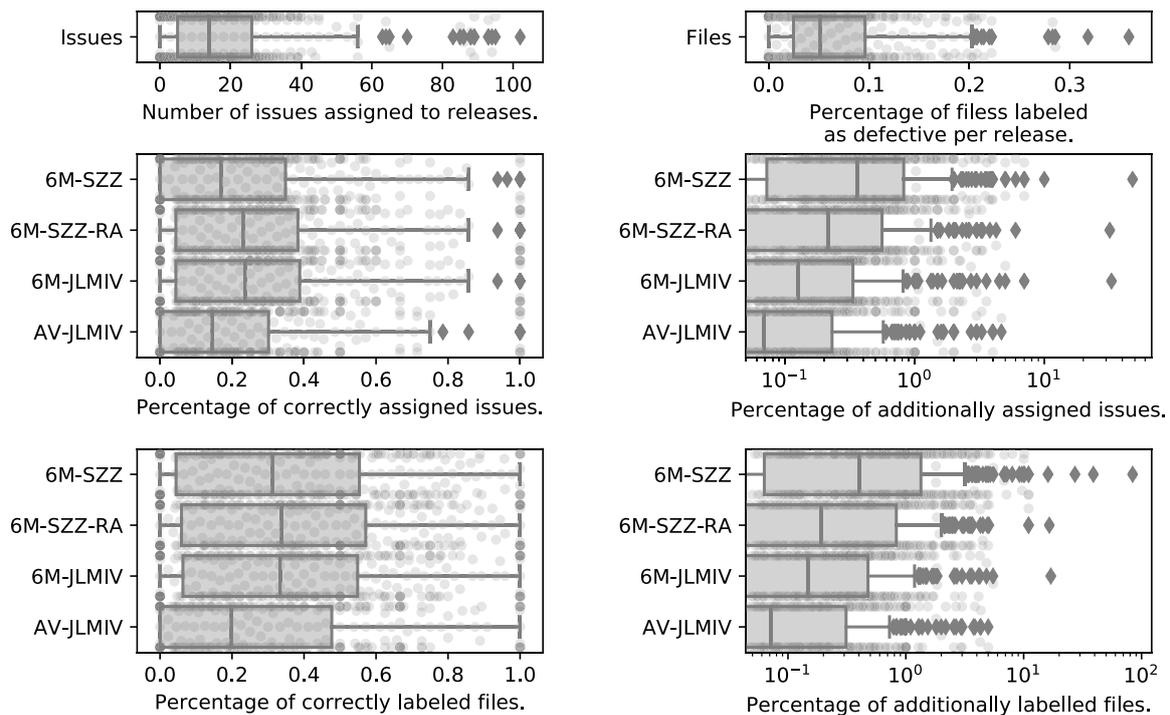


SZZ-RA similar as for the bugfixing commits, SZZ-RA finds almost all correct bug inducing commits with a median of 100% (MAD=0.0%) and a minimum of 93.3%. However, SZZ-RA finds a median of 69.8% (MAD=35.6%) false positive bug inducing commits and a median of 65.0% (MAD=48.7%) of false positive inducing file actions. Thus, the percentage of false positive inducing commits is lower than the percentage of false positive bug fixing commits (86.3%), but still relatively high. This reduction is in line with the expectation of the effect of the improvements to SZZ, i.e., ignoring changes to test, documentation, whitespaces, and refactorings, as can be seen by the difference between JLMIV and JLMIV+R. Thus, improved SZZ heuristics can resolve the relatively small problems of missing bug inducing changes due to missed issue links, but cannot resolve the problem with false positives due to wrong issue types.

## 5.8 Assignment to releases

The literature suggests either to assign all bugs that are fixed within six months after a release to the release (6M) or to use the affected versions field (AV). In this article, we propose to use the inducing changes instead (IND). We evaluate the release assignment from two perspectives: the assigned issues and the files that are labeled as defective, due to the assigned issues. Same as for the inducing changes, we do not have ground truth. Regardless, the results from Section 5.6 indicate that the assignment based on IND is the most reliable strategy, even though likely not flawless. Therefore, we evaluate the deviations of 6M and AV from IND. We use the 6M strategy with the bug fixing commits determined by SZZ (6M-SZZ), SZZ-RA<sup>27</sup> (6M-SZZ-RA), as well as those determined by JLMIV (6M-JLMIV). For the affected versions, we also use the bug fixing commits

<sup>27</sup> Technically, we only applied SZZ-RA for inducing changes. By considering which bug fixes identified by JL have an inducing change identified by SZZ-RA, we applied the improvements of SZZ-RA over SZZ to the bug fixing commits for the release assignment.



**Fig. 6** Results for the assignment of issues to releases and the labeling of files

determined by JLMIV (AV-JLMIV). For the the assignment based on including changes, we use JLMIV+R (IND-JLMIV+R).

Figure 6 summarizes the results for the release assignments. Overall, a median of 14 (MAD=14.8%) bugs affect a median of 13.0% (MAD=14.8%) files of a release. For 36 releases, we did not find any bug fixes. We marked these releases in italic in Table 4. 21 of these release are the first releases for the projects, ten releases are the last in our data. The other five releases are for stable versions of Apache Commons projects. Without these 36 releases, the median of bugs that affect a release is 16 (MAD=13.3%) and 15.5% (MAD=14.1%) files are defective. We report the results of 6M-SZZ, 6M-SZZ-RA, 6M-JLMIV and AV-JLMIV without the 36 releases that do not have any bug fix as they may skew the results.

6M-SZZ determines a median of 16.9% (MAD=25.1%) of the bugs and a median of 33.3% (MAD=38.5%) of the files correctly, and determines 33.3% (MAD=49.4%) additional bugs and 37.8% (MAD=56.0%) false positive defective files. 6M-SZZ-RA determines a median of 23.1% (MAD=26.5%) of the bugs and a median of 33.8% (MAD=38.7%) of the files correctly, and determines 20.0% (MAD=29.6%) additional bugs and 17.0% (MAD=25.2%) false positive defective files. 6M-JLMIV determines a median of 23.6% (MAD=26.7%) of the bugs and a median of 33.3% (MAD=35.3%) of the files determined correctly and a median of 11.9% (MAD=17.7%) additional bugs and 14.3% (MAD=21.2%) false positive defective files. The differences between 6M-SZZ, 6M-SZZ-RA and 6M-JLMIV are in line with the differences in the inducing changes. AV-JLMIV determines a median of 14.5% (MAD=21.6%) of the bugs and a median of 19.8% (MAD=29.4%) of the files correctly and determines 6.4% (MAD=9.6%) additional bugs and 6.6% (MAD=9.7%) false positive defective files. Thus, AV-JLMIV labels the fewest files as defective of all variants. This is in line with the results by Da Costa et al. (2017) that only a small percentage of bugs contain any data in the affected versions field. We compared these results with the ground truth data from Section 5.6. While the deviations

are not equal, they show similar trends to the sample depicted in Table 6, both regarding the 6M strategy, as well as the AV strategy.

## 5.9 Prediction models

The focus of our empirical analysis is on the quality of defect labels. However, it is unclear if and how the mislabels affect prediction models. Moreover, we have no indication if the incomplete feature sets in many data is really a problem. To assess the impact our data collection may have on the defect prediction models, we evaluate how the assessment of the best performing model from a recent benchmark on cross-project defect prediction (Herbold et al. 2017) would change. We follow the procedures for data processing outlined by Herbold et al. (2017) and use only the releases with at least 100 files and at least five defective files. We apply this criterion to both the 6M-SZZ and the IND-JLMIV+R data, i.e., there must be at least five defective files with both labels. We then use all data from projects other than the release for training, the data from the release itself for testing. This leaves us with 203 releases. Camargo Cruz and Ochimizu (2009) proposed a relatively simple transfer learning approach. The approach by Camargo Cruz and Ochimizu (2009) ranked best overall among 450 different models from the defect prediction literature that were compared on five data sets by Herbold et al. (2017) and should, therefore, be a suitable model to assess the impact of mislabels and feature sets on defect prediction models.

Camargo Cruz and Ochimizu proposed the following: first, the natural logarithm is applied to all features values plus one, then the difference between the median of the feature in the training data and the test data is subtracted, i.e., the feature  $m$  is transformed such that  $\hat{m}(s) = \log(1 + m(s)) + \text{median}(\log(1 + m(S))) - \text{median}(\log(1 + m(S^*)))$  where  $S$  is the set of files in the training data,  $S^*$  is the set of file in the test data,  $m(s)$  is the value of feature  $m$  for file  $s \in S \cup S^*$ . The transformed features are used as input into a Gaussian Naive Bayes classifier (Zhang 2004).

To evaluate the impact of features and labels on the prediction model, we train four different variants of the model by Camargo Cruz and Ochimizu (2009) based on different training data. We use two different labels: 6M-SZZ, i.e., the commonly used labeling strategy used by most defect prediction data sets and IND-JLMIV+R, i.e., the most accurate labels we determined. For each label, we use two different sets of features: 1) all features in our data (ALL) and 2) only features based on static product metric for classes<sup>28</sup> and files (SM), i.e., the similar features to the PROMISE data, one of the most popular defect prediction data sets (Hosseini et al. 2017). We denote the resulting models in the following as 6M-SZZ-ALL, 6M-SZZ-SM, IND-JLMIV+R-ALL, and IND-JLMIV+R-SM. These models are evaluated on the IND-JLMIV+R labels in the test data, i.e., with the least amount of mislabels. Through these models, we evaluate how mislabels affect the training of prediction models. Additionally, we also evaluate the models trained with 6M-SZZ labels with 6M-SZZ labels as test data and denote these models as 6M-SZZ-ALL-SZZ and 6M-SZZ-SM-SZZ. Through these models, we evaluate how mislabels affect the evaluation of prediction models.

Figure 7 summarizes the results of the prediction models. For IND-JLMIV+R-ALL the median lower boundary on  $C$  is 2141.3 (MAD=1804.9) and the median upper boundary on  $C$  is 2297.3 (MAD=1993.0). Thus, in the median, the costs of a defect must be at least

<sup>28</sup> We use summation for the aggregation to the file level.

as high as the quality assurance for 2141.3 lines of code and at most as high 2297.3 lines of code, for the model to be cost saving, i.e., the defect prediction model can only save cost under very specific assumptions in the median. For 86 of the 203 releases we use, the lower boundary is greater or equal to the upper boundary, meaning that for 42.4% of the projects the defect prediction could never save costs in comparison to a trivial approach of either doing nothing or testing everything. While these numbers are already bad, they are worse for the other models. IND-JLMIV+R-SM has a median lower boundary of 2315.2 (MAD=1817.5) and a median upper boundary of 2221.5 (MAD=1827.6), the lower boundary is greater or equal to the upper boundary for 50.7% of the projects. 6M-SZZ-ALL has a median lower boundary of 2313.4 (MAD=1996.8) and a median upper boundary of 2094.3 (MAD=1715.0), the lower boundary is greater or equal to the upper boundary for 46.3% of the projects. 6M-SZZ-SM has a median lower boundary of 2185.0 (MAD=1754.3) and a median upper boundary of 2185.0 (MAD=1754.3), the lower boundary is greater or equal to the upper boundary for 45.3% of the projects. When we train and test with the SZZ labels, the cost estimations get even worse. 6M-SZZ-ALL-SZZ has a median lower boundary of 2781.6 (MAD=2373.4) and a median upper boundary of 1755.1 (MAD=1652.8), the lower boundary is greater or equal to the upper boundary for 72.4% of the projects. 6M-SZZ-SM-SZZ has a median lower boundary of 2724.8 (MAD=2067.9) and a median upper boundary of 1467.8 (MAD=1334.5), the lower boundary is greater or equal to the upper boundary for 78.3% of the projects. Thus, for all models except IND-JLMIV+R-ALL the median lower boundary is greater or equal to the median upper boundary and the defect prediction model saves costs for fewer projects.

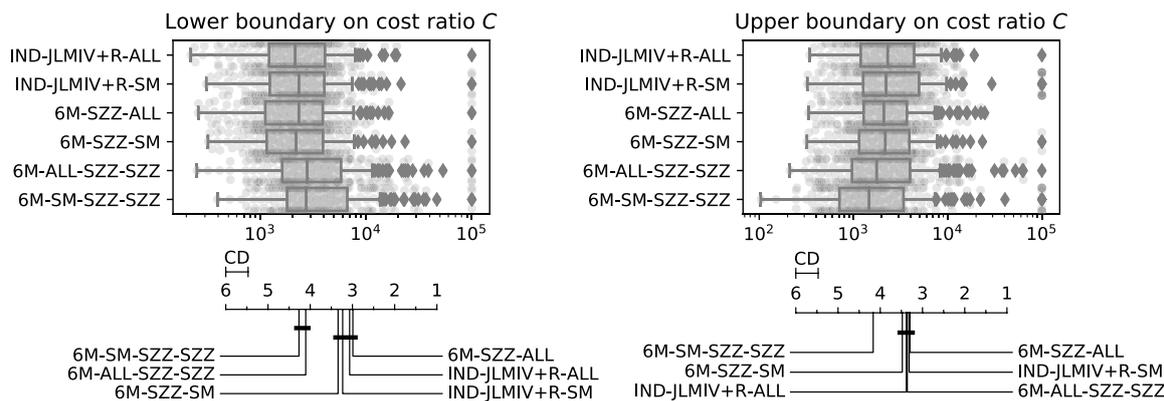
We reject the null hypothesis of the Friedman test that there are no significant differences between models for both the lower boundary ( $p\text{-value}<0.001$ ) and the upper boundary ( $p\text{-value}<0.001$ ). Figure 7 shows the critical distance diagrams of the post-hoc Nemenyi test for both boundaries. The results show that the differences in the median between the models evaluated with IND-JLMIV+R labels in the test data are not significant. Thus, contrary to Tantithamthavorn et al. (2015), we do not find that mislabels significantly impact the results, at least in terms of cost saving potential. The same is true for using more features. However, we note that while the difference may not be significant, using more features and cleaner labels still has the best median values and can be cost saving for the largest number of projects.

The models tested with 6M-SZZ labels in the test data are significantly worse with a small effect size ( $\delta \in [0.18, 0.23]$ ), with the SM features in both boundaries and with ALL features only in the lower boundary. This difference also manifests in a large drop in projects, for which the evaluation on the test data estimates that costs could be saved.

Finally, we note that the overall performance of the model is relatively poor. However, the goal of this experiment is not to show whether the model by Carmargo Cruz and Ochimizu is good, but rather to evaluate if the different labels and features impact the evaluation of the model.

## 6 Discussion

Within this section, we discuss the implications of the results of our empirical study on our research question, i.e., how issues with defect labeling and feature sets affect defect prediction data and prediction results. We consider three different aspects: the defect labels, the



**Fig. 7** Critical Distance diagrams for the comparison of significant differences the boundaries on the cost ratio  $C$

impact of features on predictions, the low performance of the defect prediction model we trained, as well as open problems.

### 6.1 Defect labels

Through our empirical study, we showed that the concerns that were raised in the recent years regarding the validity of defect labeling are valid. The two biggest factors that influence the quality of defect labels determined by SZZ are mislabeled issues in the Jira and mistakenly assigned bugs to releases. We proposed a solution for the latter problem by assigning bugs to releases based on inducing commits. The problem of mislabeled issues is more severe: to the best of our knowledge, there is no automated heuristic that can identify mislabeled issues in issue trackers. Our approach was to employ manual analysis by experts, same as Herzig et al. (2013). However, this is extremely time consuming and does not scale well. In our case, we validated 11295 issues overall for the 38 projects. While we did not measure the exact time, two authors of this article spent at least one person month each on the independent labeling, i.e., at least 176 hours. Additionally, all three authors spent at least 20 hours on the resolution of disagreements. Thus, we spent at least 412 working hours on the issue type validation, the actual number is more likely around 600 working hours. Consequently, we required between two and three minutes per issue. This estimate is similar to the four minutes that Herzig et al. (2013) reportedly required. A third factor are wrong links from commits to issues, but this can already be automatically solved by adopting the linking process to the issue tracking system, in our case Jira. However, if this is not done, this would be a third major source for noise. We note that according to our analysis of existing defect prediction data sets (see Section 2), only RNALYTICA and BUGHUNTER data avoid this problem by exploiting links from Jira (RNALYTICA) and within GitHub (BUGHUNTER). This is an indication that the state of practice may be evolving here away from the original SZZ for the identification of bug fixing changes.

We note that the mislabeled issues are mislabeled from our perspective as researchers that want to analyze defects in software repositories. Thus, issues like incompatibilities due to new Java versions, failing tests, or missing documentation, are mislabels, because these do not constitute bugs in the sense of wrong run-time behavior of the software at the time of the release. From the perspective of the developers, these may not be mislabels, because they may use a more practical definition of bug in the issue tracker: something that

is undesirable. Thus, we believe that these mislabels will remain a systematic problem for the analysis of issue tracking data, same as Herzig et al. (2013).

Regarding the impact of the problems with defect labeling, our study revealed that all problems are replicable and that the impact increases if the problems are considered together. The problems with issue linking and the issue types in the repository combined mean that **SZZ misses about one fifth of the bug fixing commits, and only about half of the commits SZZ identifies as bug fixing are actually bug fixing**. This is because SZZ only identifies about 80% of the actual bug fixing commits, but also mislabels roughly the same amount of bug fixing commits because of the mislabeled issues. If this is combined with a six month time frame for assigning defects to releases, the problem becomes even more severe. **For every file, that is correctly labeled as defective, there is roughly one file that is incorrectly labeled as defective, and two files that are incorrectly labeled as non-defective**. With implementation of the state of the art SZZ-RA variant of SZZ, these problems can be somewhat mitigated, but are still severe, because the main reasons for the deviates are the mislabeled issues and the six month time frame: **SZZ-RA misses almost no bug fixing commits but still has the problem that only about half of the commits identified as bug fixing are actually bug fixing. For every three files that are correctly labeled as defective, SZZ-RA incorrectly labels two files as defective, and misses six files that are incorrectly labeled as non-defective**. Yatish et al. (2019) proposed using the affected version field of issue trackers as a solution for the release assignment problem. While this is in principle a perfect solution, the reality of the data in the issue tracking system shows that the information contained in the affected version field is unreliable. Overall, using inducing changes currently seems to be the best heuristic.

Another important aspect to consider here is, that there is still noise in our data due to false positive defect labels. We did not validate the file actions and instead only applied a heuristic that ignored changes to whitespaces, comments, and refactorings. Thus, while we may have measured the largest part of the noise in the defect labels, there is still an uncertain region, which is also not accounted for in our data. We note that additional improvements would only lead to more deviations of SZZ from the actual data, because SZZ would identify even more false positives.

All data sets we discussed in Section 2 are affected by the problems regarding the defect labels. This is a severe threat to all publications that use this data and, therefore, basically to the complete state of the art of defect prediction. We have shown in Section 5.9 that the mislabels lead to significantly different results when used for training and evaluation. The differences are not significant if noisy data is used for training but cleaned data is used for testing. How this impacts, e.g., the comparison between defect prediction approaches is unclear. It may be that empirical results are not affected, because the signal of the defective data was still strong enough to be picked up by analysis and all approaches are affected equally. It may also be that the outcome of experiments changes, because different software artifacts are labeled as defective.

## 6.2 Features

Our initial motivation that started this research was, that we actually wanted to have a defect prediction data set with a broad selection of features, because we believed that this was the key ingredient that was missing for highly performing defect prediction models. The indications from the literature suggested that this was true (Section 3.2), especially due to churn related features. Because we considered how we should collect the defect data,

we discovered the need for our analysis of the defect labels and the focus of our research evolved: due to our findings regarding the defect labels, the analysis of the features is now only in the background of this article. Regardless, we believed that the small experiment we described in Section 5.9 would show that the large feature set is beneficial and that we could point to future research to find a suitable subset from the large amount of features we use, e.g., to minimize the effort to collect the features without a loss in prediction performance.

Our results do not support such a conclusion and instead indicate that the impact of features that go beyond static source code metrics may be relatively small. While the results were best with all features, the difference was not significant. Our replication package also contains the results for *recall* and *precision*, which show that the differences with traditional machine learning metrics are also not very strong, i.e., insignificant for *recall* and potentially small for *precision*. A closer look at the literature reveals a potential reason for this lack of a stronger effect. On the one hand, the studies that clearly find that more features, especially churn-related features, lead to better prediction results, were all conducted on relatively small data sets, i.e., on 26 releases (Ostrand et al. 2005), three releases (Moser et al. 2008), and five releases (D'Ambros et al. 2012). On the other hand, Zhang et al. (2017) used 255 and only found a small effect of using aggregations. Rahman et al. (2014) even found no statistically significant difference, when they added features based on PMD or FindBugs warnings to static metrics. Thus, the expectations that larger feature sets that go beyond static source code metrics improve predictions may be inflated.

### 6.3 Prediction performance

We note that the overall performance of the model by Camargo Cruz and Ochimizu (2009) was relatively bad. This is, from our point of view, nearly as troubling as the large amounts of mislabels, because there are two potential interpretations for this bad performance. 1) The benchmark by Herbold et al. (2017) misjudged the performance of the approach by Camargo Cruz and Ochimizu (2009) based on the prediction performance on five data sets and the approach should have been ranked lower. 2) The benchmark judged the performance correctly, and other approaches from the state of the art perform even worse. Neither interpretation is good. The first interpretation indicates a severe threat to the external validity of existing defect prediction studies, because most of them used the same data as Herbold et al. (2017). The second interpretation means that release-level defect prediction may not be able to reliably save costs. Regardless which is the case, further research in this direction is important. A potential explanation for the low performance and also possible deviations from the existing defect prediction results would be the relatively small number of defects, in comparison to all existing data set. The rigorous cleaning of false positive defect labels leads to a stronger class-level imbalance in our data, than in any of the existing data sets. Thus, newer results that were not considered in the benchmark by Herbold et al. (2017) may help.

### 6.4 Open problems

While we empirically explore many problems regarding data for defect prediction, there are still open problems left, as well as new problems we discovered due to our results.

We have only used manual validation for the bug fixing commits, but not for the file actions in those commits or for the inducing changes that were detected. While we used

smaller samples to get insights into the quality, this only helps us estimate the remaining noise in the data, as our tooling cannot identify which changes to source code actually contribute to a bug fix automatically. Thus, the first open problem is to extend this data with validated file actions for bug fixes, inducing changes, and release assignments. For the data in our empirical study we would need to manually validate 46.422 file actions for 10.515 bug fixing commits, as well as for the release assignment of 6.530 bugs. These are 34.5 times more file actions than in the study by Mills et al. (2018) with the additional effort for validation of the inducing changes. Thus, this kind of problem cannot be solved by single research groups, but must be tackled by the complete community. We already registered a study with the goal to start to address this problem with the help of the community (Herbold et al. 2020). In a first step, we want to validate which lines in bug fixes actually contribute to the bug fix and which changes are unrelated improvements.

Our IND approach for the assignment of bugs to releases also has limitations. For example, lines that are added and not modified do not have an inducing change. In general, as Rodríguez-Pérez et al. (2020) point out, there are intrinsic bugs, for which they could not locate the inducing change, even with the help of experts. Future research should consider the impact of this limitation on automated approaches, i.e., how often this is the case. Moreover, we should explore if we can improve our heuristics for indentifying inducing changes or if they may even be impossible to solve as an instance of the halting problem (Turing 1937).

Moreover, our results raise several interesting and concerning questions for further research. We can only speculate how our results regarding the defect labels affect the state of the art. We may find that the same prediction models as before are the best, simply because they are good models, independent of the data. The results may also change, because with the different data, other algorithms may perform better. We are especially looking forward to how our data affects findings that trivial baselines may outperform machine learning, e.g., by Zhou et al. (2018). Recent work already considered similar problems with other variants of SZZ, but without accounting for the biggest source of noise, i.e., the mislabeled issues (Fan et al. 2019). The results indicate that these changes will have an effect.

Through a small experiment, we have already (inadvertently) shown that some results regarding the importance of features may need to be revisited. While we found improvements, they were not significant. However, because we only performed a relatively simple study, this only means that the impact is not as obvious as we expected. Future work may uncover subsets of our features which lead to bigger improvements or demonstrate that we only found negligible differences due to our use of the approach by Camargo Cruz and Ochimizu (2009). Even if future research finds that there really is not much of an improvement if other features than static metrics are used, the question of which features are best is still interesting. For example, just-in-time defect prediction relies mostly on features that are independent of the programming language. Whether the same would be possible for release-level defect prediction without loosing predictive performance is unknown. Such results could help to broaden the scope of future research, because current research only considers a relatively small set of programming languages. Vice versa, just-in-time research avoids using static analysis tools and hence, there is a lack of research on the use of features like the complexity of code changes (Hassan 2009). Future research could explore if the language independent features are really sufficient.

Finally, there is the impact on mining defect prediction data for industrial application in tools. We rely heavily on time-consuming manual validation to increase the data quality. While we believe that this is essential for researchers that need to accurately and reliably

assess proposed approaches, this is of less importance in the industry. Here, the key question is if models trained on noisy data, e.g., collected with SZZ-RA perform as well as models trained on manually inspected data. Our results in Section 5.9 indicate that this difference may be small. Thus, such data could still be used to train domain specific models. Regardless, our results also indicate that a small sample of data should always be cleaned manually to be used as test data, because the assessment of the model performance may be unreliable. Future research should investigate if our initial results hold and automated heuristics can be safely used for the training of defect prediction models without a negative effect on the prediction performance.

## 7 Threats to validity

Due to the scope of our empirical study, there are many threats to the validity of our findings. We discuss the construct validity, internal validity, external validity, and reliability as separately, as suggested by Runeson and Höst (2008).

### 7.1 Construct validity

There are several threats to the construct validity of our experiments. We wrote a large amount of software for these experiments, which may contain defects. However, all software was tested, including the development of automated unit tests for complex and critical components like the identification of inducing changes. Furthermore, we checked the results manually. Especially the large amounts of manual analysis we conducted revealed many corner cases, which we could then handle correctly, mitigating the threats due to bugs in our software. Additionally, we may have selected unsuitable baselines for the comparison of results. To mitigate this threat, we created ground truth data as baseline where possible. In case this was not possible, we evaluated a sample from our baseline manually to establish whether our proxy for the baseline was suitable. Moreover, we cross-checked all our results with findings from related studies to evaluate the plausibility of our results. Finally, we may have used inadequate metrics for the measurement of differences. We mitigate this threat by only reporting deviations from the ground truth. To further mitigate this threat, we looked at the raw data and validated that the deviations are accurate reflections of the raw results.

### 7.2 Internal validity

The results of the analysis of the defect labeling directly follow from the properties of the defect labeling algorithms, e.g., missing links to issues are the only source for false negative bug fixing commits with SZZ in our data. The results of our sampling in Section 5.6 and the prior work by Rodríguez-Pérez et al. (2020) both indicate that finding inducing changes based on blaming prior changes may be unreliable. However, the analysis of the differences between the improved versions of this blaming, e.g., by ignoring refactorings, is still valid, because all deviations from simpler approaches are improvements, as we discuss in Section 5.7. Similarly, while the assignment of bugs to releases may be negatively affected by this, the results from Section 5.6 provide a strong indication that these results contain the least amount of noise, in comparison to the other strategies. Thus, while

concrete values with respect to the wrong assignment of bugs to releases may change with additional manual validation, the overall conclusions would likely not be affected.

The conclusion that the difference with a larger set of features is negligible may be wrong or misleading. Other factors, especially properties that we cannot easily capture with performance metrics may yield different results, e.g., the acceptance of prediction models by developers could be higher because the recall is improved. Moreover, we only consider a pure classification scenario and no ranking of files by their likelihood of defect or a regression scenario for the prediction of the number of defects in a file. The additional features may lead to bigger differences in performance under these considerations.

### 7.3 External validity

The main threat regarding the external validity of the results is due to our focus on Java projects that are developed under the umbrella of the Apache Software Foundation. Thus, it is unclear if and how our findings generalize to projects using other programming languages or software development outside of the Apache Software Foundation. We note that our analysis regarding the defect labeling problems is mostly independent of the programming language, with the exception of the identification of whitespace and comment-only changes. For example, whitespace changes may actually be changes to the logic of a program written with Python. Moreover, the Apache Software Foundation attracts a large amount of developers both from the industry as well as from the open source community. This increases the likelihood that aspects like the labeling of issues as bug or the use of the affected versions field are similar in other contexts.

Furthermore, it is unclear if our conclusions regarding performance differences are only relevant for release-level defect prediction or if they generalize to just-in-time defect prediction. However, a follow-up study indicates that there at least large performance differences between keyword-based approaches and validated data for just-in-time defect prediction (Trautsch et al. 2020).

### 7.4 Reliability

To avoid bias in the manual validation of data, we involved multiple people. The issues were validated by two authors independently, conflicts were solved by three authors. While the initial validation of the issue links was conducted only one author, two authors performed the manual analysis of a sample of 1000 issues for mislabels. Additionally, these results were cross-checked by a third person that is not an author of this manuscript. Thus, we minimized the impact of individuals on the results to mitigate this the threat to the reliability of the research.

## 8 Conclusion

Within this article we performed a critical assessment of the state of practice of the collection of defect prediction data. We summarized existing data sets and found that the SZZ algorithm is the standard approach for defect labeling and that most data sets only offer a limited set of features. This is in contrast to the state of the art that found problems with defect labeling using SZZ, as well as diverse features that should be valuable for defect prediction. To assess the impact of this difference, we performed an empirical study with the

focus on the problems of defect labeling and found that SZZ identifies one incorrect bug fixing commit for each correct bug fixing commit, while still missing about one fifth of the bug fixing commits. The main reason for the mislabeled commits are mislabeled issues, a problem initially found by Herzig et al. (2013). For release-level defect prediction data, this problem is even worse, because most data sets use a six month timeframe to assign defects to releases. The combination of these problems mean that for every correctly labeled defective file, there is one incorrectly labeled defective file and two missed defective files. Thus, there is a large amount of noise in the defect prediction data that is currently used and we can only speculate how this affects the state of the art. Future work on defect labeling should carefully manually validate data and not rely on time windows for the identification of defects to avoid the generation of noisy data.

Regarding the features, we found that the difference of the prediction performance measured with more features is not statistically significant, even though more features yielded the best overall results. This is in contrast to prior findings that highlighted the importance of, e.g., churn features. However, since our analysis of feature importances and the impact of larger feature sets was only rudimentary, additional research is required to establish what a suitable set of features for defect prediction looks like.

Another contribution of this article is a new defect prediction data set, both for just-in-time, as well as release-level defect prediction. Our data set is larger than any currently used data set, i.e., contains more releases and projects, as well as more features. We hope that the data we produced as part of our work will help the research community to resolve the problems we found. On the one hand, we are looking forward to studies of defect prediction models using our data, both replications of existing work with the de-noised data, as well as the assessment of new approaches and techniques. On the other hand, our data may also be used to improve automated defect labeling, e.g., by trying to automatically correct bug issue labels in issue trackers. Moreover, we hope that our data will be used as the foundation for the manual validation of file actions, to provide a ground truth assessment of the assignment of defects to files and releases.

## Appendix A: Summary of Acronyms

Throughout the article, we use a many acronyms to refer to different variants of data creation. We summarize these acronyms here.

SZZ	The original SZZ algorithm by Śliwerski et al. (2005) that identifies by fixing commits through numeric links to issues and uses blames to identify inducing changes.
SZZ-RA	An improved SZZ algorithm by Neto et al. (2018, 2019) that ignores whitespaces and refactorings when identifying inducing changes. We further extended SZZ-RA to ignore test and documentation changes.
JL	Identification of bug fixing commits through the Jira identifier of issues.
JLM	Extends JL with manual validation of the links from commits to issues.
JLMIV	Extends JLM with the manual validation of the type of issues to determine if they are really bugs.
JLMIV+	Extends JLMIV with the filtering of documentation, test, and whitespace changes.
JLMIV+R	Extends JLMIV+ with the filtering of refactorings. Can be seen as SZZ-RA with Jira links and manual validation of issue links and issue types.
6M-SZZ	SZZ with a six months time window to assign bugs to releases.
6M-SZZ-RA	SZZ-RA with a six-month time window to assign bugs to releases.

6M-JLMIV	JLMIV with a six month time window to assign bugs to releases.
AV-JLMIV	JLMIV with affected versions of linked issues to assign bugs to releases.
IND-JLMIV+R	JLMIV+R with inducing changes to assign bugs to releases.
IND-JLMIV+R-ALL	Release-level defect prediction data with IND-JLMIV+R for bug labeling and all our features.
IND-JLMIV+R-SM	Release-level defect prediction data with IND-JLMIV+R for bug labeling and only static product metrics as features.
6M-SZZ-ALL	Release-level defect prediction data with SZZ for bug labeling and all our features. IND-JLMIV+R labels are used for testing.
6M-SZZ-SM	Release-level defect prediction data with SZZ for bug labeling and only static product metrics as features. IND-JLMIV+R labels are used for testing.
6M-ALL-SZZ-SZZ	Release-level defect prediction data with SZZ for bug labeling and all our features. SZZ labels are used for testing.
6M-SM-SZZ-SZZ	Release-level defect prediction data with SZZ for bug labeling and only static product metrics as features. SZZ labels are used for testing.

## Appendix B: Details of the Data Collection

We used the tools `vcsSHARK`<sup>29</sup>, `mecoSHARK`, `coastSHARK`, `changeSHARK`, and `refSHARK` to collect data from the version control system. The `vcsSHARK` collects meta data about commits, e.g., the messages, the committer, as well as the actual changes, i.e., the file actions and hunks. The `mecoSHARK` is a wrapper around `SourceMeter`<sup>30</sup>, a tool that calculates static software metrics, clone metrics, as well as the warnings by the static analysis tool `PMD`. The `coastSHARK` collects AST node counts and the import statements of Java classes, i.e., low level data about the use of language constructs and the dependencies of classes. The `changeSHARK` is a wrapper around the `ChangeDistiller` (Fluri et al. 2007) that determines the types of changes performed in commits using the classification from Zhao et al. (2017). The `refSHARK` is a wrapper around the `RefDiff` tool for refactoring detection (Silva and Valente 2017). These tools are executed for every commit in the repositories to collect data about the source code evolution. Additionally, we use the tool `memeSHARK` to remove redundancies from the collected data, e.g., because the data did not change between commits, for a more efficient storage. We use the tool `issueSHARK` to collect data from the issue tracking system of the projects, e.g., the identifiers, comments, status, and other meta data about the issues.

The tools `linkSHARK`, `labelSHARK`, and `inducingSHARK` implement the approaches for issue linking, labeling of bug fixing commits, and the inference of inducing commits<sup>31</sup> that we evaluate in this empirical study. The manual validation was supported by the `visualSHARK`, a web application that presented the data that requires manual validation to the experts and stores the results of the validation in the `MongoDB`. The information that the web interface provides is similar to the `LINKSTER` tool (Bird et al. 2010).

Data for just-in-time defect prediction is available through the `MongoDB`. This data includes all metrics mentioned above, and also the data required for code ownership

<sup>29</sup> All \*SHARK tools and `mynbou` are available at <https://github.com/smarts shark>

<sup>30</sup> <https://www.sourceme ter.com/>

<sup>31</sup> We use `git blame` for the identification of inducing commits.

analysis (Bird et al. 2011). We use the tool mynbou to create CSV files with release-level data with files as level of abstraction. For each file, the data contains the software metrics and PMD warnings collected by the mecoSHARK and the coastSHARK, the number of the different kinds of changes and refactorings collected by the changeSHARK and the refSHARK from the last six months, and churn metrics proposed by Moser et al. (2008), Hassan (2009), and D'Ambros et al. (2012). Additionally, mynbou provides all thirteen aggregations that were proposed by Zhang et al. (2017) for the software metrics the mecoSHARK collected that are not on the file level, i.e., class, method, interface, enum, attribute and annotation metrics. The data set contains a total of 4198 features. We decided not to add features with code smells (Palomba et al. 2019) to the data set, because these can be calculated indirectly from the available source code metrics and definition of smells like god class may change over time. Additionally, features based on mutation testing are not available, because retrospective execution of tests is, unfortunately, often not possible (Tufano et al. 2017). We also have not added features regarding test smells (Spadini et al. 2018) and review activity (Thongtanunam et al. 2016), because current results only show correlation with defects, but have not yet shown that these features may actually improve defect prediction models.

**Acknowledgements** This work is partially funded by DFG Grant 402774445. We also want to thank the GWDG for the support in using their high performance computing infrastructure, that enabled the collection of the large amounts of software metric data.

**Funding** Open Access funding enabled and organized by Projekt DEAL.

**Open Access** This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

## References

- Altinger H, Siegl S, Dajsuren Y, Wotawa F (2015) A novel industry grade dataset for fault prediction based on model-driven developed automotive embedded software. In: Proceedings of the 12th Working Conference on Mining Software Repositories, IEEE Press, Piscataway, NJ, USA, MSR '15, pp 494–497. <http://dl.acm.org/citation.cfm?id=2820518.2820596>
- Antoniol G, Ayari K, Di Penta M, Khomh F, Guéhéneuc YG (2008) Is it a bug or an enhancement? a text-based approach to classify change requests. In: Proceedings of the 2008 Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds, Association for Computing Machinery, New York, NY, USA, CASCON '08 <https://doi.org/10.1145/1463788.1463819>.
- Bird C, Bachmann A, Aune E, Duffy J, Bernstein A, Filkov V, Devanbu P (2009a) Fair and balanced?: Bias in bug-fix datasets. In: Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, ACM, New York, NY, USA, ESEC/FSE '09, pp 121–130 <https://doi.org/10.1145/1595696.1595716>.
- Bird C, Rigby PC, Barr ET, Hamilton DJ, German DM, Devanbu P (2009b) The promises and perils of mining git. In: 2009 6th IEEE International Working Conference on Mining Software Repositories, pp 1–10 <https://doi.org/10.1109/MSR.2009.5069475>
- Bird C, Bachmann A, Rahman F, Bernstein A (2010) Linkster: Enabling efficient manual inspection and annotation of mined data. In: Proceedings of the Eighteenth ACM SIGSOFT International Symposium

- on Foundations of Software Engineering, ACM, New York, NY, USA, FSE '10, pp 369–370 <https://doi.org/10.1145/1882291.1882352>.
- Bird C, Nagappan N, Murphy B, Gall H, Devanbu P (2011) Don't touch my code! examining the effects of ownership on software quality. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, Association for Computing Machinery, New York, NY, USA, ESEC/FSE '11, p 4–14 <https://doi.org/10.1145/2025113.2025119>.
- Bissyandé TF, Thung F, Wang S, Lo D, Jiang L, Réveillère L (2013) Empirical evaluation of bug linking. In: 2013 17th European Conference on Software Maintenance and Reengineering, pp 89–98 <https://doi.org/10.1109/CSMR.2013.19>
- Bowes D, Hall T, Harman M, Jia Y, Sarro F, Wu F (2016) Mutation-aware fault prediction. In: Proceedings of the 25th International Symposium on Software Testing and Analysis, Association for Computing Machinery, New York, NY, USA, ISSTA 2016, p 330–341 <https://doi.org/10.1145/2931037.2931039>
- Camargo Cruz AE, Ochimizu K (2009) Towards logistic regression models for predicting fault-prone code across software projects. In: Proc. 3rd Int. Symp. on Empirical Softw. Eng. and Measurement (ESEM), IEEE Computer Society
- Chidamber SR, Kemerer CF (1994) A metrics suite for object oriented design. *IEEE Trans Softw Eng* 20(6):476–493. <https://doi.org/10.1109/32.295895>
- Cliff N (1993) Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological Bulletin* 114(3):494
- Cohen J (1960) A coefficient of agreement for nominal scales. *Educational and Psychological Measurement* 20(1):37–46. <https://doi.org/10.1177/001316446002000104>
- Da Costa DA, McIntosh S, Shang W, Kulesza U, Coelho R, Hassan AE (2017) A framework for evaluating the results of the szz approach for identifying bug-introducing changes. *IEEE Transactions on Software Engineering* 43(7):641–657. <https://doi.org/10.1109/TSE.2016.2616306>
- D'Ambros M, Lanza M, Robbes R (2012) Evaluating defect prediction approaches: A benchmark and an extensive comparison. *Empirical Softw Engg* 17(4–5):531–577. <https://doi.org/10.1007/s10664-011-9173-9>
- Demšar J (2006) Statistical comparisons of classifiers over multiple data sets. *J Mach Learn Res* 7:1–30
- Di Penta M, Bavota G, Zampetti F (2020) On the relationship between refactoring actions and bugs: A differentiated replication. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Association for Computing Machinery, New York, NY, USA, ESEC/FSE 2020, p 556–567 <https://doi.org/10.1145/3368089.3409695>
- Fan Y, Xia X, Alencar Da Costa D, Lo D, Hassan AE, Li S (2019) The impact of changes mislabeled by szz on just-in-time defect prediction. *IEEE Transactions on Software Engineering* pp 1 <https://doi.org/10.1109/TSE.2019.2929761>
- Ferenc R, Tóth Z, Ladányi G, Siket I, Gyimóthy T (2018) A public unified bug dataset for java. In: Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering, ACM, New York, NY, USA, PROMISE'18, pp 12–21 <https://doi.org/10.1145/3273934.3273936>
- Ferenc R, Gyimesi P, Gyimesi G, Tóth Z, Gyimóthy T (2020a) An automatically created novel bug dataset and its validation in bug prediction. *J Syst Softw* 169:110691. <https://doi.org/10.1016/j.jss.2020.110691>
- Ferenc R, Tóth Z, Ladányi G, Siket I, Gyimóthy T (2020) A public unified bug dataset for java and its assessment regarding metrics and bug prediction. *Software Quality Journal* 28(4):1447–1506. <https://doi.org/10.1007/s11219-020-09515-0>
- Fischer M, Pinzger M, Gall H (2003) Populating a release history database from version control and bug tracking systems. In: International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings., pp 23–32 <https://doi.org/10.1109/ICSM.2003.1235403>
- Fluri B, Würsch M, Pinzger M, Gall H (2007) Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering* 33(11):725–743
- Friedman M (1940) A comparison of alternative tests of significance for the problem of m rankings. *The Annals of Mathematical Statistics* 11(1):86–92
- Hall T, Beecham S, Bowes D, Gray D, Counsell S (2012) A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering* 38(6):1276–1304. <https://doi.org/10.1109/TSE.2011.103>
- Hassan AE (2009) Predicting faults using the complexity of code changes. In: Proceedings of the 31st International Conference on Software Engineering, IEEE Computer Society, Washington, DC, USA, ICSE '09, pp 78–88 <https://doi.org/10.1109/ICSE.2009.5070510>

- Herbold S (2019) On the costs and profit of software defect prediction. *IEEE Transactions on Software Engineering* pp 1 <https://doi.org/10.1109/TSE.2019.2957794>
- Herbold S, Trautsch A, Grabowski J (2017) A comparative study to benchmark cross-project defect prediction approaches. *IEEE Transactions on Software Engineering* PP(99):1 <https://doi.org/10.1109/TSE.2017.2724538>
- Herbold S, Trautsch A, Ledel B (2020) Large-scale manual validation of bugfixing changes. <https://doi.org/10.17605/OSF.IO/ACNWK>
- Herzig K, Just S, Rau A, Zeller A (2013) Predicting defects using change genealogies. In: 2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE), pp 118–127 <https://doi.org/10.1109/ISSRE.2013.6698911>
- Herzig K, Just S, Zeller A (2013) It's not a bug, it's a feature: How misclassification impacts bug prediction. In: Proceedings of the 2013 International Conference on Software Engineering, IEEE Press, Piscataway, NJ, USA, ICSE '13, pp 392–401. <http://dl.acm.org/citation.cfm?id=2486788.2486840>
- Hosseini S, Turhan B, Gunarathna D (2017) A systematic literature review and meta-analysis on cross project defect prediction. *IEEE Transactions on Software Engineering* PP(99):1 <https://doi.org/10.1109/TSE.2017.2770124>
- Jureczko M, Madeyski L (2010) Towards identifying software project clusters with regard to defect prediction. In: Proceedings of the 6th International Conference on Predictive Models in Software Engineering, ACM, New York, NY, USA, PROMISE '10, pp 9:1–9:10 <https://doi.org/10.1145/1868328.1868342>.
- Kamei Y, Shihab E, Adams B, Hassan AE, Mockus A, Sinha A, Ubayashi N (2013) A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering* 39(6):757–773. <https://doi.org/10.1109/TSE.2012.70>
- Kim S, Zimmermann T, Pan K, Jr Whitehead EJ (2006) Automatic identification of bug-introducing changes. In: 21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06), pp 81–90 <https://doi.org/10.1109/ASE.2006.23>
- Kovalenko V, Palomba F, Bacchelli A (2018) Mining file histories: Should we consider branches? In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ACM, New York, NY, USA, ASE 2018, pp 202–213 <https://doi.org/10.1145/3238147.3238169>
- Landis JR, Koch GG (1977) The measurement of observer agreement for categorical data. *Biometrics* 33(1):159–174
- Madeyski L, Jureczko M (2015) Which process metrics can significantly improve defect prediction models? an empirical study. *Software Quality Journal* 23(3):393–422. <https://doi.org/10.1007/s11219-014-9241-7>
- McIntosh S, Kamei Y (2018) Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction. *IEEE Transactions on Software Engineering* 44(05):412–428. <https://doi.org/10.1109/TSE.2017.2693980>
- Menzies T, Krishna R, Pryor D (2015) The promise repository of empirical software engineering data
- Menzies T, Krishna R, Pryor D (2017) The seacraft repository of empirical software engineering data
- Mills C, Pantiuchina J, Parra E, Bavota G, Haiduc S (2018) Are bug reports enough for text retrieval-based bug localization? In: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp 381–392 <https://doi.org/10.1109/ICSME.2018.00046>
- Mockus A (2009) Amassing and indexing a large sample of version control systems: Towards the census of public source code history. In: 2009 6th IEEE International Working Conference on Mining Software Repositories, pp 11–20 <https://doi.org/10.1109/MSR.2009.5069476>
- Moser R, Pedrycz W, Succi G (2008) A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In: Proceedings of the 30th International Conference on Software Engineering, ACM, New York, NY, USA, ICSE '08, pp 181–190 <https://doi.org/10.1145/1368088.1368114>
- NASA (2004) Nasa iv & v facility metrics data program. <http://web.archive.org/web/20110421024209/http://mdp.ivv.nasa.gov/repository.html>. Accessed 17 December 2021
- Nemenyi P (1963) Distribution-free multiple comparison. PhD thesis, Princeton University
- Neto EC, da Costa DA, Kulesza U (2018) The impact of refactoring changes on the szz algorithm: An empirical study. In: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp 380–390 <https://doi.org/10.1109/SANER.2018.8330225>
- Neto EC, d Costa DA, Kulesza U (2019) Revisiting and improving szz implementations. In: 2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), pp 1–12 <https://doi.org/10.1109/ESEM.2019.8870178>
- Ostrand T, Weyuker E, Bell R (2005) Predicting the location and number of faults in large software systems. *IEEE Trans Softw Eng* 31(4):340–355. <https://doi.org/10.1109/TSE.2005.49>

- Palomba F, Zanoni M, Fontana FA, De Lucia A, Oliveto R (2019) Toward a smell-aware bug prediction model. *IEEE Transactions on Software Engineering* 45(2):194–218. <https://doi.org/10.1109/TSE.2017.2770122>
- Pascarella L, Palomba F, Bacchelli A (2019) Fine-grained just-in-time defect prediction. *J Syst Softw* 150:22–36. <https://doi.org/10.1016/j.jss.2018.12.001>
- Plosch R, Gruber H, Hentschel A, Pomberger G, Schiffer S (2008) On the relation between external software quality and static code analysis. In: 2008 32nd Annual IEEE Software Engineering Workshop, pp 169–174 <https://doi.org/10.1109/SEW.2008.17>
- Rahman F, Posnett D, Hindle A, Barr E, Devanbu P (2011) Bugcache for inspections: Hit or miss? In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ACM <https://doi.org/10.1145/2025113.2025157>
- Rahman F, Khatri S, Barr ET, Devanbu P (2014) Comparing static bug finders and statistical prediction. In: Proceedings of the 36th International Conference on Software Engineering, ACM, New York, NY, USA, ICSE 2014, pp 424–434 <https://doi.org/10.1145/2568225.2568269>
- Rodríguez-Pérez G, Zaidman A, Serebrenik A, Robles G, González-Barahona JM (2018) What if a bug has a different origin? making sense of bugs without an explicit bug introducing change. In: Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, Association for Computing Machinery, New York, NY, USA, ESEM '18 <https://doi.org/10.1145/3239235.3267436>
- Rodríguez-Pérez G, Robles G, Serebrenik A, Zaidman A, Germán DM, Gonzalez-Barahona JM (2020) How bugs are born: a model to identify how bugs are introduced in software components. *Empirical Software Engineering*. <https://doi.org/10.1007/s10664-019-09781-y>
- Rodríguez-Pérez G, Robles G, González-Barahona JM (2018) Reproducibility and credibility in empirical software engineering: A case study based on a systematic literature review of the use of the szz algorithm. *Information and Software Technology* 99:164–176. <https://doi.org/10.1016/j.infsof.2018.03.009>
- Romano J, Kromrey J, Coraggio J, Skowronek J (2006) Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen'sd for evaluating group differences on the NSSE and other surveys? In: Annual Meeting of the Florida Association of Institutional Research, pp 1–3
- Rosa G, Pascarella L, Scalabrino S, Tufano R, Bavota G, Lanza M, Oliveto R (2021) Evaluating szz implementations through a developer-informed oracle. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), IEEE Computer Society, Los Alamitos, CA, USA, pp 436–447 <https://doi.org/10.1109/ICSE43902.2021.00049>
- Rousseeuw PJ, Croux C (1993) Alternatives to the median absolute deviation. *Journal of the American Statistical Association* 88(424):1273–1283. <https://doi.org/10.1080/01621459.1993.10476408>
- Runeson P, Höst M (2008) Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering* 14(2):131. <https://doi.org/10.1007/s10664-008-9102-8>
- Shippey T, Hall T, Counsell S, Bowes D (2016) So you need more method level datasets for your software defect prediction?: Voilà! In: Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ACM, New York, NY, USA, ESEM '16, pp 12:1–12:6 <https://doi.org/10.1145/2961111.2962620>
- Shull F, Carver J, Vegas S, Juristo N (2008) The role of replications in empirical software engineering. *Empirical Software Engineering* 13(2):211–218
- Silva D, Valente MT (2017) Refdiff: Detecting refactorings in version histories. In: 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), pp 269–279 <https://doi.org/10.1109/MSR.2017.14>
- Śliwerski J, Zimmermann T, Zeller A (2005) When do changes induce fixes? In: Proceedings of the 2005 International Workshop on Mining Software Repositories, ACM, New York, NY, USA, MSR '05, pp 1–5 <https://doi.org/10.1145/1082983.1083147>
- Spadini D, Palomba F, Zaidman A, Bruntink M, Bacchelli A (2018) On the relation of test smells to software code quality. In: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp 1–12 <https://doi.org/10.1109/ICSME.2018.00010>
- Tantithamthavorn C, McIntosh S, Hassan AE, Ihara A, Matsumoto K (2015) The impact of mislabelling on the performance and interpretation of defect prediction models. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, vol 1, pp 812–823 <https://doi.org/10.1109/ICSE.2015.93>
- Tantithamthavorn C, Hassan AE, Matsumoto K (2018) The impact of class rebalancing techniques on the performance and interpretation of defect prediction models. *IEEE Transactions on Software Engineering* pp 1 <https://doi.org/10.1109/TSE.2018.2876537>

- Thongtanunam P, McIntosh S, Hassan AE, Iida H (2016) Revisiting code ownership and its relationship with software quality in the scope of modern code review. In: 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), pp 1039–1050 <https://doi.org/10.1145/2884781.2884852>
- Tóth Z, Gyimesi P, Ferenc R (2016) A public bug database of github projects and its application in bug prediction. In: Gervasi O, Murgante B, Misra S, Rocha AMA, Torre CM, Tanar D, Apduhan BO, Stankova E, Wang S (eds) Computational Science and Its Applications - ICCSA 2016. Springer International Publishing, Cham, pp 625–638
- Trautsch A, Herbold S, Grabowski J (2020) Static source code metrics and static analysis warnings for fine-grained just-in-time defect prediction. In: 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp 127–138 <https://doi.org/10.1109/ICSME46990.2020.00022>
- Trautsch A, Trautsch F, Herbold S, Ledel B, Grabowski J (2020) The smartshark ecosystem for software repository mining. In: Proceedings of the 42st International Conference on Software Engineering - Demonstrations, ACM
- Trautsch F, Herbold S, Makedonski P, Grabowski J (2017) Addressing problems with replicability and validity of repository mining studies through a smart data platform. Empirical Software Engineering. <https://doi.org/10.1007/s10664-017-9537-x>
- Tufano M, Palomba F, Bavota G, Di Penta M, Oliveto R, De Lucia A, Poshyvanyk D (2017) There and back again: Can you compile that snapshot? Journal of Software: Evolution and Process 29(4):e1838. <https://doi.org/10.1002/smr.1838>
- Turhan B, Menzies T, Bener AB, Di Stefano J (2009) On the relative value of cross-company and within-company data for defect prediction. Empirical Software Engineering 14(5):540–578. <https://doi.org/10.1007/s10664-008-9103-7>
- Turing AM (1937) On computable numbers, with an application to the entscheidungsproblem. Proceedings of the London Mathematical Society s2-42(1):230–265 <https://doi.org/10.1112/plms/s2-42.1.230>
- Williams C, Spacco J (2008) Szz revisited: Verifying when changes induce fixes. In: Proceedings of the 2008 Workshop on Defects in Large Software Systems, Association for Computing Machinery, New York, NY, USA, DEFECTS '08, p 32–36 <https://doi.org/10.1145/1390817.1390826>
- Wu R, Zhang H, Kim S, Cheung SC (2011) Relink: Recovering links between bugs and changes. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ACM, New York, NY, USA, ESEC/FSE '11, pp 15–25 <https://doi.org/10.1145/2025113.2025120>
- Yao J, Shepperd M (2020) Assessing software defection prediction performance: Why using the matthews correlation coefficient matters. In: Proceedings of the Evaluation and Assessment in Software Engineering, Association for Computing Machinery, New York, NY, USA, EASE '20, p 120–129 <https://doi.org/10.1145/3383219.3383232>
- Yatish S, Jiarpakdee J, Thongtanunam P, Tantithamthavorn C (2019) Mining software defects: Should we consider affected releases? In: Proceedings of the 41st International Conference on Software Engineering, IEEE Press, Piscataway, NJ, USA, ICSE '19, pp 654–665 <https://doi.org/10.1109/ICSE.2019.00075>
- Zhang F, Mockus A, Keivanloo I, Zou Y (2014) Towards building a universal defect prediction model. In: Proceedings of the 11th Working Conference on Mining Software Repositories, ACM, New York, NY, USA, MSR 2014, pp 182–191 <https://doi.org/10.1145/2597073.2597078>
- Zhang F, Hassan AE, McIntosh S, Zou Y (2017) The use of summation to aggregate software metrics hinders the performance of defect prediction models. IEEE Transactions on Software Engineering 43(5):476–491. <https://doi.org/10.1109/TSE.2016.2599161>
- Zhang H (2004) The optimality of naive bayes. In: Proceedings of the Seventeenth International Florida Artificial Intelligence Research Society Conference (FLAIRS 2004), AAAI Press
- Zhao Y, Leung H, Yang Y, Zhou Y, Xu B (2017) Towards an understanding of change types in bug fixing code. Information and Software Technology 86:37–53 <https://doi.org/10.1016/j.infsof.2017.02.003>, <http://www.sciencedirect.com/science/article/pii/S0950584917301313>
- Zhou Y, Yang Y, Lu H, Chen L, Li Y, Zhao Y, Qian J, Xu B (2018) How far we have progressed in the journey? an examination of cross-project defect prediction. ACM Trans Softw Eng Methodol 27(1):1:1:51 <https://doi.org/10.1145/3183339>
- Zimmermann T, Premraj R, Zeller A (2007) Predicting defects for eclipse. In: Proceedings of the Third International Workshop on Predictor Models in Software Engineering, IEEE Computer Society, Washington, DC, USA, PROMISE '07, pp 9– <https://doi.org/10.1109/PROMISE.2007.10>

**Steffen Herbold** is a Full Professor for Methods and Applications of Machine Learning at the Institute of Software and Systems Engineering (ISSE) of the TU Clausthal, where he leads the AI Engineering research

group. His research considers the quality assurance of AI systems, the application of AI to aid software engineering, as well as the support of domain experts with the development of AI based solutions based on available data.

**Alexander Trautsch** is a PhD candidate at Software Engineering for Distributed Systems Group the Institute of Computer Science of the University of Goettingen. His research interests include mining software repositories, software evolution, and empirical software engineering. His current work involves software quality evolution with a focus on the impact of static analysis tools.

**Fabian Trautsch** works as a Full Stack Developer at the Sartorius AG. Previously, he worked as a PostDoc in the Software Engineering for Distributed Systems group at the Institute of Computer Science of the University of Goettingen. He received the BSc degree in applied computer science from the University of Goettingen in 2013, the subsequent MSc degree in 2015 and his doctorate in 2019. His research interests include mining software repositories, software evolution, software testing, and empirical software engineering.

**Benjamin Ledel** is a Ph.D. student in the AI Engineering research group of the TU Clausthal. He started his Ph.D. 2020 under supervision of Prof. Dr. Steffen Herbold. He is part of the SmartSHARK project since 2018. His main research interests are bug localization and explainable AI.

## Authors and Affiliations

Steffen Herbold<sup>1</sup>  · Alexander Trautsch<sup>2</sup> · Fabian Trautsch<sup>2</sup> · Benjamin Ledel<sup>1</sup>

Alexander Trautsch  
alexander.trautsch@cs.uni-goettingen.de

Fabian Trautsch  
fabian.trautsch@cs.uni-goettingen.de

Benjamin Ledel  
benjamin.ledel@tu-clausthal.de

<sup>1</sup> Institute of Software and Systems Engineering, TU Clausthal, Clausthal-Zellerfeld, Germany

<sup>2</sup> Institute of Computer Science, University of Goettingen, Goettingen, Germany

## **C Improving predictive models with ASAT information**

This section contains a copy of the following publications.

A. Trautsch, S. Herbold, J. Grabowski: Static source code metrics and static analysis warnings for fine-grained just-in-time defect prediction. IEEE International Conference on Software Maintenance and Evolution (ICSME), 2020

© 2020 IEEE. Reprinted, with permission, from [A. Trautsch, S. Herbold. J. Grabowski, Static source code metrics and static analysis warnings for fine-grained just-in-time defect prediction, IEEE International Conference on Software Maintenance and Evolution (ICSME), 2020]

<https://doi.org/10.1109/ICSME46990.2020.00022>

# Static source code metrics and static analysis warnings for fine-grained just-in-time defect prediction



Alexander Trautsch  
*Institute of Computer Science*  
*University of Goettingen, Germany*  
 alexander.trautsch@cs.uni-goettingen.de

Steffen Herbold  
*Institute AIFB*  
*Karlsruhe Institute of Technology, Germany*  
 steffen.herbold@kit.edu

Jens Grabowski  
*Institute of Computer Science*  
*University of Goettingen, Germany*  
 grabowski@cs.uni-goettingen.de

**Abstract**—Software quality evolution and predictive models to support decisions about resource distribution in software quality assurance tasks are an important part of software engineering research. Recently, a fine-grained just-in-time defect prediction approach was proposed which has the ability to find bug-inducing files within changes instead of only complete changes. In this work, we utilize this approach and improve it in multiple places: data collection, labeling and features. We include manually validated issue types, an improved SZZ algorithm which discards comments, whitespaces and refactorings. Additionally, we include static source code metrics as well as static analysis warnings and warning density derived metrics as features. To assess whether we can save cost we incorporate a specialized defect prediction cost model. To evaluate our proposed improvements of the fine-grained just-in-time defect prediction approach we conduct a case study that encompasses 38 Java projects, 492,241 file changes in 73,598 commits and spans 15 years. We find that static source code metrics and static analysis warnings are correlated with bugs and that they can improve the quality and cost saving potential of just-in-time defect prediction models.

**Index Terms**—Software quality, Software metrics

## I. INTRODUCTION

Quality assurance budgets are limited. A risk analysis for changes introduced to software would provide hints for quality assurance personal on how to make the most of their limited resources. Just-in-time defect prediction models are predictive models that assign a risk to changes, or files within a change, of being defect-inducing. Because just-in-time models are able to provide feedback directly after a change happened, they can reduce the cost of bug removal.

Just-in-time defect prediction is an active research topic which tries to enable the aforementioned theoretical risk probabilities on a per-change basis. A lot of research is being conducted in this area, e.g., improving the granularity of the predictions [1], adding features, e.g., code review [2], change context [3], or applying deep learning models [4].

Just-in-time defect prediction models are trained on bug-inducing changes, which are found by tracing back bug-fixing changes, e.g., with the SZZ algorithm [5]. Some researchers utilize keyword only approaches that scan the commit messages for certain keywords, e.g., fixes, fixed or patch, to find bug-fixing commits, e.g., [1, 3, 6]. We refer to this approach as

ad-hoc SZZ. Others apply full SZZ which requires a link from the commit to the Issue Tracking System (ITS) for identifying bug-fixing commits, e.g., [2, 7, 8]. We refer to this approach as ITS SZZ. In addition to this information, features that describe the change are used as independent variables, e.g., size of the change or diffusion, i.e., how many different subsystems are changed [1, 7, 9].

In contrast to just-in-time defect prediction, release-level defect prediction utilizes features describing files, classes or methods. These features encompass size and complexity metrics as well as object oriented metrics extracted from the files. D'Ambros et al. [10] incorporated change level features into release level defect prediction by including a time-frame before the release for change metric calculation. In addition to static, size and complexity metrics, static analysis warnings were also investigated in the context of quality insurance. Rahman et al. [11] investigated static analysis warnings in the context of release-level defect prediction. Zheng et al. [12] found that the number of static analysis warnings may help to identify defective modules.

Static analysis warnings are generated by tools which inspect different source code representations, e.g., Abstract Syntax Trees (ASTs) or call graphs and find patterns that are known to be problematic. If a problematic pattern is found a warning for the line in the code is generated for the developer. The combination of pattern and generated warning is defined in a rule, these tools allow the developers to define which rules should be utilized by the tool. The rules depend on the tool but most are concerned with readability, common coding mistakes as well as size and complexity thresholds. Results of questionnaires show that developers assign importance to static analysis software, especially at code review time [13], see also Panichella et al. [14].

Recently, Pascarella et al. [1] introduced a fine-grained just-in-time defect prediction approach where instead of complete changes the files contained in these changes are used to train predictive models. Static analysis warnings were not included, however the authors adopted change features for their fine-grained approach together with an ad-hoc SZZ implementation. While Querel et al. [15] included static analysis warnings

for just-in-time defect prediction models, this information has not yet been included in a fine-grained approach. Fan et al. [16] investigated the impact of mislabels by SZZ on just-in-time defect prediction models and found that, depending on the SZZ variant, there can be a significant impact on the models performance.

Independent of the implemented SZZ variant, if a valid link to the ITS is required, there may be additional data validity problems regarding the chosen type of the issue. Prior research by multiple groups found that not every issue classified as a bug in the ITS is actually a bug [17]–[19].

In this work we combine the fine-grained approach by Pascarella et al. [1] with static source code metrics and static analysis warnings. In addition, we include an improved SZZ algorithm which works similar to the approach proposed by Neto et al. [20]. Instead of keyword matches this approach requires valid links to the ITS for each bug-fixing commit. Similar to the approach used by Pascarella et al., its implementation ignores whitespace and comment changes. In addition, it also ignores refactorings. The links between commits and the ITS as well as the types of the linked issues are manually validated. We explore the impact that this SZZ approach has on the resulting models performance in comparison to a keyword based ad-hoc SZZ approach.

We are interested in the impact of additional features on the performance of fine-grained just-in-time defect prediction models. Similar to previous just-in-time defect prediction approaches we include a model to estimate effort. In contrast to effort based on lines of code, the cost model we incorporate is a specialized defect prediction model by Herbold [21] which calculates whether we can save cost by implementing our predictive model.

To summarize the contributions of this work:

- An evaluation of the impact of static source code metrics and static analysis warnings on fine-grained just-in-time defect prediction models.
- Three novel features based on static analysis warning density designed to capture quality evolution regarding static analysis warnings.
- Combination of a specialized defect prediction cost model [21] with a fine-grained just-in-time approach.
- Evaluation of two common labeling strategies in a fine-grained approach, ad-hoc SZZ [1, 3, 6] and ITS SZZ [7, 9, 22].

The rest of this article is structured as follows. In Section II, we introduce prior publications on the topic and relate our current article to it. In Section III, we motivate and define the research questions that we want to answer. Afterwards, we define our case study in Section IV. Section V presents the results of the case study which we discuss in Section VI. After that, we describe threats to validity in Section VII. Finally, we present a short conclusion in Section VIII.

## II. RELATED WORK

Just-in-time defect prediction has been an active area of research. In this section, we discuss the relevant related work

and draw comparisons with our own.

Kamei et al. [7] performed a large-scale empirical study of just-in-time defect prediction. They build predictive models for bug-inducing changes, including effort awareness and also investigate the difference between bug-inducing changes and the rest of the changes. To this end the authors introduced change based metrics which incorporate size, diffusion, purpose, and the history of a change as well as developer experience. The authors use ITS SZZ to find bug-inducing commits without falling back on a keyword based approach. This has a detrimental effect on the performance of their models due to heavy class imbalance as the authors note in the discussion. The predictive models are evaluated with 10-fold-cross-validation.

Tan et al. [8] apply online defect prediction where the window for the training data expands stepwise from the beginning of the project. The authors utilize the commit message, the characteristic vector [23], and churn metrics to build models for 6 open source and one proprietary project. Ad-hoc SZZ is utilized to find bug-fixing commits. The authors point out that cross-validation is not a realistic scenario for just-in-time defect prediction due to it including information from the future. They point out that due to this limitation their model performance is impacted negatively. To mitigate class-imbalance the authors apply and discuss four sampling variants.

Yang et al. [9] further investigated the model complexity utilizing the same data as Kamei et al. [7]. They found that sometimes simple unsupervised models are better than the model introduced by Kamei et al. [7]. The authors also found no big difference between cross-validation and a time-sensitive approach when evaluating the models.

McIntosh et al. [2] investigate whether the properties of bug-inducing changes change over time. The authors utilize change metrics proposed by Kamei et al. [7] and include code review metrics for the predictive models. They analyze the evolution of two open source projects.

Pascarella et al. [1] combine the features used previously by Kamei et al. [7] and Rahman et al. [24] with a fine-grained approach. Instead of predicting bug-inducing changes at the commit level they predict bug-inducing files within one commit. An ad-hoc SZZ implementation is used. To predict one instance the authors use the previous three months of data as training data.

Querel et al. [15] present an addition to commit guru [25] which includes static analysis warnings for building just-in-time defect prediction models. They show that they are able to improve the predictive models with the additional information. Their result complements the results of our case study.

Almost every prior work regarding just-in-time defect prediction relies on some variant of the SZZ algorithm. Although there are differences in its implementation. Some publications use a modified version of the SZZ algorithm which does not utilize an ITS. The original SZZ algorithm does not work as well without an ITS. Without an ITS there is no way to define a suspect boundary date [5]. This results in more bug-fixes

and bug-inducing changes with keyword only ad-hoc SZZ approaches.

In our case study, we investigate this difference by including the ad-hoc SZZ keyword based approach as well as the ITS SZZ approach which requires bug-fixing changes to have a link to a valid ITS issue. The dataset we build upon contains manually validated issue types for every issue that is linked to a bug-fixing commit to account for wrongly classified issues [19].

None of the prior work investigated if static source code metrics or static analysis warnings can improve just-in-time defect prediction models in a fine-grained context. Moreover, none of the prior studies compared how the difference between ad-hoc SZZ and ITS SZZ impact the results of defect prediction. Finally, while some publications considered aspects related to the costs [1, 9, 22] this is the first publication that applies a complete cost model [21] to evaluate the cost saving potential of just-in-time defect prediction.

### III. RESEARCH QUESTIONS AND ANALYSIS PROCEDURE

We hypothesize that additional features in the form of static source code metrics and static analysis warnings may improve just-in-time defect prediction models. The commonly used features are change metrics, e.g., [7, 24]. They capture information about the change and itself and the process, e.g., developer experience, size and diffusion of the change. Static source code metrics, e.g., Logical Lines of Code (LLOC), McCabe complexity [26] or object oriented metrics [27] would add additional information about the structure of the files that are contained in the change. Static analysis warnings can add information about violated best practices or naming conventions within the changed files. These features are commonly used for release-level defect prediction and perform well [28]. Moreover, a combination of different sets of features seem promising [29]. Both of the additional sets of features offer different additional information that might positively affect just-in-time defect prediction models. Our investigation into the impact of different features and SZZ approaches on just-in-time defect prediction is driven by the following research questions.

#### **RQ1: Which feature types are correlated with bug-inducing changes?**

This question aims to quantify the impact of the features we chose on identifying bug-inducing changes. We are utilizing a linear model which regularizes collinearity between features so that we can focus on the direct impact of each feature on the dependent variable. To broaden our view we also utilize a non-linear model which can also handle collinear features in addition to the linear model.

#### **RQ2: Which feature types improve just-in-time defect prediction?**

For this question, we combine recent state-of-the art data extraction and features for just-in-time defect prediction with features that are commonly used for release-level defect prediction. We hope to shed some light on how

much release-level feature sets, including static analysis warnings, can improve just-in-time defect prediction.

#### **RQ3: Are static features improving cost effectiveness in just-in-time defect prediction?**

Cost awareness is important to estimate the usefulness of the created models. To estimate the cost effectiveness we utilize a cost model explicitly created for defect prediction.

To answer *RQ1* we build two models, a linear logistic regression model and a non-linear random forest [30] model. The data for the linear model is scaled by a z-transformation [31] to prevent features with different scale magnitudes to dominate the objective function. The linear model is regularized via elastic net to remove collinear features. The data is not scaled for the random forest as it is robust to scale differences. The random forest chooses relevant features via gini impurity which also mitigates collinear features.

The models that are build for *RQ1* contain perfect knowledge, e.g., all information independent of the time it became available is included. As both models get all data, we do not perform sampling to mitigate class imbalance here. Both models are used to rank the features by their importance in the predictive model. The random forest provides this information directly via a feature importance score. For the regularized logistic regression we determine the feature importance by the absolute value of the coefficients, i. e., their impact on the prediction.

To answer *RQ2* we utilize both models as they were used previously in *RQ1*. As the first performance metric for the evaluation of our models we use the harmonic mean of precision and recall, F-measure.

$$\begin{aligned} \text{precision} &= \frac{TP}{TP + FP} \\ \text{recall} &= \frac{TP}{TP + FN} \\ \text{F-measure} &= \frac{2 \cdot \text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} \end{aligned}$$

TP are the bug-inducing instances correctly predicted by the model, FP non bug-inducing instances incorrectly predicted as bug-inducing. TN are non bug-inducing instances correctly predicted as such and FN are bug-inducing instances incorrectly predicted as non bug-inducing.

Additionally, we include AUC as a model performance measure that is not as impacted by highly imbalanced data. AUC is defined as the area under the Receiver Operating Characteristic (ROC) curve which is a plot of the false positive rate, against the true positive rate. AUC values range from 0 to 1 with 0.5 being equivalent to random guessing and 1 being the perfect value with no false positives and every positive correctly identified. AUC and F-measure are common choices in evaluating model performance, e.g., [1, 4, 7, 8]

To mitigate the class imbalance in our data, we perform SMOTE [32] sampling. SMOTE performs an oversampling of the minority class by creating additional instances which

are similar but not identical to the existing instances of the respective class.

Both classifiers are trained and evaluated on all projects. The models are evaluated for both labeling strategies (ad-hoc SZZ and ITS SZZ) as well as a train/test split as it is done in the replication kit by Pascarella et al. [1]. Additionally, to allow a comparison we replicate the commit label used in the replication kit. The commit label marks every commit in which a file was found inducing as bug-inducing, subsequently every file changed in an inducing commit as inducing.

Additionally, we include a time-sensitive interval approach where we use 3 months as training data and 1 month as testing data. The choice of 3 months is common in related literature, e.g., [1, 7, 8]. The first and last 3 months of each study subject are dropped from the analysis. After that, a sliding window approach is used to train and evaluate a model over the remaining time frame for each study subject.

Restricting the time frame in which training and test data is collected has certain drawbacks. Most prominently we may simply not have enough data to train a model. Therefore we relax the timeframe for the sliding window under certain conditions. 1) Sample size: we select a minimum sample size of the mean number of commits for one month over the project history. For training data this number is multiplied with 3 because the train window size is 3 months. 2) Insufficient positive instances: to perform SMOTE on the training data a minimum number of 5 instances of the minority class is needed. If the training data does not fulfill these conditions, we extend the timeframe of the training data further until we met the conditions.

To compare their performance with different sets of features the results are first combined into boxplots. After that, we perform prerequisite tests for selecting a statistical test to compare the difference between the feature sets. We use autorank [33] to conduct the statistical tests. Autorank implements Demsar’s guidelines [34] for the comparison of classifiers. It tests the data for normality and homoscedacity and then automatically selects suitable tests for the data: repeated measures ANOVA as omnibus tests with a post-hoc Tukey HSD test [35] in case the data is normal and homoscedastic and Friedman test [36] as omnibus test with a post-hoc Nemenyi test [37] otherwise. In case of normally distributed data we calculate effects sizes with Cohen’s  $d$  [38]. If the data is not normal we use Cliff’s  $\delta$  [39] for effect sizes.

We chose a significance level of  $\alpha = 0.05$ . After Bonferroni [40] correction for 16 statistical tests (4 model performance metrics, 2 labeling strategies for both train/test split and interval approach) we reject the  $H_0$  hypothesis that there is no difference in model performance at  $p < 0.003$ . We also include critical distance diagrams and plots for the confidence intervals for a combination of both classifiers for both labels, all performance metrics and all feature sets.

For  $RQ3$  we calculate whether costs can be saved by utilizing a predictive model for directing quality assurance with a cost model introduced by Herbold [21]. The cost model estimates boundaries on the ratio between costs of quality

assurance and costs of bugs ( $C$ ). Whether defect prediction can save cost for a project depends on this ratio. To this end, the cost model estimates lower and upper boundaries for  $C$  that give a range for which cost can be saved by a predictive model. As not every bug is fixed in one file, the cost model also accounts for  $m$ -to- $n$  relationships between bugs and files. Therefore, it does not work with the confusion matrix but instead a bug-issue matrix that is generated in the mining process which maps every bug to the changes in files that induced the bug. The cost model uses LLOC as a proxy for quality assurance effort. The boundaries are calculated as follows.

$$\frac{\sum_{s \in S: h(s)=1} size(s)}{|D_{PRED}|} < C < \frac{\sum_{s \in S: h(s)=0} size(s)}{|D_{MISS}|}$$

$S$  is the set of files which are predicted as bug-inducing,  $h$  is the prediction model,  $D$  is the set of bugs,  $D_{PRED} = \{d \in D : \forall s \in d \mid h(s) = 1\}$  is the set of predicted bugs, and  $D_{MISS} = \{d \in D : \exists s \in d \mid h(s) = 0\}$  is the set of missed bugs.

We count for how many projects our models can save costs and include the upper and lower cost boundaries as further model performance metrics in our ranking.

#### IV. CASE STUDY

In this case study, we investigate the changes introduced into a codebase over a multi-year time period. We use 38 Java open source projects of the Apache Software Foundation from Herbold et al. [19]. All data is available in our replication kit<sup>1</sup>.

Table I shows the summary statistics of the projects. The number of bug-inducing commits and files is small when we only consider ITS SZZ labels (denoted %its). If we consider ad-hoc SZZ fixes (denoted %adh) the number of bug-inducing commits and files increases significantly. The number of commits only shows the commits where Java source code files were changed. All other commits are ignored.

The data collection by Herbold et al. [19] was performed by SmartSHARK [41]. To utilize the data for a just-in-time defect prediction case study, we implemented an extraction on top of the SmartSHARK database snapshot provided in [19].

We base our extraction on the replication kit by Pascarella et al. [1]. In addition to change features, the extraction provides static source code metrics as well as static analysis warnings as additional features from [19]. Moreover, it provides bug-fixing commits with valid links to the ITS and manually validated bug issues from [19]. This data is integrated into our approach as ITS SZZ labels. Ad-hoc SZZ labels are extracted analogous to [1]. As we want to maximize the data we use all branches, i.e., the complete commit graph.

Although the extraction is based on Pascarella et al. [1] we extend it in three places. First, to improve the linking between bug-fixing and bug-inducing files, we directly utilize the underlying GitPython<sup>2</sup> instead of the wrapper provided by Pydriller [42]. This allows us to directly access the name of

<sup>1</sup><https://doi.org/10.5281/zenodo.3974204>

<sup>2</sup><https://pypi.org/project/GitPython/>

TABLE I  
NUMBER OF COMMITS, FILES AND DEFECTIVE RATES OF OUR STUDY  
SUBJECTS FOR AD-HOC SZZ AND ITS SZZ

Project	#com	%its	%adh	#files	%its	%adh
ant-ivy	1917	3.29%	25.98%	11581	4.83%	29.91%
archiva	3873	2.89%	11.72%	23899	3.46%	12.25%
calcite	2056	1.07%	12.89%	24653	4.53%	29.81%
cayenne	4157	1.95%	7.60%	42203	3.18%	9.54%
c-bcel	957	1.57%	7.21%	10842	1.02%	10.15%
c-beanutils	741	1.35%	13.09%	4760	0.82%	10.23%
c-codec	1093	0.73%	12.63%	3299	1.76%	14.55%
c-collections	2229	0.58%	8.97%	18362	0.43%	7.26%
c-compress	1765	2.38%	5.38%	5026	4.12%	7.54%
c-configuration	2010	1.24%	8.71%	7011	2.31%	12.22%
c-dbcp	1004	1.99%	21.61%	3459	2.66%	22.95%
c-digester	1375	0.73%	8.44%	5684	0.48%	6.30%
c-io	1411	1.42%	9.99%	4912	1.71%	9.85%
c-jcs	942	2.02%	14.01%	10905	2.60%	20.02%
c-jexl	884	2.15%	15.05%	3962	5.98%	20.92%
c-lang	3966	1.64%	10.26%	11962	1.74%	10.08%
c-math	5098	0.82%	8.14%	32421	1.55%	9.51%
c-net	1435	4.46%	5.64%	6645	2.48%	5.64%
c-scxml	620	1.13%	29.35%	2898	3.11%	39.41%
c-validator	724	2.07%	15.47%	2356	2.12%	13.03%
c-vfs	1378	1.45%	17.78%	9360	1.63%	14.97%
deltaspike	1519	1.97%	3.75%	7464	3.56%	8.87%
eagle	609	0.82%	10.67%	8989	3.39%	32.86%
giraph	861	1.86%	8.83%	9760	3.65%	15.28%
gora	568	1.41%	4.58%	3250	2.62%	7.45%
jspwiki	5086	2.22%	22.87%	20233	1.57%	15.27%
knox	841	2.02%	5.23%	7667	5.16%	9.69%
kylin	4362	3.07%	7.47%	31027	3.64%	11.09%
lens	1491	2.15%	12.41%	11207	5.84%	24.19%
mahout	2393	1.55%	10.03%	26713	2.74%	18.11%
manifoldcf	2609	7.51%	19.93%	17096	4.01%	11.61%
nutch	1536	6.45%	15.17%	7805	6.43%	21.42%
opennlp	1288	2.72%	12.66%	11490	2.04%	10.53%
parquet-mr	1184	1.10%	40.79%	8016	2.88%	47.32%
santuario-java	1432	1.19%	20.11%	12190	1.14%	16.46%
systemml	3645	1.48%	19.56%	36761	2.33%	24.23%
tika	2640	3.64%	10.34%	8797	6.45%	18.17%
wss4j	1899	1.42%	8.64%	17576	1.92%	10.55%

the file at the time when the bug-inducing change happened instead of the current file name. This is important as we label bug-inducing changes and the file may have been renamed later.

Second, we implemented a traversal algorithm on top of a Directed Acyclic Graph (DAG) constructed from a complete traversal by Pydriller. By traversing the constructed graph instead of a date ordered list of commits we gain improvements with regard to changes on different branches. We can keep track of files during subsequent renaming or additions and deletions happening on different branches. Furthermore, we can accumulate state information, e.g., number of changes to a file, even if it was renamed on a different branch.

Third, due to the implemented traversal algorithm we can not just ignore merge commits. As Pydriller currently does not support returning modifications on merge commits we use the underlying GitPython library to directly access the modifications.

We now describe details of the data collection for our predictive models, namely labels and features. We start by introducing the basic SZZ [5] algorithm, the improvements available from [19] and then both of our labeling strategies. After that we introduce the additional features our models use.

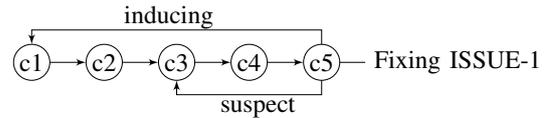


Fig. 1. SZZ algorithm

TABLE II  
LABELING STRATEGIES USED IN THIS CASE STUDY.

Label	Description
ITS SZZ	Only links to ITS, manually validated issue types and links, discard whitespace, comments and refactorings.
Ad-hoc SZZ	Keywords only (fix, bug, repair, issue, error), discard whitespace and comments.

### A. Label

Supervised learning models require labeled data, which in our case would be whether a change introduced a bug or not. The purpose of SZZ is to link bug-fixing commits with their respective bug-reports in the ITS and to link each bug-fixing change to a list of probable bug-inducing changes. Figure 1 shows the basic SZZ algorithm. Changes are denoted as  $c1-5$ , where  $c5$  is a bug-fixing change. The time in which ISSUE-1 is created is defined as the suspect boundary, changes that happen before the suspect boundary are bug-inducing changes. Changes after the suspect boundary are suspects and further divided. A suspect change is a partial fix if the suspect change is a fix for another bug. A suspect change is a weak suspect if it is a bug-inducing change (non suspect) for another bug. A suspect change is a hard suspect if it is neither a partial fix or a weak suspect. Hard suspects are discarded while partial fixes and changes inducing another bug are both counted towards bug-inducing changes.

In this work we use two labeling strategies. One uses the ITS and discards hard suspects as described above. The ITS SZZ approach discards whitespace, comments and refactorings in changes and also uses manually validated data as it uses the data from [19]. The second uses an ad-hoc SZZ keyword only approach. It filters whitespace changes and comment only changes but does not filter refactoring changes as it is based on Pydriller [42] and not part of the SmartSHARK infrastructure. This approach is similar to the data collection used by Pascarella et al. [1].

Table II provides an overview of the labeling strategies. Table I shows the defective rates for our study subjects of the commits and files. A commit is counted as defective if at least one file contained in the commit is defect-inducing. A file is defective if at least one line in the change for that file is defect-inducing.

### B. Features

The features our supervised learning models use to predict potential bug-inducing changes are based on prior publications. We include all features used by Pascarella et al. [1]. They consist of features introduced by Kamei et al. [7] and Rahman et al. [24] adopted for fine-grained just-in-time defect prediction. The features introduced by Kamei et al. are used

TABLE III  
FEATURES USED IN THE FEATURE SETS.

Name	Features
jit	COMM, ADEV, ADD, DEL, OWN, MINOR, SCTR, NADEV, NCOMM, NSCTR, OEXP, EXP, ND, ENTROPY, LA, LD, IT, AGE, NUC, CEXP, SEXP, REXP, FIX, BUG
static	PDA, LOC, CLOC, PUA, McCC, LLOC, LDC, NOS, MISM, CCL, TNOS, TLLOC, NLE, CI, HPL, MI, HPV, CD, NOI, NUMPAR, MISEI, CC, LLDC, NII, CCO, CLC, TCD, NL, TLOC, CLLC, TCLOC, MIMS, HDIF, DLOC, NLM, DIT, NPA, TNLPM, TNLA, NLA, AD, TNLPA, NM, TNG, NLP, NM, NOC, NOD, NOP, NLS, NG, TNLG, CBOI, RFC, NLG, TNLS, TNA, NLPA, NOA, WMC, NPM, TNPM, TNS, NA, LCOMS, NS, CBO, TNLM, TNPA
pmd	ABSALIL, ADLIBDC, AMUO, ATG, AUHCIP, AUOV, BIL, BI, BNC, CRS, CSR, CCEWTA, CIS, DCTR, DUFTFLI, DCL, ECB, EFB, EIS, EmSB, ESNIL, ESI, ESS, ESB, ETB, EWS, EO, FLSBWL, JI, MNC, OBEAH, RFFB, UIS, UCT, UNCIE, UOOL, UOM, FLMUB, IESMUB, ISMUB, WLMUB, CTCNSE, PCL, AIO, AAA, APMP, AUNC, DP, DNCGCE, DIS, ODPL, SOE, UC, ACWAM, AbCWAM, ATNFS, ACI, AICCC, APFIC, APMIFCNE, ARP, ASAML, BC, CWOPCSBF, CIR, CCOM, DLNLISS, EMIACSBA, EN, FDSBASOC, FFCBS, IO, IF, ITGC, LI, MBIS, MSMINIC, NCLISS, NSI, NTSS, OTAC, PLFIC, PLFIC, PST, REARTN, SDFNL, SBE, SBR, SC, SF, SSSH, TFBFASS, UEC, UEM, ULBR, USDF, UCIE, ULWCC, UNAION, UV, ACE, EF, FDNCSE, FOCSE, FO, FSBP, DJL, DI, IFSP, TMSI, UFQN, DNCSE, LHNC, LISNC, MDBASBNC, RINC, RSINC, SEJBSBF, JUASIM, JUS, JUSS, JUTCTMA, JUTSIA, SBA, TCWTC, UBA, UAEIOAT, UANIOAT, UASIOAT, UATIOAE, GDL, GLS, PL, UCEL, APST, GLSIU, LINSF, MTOL, SP, MSVUID, ADS, AFNMMN, AFNMTN, BGMN, CNC, GN, MeNC, MWSNAEC, NP, PC, SCN, SMN, SCFN, SEMN, SHMN, VNC, AES, AAL, RFI, UWOC, UALIOV, UAAL, USBFSA, AISD, MRJA, ACGE, ACNPE, ACT, ALEI, ARE, ATNIOSE, ATNPE, ATRET, DNEJLE, DNTEIF, EAFC, ADL, ASBF, CASR, CLA, ISB, SBIWC, SdI, STS, UCC, UETCS, CIMMIC, LoC, SIDTE, UnI, ULV, UPF, UPM, System/WD, File/System/WD, Author/Delta/WD

TABLE IV  
FEATURE SETS USED IN OUR CASE STUDY.

Name	Feature set description
combined	All features combined
jit	Change features commonly used in just-in-time defect prediction adopted for a fine-grained scenario by Pascarella et al. [1].
static	Static source code metrics by OpenStaticAnalyzer. A full list is available online <sup>5</sup>
pmd	Static analysis warnings by PMD also collected via OpenStaticAnalyzer. A full list is available online <sup>5</sup>

TABLE V  
WARNING DENSITY BASED FEATURES INTRODUCED IN OUR CASE STUDY.

Name	Description
System/WD	The warning density of the project.
File/System/WD	The cumulative difference between warning density of the file and the project as a whole.
Author/Delta/WD	The cumulative sum of the changes in warning density by the author.

frequently in just-in-time defect prediction, e.g., [9, 43, 44]. They contain features such as the number of lines added, experience of developers and ages on a per file basis.

Additionally, we include features consisting of static analysis warnings by PMD<sup>3</sup> and static source code metrics by OpenStaticAnalyzer<sup>4</sup>. The static source code metrics include object oriented metrics as well as size and complexity metrics, a full list is available online<sup>5</sup>.

The static analysis warnings by PMD contain a broad range of rules. From formatting rules, e.g., class names must be in CamelCase over rules regarding empty catch blocks up to very specific rules regarding BigDecimal usage. The static analysis warnings and source code metrics are collected for each change and its parent, then a delta is calculated from the current change to its parent change. This allows the included feature to quantify the impact of the change as well as its current and previous value. Table III shows all features included in our case study and their respective feature set. Table IV shows the all feature sets and a short description. In addition to the sum of static metrics and static analysis warnings we introduce new change based metrics utilizing warning density.

$$\text{Warning density} = \frac{\text{Number of static analysis warnings}}{\text{Product size}}$$

Warning density, analogous to defect density [45], describes the ratio of the sum of static analysis warnings to the size of the product, in our cases the LLOC of a file or a whole project. Table V describes the additional features we introduce based on warning density. With these additional features we hope to capture quality evolution regarding static analysis warnings. If

<sup>3</sup><https://pmd.github.io/>

<sup>4</sup><https://github.com/sed-inf-u-szeged/OpenStaticAnalyzer>

<sup>5</sup><https://www.sourcemeeter.com/resources/java/>

a modified file is consistently below the warning density of the whole project, i.e., contains less static analysis warnings per LLoC, it may be helpful in estimating its quality. Analogous, if the author of a commit consistently lowers the warning density it may also be a good indicator whether a commit by that author may induce defects or not.

## V. RESULTS

To answer *RQ1*, we applied a linear logistic regression and a non-linear random forest classification model to a complete set of our data.

Table VI shows the top ten features for both of our classifiers. We note that new static and pmd features are in the top ten for all combinations except for the logistic regression with ad-hoc *SZZ* labels. The random forest classifier contains warning density based features in its most important features. Jit features, e.g., lines added, deleted or author experience remain important for both classifiers and both labeling strategies. However, this result indicates that we may be able to improve just-in-time defect prediction models with additional static source code metrics and static analysis warnings.

*RQ1 Summary:* Static as well as pmd warning density based features appear in the top 10 features in 3 out of 4 combinations.

To answer *RQ2*, we start with first replicating the approach utilized in the replication kit by Pascarella et al. [1], i.e., the commit label in a train/test split. Figure 2 shows both performance metrics of both of our models. We can see, that the random forest model performs best with only the jit features while the logistic regression model somewhat performs the same, with the combined features. Our results are consistent with the results obtained by Pascarella et al. [1].

We restrict the labeling now to ad-hoc and ITS *SZZ* labels. As described in Section IV-A with ad-hoc and ITS *SZZ* we only label the bug-inducing files themselves as bug-inducing independent of the commit. This reduces our positive instances significantly as shown in Table I and also impacts the overall performance.

Figure 3 shows the performance metrics on a train/test split on all of our available data. In comparison with the commit

TABLE VI  
TOP 10 FEATURES OF BOTH CLASSIFIERS

Logistic regression				Random forest			
ITS SZZ label		Ad-hoc SZZ label		ITS SZZ label		Ad-hoc SZZ label	
la (jit)	0.1085	la (jit)	0.3325	la (jit)	0.0143	la (jit)	0.0276
add (jit)	0.0812	age (jit)	-0.2253	file/system/WD (pmd)	0.0101	add (jit)	0.0208
del (jit)	0.0689	sctr (jit)	-0.2053	system/WD (pmd)	0.0101	exp (jit)	0.0142
entropy (jit)	-0.0656	add (jit)	0.1926	add (jit)	0.0099	oexp (jit)	0.0135
delta_CBO (static)	0.0472	nsctr (jit)	-0.1753	author/delta/WD (pmd)	0.0096	system/WD (pmd)	0.0134
current_NL (static)	0.0438	oexp (jit)	0.1722	ld (jit)	0.0095	entropy (jit)	0.0133
age (jit)	-0.0421	fix_bug (jit)	0.1335	exp (jit)	0.0094	author/delta/WD (pmd)	0.0126
current_NLE (static)	0.0398	ld (jit)	-0.1176	oexp (jit)	0.0085	sctr (jit)	0.0114
system/WD (pmd)	-0.0343	minor (jit)	-0.1113	entropy (jit)	0.0085	delta_HPL (static)	0.0106
current_NUMPAR (static)	0.0340	own (jit)	0.1087	sctr (jit)	0.0078	nd (jit)	0.0101

label we can see that with the ad-hoc label both classifiers performance improves slightly with regards to the combined feature set. The combined feature set does not perform best with ad-hoc but the improvement may be an indication that there is a possibility of the model performing better with the combined features. Our next step is to restrict analysis to the ITS SZZ label.

Figure 4 shows the performance metrics for the ITS SZZ label. We observe that the F-measure is significantly lower than with the ad-hoc SZZ labels. However, we can see that both classifiers perform slightly better for the combined feature set.

While until now we performed a train/test split of our data as is done in the replication kit of Pascarella et al. [1]. We now explore whether our assumption holds when evaluating our models in a time-sensitive approach.

Figure 5 and Figure 6 show both classifiers with the interval approach. There is a drop in model performance, especially the F-measure. Regardless of the limited power of the predictive models, as shown by their F-measure, we can see that what we previously demonstrated holds. Adding additional features consisting of static source code metrics and static analysis warnings can improve fine-grained just-in-time defect prediction models, especially if we consider the ITS SZZ labels.

We now rank the performance of both classifiers for all feature sets for each model performance metric using statistical tests. If the data is normally distributed and homoscedastic, we plot the confidence interval and mean for each feature set. Otherwise we plot the critical distance diagram. Figure 7 shows the confidence intervals as well as the critical distance diagrams for both classifiers combined and all feature sets in the train/test split setting. For AUC and F-measure the combined feature set is ranked first. The difference to the second rank is not significant for the ad-hoc SZZ label. However, the difference between first and second rank is significant for the ITS SZZ label for AUC and close to significant for F-measure. Moreover, while the jit feature set is second for the ad-hoc SZZ label this rank is occupied by the static feature set for the ITS SZZ label. Figure 8 shows the critical distance diagrams for both classifiers combined and all feature sets for the interval approach. Again, the combined feature set is ranked first for ad-hoc as well as ITS SZZ. However, the difference to the static features is not significant for the F-measure. We notice that for the ITS SZZ label the static metrics are more important than the jit metrics as was the case for the train/test split.

So far we determined that the combined feature set is ranked first for both AUC and F-measure for both labeling strategies as well as train/test split and interval approaches. However the difference is only significant in some cases. Table VII provides additional details. In addition to mean, standard deviation or in the case of critical distance diagrams, median and median absolute deviation, they provide effect sizes in the form of Cohen's  $d$  and Cliff's  $\delta$  as well as the confidence intervals.

The effect sizes indicate that the differences between the best ranked combined feature set and the second ranked feature set is often negligible. Thus, there is always at least one other feature set that performs similar to the combined feature set. For the ad-hoc SZZ labels, this is the jit feature set, for the ITS SZZ labels, this is the static feature set. However, the difference between combined features and the jit features is large with AUC with the ITS SZZ labels. Similarly, the difference between the combined features and the static features is large with AUC and medium with F-Measure for the ad-hoc labels. This means the combined feature set is the save choice, regardless of the type of labels.

*RQ2 Summary:* The combined feature set ranks first for AUC and F-measure in every configuration. While the difference to the second ranked feature set is negligible, all other feature sets rank significantly worse with a large effect size for at least once, indicating that the combined features improve the stability of just-in-time defect prediction.

For *RQ3* we calculate if cost savings are possible with the cost model for defect prediction introduced by Herbold [21]. The cost model provides boundary conditions for saving cost by taking predictions for bug-inducing files and the number of bugs into account. By inspecting lower and upper boundaries for each project we can see if we are able to save costs in more projects if we train the predictive model with more features.

Table VIII shows the end result of the cost model boundary calculations. For each label, feature set and classifier it shows the number of projects for which cost saving is possible depending on the costs of defects. For the interval approach it shows the number of intervals for which cost saving is possible. We can see that the number increases between the jit and combined feature sets for both classifiers and both labels. This further indicates that by including static source code metrics and static analysis warnings we may improve a

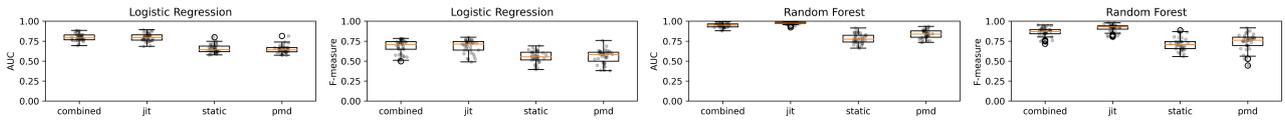


Fig. 2. Model performance metrics with ad-hoc SZZ commit label and train/test split

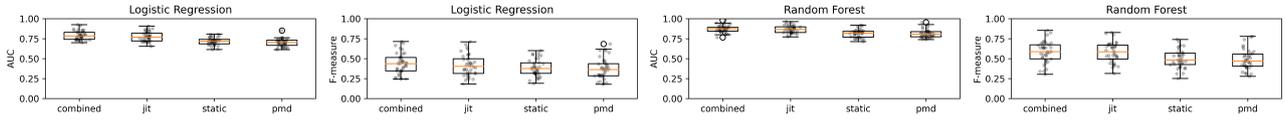


Fig. 3. Model performance metrics with ad-hoc SZZ label and train/test split

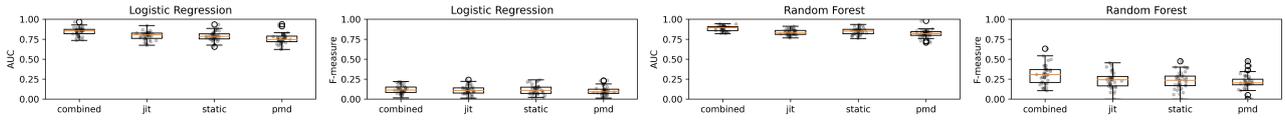


Fig. 4. Model performance metrics with ITS SZZ label and train/test split

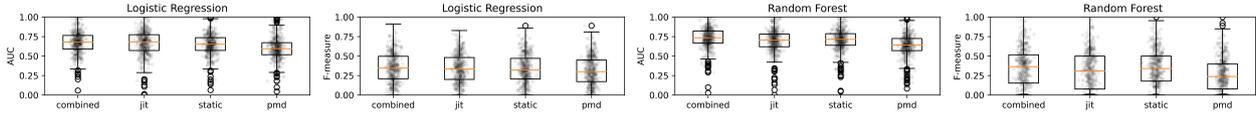


Fig. 5. Model performance metrics with ad-hoc SZZ label and interval approach

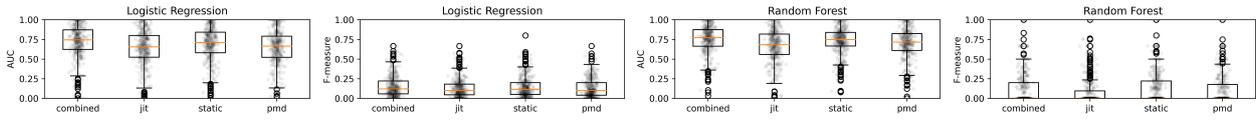


Fig. 6. Model performance metrics with ITS SZZ label and interval approach

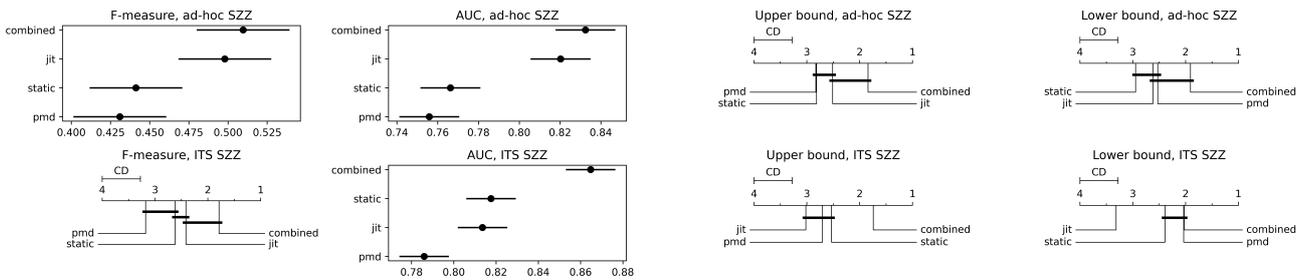


Fig. 7. Ranking of model performance metrics and cost boundaries for the train/test split

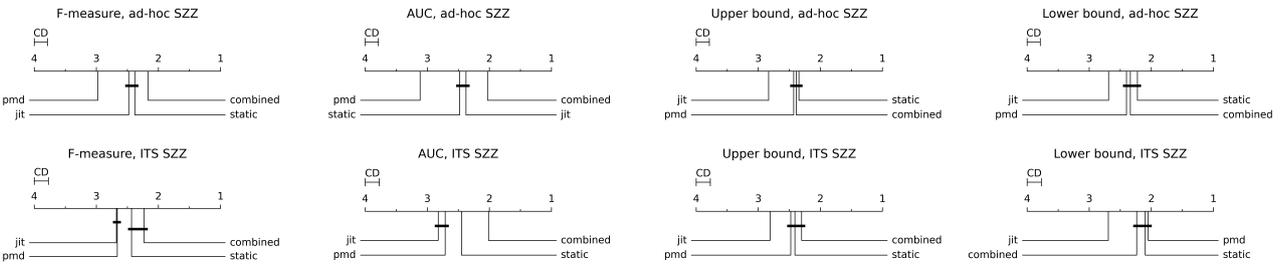


Fig. 8. Ranking of model performance metrics and cost boundaries for the interval approach

TABLE VII  
RANKING OF MODEL PERFORMANCE METRICS, MEAN (M), STANDARD DEVIATION (SD), MEDIAN (MED), MEAN ABSOLUTE ERROR (MAD), CONFIDENCE INTERVAL (CI), COHEN'S  $d$  ( $d$ ), CLIFF'S  $\delta$  ( $\delta$ ) AND EFFECT SIZE MAGNITUDES NEGLIGIBLE ( $n$ ), SMALL ( $s$ ), MEDIUM ( $m$ ), LARGE ( $l$ ). BOLDING DENOTES A STATISTICALLY SIGNIFICANT DIFFERENCE TO THE FIRST RANK

		Train/test split				
		M	SD	CI	$d$	
ad-hoc SZZ	AUC	combined	0.832	0.064	[0.818, 0.847]	0.000 ( $n$ )
		jit	0.820	0.071	[0.806, 0.835]	0.181 ( $n$ )
		static	0.766	0.070	[0.752, 0.781]	<b>0.988</b> ( $l$ )
		pmd	0.756	0.075	[0.741, 0.771]	<b>1.091</b> ( $l$ )
	F-measure	combined	0.510	0.149	[0.480, 0.539]	0.000 ( $n$ )
		jit	0.498	0.152	[0.468, 0.528]	0.078 ( $n$ )
		static	0.441	0.123	[0.412, 0.471]	<b>0.500</b> ( $m$ )
		pmd	0.431	0.139	[0.401, 0.461]	<b>0.545</b> ( $m$ )
ITS SZZ	AUC	combined	0.865	0.048	[0.853, 0.876]	0.000 ( $n$ )
		static	0.818	0.059	[0.806, 0.829]	<b>0.876</b> ( $l$ )
		jit	0.814	0.050	[0.802, 0.825]	<b>1.051</b> ( $l$ )
		pmd	0.786	0.065	[0.774, 0.798]	<b>1.376</b> ( $l$ )
	F-measure	combined	0.190	0.126	[0.110, 0.320]	0 ( $n$ )
		jit	0.151	0.103	[0.090, 0.250]	0.152 ( $s$ )
		static	0.152	0.111	[0.094, 0.247]	<b>0.156</b> ( $s$ )
		pmd	0.134	0.089	[0.080, 0.223]	<b>0.249</b> ( $s$ )
Interval approach						
ad-hoc SZZ	AUC	combined	0.707	0.121	[0.685, 0.732]	0.000 ( $n$ )
		jit	0.695	0.136	[0.664, 0.716]	<b>0.078</b> ( $n$ )
		static	0.681	0.126	[0.657, 0.709]	<b>0.110</b> ( $n$ )
		pmd	0.625	0.123	[0.597, 0.645]	<b>0.351</b> ( $n$ )
	F-measure	combined	0.350	0.236	[0.304, 0.400]	-0.000 ( $n$ )
		static	0.333	0.225	[0.286, 0.382]	<b>0.015</b> ( $n$ )
		jit	0.320	0.250	[0.273, 0.370]	<b>0.063</b> ( $n$ )
		pmd	0.272	0.227	[0.233, 0.320]	<b>0.158</b> ( $s$ )
ITS SZZ	AUC	combined	0.759	0.170	[0.730, 0.795]	0.000 ( $n$ )
		static	0.733	0.162	[0.703, 0.773]	<b>0.088</b> ( $n$ )
		pmd	0.697	0.186	[0.657, 0.727]	<b>0.202</b> ( $s$ )
		jit	0.672	0.199	[0.632, 0.716]	<b>0.247</b> ( $s$ )
	F-measure	combined	0.086	0.128	[0.049, 0.126]	0.000 ( $n$ )
		static	0.091	0.135	[0.055, 0.127]	-0.011 ( $n$ )
		pmd	0.062	0.091	[0.029, 0.100]	<b>0.057</b> ( $n$ )
		jit	0.054	0.080	[0.000, 0.087]	<b>0.119</b> ( $n$ )

TABLE VIII  
NUMBER OF PROJECTS/INTERVALS WHERE COST CAN BE SAVED FOR BOTH CLASSIFIERS AND MEAN NUMBER OF PROJECTS.

Label	Feature set	$\frac{1}{2}(\#LR + \#RF)$	#LR	#RF	
train/test split	ad-hoc SZZ	jit	23.0	24	22
		static	19.5	15	24
		pmd	20.5	13	28
		combined	28.5	26	31
	ITS SZZ	jit	24.5	23	26
		static	35.0	37	33
		pmd	32	35	29
		combined	34	32	36
interval	ad-hoc SZZ	jit	109	111	107
		static	170.5	162	179
		pmd	160.0	152	168
		combined	162.5	150	175
	ITS SZZ	jit	87.5	109	66
		static	129.5	146	113
		pmd	137	168	106
		combined	121	99	143

fine-grained just-in-time defect prediction approach and also save cost for software projects using the approach.

In addition to Table VIII, Figure 7 and Figure 8 show the upper and lower bounds ranked for each feature set. As the cost model defines the potential for cost saving the lower bound should be as low as possible while the upper bound as high as possible. To simplify a visual ranking we reversed the rank order for the lower bound the plots. For the train/test split in Figure 7 we can see that for the ad-hoc SZZ label the combined feature set is ranked first for upper and lower bound. While this indicates that more cost savings are possible with the combined feature set the critical distance to the next rank is not exceeded. For the ITS SZZ label we see that while the combined feature set is ranked first for the upper bound the best feature set for the lower bound is static. Although we note the large difference of the jit feature set to the others.

For the interval approach depicted in Figure 8 we can see that static performs best for the ad-hoc SZZ, with combined second. However the critical distance is not exceeded except for the jit feature set which performs worse. For the ITS SZZ label, combined is again best, although critical distance is only exceeded again for jit which performs worse. The lower bounds show that static is the best feature set for ad-hoc SZZ and pmd for ITS SZZ. However the critical distance to the combined feature set is not exceeded. This is also shown in Table VIII, we can see that models build with ITS SZZ and static/pmd features are able to save cost in more projects.

*RQ3 Summary:* The potential for cost saving is higher with a combined feature set than with only jit features. However, static and pmd features perform better with the ITS SZZ labeling strategy.

## VI. DISCUSSION

In the answer to our first research question regarding the importance of adding static features and static analysis warnings to just-in-time defect prediction we first find that the top 10 features for our regularized linear model and random forest contain static and pmd features in 3 out of 4 combinations. The linear model with ad-hoc SZZ labels is the only one which contains only jit features. This analysis shows that, given perfect knowledge, both ways to measure the importance of features indicate that static metrics can have correlations with defects. Since we use regularization to account for collinearity these correlations provide an indication that static source code metrics carry useful information about defects that is not contained in the features proposed by Kamei et al. [7] which are the standard choice for just-in-time defect prediction.

The results of the model evaluation show that for the label (commit) also used by Pascarella et al. [1] in their replication kit there is no performance gain when using additional metrics. Although, the more detailed the labeling process gets, i.e., ad-hoc SZZ for keyword only SZZ, ITS SZZ for full SZZ, the more positive impact additional static source code metrics and static analysis warnings as features have on the predictive models. This is also reflected by our final analysis which

incorporates a sliding window approach for time-sensitive analysis. The combined feature set is ranked first in every case. The performance drops between train/test split and interval are also in line with the literature, e.g., Tan et al. [32].

While PMD itself may be able to warn about issues that are responsible for bugs, it is not the primary use case as with FindBugs/SpotBugs. We inspected a small sample of bug-fixes from our data and found no removed warnings in the bug-fix changes. We believe that PMD and warning density may be useful features in a long term maintenance perspective, i.e., files that contain less static analysis warnings throughout their lifetime are better maintained, therefore they contain less bugs.

The results for *RQ3* show that in our reproduction of Pascarella et al. [1] our created models can save cost. We see that the combined feature set allows us to utilize the predictive model to save cost in more cases than the jit feature set. However, with ITS SZZ labels we see that the models built with static and pmd feature sets are able to save cost in more cases than the combined feature set. This is another indication that file-based metrics are more important for ITS SZZ labels than in an ad-hoc SZZ labeling strategy.

As a final note, we believe that both labeling strategies have their use. Ad-hoc SZZ labels can be used to distinguish possible quick fixes developers apply from possible bigger issues that are more indicative of an entry in an ITS. However, we have shown that it is very important to be aware of this as the defective rates for both approaches differ significantly, especially in a fine-grained scenario.

## VII. THREATS TO VALIDITY

In this section we discuss the threats to validity we identified for our work. To structure this section we discuss four basic types of validity, as suggested by Wohlin et al. [46].

### A. Construct Validity

The link between bug-fixing and bug-inducing commits is at the heart of this study. We are aware that some variants of the SZZ [5] algorithm have a certain imprecision [16, 47]. The ad-hoc SZZ label in our study ignores whitespace and comment changes while the ITS SZZ label additionally ignores refactoring changes which removes more false positives [20].

The ITS SZZ label in our study relies on a link between the ITS and the Version Control System (VCS), i.e., the bug-fixing commit must be linked to a valid issue of the type bug in the ITS. The type of the issue in the ITS may not reflect the real type but instead feature requests or other change requests [17, 18]. To mitigate this threat, our study subjects are based on a convenience sample of the Apache Software Foundation ecosystem. Not only do the ASF developers a good job of linking changes to issues, this sample also includes manually validated issue types and links [19].

### B. Internal Validity

Our results are influenced by the data collected from our study subjects. Factors that we are not able to change include the number of changes over time. This has a pronounced

impact on model performance as can be seen in Figure 5. As we do not want to choose our study subjects based on their commit history we are forced to handle fluctuating change histories. We do this by relaxing a strict time window as used in prior publications by also requiring a minimum number of changes for the dataset. Instead of choosing hard values for the number of changes we require the average number of changes for that time frame over the complete change history of the considered study subject. This improves the performance of the models and, in our eyes, is a reasonable choice. Nevertheless, this still is a factor that impacts our internal validity and warrants future research, i.e., how can just-in-time defect prediction work with all kinds of projects.

### C. External Validity

A threat to external validity is our project selection. Although the projects are all Java and originate from the same organization they contain a variety of developers due to their open source nature. Moreover, our sample contains a diverse set of application domains, e.g., wiki software, math libraries and build systems. Nevertheless, our results may not be applicable to all Java projects of the Apache Software Foundation much less all Java projects in existence.

### D. Conclusion Validity

As our study investigates many features we perform regularization on our linear logistic regression classifier. To further complete our view we additionally include a non linear random forest classifier. Both should be able to handle collinear features. Moreover, we apply statistical tests to enhance the validity of our conclusion for *RQ2* and *RQ3*.

## VIII. CONCLUSION

In this work we combined a state-of-the art just-in-time defect prediction approach with additional static source code metrics from OpenStaticAnalyzer and static analysis warnings from a well known Java static analysis tool (PMD). We create additional features based on warning density and show that additional features can improve just-in-time defect prediction models depending on the granularity of the labeling strategy. We investigated two labeling strategies in depth, ad-hoc SZZ and ITS SZZ and found that the more targeted the label the more the models performance is positively impacted by the additional features. We conclude that highly targeted models, i.e., models that target bugs linked to an ITS profit from the additional features.

We applied a defect prediction cost model to investigate if cost saving is possible with our created models. The number of projects where cost can be saved increases between jit only and combined feature sets. For ITS SZZ static and pmd feature sets provide more cost saving opportunities.

## IX. ACKNOWLEDGEMENTS

This work was partly funded by the German Research Foundation (DFG) through the project DEFECTS, grant 402774445.

## REFERENCES

- [1] L. Pascarella, F. Palomba, and A. Bacchelli, "Fine-grained just-in-time defect prediction," *Journal of Systems and Software*, vol. 150, pp. 22–36, 2019. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S0164121218302656>
- [2] S. McIntosh and Y. Kamei, "Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction," *IEEE Transactions on Software Engineering*, vol. 44, no. 5, pp. 412–428, May 2018.
- [3] M. Kondo, D. M. German, O. Mizuno, and E.-H. Choi, "The impact of context metrics on just-in-time defect prediction," *Empirical Software Engineering*, vol. 25, no. 1, pp. 890–939, 2020.
- [4] T. Hoang, H. Khanh Dam, Y. Kamei, D. Lo, and N. Ubayashi, "Deepjit: An end-to-end deep learning framework for just-in-time defect prediction," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, 2019, pp. 34–45.
- [5] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 4, pp. 1–5, May 2005.
- [6] G. G. Cabral, L. L. Minku, E. Shihab, and S. Mujahid, "Class imbalance evolution and verification latency in just-in-time software defect prediction," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, May 2019, pp. 666–676.
- [7] 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*, vol. 39, no. 6, pp. 757–773, June 2013.
- [8] M. Tan, L. Tan, S. Dara, and C. Mayeux, "Online defect prediction for imbalanced data," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2, May 2015, pp. 99–108.
- [9] Y. Yang, Y. Zhou, J. Liu, Y. Zhao, H. Lu, L. Xu, B. Xu, and H. Leung, "Effort-aware just-in-time defect prediction: Simple unsupervised models could be better than supervised models," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 157–168. [Online]. Available: <https://doi.org/10.1145/2950290.2950353>
- [10] M. D'Ambros, M. Lanza, and R. Robbes, "Evaluating defect prediction approaches: A benchmark and an extensive comparison," *Empirical Softw. Engg.*, vol. 17, no. 4-5, pp. 531–577, Aug. 2012. [Online]. Available: <http://dx.doi.org/10.1007/s10664-011-9173-9>
- [11] F. Rahman, S. Khatri, E. T. Barr, and P. Devanbu, "Comparing static bug finders and statistical prediction," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 424–434. [Online]. Available: <http://doi.acm.org/10.1145/2568225.2568269>
- [12] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl, and M. A. Vouk, "On the value of static analysis for fault detection in software," *IEEE Transactions on Software Engineering*, vol. 32, no. 4, pp. 240–253, April 2006.
- [13] P. Devanbu, T. Zimmermann, and C. Bird, "Belief evidence in empirical software engineering," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, May 2016, pp. 108–119.
- [14] S. Panichella, V. Arnaudova, M. D. Penta, and G. Antoniol, "Would static analysis tools help developers with code reviews?" in *2015 IEEE 22nd International Conference on Software Analysis, Evolution and Reengineering (SANER)*, vol. 00, March 2015, pp. 161–170.
- [15] L.-P. Querel and P. C. Rigby, "Warningsguru: Integrating statistical bug models with static analysis to provide timely and specific bug warnings," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 892–895. [Online]. Available: <https://doi.org/10.1145/3236024.3264599>
- [16] Y. Fan, D. Alencar da Costa, D. Lo, A. E. Hassan, and L. Shaping, "The impact of mislabeled changes by szz on just-in-time defect prediction," *IEEE Transactions on Software Engineering*, 2020.
- [17] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y.-G. Guéhéneuc, "Is it a bug or an enhancement?: A text-based approach to classify change requests," in *Proceedings of the 2008 Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds*, ser. CASCON '08. New York, NY, USA: ACM, 2008, pp. 23:304–23:318. [Online]. Available: <http://doi.acm.org/10.1145/1463788.1463819>
- [18] 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*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 392–401. [Online]. Available: <http://dl.acm.org/citation.cfm?id=2486788.2486840>
- [19] S. Herbold, A. Trautsch, F. Trautsch, and B. Ledel, "Issues with szz: An empirical study of the state of practice of defect prediction data collection," *Submitted to: Empirical Software Engineering*, 2020. [Online]. Available: <https://arxiv.org/abs/1911.08938>
- [20] E. C. Neto, D. A. da Costa, and U. Kulesza, "The impact of refactoring changes on the szz algorithm: An empirical study," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, March 2018, pp. 380–390.
- [21] S. Herbold, "On the costs and profit of software defect prediction," *IEEE Transactions on Software Engineering*, pp. 1–1, 2019.
- [22] Q. Huang, X. Xia, and D. Lo, "Supervised vs unsupervised models: A holistic look at effort-aware just-in-time defect prediction," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2017, pp. 159–170.
- [23] T. Jiang, L. Tan, and S. Kim, "Personalized defect prediction," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Nov 2013, pp. 279–289.
- [24] F. Rahman and P. Devanbu, "How, and why, process metrics are better," in *2013 35th International Conference on Software Engineering (ICSE)*, May 2013, pp. 432–441.
- [25] C. Rosen, B. Grawi, and E. Shihab, "Commit guru: Analytics and risk prediction of software commits," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 966–969. [Online]. Available: <https://doi.org/10.1145/2786805.2803183>
- [26] T. J. McCabe, "A complexity measure," *IEEE Trans. Softw. Eng.*, vol. 2, no. 4, pp. 308–320, Jul. 1976.
- [27] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. Softw. Eng.*, vol. 20, no. 6, pp. 476–493, Jun. 1994.
- [28] S. Hosseini, B. Turhan, and D. Gunarathna, "A systematic literature review and meta-analysis on cross project defect prediction," *IEEE Transactions on Software Engineering*, vol. PP, no. 99, pp. 1–1, 2017.
- [29] 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*, vol. 38, no. 6, pp. 1276–1304, Nov 2012.
- [30] L. Breiman, "Random forests," *Mach. Learn.*, vol. 45, no. 1, pp. 5–32, Oct. 2001. [Online]. Available: <http://dx.doi.org/10.1023/A:1010933404324>
- [31] E. Kreyszig, *Advanced Engineering Mathematics: Maple Computer Guide*, 8th ed. New York, NY, USA: John Wiley & Sons, Inc., 2000.
- [32] M. Tan, L. Tan, S. Dara, and C. Mayeux, "Online defect prediction for imbalanced data," in *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 99–108.
- [33] S. Herbold, "Autorank: A python package for automated ranking of classifiers," *Journal of Open Source Software*, vol. 5, no. 48, p. 2173, 2020. [Online]. Available: <https://doi.org/10.21105/joss.02173>
- [34] J. Demšar, "Statistical comparisons of classifiers over multiple data sets," *J. Mach. Learn. Res.*, vol. 7, pp. 1–30, Dec. 2006.
- [35] J. W. Tukey, "Comparing individual means in the analysis of variance," *Biometrics*, vol. 5, no. 2, pp. 99–114, 1949. [Online]. Available: <http://www.jstor.org/stable/3001913>
- [36] M. Friedman, "A comparison of alternative tests of significance for the problem of m rankings," *The Annals of Mathematical Statistics*, vol. 11, no. 1, pp. 86–92, 1940.
- [37] P. Nemenyi, "Distribution-free multiple comparison," Ph.D. dissertation, Princeton University, 1963.
- [38] J. Cohen, *Statistical power analysis for the behavioral sciences*. L. Erlbaum Associates, 1988.
- [39] N. Cliff, "Dominance statistics: Ordinal analyses to answer ordinal questions," *Psychological Bulletin*, vol. 114, no. 3, p. 494, 1993.
- [40] H. Abdi, "Bonferroni and Sidak corrections for multiple comparisons," in *Encyclopedia of Measurement and Statistics*. Sage, Thousand Oaks, CA, 2007, pp. 103–107.
- [41] F. Trautsch, S. Herbold, P. Makedonski, and J. Grabowski, "Addressing problems with replicability and validity of repository mining studies through a smart data platform," *Empirical Software Engineering*, Aug. 2017.

- [42] D. Spadini, M. Aniche, and A. Bacchelli, "PyDriller: Python framework for mining software repositories," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018*. New York, New York, USA: ACM Press, 2018, pp. 908–911. [Online]. Available: <http://dl.acm.org/citation.cfm?doid=3236024.3264598>
- [43] Q. Huang, X. Xia, and D. Lo, "Revisiting supervised and unsupervised models for effort-aware just-in-time defect prediction," *Empirical Software Engineering*, pp. 1–40, 2018.
- [44] X. Yang, D. Lo, X. Xia, and J. Sun, "Tlel: A two-layer ensemble learning approach for just-in-time defect prediction," *Information and Software Technology*, vol. 87, pp. 206 – 220, 2017. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S0950584917302501>
- [45] N. Fenton and J. Bieman, *Software Metrics: A Rigorous and Practical Approach, Third Edition*, 3rd ed. Boca Raton, FL, USA: CRC Press, Inc., 2014.
- [46] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering: An Introduction*. Norwell, MA, USA: Kluwer Academic Publishers, 2000.
- [47] D. A. da Costa, S. McIntosh, W. Shang, U. Kulesza, R. Coelho, and A. E. Hassan, "A framework for evaluating the results of the szz approach for identifying bug-introducing changes," *IEEE Transactions on Software Engineering*, vol. 43, no. 7, pp. 641–657, 2017.



## **D Improving data validity via automatic issue classification**

This section contains a copy of the following publication.

S. Herbold, A. Trautsch, F. Trautsch: On the feasibility of automated prediction of bug and non-bug issues. *Empirical Software Engineering* (2020) 25:5333–5369. Springer Nature

© 2020, The Author(s). Reprinted with permission.

<https://doi.org/10.1007/s10664-020-09880-1>



# On the feasibility of automated prediction of bug and non-bug issues

Steffen Herbold<sup>1</sup>  · Alexander Trautsch<sup>2</sup> · Fabian Trautsch<sup>2</sup>

Published online: 14 September 2020  
© The Author(s) 2020, corrected publication 2020

## Abstract

**Context** Issue tracking systems are used to track and describe tasks in the development process, e.g., requested feature improvements or reported bugs. However, past research has shown that the reported issue types often do not match the description of the issue.

**Objective** We want to understand the overall maturity of the state of the art of issue type prediction with the goal to predict if issues are bugs and evaluate if we can improve existing models by incorporating manually specified knowledge about issues.

**Method** We train different models for the title and description of the issue to account for the difference in structure between these fields, e.g., the length. Moreover, we manually detect issues whose description contains a null pointer exception, as these are strong indicators that issues are bugs.

**Results** Our approach performs best overall, but not significantly different from an approach from the literature based on the fastText classifier from Facebook AI Research. The small improvements in prediction performance are due to structural information about the issues we used. We found that using information about the content of issues in form of null pointer exceptions is not useful. We demonstrate the usefulness of issue type prediction through the example of labelling bugfixing commits.

**Conclusions** Issue type prediction can be a useful tool if the use case allows either for a certain amount of missed bug reports or the prediction of too many issues as bug is acceptable.

**Keywords** Issue type prediction · Mislabeled issues · Issue tracking

---

Communicated by: Burak Turhan

✉ Steffen Herbold  
steffen.herbold@kit.edu

Alexander Trautsch  
alexander.trautsch@cs.uni-goettingen.de

Fabian Trautsch  
fabian.trautsch@cs.uni-goettingen.de

<sup>1</sup> Institute AIFB, Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany

<sup>2</sup> Institute of Computer Science, University of Goettingen, Göttingen, Germany

## 1 Introduction

The tracking of tasks and issues is a common part of modern software engineering, e.g., through dedicated systems like Jira and Bugzilla, or integrated into other other systems like GitHub Issues. Developers and sometimes users of software file issues, e.g., to describe bugs, request improvements, organize work, or ask for feedback. This manifests in different *types* into which the issues are classified. However, past research has shown that the issue types are often not correct with respect to the content (Antoniol et al. 2008; Herzig et al. 2013; Herbold et al. 2020).

Wrong types of issues can have different kinds of negative consequences, depending on the use of the issue tracking system. We distinguish between two important use cases, that are negatively affected by misclassifications. First, the types of issues are important for measuring the progress of projects and project planing. For example, projects may define a quality gate that specifies that all issues of type bug with a major priority must be resolved prior to a release. If a feature request is misclassified as bug this may hold up a release. Second, there are many Mining Software Repositories (MSR) approaches that rely on issue types, especially the issue type bug, e.g., for bug localization (e.g., Marcus et al. 2004, Lukins et al. 2008, Rao and Kak 2011, Mills et al. 2018) or the labeling of commits as defective with the SZZ algorithm (Śliwerski et al. 2005) and the subsequent use of these labels, e.g., for defect prediction (e.g., Hall et al. 2012, Hosseini et al. 2017, Herbold et al. 2018) or the creation of fine-grained data (e.g., Just et al. 2014). Mislabeled issues threaten the validity of the research and would also degenerate the performance of approaches based on this data that are implemented in tools and used by practitioners. Thus, mislabeled issues may have direct negative consequences on development processes as well as indirect consequences due to the downstream use of possibly noisy data. Studies by Herzig et al. (2013) and Herbold et al. (2020) have independently and on different data shown that on average about 40% issues are mislabelled, and most mislabels are issues wrongly classified as BUG.

There are several ways on how to deal with mislabels. For example, the mislabels could be ignored and in case of invalid blockers manually corrected by developers. Anecdotal evidence suggests that this is common in the current state of practice. Only mislabels that directly impact the development, e.g., because they are blockers, are manually corrected by developers. With this approach the impact of mislabels on the software development processes is reduced, but the mislabels may still negatively affect processes, e.g., because the amount of bugs is overestimated or because the focus is inadvertently on the addition of features instead of bug fixing. MSR would also still be affected by the mislabels, unless manual validation of the data is done, which is very time consuming (Herzig et al. 2013; Herbold et al. 2020).

Researchers suggested an alternative through the automated classification of issue types by analyzing the issue titles and descriptions with unsupervised machine learning based on clustering the issues Limsettho et al. (2014b), Hammad et al. (2018) and Chawla and Singh (2018) and supervised machine learning that create classification models (Antoniol et al. 2008; Pingclasai et al. 2013; Limsettho et al. 2014a; Chawla and Singh 2015; Zhou et al. 2016; Terdchanakul et al. 2017; Pandey et al. 2018; Qin and Sun 2018; Zolkeply and Shao 2019; Otoom et al. 2019; Kallis et al. 2019). There are two possible use cases for such automated classification models. First, they could be integrated into the issue tracking system and provide recommendations to the reporter of the issue. This way, mislabeled data in the issue tracking system could potentially be prevented, which would be the ideal solution. The alternative is to leave the data in the issue tracking unchanged, but use machine learning as part of software repository mining pipelines to correct mislabeled issues. In

this case, the status quo of software development would remain the same, but the validity of MSR research results and the quality of MSR tools based on the issue types would be improved.

Within this article, we want to investigate if machine learning models for the prediction of issue types can be improved by incorporating a-priori knowledge about the problem through predefined rules. For example, null pointer exceptions are almost always associated with bugs. Thus, we investigate if separating issues that report null pointers from those that do not contain null pointers improves the outcome. Moreover, current issue type prediction approaches ignore that the title and description of issues are structurally different and simply concatenate the field for the learning. However, the title is usually shorter than the description which may lead to information from the description suppressing information from the title. We investigate if we can improve issue type prediction by accounting for this structural difference by treating the title and description separately. Additionally, we investigate if mislabels in the training data are really problematic or if they do not negatively affect the decisions made by classification models. To this aim, we compare how the training with large amounts of data that contains mislabels performs in comparison to training with a smaller amount of data that was manually validated. Finally, we address the question how mature machine learning based issue type correction is and evaluate how our proposed approach, as well as the approaches from the literature, perform in two scenarios: 1) the classification of all issues regardless of their type and 2) the classification of only issues that are reported as bug. The first scenario evaluates how good the approaches would work in recommendation systems where a label must be suggested for every incoming issue. The second scenario evaluates how good the approaches would work to correct data for MSR. We only consider the correction of issues of type bug, because both Herzig et al. (2013) and Herbold et al. (2020) found that mislabels mostly affect issues of type bug. Moreover, many MSR approaches are interested in identifying bugs.

Thus, the research questions we address in the article are the following.

- **RQ1:** Can manually specified logical rules derived from knowledge about issues be used to improve issue type classification?
- **RQ2:** Does training data have to be manually validated or can a large amount of unvalidated data also lead to good classification models?
- **RQ3:** How good are issue type classification models at recognizing bug issues and are the results useful for practical applications?

We provide the following contributions to the state of the art through the study of these research questions.

- We determined that the difference in the structure of the issue title and description may be used to slightly enhance prediction models by training separate predictors for the title and the description of issues.
- We found that rules that determine supposedly easy subsets of data based on null pointers do not help to improve the quality of issue type prediction models aimed at identifying bugs.
- We were successfully able to use unvalidated data to train issue type prediction models that perform well on manually validated test data with performance comparable to the currently assigned labels by developers.
- We showed that issue type prediction is a useful tool for researchers interested in improving the detection of bugfixing commits. The quality of the prediction

models also indicate that issue type prediction may be useful for other purposes, e.g., as recommendation system.

- We provide open source implementations of the state of the art of automated issue type prediction as a Python package.

The remainder of this paper is structured as follows. We describe the terminology we use in this paper and the problems we are analyzing in Section 2, followed by a summary of the related work on issue type prediction in Section 3. Afterwards, we discuss our proposed improvements to the state of the art in the sections 4 and 5. We present the design and results of our empirical study of issue type prediction in Section 6 and further discuss our findings in Section 7. Finally, we discuss the treats to the validity of our work in Section 8 before we conclude in Section 9.

## 2 Terminology and Problem Description

Before we proceed with the details of the related work and our approach, we want to establish a common terminology and describe the underlying problem. Figure 1 shows a screenshot of the issue MATH-533 from the Jira issue tracking system of the Apache Software Foundation. Depending on the development process and the project, issues can either be reported by anyone or just by a restricted group of users, e.g., developers, or users with paid maintenance contracts. In open source projects, it is common that everybody can report issues. Each issue contains several fields with information. For our work, the title, description, type, and discussion are relevant. The title contains a (very) brief textual summary of the issue, the description contains a longer textual description that should provide all relevant details. The reporter of an issue specifies the type and the title, although they may be edited later. The reporter of an issue also specifies the type, e.g., bug, improvement,

**Fig. 1** Example of a Jira Issue from the Apache Commons Math project. Names were redacted due to data privacy concerns

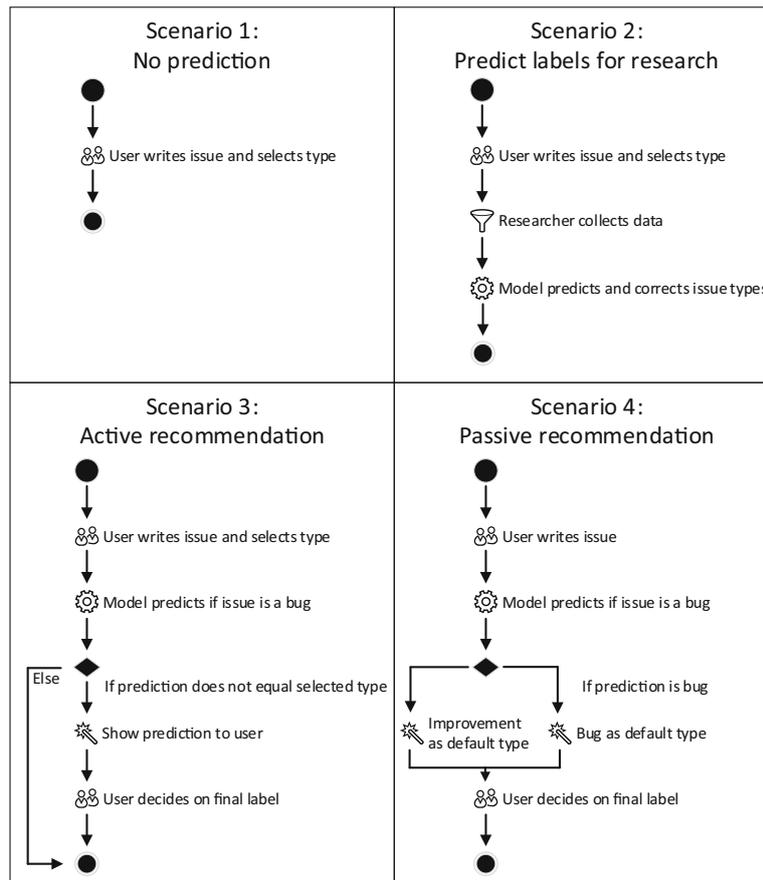
documentation change. The concrete types that are available are usually configurable and may be project dependent. However, the type bug exists almost universally.<sup>1</sup> Once the issue is reported, others can comment on the issue and, e.g., discuss potential solutions or request additional information. While the above example is for the Jira issue tracking system, similar fields can be found in other issue trackers as well, e.g., Bugzilla, Github Issues, and Gitlab Issues.

We speak of *mislabeled* issues, when the issue type does not match the description of the problem. Herzig et al. (2013) created a schema that can be used to identify the type of issues as either *bug* (e.g., crashes), *request for improvements* (e.g., update of a dependency), *feature requests* (e.g., support for a new communication protocol), *refactoring* (non-semantic change to the internal structure), *documentation* (change of the documentation), or *other* (e.g., changes to the build system or to the licenses). Herbold et al. (2020) used a similar schema, but merged the categories request for improvements, feature request, and refactoring into a single category called *improvement* and added the category *tests* (changes to tests). Figure 1 shows an example for a mislabel. The reported problem is a missing Javadoc tag, i.e., the issue should be of type documentation. However, the issue is reported as bug instead.

Since both Herzig et al. (2013) and Herbold et al. (2020) found that the main source of mislabels are issues that are reported as bug, even though they do not constitute bugs, but rather improvements of potentially sub optimal situations, we restrict our problem from a general prediction system for any type of issue to a prediction system for bugs. This is in line with the prior related work, with the exception of Antoniol et al. (2008) and (Kallis et al. 2019), who considered additional classes. Thus, we have a binary classification problem, with the label *true* for issues that describe bugs, and *false* for issues that do not describe bugs. Formally, the prediction model is a function  $h_{all} : ISSUE \rightarrow \{true, false\}$ , where *ISSUE* is the space of all issues. In practice, not all information from the issue used, but instead, e.g., only the title and/or description. Depending on the scenarios we describe in the following, the information available to the prediction system may be limited.

There are several ways such a recommendation system can be used, which we describe in Fig. 2. The Scenario 1 is just the status quo, i.e., a user creates an issue and selects the type. In Scenario 2, no changes are made to the actual issue tracking system. Instead, researchers use a prediction system as part of a MSR pipeline to predict issue types and, thereby, correct mislabels. In this scenario, all information from the issue tracking system is available, including changes made to the issue description, comments, and potentially even the source code change related to the issue resolution. The third and fourth scenario show how a prediction system can be integrated into an issue tracker, without taking control from the users. In Scenario 3, the prediction system gives active feedback to the users, i.e., the users decide on a label on their own and in case the prediction system detects a potential mistake, the users are asked to either confirm or change their decision. Ideally, the issue tracking system would show additional information to the user, e.g., the reason why the system thinks should be of a different type. Scenario 4 acts passively by prescribing different default values in the issue system, depending on the prediction. The rationale behind Scenario 4 is that Herzig et al. (2013) found that Bugzilla's default issue type of BUG led to more mislabels, because this was never changed by users. In Scenario 3 and Scenario 4 the information available to the prediction system is limited to the information users provide upon reporting the issue, i.e., subsequent changes or discussions may not be used. A variant of Scenario 4 would be

<sup>1</sup>At least we have never seen an issue tracking system for software projects without this type.



**Fig. 2** Overview of the scenarios how prediction systems for bug issues can be used

a fully automated approach, where the label is directly assigned and the users do not have to confirm the label but would have to actively modify it afterwards. This is the approach implemented in the Ticket Tagger by Kallis et al. (2019).

Another aspect in which the scenarios differ is to which issues the prediction model is applied, depending on the goal. For example, a lot of research is interested specifically in bugs. Herzig et al. (2013) and Herbold et al. (2020) both found that almost all bugs are classified by users as type bug, i.e., there are only very few bugs that are classified otherwise in the system. To simplify the problem, one could therefore build a prediction model  $h_{bug} : BUG \rightarrow \{true, false\}$  where  $BUG \subset ISSUE$  are only issues which users labeled as bug. Working with such a subset may improve the prediction model for this subset, because the problem space is restricted and the model can be more specific. However, such a model would only work in Scenario 2, i.e., for use by researchers only, or Scenario 3, in case the goal is just to prevent mislabeled bugs. Scenario 4 would, therefore, not work with the  $h_{bug}$  model.

### 3 Related Work

That classifications of issue types have a large impact on, e.g., defect prediction research was first shown by Herzig et al. (2013). They manually validated 7,401 issue types of five projects and provided an analysis of the impact of misclassifications. They found that every

third issue that is labeled as defect in the issue tracking systems is not a defect. This introduces a large bias in defect prediction models, as 39% of files are wrongly classified as defective due to the misclassified issues that are linked to changes in the version control system. Herbold et al. (2020) independently confirmed the results by Herzig et al. (2013) and demonstrated how this and other issues negatively impact defect prediction data. However, while both Herzig et al. (2013) and Herbold et al. (2020) study the impact of mislabels of defect prediction, any software repository mining research that studies defects suffers from similar consequences, e.g., bug localization (e.g., Marcus et al. 2004, Lukins et al. 2008, Rao and Kak 2011, Mills et al. 2018). In the literature, there are several approaches that try to address the issue of mislabels in issue systems through machine learning. These approaches can be divided into unsupervised approaches and supervised approaches.

### 3.1 Unsupervised Approaches

The unsupervised approaches work on clustering the issues into groups and then identifying for each group their likely label. For example (Limsettho et al. 2014b; 2016) use Xmeans and EM clustering, Chawla and Singh (2018) use Fuzzy C Means clustering and Hammad et al. (2018) use agglomerative hierarchical clustering. However, the inherent problem of these unsupervised approaches is that they do not allow for an automated identification of the label for each cluster, i.e., the type of issue per cluster. As a consequence, these approaches are unsuited for the creation of automated recommendation systems or the use as automated heuristics to improve data and not discussed further in this article.

### 3.2 Supervised Approaches

The supervised approaches directly build classification models that predict the type of the issues. To the best of our knowledge, the first approach in this category was published by Antoniol et al. (2008). Their approach uses the descriptions of the issues as input, which are preprocessed by tokenization, splitting of camel case characters and stemming. Afterwards, a Term Frequency Matrix (TFM) is built including the raw term frequencies for each issue and each term. The TFM is not directly used to describe the features used as input for the classification algorithm. Instead, Antoniol et al. (2008) first use symmetrical uncertainty attribute selection to identify relevant features. For the classification, they propose to use Naïve Bayes (NB), Logistic Regression (LR), or Alternating Decision Trees (ADT).

The TFM is also used by other researchers to describe the features. Chawla and Singh (2015) propose to use fuzzy logic based the TFM on the issue title. The fuzzy logic classifier is structurally similar to a NB classifier, but uses a slightly different scoring function. Pandey et al. (2018) propose to use the TFM of the issue titles as input for NB, Support Vector Machine (SVM), or LR classifiers. Otoom et al. (2019) propose to use a variant of the TFM with a fixed word set. They use a list of 15 keywords related to non-bug issues (e.g., enhancement, improvement, refactoring) and calculate the term frequencies for them based on the title and description of the issue. This reduced TFM is then used as an input for NB, SVM, or Random Forest (RF). Zolkeply and Shao (2019) propose to not use TFM frequencies, but simply the occurrence of one of 60 keywords as binary features and use these to train a Classification Association Rule Mining (CARM). Terdchanakul et al. (2017) propose to go beyond the TFM and instead use the Inverse Document Frequency (IDF) of n-grams for the title and descriptions of the issues as input for either LR or RF as classifier.

Zhou et al. (2016) propose an approach that combines the TFM from the issue title with structured information about the issue, e.g., the priority and the severity. The titles are

classified into the categories high (can be clearly classified as bug), low (can be clearly classified as non-bug), and middle (hard to decide) and they use the TFM to train a NB classifier for these categories. The outcome of the NB is then combined with the structural information as features used to train a Bayesian Network (BN) for the binary classification into bug or not a bug.

There are also approaches that do not rely on the TFM. Pingclasai et al. (2013) published an approach based on topic modeling via the Latent Dirichlet Allocation (LDA). Their approach uses the title, description, and discussion of the issues, preprocesses them, and calculates the topic-membership vectors via LDA as features. Pingclasai et al. (2013) propose to use either Decision Trees (DT), NB, or LR as classification algorithm. Limsettho et al. (2014a) propose a similar approach to derive features via topic-modeling. They propose to use LDA or Hierarchical Dirichlet Process (HDP) on the title, description, and discussion of the issues to calculate the topic-membership vectors as features. For the classification, they propose to use ADT, NB, or LR. In their case study, they have shown that LDA is superior to HDP. We note that the approaches by Pingclasai et al. (2013) and Limsettho et al. (2014b) both cannot be used for recommendation systems, because the discussion is not available at the time of reporting the issue. Qin and Sun (2018) propose to use word embeddings of the title and description of the issue as features and use these to train a Long Short-Term Memory (LSTM). Palacio et al. (2019) propose to use the SecureReqNet (shallow)<sup>2</sup>

Kallis et al. (2019) created the tool Ticket Tagger that can be directly integrated into GitHub as a recommendation system for issue type classification. The Ticket Tagger uses the fastText Facebook AI Research (2019) algorithm, which uses the text as input and internally calculates a feature representation that is based on n-grams, but not of the words, but of the letters within the words. These feature are used to train a neural network for the text classification.

The above approaches all rely on fairly common text processing pipelines to define features, i.e., the TFM, n-grams, IDF, topic modeling or word embeddings to derive numerical features from the textual data as input for various classifiers. In general, our proposed approach is in line with the related work, i.e., we also either rely on a standard text processing pipeline based on the TFM and IDF as input for common classification models or use the fastText algorithm which directly combines the different aspects. The approaches by Otoom et al. (2019) and Zolkeply and Shao (2019) try to incorporate manually specified knowledge into the learning process through curated keyword lists. Our approach to incorporate knowledge is a bit different, because we rather rely on rules that specify different training data sets and do not restrict the feature space.

## 4 Approach

Within this section, we describe our approach for issue type prediction that allows us to use simple rules to incorporate knowledge about the structure of issues and the issues types in the learning process in order to study RQ1.

---

<sup>2</sup>The network is still a deep neural network, the (shallow) means that this is the less deep variant that was used in by Palacio et al. (2019), because they found that this performs better. neural network based on work by Han et al. (2017) for the labeling of issues as vulnerabilities. The neural network uses word embeddings and a convolutional layer that performs 1-gram, 3-gram, and 5-gram convolutions that are then combined using max-pooling and a fully connected layer to determine the classification.

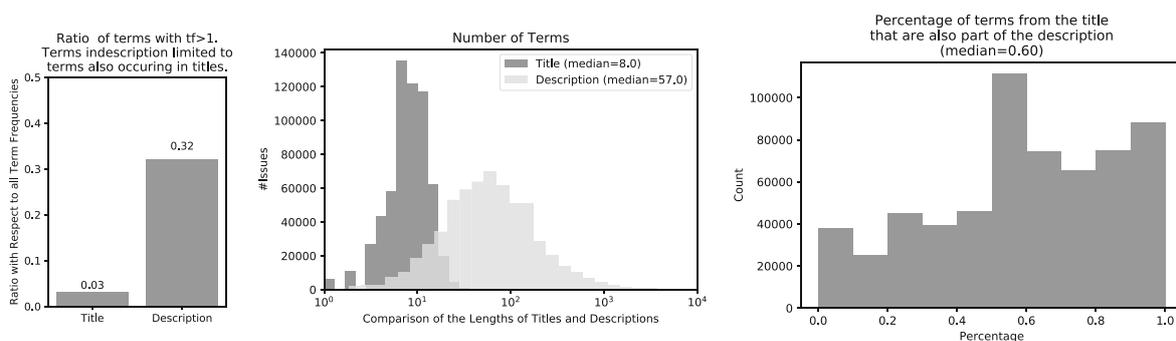
## 4.1 Title and Description

We noticed that in the related work, researchers used the title and description together, i.e., as a single document in which the title and description are just concatenated. From our point of view, this ignores the properties of the fields, most notably the length of the descriptions. Figure 3 shows data for the comparison of title and description. The title field is more succinct, there are almost never duplicate words, i.e., term frequency will almost always be zero or one. Moreover, the titles are very short. Thus, the occurrence of a term in a title is more specific for the issue than the occurrence of a term in the description. The description on the other hand is more verbose, and may even contain lengthy code fragments or stack traces. Therefore, many terms occur multiple times and the occurrence of terms is less specific. This is further highlighted by the overlap of terms between title and description. Most terms from the title occur also in the description and the terms lose their uniqueness when the title and description are considered together. As a result, merging of the title and description field may lead to suppressing information from the title in favor of information from the description, just due to the overall lengths of the fields and the higher term frequencies. This loss of information due to the structure of the features is undesirable.

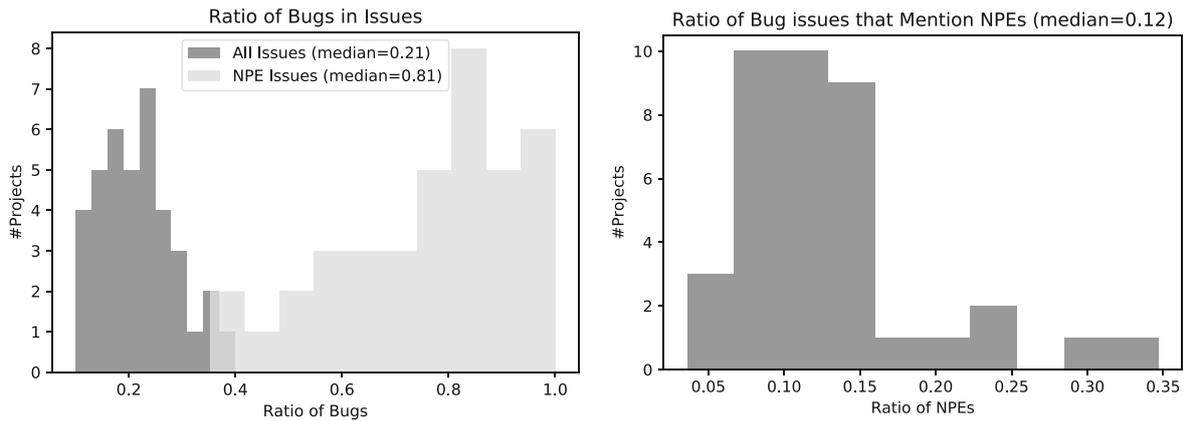
We propose a very simple solution to this problem, i.e., the training of different prediction models for title and description. The results of both models can then be combined into a single result. Specifically, we suggest that classifiers that provide scores, e.g., probabilities for classes are used and the mean value of the scores is then used as prediction. Thus, we have two classifiers  $h_{title}$  and  $h_{description}$  that both predict values in  $[0, 1]$  and our overall probability that the issue is a bug is  $\frac{h_{title}+h_{description}}{2}$ . This generic approach works with any classifier, and could, also be extended with a third classifier for the discussion of the issues.

## 4.2 Easy Subsets

An important aspect we noted during our manual validation of bugs for our prior work (Herbold et al. 2020) was that not all bugs are equal, because some rules from Herzig et al. (2013) are pretty clear. The most obvious example is that almost anything related to an unwanted null pointer exception is a bug. Figure 4 shows that mentioning a null pointer is a good indicator for a bug and that there is a non-trivial ratio of issues that mention null pointers. However, the data also shows that a null pointer is no sure indication that the issue is actually a bug and cannot be used as a static rule. Instead, we wanted to know if we can enhance the machine learning models by giving them the advantage of knowing that something is related to a null pointer. We used the same approach as above. We decided to



**Fig. 3** Visualization of the structural differences of issue titles and descriptions based on the 607,636 Jira Issues from the UNVALIDATED data (see Section 6.1)



**Fig. 4** Data on the usage of the terms NullPointerException, NPE, and NullPointer in issues based on 30,922 issues from the CV data (see Section 6.1)

train one classifier for all issues that mention the terms “NullPointerException”, “NPE”, or “NullPointer” in the title or description, and a second classifier for all other issues. Together with the separate classifiers for title and description, we now have four classifiers, i.e., one classifier for the title of null pointer issues, one classifier for the description of null pointer issues, one classifier for the title of the other issues, and one classifier for the description of the other issues. We do not just use the average of these four classifiers. Instead, the prediction model checks if an issue mentions a null pointer and then uses either the classifiers for null pointer issues or for the other issues.

### 4.3 Classification Model

So far, we only described that we want to train different classifiers to incorporate knowledge about issues into the learning process. However, we have not yet discussed the classifiers we propose. We consider two approaches that are both in line with the current state of the art in issue type prediction (see Section 3).

The first approach is a simple text processing pipeline as can be found in online tutorials on text mining<sup>3</sup> and is similar to the TFM based approaches from the literature. As features, we use the TF-IDF of the terms in the documents. This approach is related to the TFM but uses the IDF as scaling factor. The IDF is based on the number of issues in which a term occurs, i.e.,

$$IDF(t) = \log \frac{n}{df(t)} + 1 \quad (1)$$

where  $n$  is the number of issues and  $df(t)$  is the number of issues in which the term  $t$  occurs. The TF-IDF of a term  $t$  in an issue  $d$  is computed as

$$TF - IDF(t, d) = TF(t, d) \cdot IDF(t) \quad (2)$$

where  $TF(t, d)$  is the term frequency of  $t$  in  $d$ . The idea behind using TF-IDF instead of just TF is that terms that occur in many documents may be less informative and are, therefore, down-scaled by the IDF. We use the TF-IDF of the terms in the issues as features for our first approach and use multinomial NB and RF as classification models. We use the TF-IDF

<sup>3</sup>e.g., <https://www.hackerearth.com/de/practice/machine-learning/advanced-techniques/text-mining-feature-engineering-r/tutorial/> [https://scikit-learn.org/stable/tutorial/text\\_analytics/working\\_with\\_text\\_data.html](https://scikit-learn.org/stable/tutorial/text_analytics/working_with_text_data.html)

implementation from Scikit-Learn (Pedregosa et al. 2011) with default parameters, i.e., we use the lower-case version of all terms without additional processing.

Our second approach is even simpler, taking pattern from Kallis et al. (2019). We just use the fastText algorithm (Facebook AI Research 2019) that supposedly does state of the art text mining on its own, and just takes the data as is. The idea behind this is that we just rely on the expertise of one of the most prominent text mining teams, instead of defining any own text processing pipeline. We apply the fastText algorithm once with the same parameters as were used by Kallis et al. (2019) and once with an automated parameter tuning that was recently made available for fastText<sup>4</sup>. The automated parameter tuning does not perform a grid search, but instead uses a guided randomized strategy for the hyper parameter optimization. A fixed amount of time is used to bound this search. We found that 90 seconds was sufficient for our data, but other data sets may require longer time. In the following, we refer to fastText as FT and the autotuned fastText as FTA.

Please note that we do not consider any deep learning based text mining techniques (e.g., BERT by Devlin et al. (2018)) for the creation of a classifier, because we believe that we do not have enough (validated) data to train a deep neural network. We actually have empirical evidence for this, as the deep neural networks we used in our experiments do not perform well (see Section 6, Qin2018-LSTM, Palacio2019-SRN). Deep learning should be re-considered for this purpose once the requirements on data are met, e.g., through pre-trained word embeddings based on all issues reported at GitHub.

#### 4.4 Putting it all Together

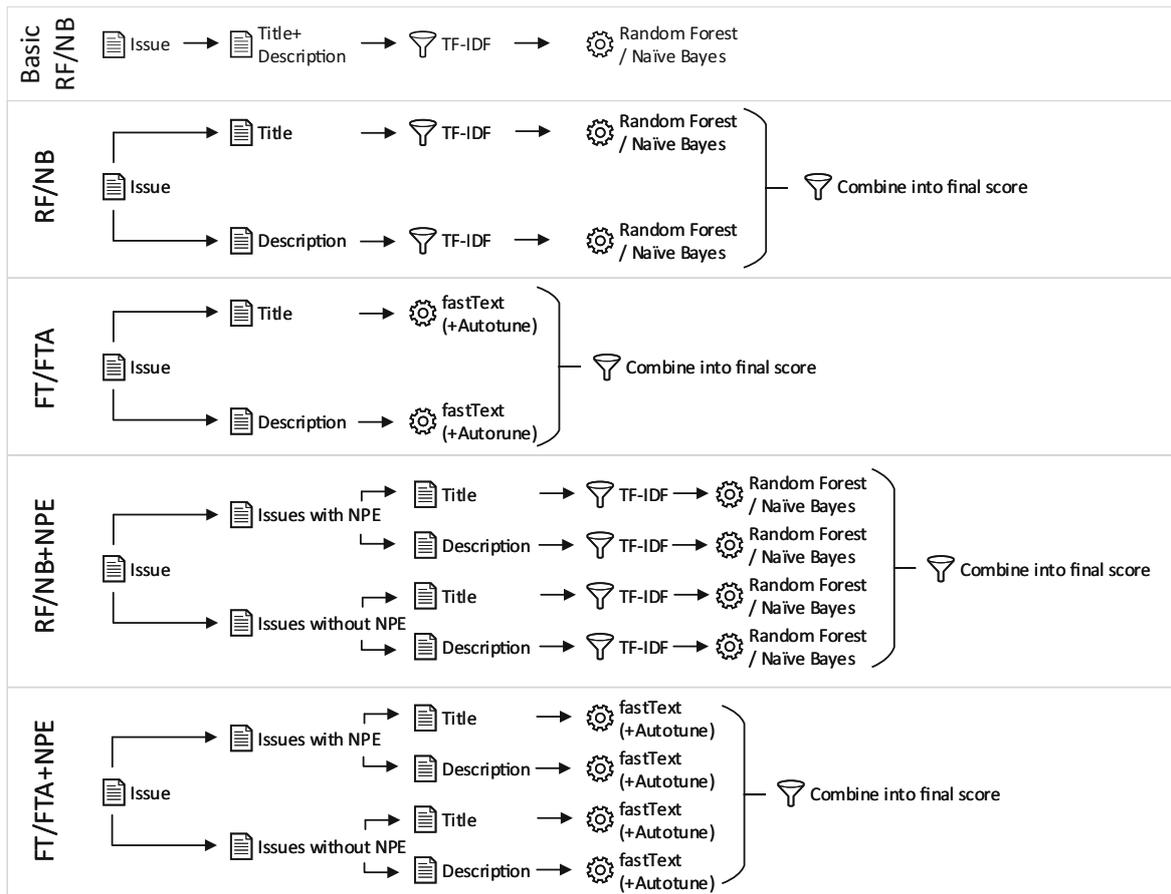
From the different combinations of rules and classifiers, we get ten different classification models for our approach that we want to evaluate, which we summarize in Fig. 5. First, we have Basic-RF and Basic-NB, which train classifiers on the merged title and description, i.e., a basic text processing approach without any additional knowledge about the issues provided by us.<sup>5</sup> This baseline allows us to estimate if our rules actually have a positive effect over not using any rules. Next, we have RF, NB, FT, and FTA which train different classifiers for the title and description as described in Section 4.1. Finally, we extend this with separate classifiers for null pointers and have the models RF+NPE, NB+NPE, FT+NPE, and FTA+NPE.

### 5 Unvalidated Data

A critical issue with any machine learning approach is the amount of data is available for the training. The validated data about the issue types that accounts for mislabels is limited, i.e., there are only the data sets by Herzig et al. (2013) and Herbold et al. (2020). Combined, they contain validated data about roughly 15,000 bugs. While this may be sufficient to train a good issue prediction model with machine learning, the likelihood of getting a good model that generalizes to many issues increases with more data. However, it is unrealistic that vast amounts of manually labelled data become available, because of the large amount of manual

<sup>4</sup><https://ai.facebook.com/blog/fasttext-blog-post-open-source-in-brief/>

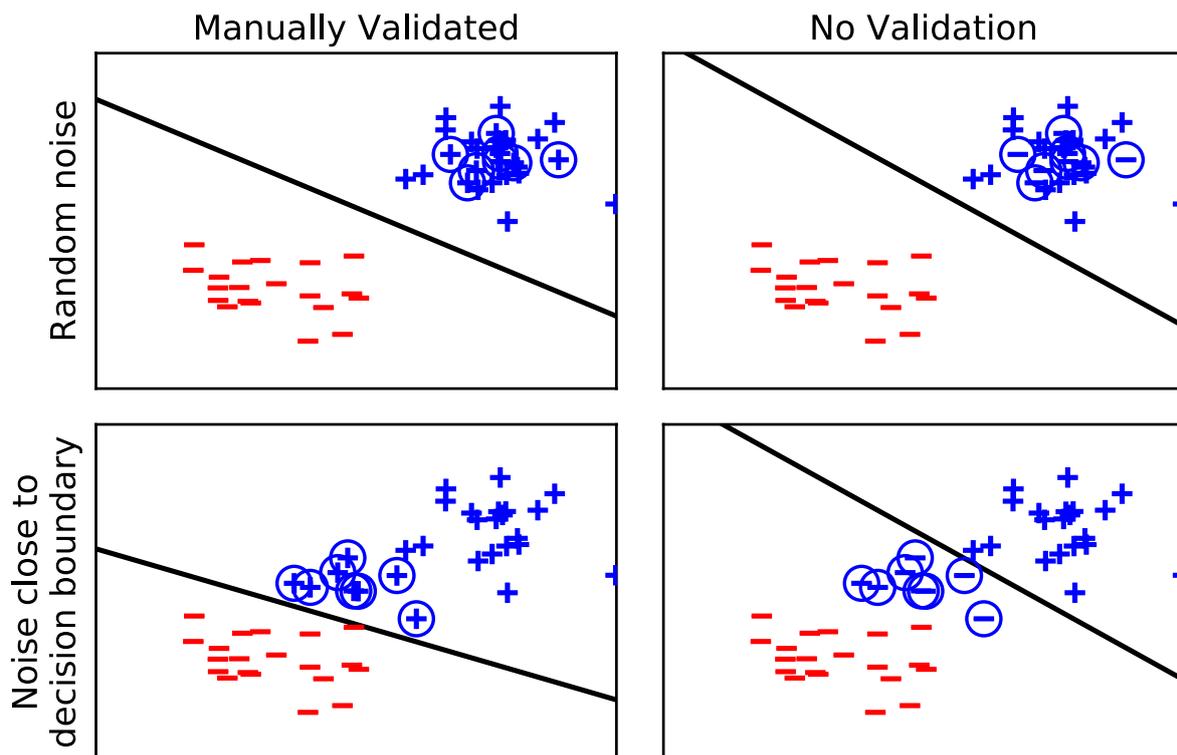
<sup>5</sup>Basic-FT is omitted, because this is the same as the work by Kallis et al. (2019) and, therefore, already covered by the literature and in our experiments in Section 6.



**Fig. 5** Summary of our approach

effort involved. The alternative is to use data that was not manually labelled, but instead use the user classification for the training. In this case, all issues from more or less any issue tracker can be used as training data. Thus, the amount of data available is huge. However, the problem is that the resulting models may not be very good, because the training data contains mislabels that were not manually corrected. This is the same as noise in the training data. While this may be a problem, it depends on where the mislabels are, and also on how much data there is that is correctly labelled.

Figure 6 shows an example that demonstrates why training with unvalidated data may work and why it may fail. The first column shows data that was manually corrected, the second column shows data that was not corrected and contains mislabels. In the first row, the mislabels are random, i.e., random issues that are not a bug are mislabeled as bugs. In this case, there is almost no effect on the training, as long as there are more correctly labelled instances than noisy instances. Even better, the prediction model will even predict the noisy instances correctly, i.e., the prediction would actually be better than the labels of the training data. Thus, noise as in the first example can be ignored for training the classifier. This is line with learning theory, e.g., established by Kearns (1998) who demonstrated with the statistical query model that learning in the presence of noise is possible, if the noise is randomly distributed. In the second row, the mislabels are not random, but close to the decision boundary, i.e., the issues that are most similar to bugs are mislabeled as bugs. In this case, the decision boundary is affected by the noise and would be moved to the top-right of the area without manual validation. Consequently, the trained



**Fig. 6** Example for the possible effect of mislabels in the training data on prediction models. The color indicates the correct labels, i.e., red for bugs and blue for other issues. The marker indicates the label in the training data, - for bugs, + for other issues. Circled instances are manually corrected on the left side and mislabels on the right side. The line indicates the decision boundary of the classifier. Everything below the line is predicted as a bug, everything above the line is predicted as not a bug

model would still mislabel all instances that are mislabeled in the training data. In this case, the noise would lead to a performance degradation of the training and cannot be ignored.

Our hope is that mislabeled issues are mostly of the first kind, i.e., randomly distributed honest mistakes. In this case, a classifier trained with larger amounts of unlabeled data should perform similar or possibly even better than a classifier trained with a smaller amount of validated data.

## 6 Experiments

We now describe the experiments we conducted to analyze issue type prediction. The experiments are based on the Python library `icb`<sup>6</sup> that we created as part of our work. `icb` provides implementations for the complete state of the art of supervised issue type prediction (Section 3.2) with the exceptions described in Section 6.2. The additional code to conduct our experiments is provided as a replication package<sup>7</sup>.

<sup>6</sup><https://github.com/smartshark/icb>

<sup>7</sup><https://doi.org/10.5281/zenodo.3994254>

## 6.1 Data

We use four data sets to conduct our experiments. Table 1 lists statistics about the data sets. First, we use the data by Herzig et al. (2013). This data contains manually validated data for 7,297 issues from five projects. Three projects used Jira as issue tracker (`httpcomponents-client`, `jackrabbit`, `lucene-solr`), the other two used Bugzilla as issue tracker (`rhino`, `tomcat`). The data shared by Herzig et al. (2013) only contains the issue IDs and the correct labels. We collected the issue titles, descriptions, and discussions for these issues with SmartSHARK (Trautsch et al. 2018; Trautsch et al. 2020). The primary purpose of the data by (Herzig et al. 2013) in our experiments is the use as test data. Therefore, we refer to this data set in the following as TEST.

Second, we use the data by Herbold et al. (2020). This data contains manually validated data for all 11,154 bugs of 38 projects. Issues that are not bugs were not manually validated. However, Herbold et al. (2020) confirmed the result by Herzig et al. (2013) using sampling that only about 1% of issues that are not labeled as bugs are actually bugs. Consequently, Herbold et al. (2020) decided to ignore this small amount of noise, which we also do in this article, i.e., we assume that everything that is not labeled as bug in the data by Herbold et al. (2020) is not a bug. The primary purpose of the data by Herbold et al. (2020) in our experiments is the use in a leave-one-project-out cross-validation experiment. Therefore, we refer to this data as CV in the following.

The third data set was collected by Ortu et al. (2015). This data set contains 701,002 Jira issues of 1,238 projects. However, no manual validation of the issue types is available for the data by Ortu et al. (2015). We drop all issues that have no description and all issues of projects that are also included in the data by Herzig et al. (2013) or Herbold et al. (2020). This leaves us with 607,636 issues of 1,198 projects. Since we use this data to evaluate the impact of not validating data, we refer to this data as UNVALIDATED in the following.

We use two variants of the data by Herzig et al. (2013) and Herbold et al. (2020): 1) only the issues that were labelled as bug in the issue tracker; and 2) all issues regardless of their type. Our rationale for this are the different possible use cases for issue type prediction, we outlined in Section 2. Using these different sets, we evaluate how good issue type prediction works in different circumstances. With the first variant, we evaluate how good the issue type prediction models work for the correction of mislabeled bugs either as recommendation system or by researchers. With the second variant we evaluate how good the models are as general recommendation systems. We refer to these variants as  $TEST_{BUG}$ ,  $CV_{BUG}$ ,  $TEST_{ALL}$ , and  $CV_{ALL}$ . We note that such a distinction is only possible with data that was manually validated, hence, there is no such distinction for the UNVALIDATED data.

We also use a combination of the UNVALIDATED and the CV data. The latest issue in the UNVALIDATED data was reported on 2014-01-06. We extend this data with all issues from the CV data that were reported prior to this date. We use the original labels from the Jira instead of the manually validated labels from Herbold et al. (2020), i.e., an unvalidated version of this data that is cut off at the same time as the UNVALIDATED data. We refer to this data as UNVALIDATED+CV. Similarly, we use a subset of the  $CV_{ALL}$  data, that only consists of the issues that were reported after 2014-01-06. Since we will use this data for evaluation, we use the manually validated labels by (Herbold et al. 2020). We drop the commons-digester project from this data, because only nine issues were reported after 2014-01-06, none of which were bugs. We refer to this data as  $CV_{2014+}$ .

**Table 1** Statistics about the data we used, i.e., the number of issues in the projects (All), the number of issues that developers labeled as bug (Dev. Bugs), the number of issues that are validated as bugs (Val. Bugs), the number of bugfixing commits without issue type validation (No Val.) and the number of bugfixing commits with issue type validation (Val.)

	Issues			Bugfixing Commits	
	All	Dev. Bugs	Val. Bugs	No Val.	Val.
httpcomponents-client	744	468	304	–	–
jackrabbit	2344	1198	925	–	–
lucene-solr	2399	1023	688	–	–
rhino	584	500	302	–	–
tomcat	1226	1077	672	–	–
TEST Total	7297	4266	2891	–	–
ant-ivy	1168	544	425	708	568
archiva	1121	504	296	940	543
calcite	1432	830	393	923	427
cayenne	1714	543	379	1272	850
commons-bcel	127	58	36	85	49
commons-beanutils	276	88	51	118	59
commons-codec	183	67	32	137	59
commons-collections	425	122	49	180	88
commons-compress	376	182	124	291	206
commons-configuration	482	193	139	340	243
commons-dbcp	296	131	71	191	106
commons-digester	97	26	17	38	26
commons-io	428	133	75	216	129
commons-jcs	133	72	53	104	72
commons-jexl	233	87	58	239	161
commons-lang	1074	342	159	521	242
commons-math	1170	430	242	721	396
commons-net	377	183	135	235	176
commons-scxml	234	71	47	123	67
commons-validator	265	78	59	101	73
commons-vfs	414	161	92	195	113
deltaspikes	915	279	134	490	217
eagle	851	230	125	248	130
giraph	955	318	129	360	141
gora	472	112	56	208	99
jspwiki	682	288	180	370	233
knox	1125	532	214	860	348
kylin	2022	698	464	1971	1264
lens	945	332	192	497	276
mahout	1669	499	241	710	328
manifoldcf	1396	641	310	1340	671

**Table 1** (continued)

	Issues			Bugfixing Commits	
	All	Dev. Bugs	Val. Bugs	No Val.	Val.
nutch	2001	641	356	976	549
opennlp	1015	208	102	353	144
parquet-mr	746	176	81	241	120
santuario-java	203	85	52	144	95
systemml	1452	395	241	583	304
tika	1915	633	370	1118	670
wss4j	533	242	154	392	244
CV Total	30922	11154	6333	18539	10486
UNVALIDATED Total	607636	346621	—	—	—

The statistics for the BUGFIXES data set are shown in the last two columns of the CV data

Finally, we also use data about validated bugfixing commits. The data we are using also comes from Herbold et al. (2020), who in addition to the validation of issue types also validated the links between commits and issues. They found that the main source of mislabels for bug fixing commits are mislabeled issue types, i.e., bugs that are not actually bugs. We use the validated links and validated bug fix labels from Herbold et al. (2020). Since the projects are the same as for the CV data, we list the data about the number of bug fixing commits per project in Table 1 together with the CV data, but refer to this data in the following as BUGFIXES.

## 6.2 Baselines

Within our experiments, we not only evaluate our own approach which we discussed in Section 4, but also compare our work to several baselines. First, we use a trivial baseline which assumes that all issues are bugs. Second, we use the approaches from the literature as comparison. We implemented the approaches as they were described and refer to them by the family name of the first author, year of publication, and acronym for the classifier. The approaches from the literature we consider are (in alphabetical order) Kallis2019-FT by Kallis et al. (2019), Palacio-2019-SRN by Palacio et al. (2019), Pandey2018-LR and Pandey2018-NB by Pandey et al. (2018), Qin2018-LSTM by Qin and Sun (2018), Otoom2019-SVC, Otoom2019-NB, and Otoom2019-RF by Otoom et al. (2019), Pingclasai2013-LR and Pingclasai2013-NB by Pingclasai et al. (2013), Limsettho2014-LR and Limsettho2014-NB by Limsettho et al. (2014a), and Terdchanakul2017-LR and Terdchanakul2017-RF by Terdchanakul et al. (2017).

We note that this is, unfortunately, a subset of the related work discussed in Section 3. We omitted all unsupervised approaches, because they require manual interaction to determine the issue type for the determined clusters. The other supervised approaches were omitted due to different reasons. Antoniol et al. (2008) perform feature selection based on a TFM by pair-wise comparisons of all features. In comparison to Antoniol et al. (2008), we used data sets with more issues which increased the number of distinct terms in the TFM. As a result, the quadratic growth of the runtime complexity required for the proposed feature

selection did not terminate, even after waiting several days. Zhou et al. (2016) could not be used, because their approach is based on different assumptions on the training data, i.e., that issues are manually classified using only the title, but with different certainties. This data can only be generated by manual validation and is not available in any of the data sets we use. Zolkeply and Shao (2019) could not be replicated because the authors do not state which 60 keywords they used in their approach.

### 6.3 Performance Metrics

We take pattern from the literature (e.g. Antoniol et al. 2008, Chawla and Singh 2015, Terdchanakul et al. 2017, Pandey et al. 2018, Qin and Sun 2018, Kallis et al. 2019) and base our evaluation on the *recall*, *precision*, and *F1 score*, which are defined as

$$\begin{aligned} recall &= \frac{tp}{tp + fn} \\ precision &= \frac{tp}{tp + fp} \\ F1\ score &= 2 \cdot \frac{recall \cdot precision}{recall + precision} \end{aligned}$$

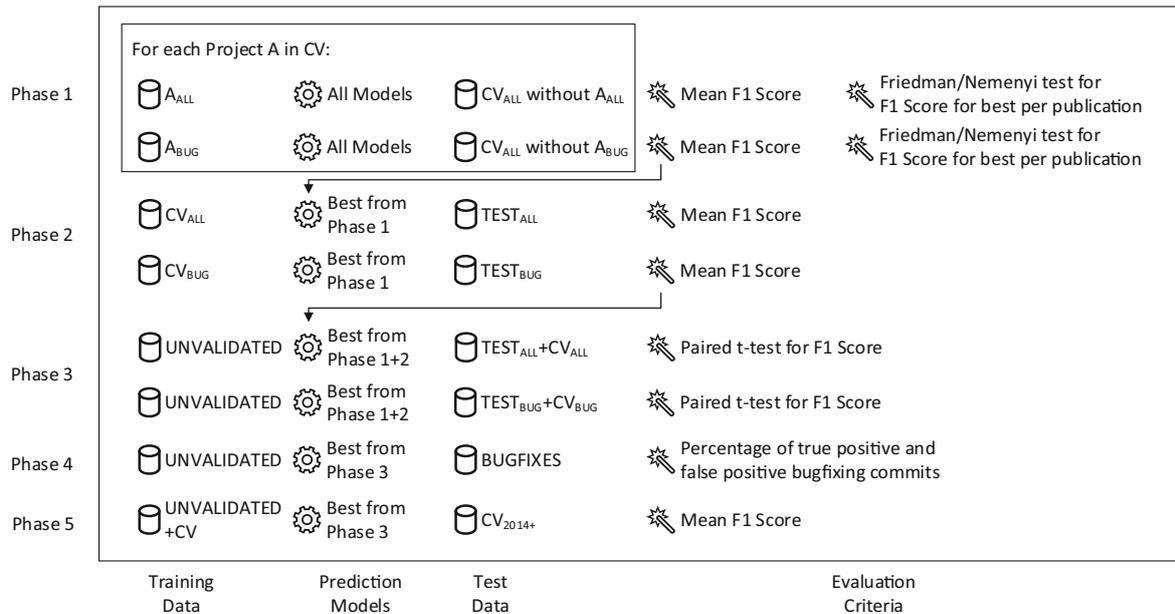
where *tp* are true positives, i.e., bugs that are classified as bugs, *tn* true negatives, i.e., non bugs classified as non bugs, *fp* bugs not classified as bugs and *fn* non bugs classified as bugs. The *recall* measures the percentage of bugs that are correctly identified as bugs. Thus, a high *recall* means that the model correctly finds most bugs. The *precision* measures the percentage of bugs among all predictions of bugs. Thus, a high *precision* means that there is strong likelihood that issues that are predicted as bugs are actually bugs. The *F1 score* is the harmonic mean of *recall* and *precision*. Thus, a high *F1 score* means that the model is good at both predicting all bugs correctly and at not polluting the predicted bugs with too many other issues.

### 6.4 Methodology

Figure 7 summarizes our general methodology for the experiments, which consists of four phases. In Phase 1, we conduct a leave-one-project-out cross validation experiment with the CV data. This means that we use each project once as test data and train with all other projects. We determine the *recall*, *precision*, and *F1 score* for all ten models we propose in Section 4.4 as well as all baselines this way for both the  $CV_{ALL}$  and the  $CV_{BUG}$  data. In case there are multiple variants, e.g., our ten approaches or different classifiers proposed for a baseline, we select the one that has the best overall mean value on the  $CV_{ALL}$  and the  $CV_{BUG}$  data combined. This way, we get a single model for each baseline, as well as for our approach, that we determined works best on the CV data. We then follow the guidelines from Demšar (2006) for the comparison of multiple classifiers. Since the data is almost always normal, except for trivial models that almost always yield 0 as performance value, we report the mean value, standard deviation, and the confidence interval of the mean value of the results. The confidence interval with a confidence level of  $\alpha$  for normally distributed samples is calculated as

$$mean \pm \frac{sd}{\sqrt{n}} Z_{\alpha} \quad (3)$$

where the *sd* is the standard deviation, *n* the sample size, and  $Z_{\alpha}$  the  $\frac{\alpha}{2}$  percentile of the t-distribution with  $n - 1$  degrees of freedom. However, the variances are not equal, i.e., the



**Fig. 7** Overview of the experiment methodology. The training and evaluation in all phases is conducted with all issues and with only bug issues

assumption of homoscedacity is not fulfilled. Therefore, we use the Friedman test (Friedman 1940) with the post-hoc Nemenyi test Nemenyi (1963) to evaluate significant differences between the issue prediction approaches. The Friedman test is an omnibus test that determines if there is any difference in the central tendency of a group of paired samples with equal group sizes. If the outcome of the Friedman test is significant, the Nemenyi test evaluates which differences between approaches are significant based on the critical distance, which is defined as

$$CD = \sqrt{\frac{k(k+1)}{12N}} q_{\alpha, N} \quad (4)$$

where  $k$  is the number of approaches that are compared,  $N$  is the number of distinct values for each approach, i.e., in our case the number of projects in a data set, and  $q_{\alpha, N}$  is the  $\alpha$  percentile of the studentized range distribution for  $N$  groups and infinite degrees of freedom. Two approaches are significantly different, if the difference in the mean ranking between the performance of the approaches is greater than the critical distance. We use Cohen's  $d$  Cohen (1988) which is defined as

$$d = \frac{mean_1 - mean_2}{\sqrt{\frac{sd_1 + sd_2}{2}}} \quad (5)$$

to report the effect sizes in comparison to the best performing approach. Table 2 shows the magnitude of the effect sizes for Cohen's  $d$ . We will use the results from the first phase to evaluate RQ1, i.e., to see if our rules improved the issue type prediction. Moreover, the performance values will be used as indicators for RQ3.

In Phase 2, we use the CV data as training to train a single model for the best performing approaches from Phase 1. This classifier is then applied to the TEST data. We report the mean value and standard deviation of the results. However, we do not conduct any statistical tests, because there are only five projects in the TEST data, which is insufficient for a statistical analysis. Through the results of Phase 2 we will try to confirm if the results from Phase 1 hold on unseen data. Moreover, we get insights into the reliability of the manually validated data, since different teams of researchers validated the CV and the TEST data. In

**Table 2** Magnitude of effect sizes of Cohen's  $d$ 

$d$	Magnitude
$d < 0.2$	Negligible
$0.2 \leq d < 0.5$	Small
$0.5 \leq d < 0.8$	Medium
$0.8 \leq d$	Large

case the performance is stable, we have a good indication that our estimated performance from Phase 1 generalizes. This is especially important, because Phase 2 is biased in favor of the state of the art, while Phase 1 is biased in favor of our approach. The reason for this is that most approaches from the state of the art were developed and tuned on the TEST data, while our approach was developed and tuned on the CV data. Therefore, the evaluation on the TEST data also serves as counter evaluation to ensure that the results are not biased due to the data that was used for the development and tuning of an approach, as stable results across the data sets would indicate that this is not the case. Thus, the results from Phase 2 are used to further evaluate RQ3 and to increase the validity of our results.

In Phase 3, we evaluate the use of unvalidated data for the training, i.e., data about issues where the labels were not manually validated by researchers. For this, we compare the results of the best approach from Phase 1 and Phase 2 with the same approach, but trained with the UNVALIDATED data. Through this, we analyze RQ2 to see if we really require validated data or if unvalidated data works as well. Because the data is normal, we use the paired t-test to test if the differences between results in the *F1 score* are significant and Cohen's  $d$  to calculate the effect size. Moreover, we consider how the *recall* and *precision* are affected by the unvalidated data, to better understand how training with the UNVALIDATED data affects the results.

We apply the approach we deem best suited in Phase 4 to a relevant problem of the Scenario 2 for issue type prediction discussed in Section 2. Concretely, we analyze if issue type prediction can be used to improve the identification of bugfixing commits. Herbold et al. (2020) found that mislabelled issues are the main source for the wrong identification of bugfixing commits. Therefore, an accurate issue type prediction model would be very helpful for any software repository mining tasks that relies on bugfixing commits. To evaluate the impact of the issue type prediction on the identification of bug fixing commits, we use two metrics. First, the percentage of actual bug fixing commits, that are found if issue type prediction is used (true positives). This is basically the same as the *recall* of bugfixing commits. Second, the percentage of additional bugfixing commits that are found in additionally in relation to the actual number of bug fixing commits. This is indirectly related to the *precision*, because such additional commits are the result of false positive prediction.

Finally, we apply the best approach in a setting that could be Scenario 3 or the Scenario 4 outlined in Section 2. An important aspect we ignored so far is the potential information leakage because we did not consider the time when issues were reported. In a realistic scenario where we apply the prediction live within an issue tracking system, data from the future is not available. Instead, only past issues may be used to train the model. For this scenario, we decide on a fixed cutoff date for the training data. All data prior to this cutoff is used for training, all data afterwards for testing of the prediction performance. This is realistic, because such models are often trained at some point and the trained model is then implemented in the live system and must be actively updated as a maintenance task. We use the UNVALIDATED+CV data for training and the CV<sub>2014+</sub> for testing in this phase. We

compare the results with the performance we measured in Phase 3 of the experiments, to understand how this realistic setting affects our performance estimations in comparison to the less realistic results that ignore the potential information leakage because of overlaps in time between the training and test data.

For our experiments, we conduct many statistical tests. We use Bonferroni correction (Dunn 1961) to account for false positive due to the repeated tests and have an overall significance level of 0.05, resp. confidence level of 0.95 for the confidence intervals. We use a significance level of  $\frac{0.05}{22} = 0.0023$  for the Shapiro-Wilk tests for normality (Shapiro and Wilk 1965), because we perform nine tests for normality for the best performing approaches for both all issues and only bugs in both Phase 1 and four additional tests for normality in Phase 3. We conduct two Bartlett tests for homoscedacity (Bartlett 1937) in Phase 1 with a significance level of  $\frac{0.05}{2} = 0.025$ . We conduct four tests for the significance of differences between classifiers with a significance level of  $\frac{0.05}{4}$ , i.e., two Friedman tests in Phase 1 and two paired t-tests in Phase 3. Moreover, we calculate the confidence intervals for all results in Phase 1 and Phase 3, i.e., 25 results for both  $CV_{ALL}$  and  $CV_{BUG}$  and four results for Phase 3. Hence, we use a confidence level of  $1 - \frac{0.05}{54} = 0.999$  for these confidence intervals.

## 6.5 Results

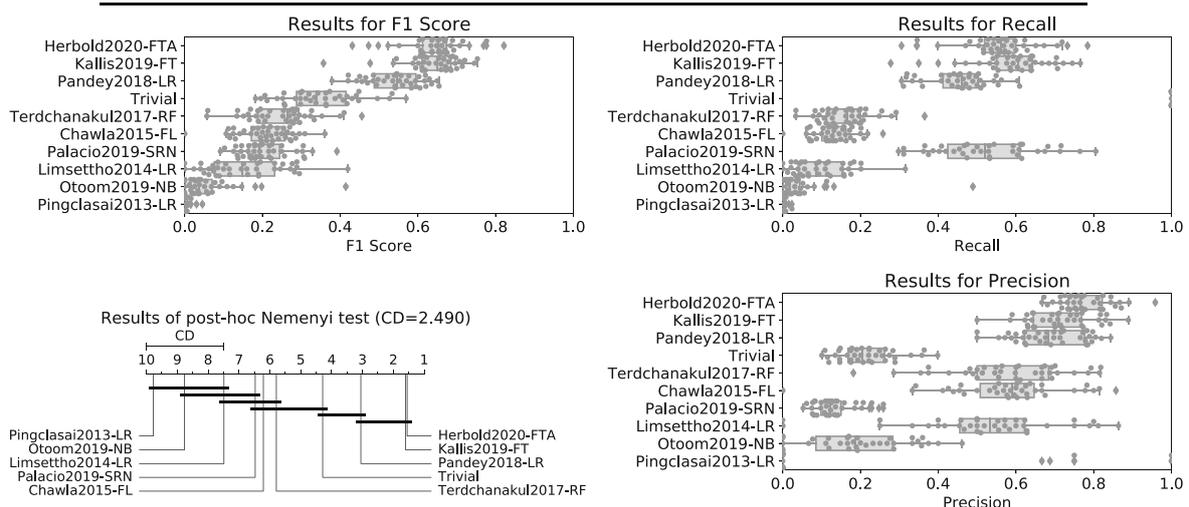
### 6.5.1 Results for Phase 1

Figures 8 and 9 show the the results for the first phase of the experiment on the  $CV_{ALL}$  and  $CV_{BUGS}$  data, respectively. We observe that while there is a strong variance in the *F1 score* using the  $CV_{ALL}$  data with values between 0.0 (Limsettho2014-NB) and 0.643 (Herbold2020-FTA), the results on the  $CV_{BUG}$  data are more stable with values between 0.610 (Terdchanakul2017-RF) and 0.809 (Herbold2020-RF). The strong performance on  $CV_{BUG}$  includes the Trival approach, i.e., simply predicted everything as bug is already relatively hard to beat, because the *recall* is perfect and the *precision* is at about 60%.<sup>8</sup> This finding is different for the  $CV_{ALL}$ , because here the class level imbalance is the other way around and the *precision* drops to roughly 20%. In general, we observe that the *F1 score* with the  $CV_{ALL}$  data is lower than with  $CV_{BUG}$ .

This also shows in the results for the different variants we proposed in Section 4.4, were we observe big difference with the  $CV_{ALL}$  data and only relatively small differences with the  $CV_{BUG}$  data. On the  $CV_{ALL}$  data, the strongest driver of the differences is the choice of the classifier. The models using FTA perform best, followed by FT classifier which has a slightly worse performance. The drop between FT and RF is steep, NB performs worse and predicts only few bugs. Using distinct classifiers for the title and description improves the performance slightly. However, additional classifiers for null pointers lead to slightly worse results. Thus, we find that Herbold2020-FTA with separate classifiers for title and descriptions performs best for  $CV_{ALL}$ , even though the difference to the other variants with FTA/FT is small. On the  $CV_{BUG}$ , the *F1 score* of all models that use different separate classifiers for title and description is within the interval [0.790, 0.809].

<sup>8</sup>60% is roughly the amount of bugs within the data

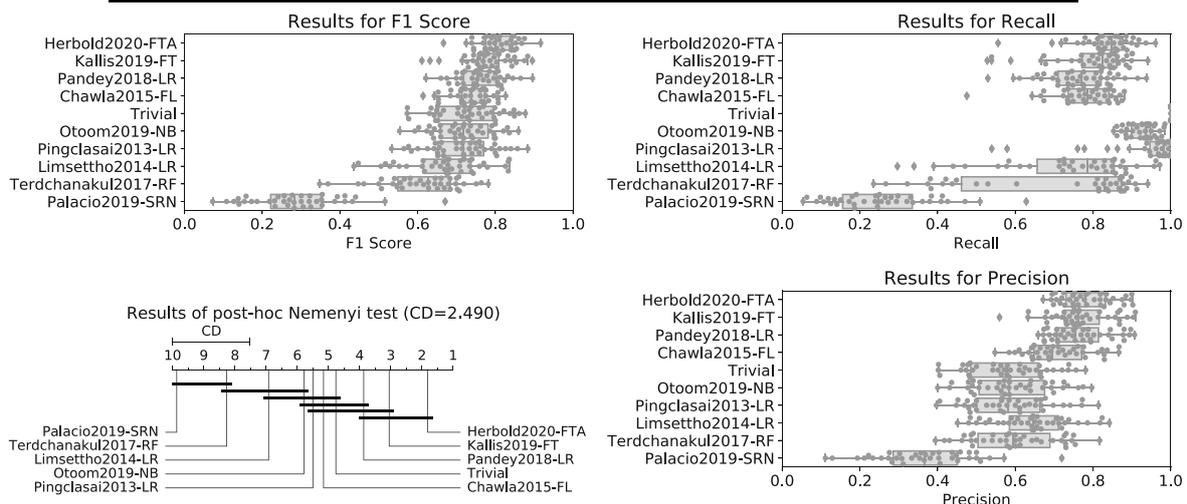
	mean	sd	CI	<i>d</i>
<b>Herbold2020-FTA</b>	<b>0.643</b>	<b>0.077</b>	<b>[0.598, 0.689]</b>	<b>0.000</b>
Herbold2020-FTA+NPE	0.639	0.073	[0.596, 0.681]	0.064
Herbold2020-FT	0.627	0.072	[0.585, 0.669]	0.218
Herbold2020-FT+NPE	0.619	0.072	[0.577, 0.661]	0.326
Herbold2020-RF	0.313	0.092	[0.260, 0.367]	3.896
Herbold2020-RF+NPE	0.307	0.077	[0.262, 0.352]	4.356
Herbold2020-Basic-RF	0.222	0.104	[0.161, 0.283]	4.606
Herbold2020-NB+NPE	0.169	0.060	[0.134, 0.204]	6.861
Herbold2020-Basic-NB	0.017	0.029	[0.000, 0.033]	10.759
Herbold2020-NB	0.015	0.018	[0.005, 0.026]	11.205
<b>Kallis2019-FT</b>	<b>0.638</b>	<b>0.074</b>	<b>[0.594, 0.681]</b>	<b>0.073</b>
<b>Pandey2018-LR</b>	<b>0.541</b>	<b>0.070</b>	<b>[0.500, 0.582]</b>	<b>1.387</b>
Pandey2018-NB	0.503	0.111	[0.439, 0.568]	1.469
Qin2018-LSTM	0.349	0.096	[0.293, 0.405]	3.370
<b>Trivial</b>	<b>0.349</b>	<b>0.096</b>	<b>[0.293, 0.405]</b>	<b>3.370</b>
<b>Chawla2015-FL</b>	<b>0.208</b>	<b>0.070</b>	<b>[0.168, 0.249]</b>	<b>5.913</b>
<b>Palacio2019-SRN</b>	<b>0.206</b>	<b>0.068</b>	<b>[0.166, 0.246]</b>	<b>6.017</b>
<b>Limsettho2014-LR</b>	<b>0.154</b>	<b>0.100</b>	<b>[0.096, 0.212]</b>	<b>5.490</b>
Limsettho2014-NB	0.000	0.001	[0.000, 0.001]	11.777
<b>Terdchanakul2017-RF</b>	<b>0.248</b>	<b>0.088</b>	<b>[0.196, 0.299]</b>	<b>4.775</b>
Terdchanakul2017-LR	0.076	0.068	[0.037, 0.116]	7.796
<b>Otoom2019-NB</b>	<b>0.059</b>	<b>0.077</b>	<b>[0.013, 0.104]</b>	<b>7.561</b>
Otoom2019-RF	0.010	0.024	[0.000, 0.024]	11.070
Otoom2019-SVC	0.003	0.010	[0.000, 0.008]	11.638
<b>Pingclasai2013-LR</b>	<b>0.003</b>	<b>0.009</b>	<b>[0.000, 0.009]</b>	<b>11.636</b>
Pingclasai2013-NB	0.004	0.011	[0.000, 0.010]	11.599



**Fig. 8** Results of leave-one-project-out cross validation with the  $CV_{ALL}$  data. The bold-faced approaches where the best for a publication and are used in the statistical analysis

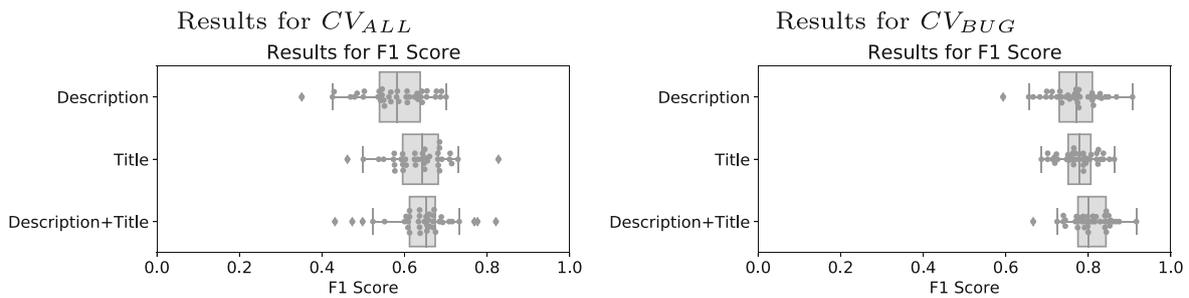
Thus, we find that overall, Herbold2020-FTA performs best among the approaches discussed in Section 4.4. Figure 10 allows us to gain further insights and to understand how the approach achieves the performance. The figure shows the performance of the two predictors

	mean	sd	CI	<i>d</i>
Herbold2020-RF	0.809	0.051	[0.779, 0.839]	-0.082
<b>Herbold2020-FTA</b>	<b>0.805</b>	<b>0.049</b>	<b>[0.776, 0.834]</b>	<b>0.000</b>
Herbold2020-FT	0.803	0.050	[0.774, 0.832]	0.041
Herbold2020-RF+NPE	0.803	0.046	[0.776, 0.830]	0.044
Herbold2020-FT+NPE	0.801	0.046	[0.774, 0.828]	0.088
Herbold2020-FTA+NPE	0.802	0.048	[0.774, 0.830]	0.057
Herbold2020-NB+NPE	0.799	0.067	[0.760, 0.839]	0.098
Herbold2020-NB	0.790	0.066	[0.751, 0.829]	0.252
Herbold2020-Basic-NB	0.767	0.083	[0.718, 0.816]	0.563
Herbold2020-Basic-RF	0.776	0.056	[0.743, 0.809]	0.546
<b>Kallis2019-FT</b>	<b>0.777</b>	<b>0.062</b>	<b>[0.740, 0.814]</b>	<b>0.503</b>
<b>Pandey2018-LR</b>	<b>0.760</b>	<b>0.058</b>	<b>[0.726, 0.794]</b>	<b>0.839</b>
Pandey2018-NB	0.765	0.069	[0.725, 0.806]	0.667
<b>Pandey2018-NB</b>	<b>0.765</b>	<b>0.069</b>	<b>[0.725, 0.806]</b>	<b>0.667</b>
Qin2018-LSTM	0.729	0.083	[0.681, 0.778]	1.115
<b>Trivial</b>	<b>0.729</b>	<b>0.083</b>	<b>[0.681, 0.778]</b>	<b>1.115</b>
<b>Otoom2019-SVC</b>	<b>0.722</b>	<b>0.074</b>	<b>[0.678, 0.765]</b>	<b>1.324</b>
Otoom2019-NB	0.718	0.075	[0.674, 0.762]	1.382
Otoom2019-RF	0.718	0.071	[0.676, 0.760]	1.429
<b>Pingclasai2013-LR</b>	<b>0.715</b>	<b>0.085</b>	<b>[0.665, 0.765]</b>	<b>1.297</b>
Pingclasai2013-NB	0.671	0.131	[0.594, 0.748]	1.361
<b>Limsettho2014-LR</b>	<b>0.670</b>	<b>0.103</b>	<b>[0.610, 0.731]</b>	<b>1.671</b>
Limsettho2014-NB	0.734	0.078	[0.688, 0.780]	1.090
Terdchanakul2017-LR	0.728	0.082	[0.680, 0.777]	1.136
<b>Terdchanakul2017-RF</b>	<b>0.610</b>	<b>0.103</b>	<b>[0.550, 0.671]</b>	<b>2.425</b>
<b>Palacio2019-SRN</b>	<b>0.290</b>	<b>0.121</b>	<b>[0.218, 0.361]</b>	<b>5.600</b>



**Fig. 9** Results of leave-one-project-out cross validation with the  $CV_{BUG}$  data. The bold-faced approaches where the best for a publication and are used in the statistical analysis

that are internally used in comparison to the overall performance. We see that for both the  $CV_{ALL}$  and the  $CV_{BUG}$  data, that the performance of using only the title or description yields worse results than the combination of both classifiers. This indicates that the



**Fig. 10** Results for the different classifiers for title and description in comparison to their combination to gain insights into the Herbold2020-FTA model

differences between title and description we describe in Section 4.1 are indeed relevant. Moreover, because the combination of both classifiers outperforms each classifier on its own, this indicates that some issues that are hard to predict based on title can be predicted based on the description, and vice versa.

For the related work that suggested multiple classifiers, our observations are similar. The differences on the  $CV_{BUG}$  data are small, the performance on the  $CV_{ALL}$  data is always worse and, in many cases, very bad with only very few bugs predicted. The only outlier is Palacio2019-SRN, which performs better on the  $CV_{ALL}$  data than on the  $CV_{BUG}$ , but relatively bad on both data sets. We hypothesize that this is because there are fewer training issues available in  $CV_{BUG}$  and see this as an indicator that Palacio2019-SRN may perform better in future benchmarks, given that the amount of data is increased.

Of the approaches from the related work, only Kallis2019-FT and Pandey2018-LR achieve an *F1 score* of over 0.5 on  $CV_{ALL}$ . We also note that the approach Qin2018-LSTM degenerated into a trivial model that predicts everything as bug. Consequently, we exclude Qin2018-LSTM from our subsequent statistical analysis, because we would, otherwise, have the same model twice.

We reject the null hypothesis of the Friedman test that there are no significant differences between the approaches on the  $CV_{ALL}$  data ( $p - value < 0.001$ ). The post-hoc Nemenyi test found that while our Herbold2020-FTA approach has the best mean ranking, the difference is not significant in comparison to Kallis2019-FT and Pandey2018-LR. The difference in *F1 score* between our Herbold2020-FTA and Kallis2019-FT is almost non-existent. Thus, the advantage of automatically tuning the FT classifier and the usage of different classifiers for title and description is very small. The magnitude of the (non-significant) effect size between Herbold2020-FTA and Kallis2019-FT is negligible with  $d = 0.073$ . However, there is a difference between the *recall* and *precision* of Herbold2020-FTA and Kallis2019-FT. Herbold2020-FTA has a higher *precision*, while Kallis2019-FT has a higher *recall*. While not significant, the difference between Pandey2018-LR on the one hand and Herbold2020-FTA and Kallis2019-FT on the other hand is larger, both in the mean value and also in the (non-significant) effect size which has a large magnitude with  $d = 1.387$ . All other models from the state of the art are significantly different from Herbold2020-FTA and Kallis2019-FT with a large effect size of at least  $d = 3.370$ . We note that Pandey2018-LR is not significantly different from the trivial model. Moreover, we observe that all other approaches are all equal to or worse than trivially predicting everything as bug in the mean *F1 score*, because of a low *recall*. Pinglasai2013-LR, Otoom2019-NB, and Limsettho2014-LR are even significantly worse than the trivial model.

We reject the null hypothesis of the Friedman test that there are no significant differences between the approaches on the  $CV_{BUG}$  data ( $p - value < 0.001$ ). The post-hoc Nemenyi test found that our Herbold2020-FTA has the best best mean ranking, but the difference is not significant in comparison to Kallis2019-FT and Pandey2018-LR. However, in comparison to the  $CV_{ALL}$  data, there is some gap between Herbold2020-FTA and Kallis2019-FT both in the mean  $F1$  score and the magnitude of the effect size would be medium with  $d = 0.503$  if it were significant. Moreover, Herbold2020-FTA yields a slightly better performance in both *recall* and *precision*, i.e., there is no trade-off between Herbold2020-FTA and Kallis2019-FT between *recall* and *precision*. The gap between Herbold2020-FTA and Pandey2018-LR is similar to the gap on the  $CV_{ALL}$  of effect and the effect size would be large with  $d = 0.839$ . All other approaches from the state of the art are significantly worse than Herbold2020-FTA with a large effect size of at least  $d = 1.115$ . We note that Kallis2019-FT and Pandey2018-LR are not significantly different from the trivial model. Same as for the  $CV_{BUG}$  data, we find that all other approaches have a worse mean ranking than the trivial approach, even though Chawla2015-FL has a slightly higher mean value than the trivial model. However, only Terdchanakul2017-RF is significantly worse than the trivial model.

When we consider the results on  $CV_{BUG}$  and  $CV_{ALL}$  together, the approaches Herbold2020-FTA, Kallis2019-FT and Pandey2018-LR are consistently ranked first, with their mean ranks in that order. The differences between these three approaches are not significant. However, we note that there is a considerable gap in the mean ranks reported on the CD diagrams between Herbold2020-FTA and Kallis2019-FT on the hand, and Pandey2018-LR on the other hand. This gap also shows in the mean  $F1$  score, which is lower on both the  $CV_{ALL}$  and  $CV_{BUG}$  data. In fact, Pandey2018-LR almost always ranks worse than both Herbold2020-FTA and Kallis2019-FT. Thus, we believe that the reason that Pandey2018-LR is not significantly different from Herbold2020-FTA and Kallis2019-FT is our limited sample size of 38 projects, which leads to a critical distance of 2.216 for the Nemenyi test. Thus, even if Herbold2020-FTA would always have the best score, Kallis2019-FT would always have the second best score, and Pandey2018-LR would always have the third best score, the difference would still not be significant. The reason for this is that we have not enough data, given that we are not just comparing these three populations, but a total of nine approaches at the same time. Therefore, we predict that with more data Herbold2020-FTA and Kallis2019-FT would significantly outperform Pandey2018-LR, but do not have the data to substantiate our claim.

Given this prediction, we conclude from Phase 1 that Facebook AI Research did a very good job on the design of FT that outperforms other classification algorithms, but that the automated tuning may not always yield better results, e.g., because there is not enough data. Similarly, different classifiers for title and description may improve the results of FT, but not significantly.

We also note that approaches that use NB as classifier perform a lot worse on the  $CV_{ALL}$  data than on the  $CV_{BUG}$  data. We believe this may be because the models are not actually learning a good scoring function, but rather relatively randomly guessing the number of bugs based on the a-priori probability of the class in the training data. This should work reasonably well in the  $CV_{BUG}$  data because of the class level imbalance in favor of bug issues, but should fail in case of the  $CV_{ALL}$  data because the chance of correctly hitting bugs when roughly 20% of the issues are randomly predicted is relatively low.

### 6.5.2 Results of Phase 2

Figures 11 and 12 show the results for the second phase of the experiment, i.e., training with the  $CV_{ALL}$ , resp.  $CV_{BUG}$  data and testing with the  $TEST_{ALL}$ , resp.  $TEST_{BUG}$  data. The results are consistent with our results from Phase 1. Herbold2020-FTA and Kallis2019-FT perform best on both data sets, there is almost no difference on the  $TEST_{ALL}$  data and a slightly better  $F1$  score for Herbold2020-FTA on the  $TEST_{BUG}$  data. The mean  $F1$  score of both approaches is within the confidence interval of the results from Phase 1. In general, we find that even though we just have five projects in the TEST data and the data was independently labelled from the CV data, that the  $F1$  score of 12 out of 18 approaches is within the CI, in the other six cases (three on  $TEST_{ALL}$  and  $TEST_{CV}$  each) the values on the TEST data are only slightly higher than the upper bound of the confidence interval on the CV data. This slight improvement of approaches on the TEST data can be explained by the fact that these approaches were developed on the TEST data, which should give them an advantage in comparison to the CV data. Overall, these results are a strong indicator that our findings from Phase 1 generalize to a broader population of projects.

### 6.5.3 Results of Phase 3

Figures 13 and 14 show the results of the training with the UNVALIDATED data in comparison to the results of Phase 1 and Phase 2 combined. The training with UNVALIDATED

	http-comp.	jack-rabbit	lucene-solr	rhino	tomcat	mean	std
Herbold2020-FTA	0.692	0.732	0.634	0.705	0.641	0.681	0.038
Kallis2019-FT	0.706	0.707	0.620	0.719	0.633	0.677	0.042
Pandey2018-LR	0.630	0.610	0.574	0.582	0.594	0.598	0.020
Qin2018-LSTM	0.580	0.566	0.446	0.682	0.708	0.596	0.093
Palacio2019-SRN	0.385	0.366	0.340	0.406	0.520	0.404	0.062
Terdchanakul2017-RF	0.203	0.285	0.252	0.246	0.321	0.261	0.040
Chawla2015-FL	0.276	0.238	0.266	0.179	0.186	0.229	0.040
Limsettho2014-LR	0.192	0.278	0.045	0.202	0.384	0.221	0.111
Otoom2019-NB	0.037	0.064	0.016	0.026	0.092	0.047	0.028
Pingclasai2013-LR	0.000	0.000	0.000	0.000	0.000	0.000	0.000

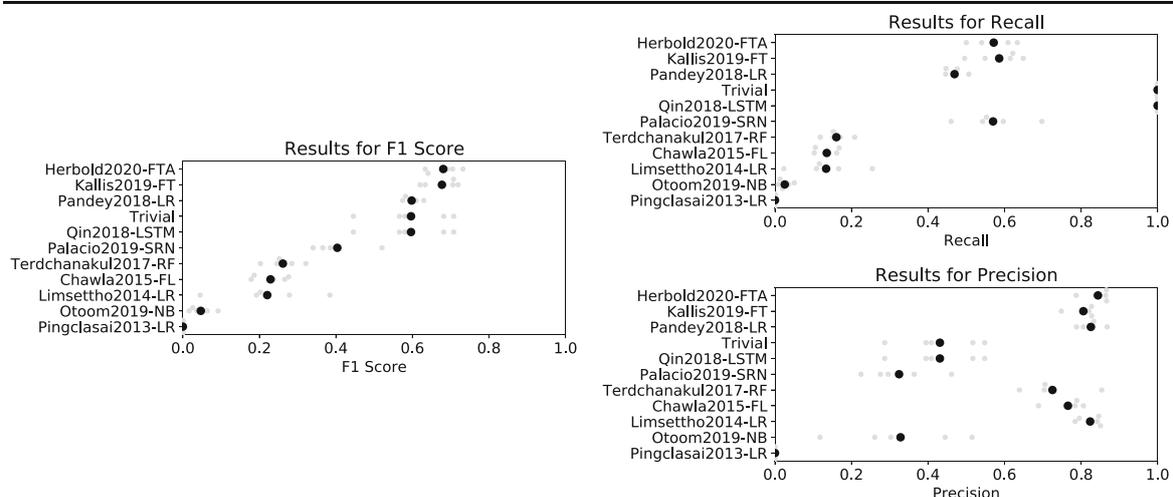


Fig. 11 Results of training with the  $CV_{ALL}$  and testing with the  $TEST_{ALL}$  data

	http-comp.	jack-rabbit	lucene-solr	rhino	tomcat	mean	std
Herbold2020-FTA	0.832	0.874	0.777	0.813	0.783	0.816	0.035
Kallis2019-FT	0.821	0.841	0.767	0.803	0.783	0.803	0.026
Qin2018-LSTM	0.776	0.857	0.791	0.748	0.760	0.786	0.038
Pandey2018-LR	0.806	0.826	0.797	0.760	0.743	0.786	0.031
Otoom2019-NB	0.776	0.826	0.789	0.764	0.751	0.781	0.025
Chawla2015-FL	0.777	0.801	0.756	0.712	0.726	0.755	0.033
Pingclasai2013-LR	0.778	0.848	0.777	0.742	0.602	0.750	0.081
Limsettho2014-LR	0.765	0.774	0.691	0.732	0.759	0.744	0.030
Terdchanakul2017-RF	0.561	0.546	0.442	0.690	0.717	0.591	0.101
Palacio2019-SRN	0.169	0.212	0.273	0.150	0.475	0.256	0.118

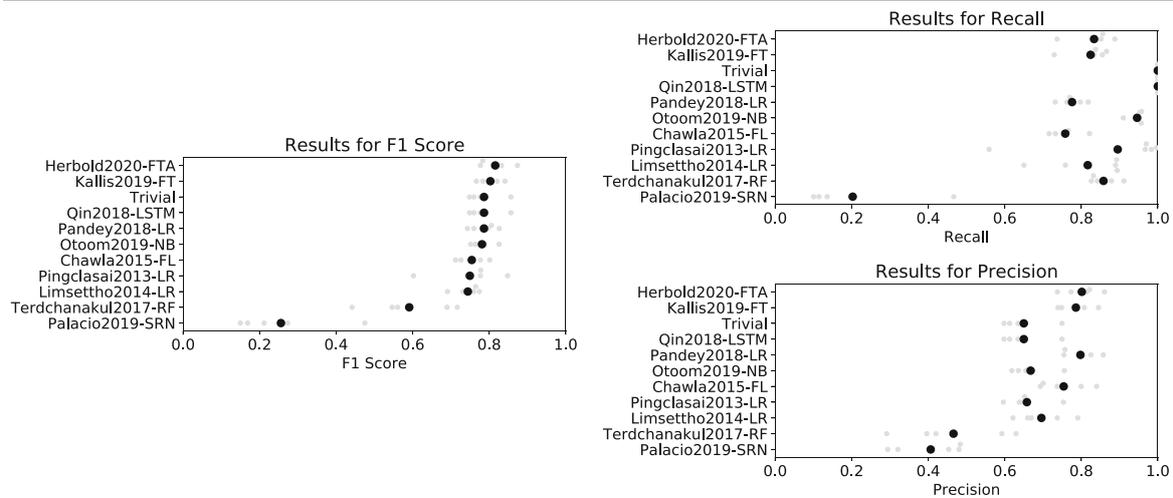


Fig. 12 Results of training with the  $CV_{BUG}$  and testing with the  $TEST_{BUG}$  data

data actually outperforms the training with validated data in case all issues are performed if we consider the mean  $F1$  score. However, we fail to reject the null hypothesis of the paired t-test that there is no difference ( $p - value = 0.385$ ). Therefore, we conclude that training without manual validation is not significantly different for the identification of bugs among all issues in terms of  $F1$  score. Regardless, the  $recall$  and  $precision$  show that while the  $F1$  score is not affected, the way this  $F1$  score is achieved is very different between validated

	mean	sd	CI	$d$
Herbold2020-FTA-UV	0.662	0.083	[0.615, 0.709]	0.000
Herbold2020-FTA	0.648	0.075	[0.606, 0.690]	0.179

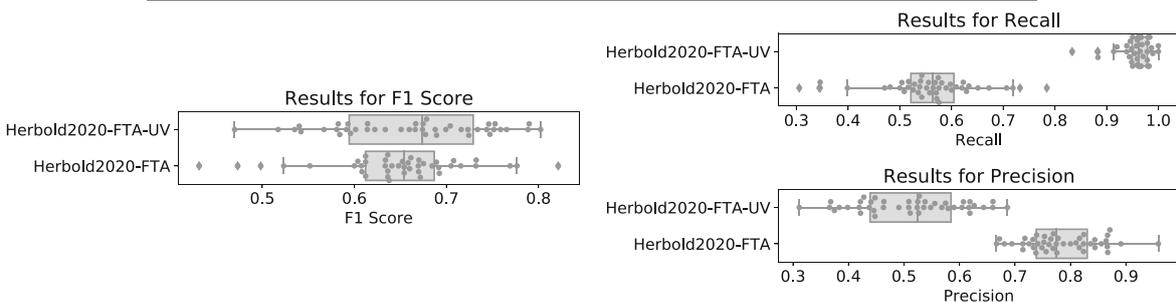
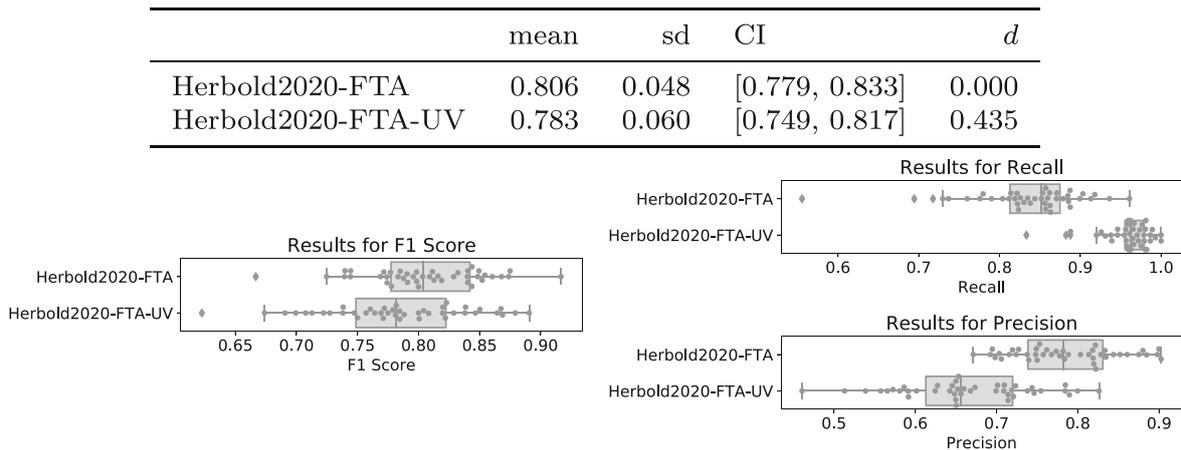


Fig. 13 Comparison of the results from Phase 1 and Phase 2 with training with the UNVALIDATED data and testing with  $CV_{ALL}$  and  $TEST_{ALL}$



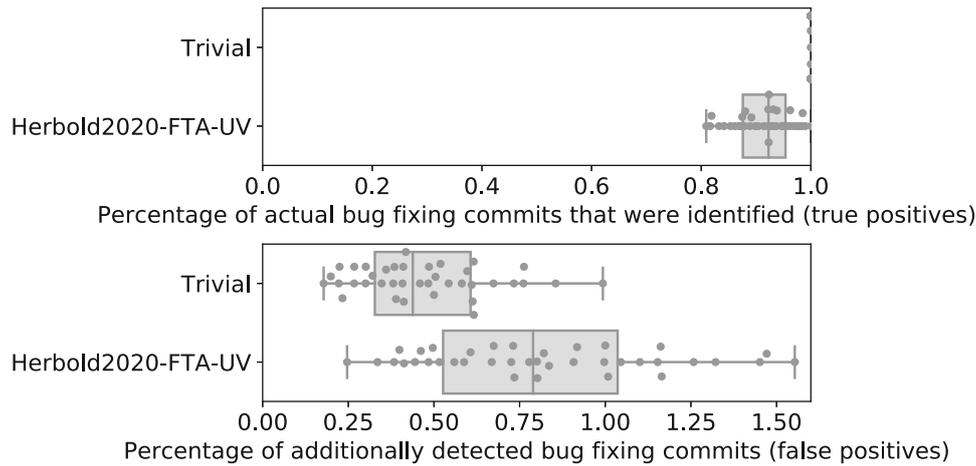
**Fig. 14** Comparison of the results from Phase 1 and Phase 2 with training with the UNVALIDATED data and testing with  $CV_{BUG}$  and  $TEST_{BUG}$

and unvalidated data. The Herbold2020-FTA-UV has a very large recall nearly always over 0.9, but a relatively low precision with values between 0.3 and 0.7. Thus, the strong *F1 score* is achieved by predicting nearly all bugs at the cost of a mediocre *precision*. With the validated data, this is the opposite. The *recall* is much lower and between 0.3 and 0.8, but the *precision* is higher with values between 0.65 and 1.0.

If we apply the classifier trained on the UNVALIDATED data to only the bugs, the *F1 score* is still similar, but slightly worse than training with validated data. We reject the null hypothesis of the paired t-test that there is no difference ( $p$  – value = 0.010) and find that the effect size of this difference is small ( $d = 0.435$ ). In terms of *recall* and *precision* the differences between training with and without validation are similar. However, the *precision* of training without validation is increased to values between 0.45 and 0.85 and the *recall* of training with validation is increased to 0.45 to 0.95. Overall, our results show that training with unvalidated data is an option, especially if *recall* is more important than *precision*.

#### 6.5.4 Results for Phase 4

For Phase 4, we decided to use the Herbold2020-FTA-UV approach, even though Herbold2020-FTA performed slightly better if only bugs are considered. Our reason for this is that we have a defect prediction use case in mind, where false negatives would mean that bugs are missed. Consequently, we value *recall* higher than *precision*, which means that Herbold2020-FTA-UV is preferable over Herbold2020-FTA. Figure 15 shows how the labeling of bugfixing commits is changed by using issue type prediction in comparison to trivially assuming that the developer classification is correct without validation. With the trivial approach, all actual bugfixing commits are identified. This is a property of the data, since the manual validation only reduces the amount of bug fixes. The issue type prediction with Herbold2020-FTA-UV finds on average 91.3% of the bugfixing commits, the worst case in our data is that only 80.9% of the bugfixing commits are identified. When we consider the false positive bugfixing commits, we see that the issue type prediction has a strong positive impact. The mean percentage of additional bugfixing commits is at 47%, in comparison to 81% with the trivial approach. Thus, the amount of additional bugfixing commits, that are actually, e.g., feature additions, is greatly reduced by using the issue type prediction. While the resulting data still contains mislabels, the amount of mislabels is reduced.

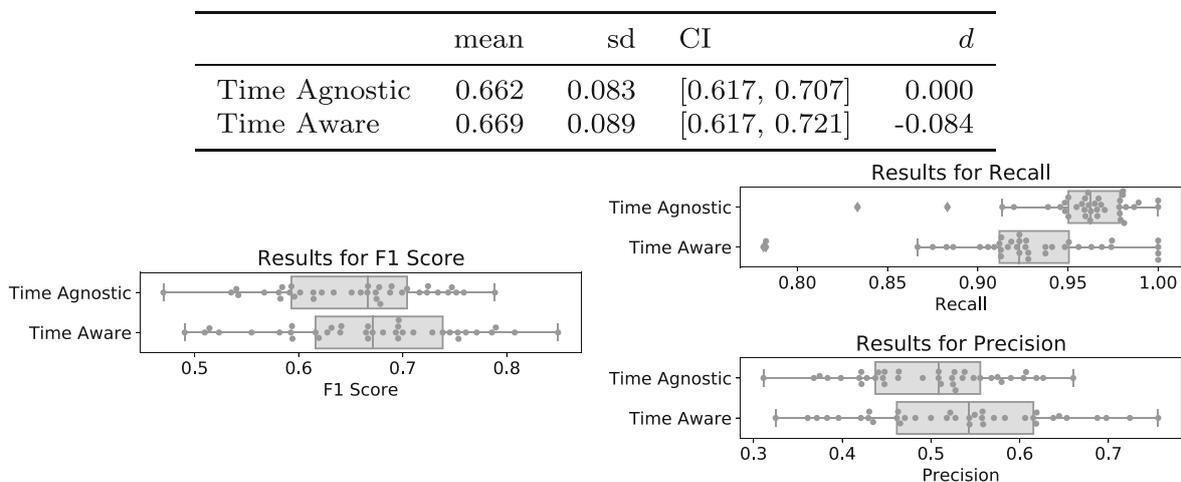


**Fig. 15** Impact on using Herbold2020-FTA-UV for the labeling of bugfixing commits

However, even with the reduction there is still much noise left, i.e., about one third of bugfixing commits the data would still be mislabels. Hence, for the use case of the creation of defect prediction data, issue type prediction could potentially replace manual validation for the creation of training data, but we suggest to rely on manual validation for the creation of test data.

### 6.5.5 Results for Phase 5

Same as for Phase 4, we decided to use the Herbold2020-FTA-UV approach, because this approach performs best when applied to all issues. We use the UNVALIDATED+CV data for training and the CV<sub>2014+</sub> for testing. This combination of training and test data is realistic for practical scenarios, because there is no temporal overlap between the training and test data. Moreover, the UNVALIDATED+CV data contains historic data from the test projects, that would be readily available without large effort. The advantage of adding this data for the training is that the classifier could possibly also learn project specific information.



**Fig. 16** Evaluation in a realistic time aware setting in comparison to the time agnostic results from Phase 3. The results from Phase 3 are restricted to the projects we evaluate in Phase 5, i.e., the projects from CV without commons-digester

Figure 16 shows the results of Phase 5. In general, the results are similar to the results of Phase 4. There is a small drop in *recall* and a slight increase in *precision* with the time awareness, but almost no difference in the F-measure. This is a strong indicator that our results from Phase 3 generalize and that this approach can also be reasonably be used for prediction within issue tracking system. We note that based on the data from Herzig et al. (2013) and Herbold et al. (2020) this prediction system would perform almost the same as developers: most bugs would be correctly labelled as bug (very high recall over 90%), but only about 55% of the issues labeled as bug would actually be bugs, which is about 5% worse than developers.

## 7 Discussion

We now discuss the answers to our research questions and how our results relate to findings from the literature.

### 7.1 Answers to Research Questions

#### 7.1.1 RQ1: Can manually specified logical rules derived from knowledge about issues be used to improve issue type classification?

On the one hand, our results show a small improvement in the performance of issue type prediction due to the use of two classifiers, i.e., because of the structural information about the difference between title and description that we incorporate in the learning process. On the other hand, the knowledge about null pointers we integrated to the learning process did not have a positive effect. For us, this indicates two things. First, if we can help the model to better understand the structure of the data, e.g., by accounting for the separate fields, this can have a positive effect, even though the improvement is likely small. Other aspects that could be considered here would be parsing of structural information that is available in the issue, e.g., HTML or Markdown syntax, to enable a better pre-processing of the data. Second, defining logical rules that mimic (parts of) the guidelines for labeling issues by Herzig et al. (2013) does not seem helpful. It seems like the classifiers can infer these rules, or at least similar rules themselves and the definition of hard coded rules may actually inhibit the learning process, because they either restrict the solution space or the available data unnecessarily. In summary, we answer RQ1 as follows.

**Answer to RQ1:** Rules that describe the general structure of issues may improve issue type classification. Rules that describe specific issue types and interfere with the classification should be avoided.

#### 7.1.2 RQ2: Does training data have to be manually validated or can large amount of unvalidated data also lead to good classification models?

Our results show that training with unvalidated data leads to a comparable mean performance than training with validated data. However, we also find that there are strong differences in how this performance is achieved. Our results provide a strong indication that manual validation leads to models with a better *precision*, i.e., fewer false positive

predictions of bugs. On the other hand, large amounts of unvalidated data achieve only a mediocre *precision* that is counterbalanced by a very high recall, i.e., few bugs that are missed. Thus, the choice whether to use unvalidated data or validated data should be done with the use case of the issue type prediction in mind. For example, for defect prediction research few false negatives are important, i.e., unvalidated data is better. If a sample of bug-fixing commits should be manually validated line by line to create data like Defects4J (Just et al. 2014), few false negatives are more important and training with validated data would be preferable. In summary, we answer RQ2 as follows.

**Answer to RQ2:** Unvalidated data is useful, if the goal is to miss as few bugs as possible. If there should be few false positive, validated data should be used for training.

### 7.1.3 RQ3: How good are issue type classification models at recognizing bug issues?

According to our results, users of issue type prediction can expect a *F1 score* of about 0.65 if applied to all issues and of 0.80 if applied to only bugs. Depending on the use case, one can either use validated or unvalidated data and thereby, get usable models for the prediction of bug issues both with few false positives (less than 25%) or few false negatives (less than 5%), but not at the same time. However, our results also show the limitations of the current state of the art of issue type prediction. Models that perform universally well with both few false positives and few false negatives are not possible within the current state of the art and the available data. We believe that this could only change, if massive amounts of validated data would be available, i.e., not in the order of  $10^5$  as is currently the case, but rather in the order of at least  $10^6$  or more. However, given that (Herzig et al. 2013) and (Herbold et al. 2020) report that the manual issue type classification is very time consuming, it may be problematic to achieve this. In summary, we answer RQ3 as follows.

**Answer to RQ3:** Issue type classification models are good at recognizing bugs, in case the use case allows for either some false positives or some false negatives. Current models cannot minimize both false positives and false negatives.

## 7.2 Recommendations for Using Issue Type Prediction

Based on our results, we have the following recommendations for researchers and practitioners with respect to the scenarios we outlined in Section 2.

- Researchers may use issue type prediction to reduce the false positive issue labels (Scenario 2), but the resulting data will still contain mislabels and does not constitute ground truth. Models for this purpose should be trained with manually validated data. In case a very high data quality is required, issue type prediction is not a suitable replacement for manual validation.
- Practitioners may use issue type prediction to automatically differentiate between bugs and other issue types in bug trackers and get comparable results to the currently assigned labels by developers. Models for this purpose should be trained with a large

corpus of data that does not require manual validation. We recommend to use this as a passive recommendation (Scenario 4) and not active recommendation (Scenario 3), because too many wrong active predictions may lead to the rejection of the approach.

### 7.3 Comparison with the Literature

An important part of our work was the replication of the existing approaches from the literature and the evaluation of their performance on independent data and, in general, for more than the usually used five projects by Herzig et al. (2013). The literature generally reported very good results with performance values higher than the best results we observed in our study. We could not replicate this but are aware of several reasons for the differences between our work and the literature. Most importantly, we used more data. Thus, if an approach randomly works on two or three projects, but fails otherwise, this would be detected by our work, but not necessarily by the smaller case studies in the related work. Also, we used a different case study setup. We clearly separated the training and test data, to avoid any kind of information leakage, e.g., because timing aspects were not considered while doing cross-validation. When we looked at the literature, all prior publications performed some sort of train/test split within the projects, sometimes with accounting for the timing (Terdchanakul et al. 2017; Otoom et al. 2019; Pingclasai et al. 2013), but sometimes not Limsettho et al. (2014a), Chawla and Singh (2015), and Qin and Sun (2018). In addition to the possible information leakage, this kind of train/test splits reduced the amount of test data to at most 20% of all the data of a project. Thus, the related work not only used fewer projects, but also less data of these projects for testing. Finally, most of the literature did not even use all five projects from Herzig et al. (2013), but subsets of this data, further reducing the evidence available for drawing conclusions. Interestingly, the best performing approach from the literature was never evaluated on validated data Kallis et al. (2019). However, the authors used 30,000 unvalidated issues for the performance estimation, i.e., more evidence than the other approaches from the state of the art.

In conclusion, we saw that the literature on this topic was not reliable so far and hope that our work not only sheds light on how well issue type prediction actually works, but also serves as guideline for future studies on this topic.

## 8 Threats to Validity

There are several threats to the validity of our work, which we report following the classification by Wohlin et al. (2012).

### 8.1 Construct Validity

The design of experiments may not be suitable to analyze our research question. The biggest threat is to the analysis of RQ1, because we only evaluated the impact of two manually designed rules and only in the single way of using different classifiers. Other manual knowledge or other ways to incorporate this knowledge into the learning process may lead to different results. Moreover, the results of all three research questions are impacted by our choice of *F1 score*, *recall*, and *precision* as performance metrics. While these metrics are well accepted in the state of the art and reasonable for the use case, other metrics, especially

metrics that would consider different costs for different kinds of misclassification, may lead to different results.

Regarding the statistical methods, the biggest threat is that we did not perform statistical tests for the selection of the best classifier per publication, but simply took the one with the best mean value. Thus, other classifiers from the same publication might not be statistically significantly different. Unfortunately, performing additional tests was not reasonably possible, because if we would have done one statistical test per publication, we would have a very small significance level for the subsequent tests where we compare publications due to the Bonferroni correction. If we would just have applied the Nemenyi post-hoc test to all 25 approaches, we would have found much fewer significant difference, because the critical distance increases linearly with the number of approaches, i.e., our tests would not have been very powerful. Therefore, doing a selection just with the mean values was the only viable option with the amount of data we have available. Another potential threat is that a Bayesian approach for the statistical analysis as was suggested by Benavoli et al. (2017) as an update of the guidelines by Demšar (2006) may lead to different results. We mitigated this threat by not using pure null hypothesis testing, but also considering the confidence intervals and the effect sizes.

## 8.2 Internal Validity

Our interpretation that the Pandey2018-LR is actually worse than Herbold2020-FTA and Kallis2019-FT is only conjectured from the properties of the statistical tests, but not directly supported by the statistical analysis. Moreover, our conclusions regarding the differences between the models trained with unvalidated and validate data may also be wrong. The unvalidated data contains more issues, i.e., the size of the data set could also be responsible for the differences in *recall* and *precision*. We believe that the difference in *recall* may be due to the size and decrease with more validated data, but cannot reasonably determine this without more validated data. However, we believe that the differences in *precision* are unlikely to go away, because this would mean a performance degradation due to more validated data, which is unlikely.

## 8.3 External Validity

The manually validated data was mostly for Java projects of the Apache Software Foundation that use Jira as issue tracker. Only two projects used Bugzilla and one project was from the Mozilla Foundation. The unvalidated data we used was from a diverse range of software projects written in different languages and owned by different organizations, but also limited to Jira as issue tracker. Therefore, it is unclear how well our results generalize beyond Apache projects, the Java programming language, and the Jira issue tracker. However, the good results with the unvalidated data indicate that generalization to different organizations and programming languages is likely. Generalization to other issue trackers is also likely, because we do not observe a difference between the two Bugzilla projects and the Jira projects in our results. Moreover, the results from Phase 2 of our experiments show that our observations hold on two data sets, i.e., at least generalize to some degree.

## 8.4 Reliability

There are no threats to the reliability of our research, other than the threats to the reliability of the prior research by Herzig et al. (2013) and Herbold et al. (2020), i.e., the reliability

of the manually validated data that we used. However, we note that the work by Herbold et al. (2020) is a successful independent conceptual replication of the work by Herzig et al. (2013).

## 9 Conclusion

Issue tracking systems play a central role in the organization of modern software development. Issues that are raised guide the development process and describe, e.g., requested features or reported bugs. Each issue has a type that is assigned by the reporter of the issue and only seldom changed afterwards. From prior research, we know that the issue type does often not match with the content of the issue, e.g., because feature requests are incorrectly labelled as bug. Within this article, we analyzed the state of the art of automated issue type prediction with machine learning and focused on the prediction of whether an issue describes a bug or not. We analyzed if we can improve issue type prediction with rules that account for structural information or rules about whether an issue is a bug and found that accounting for the structure of issues may slightly improve predictions. Moreover, we determined that data that contains mislabeled issues may still leads to good prediction models regardless, especially with respect to their ability to correctly identify all bugs. In comparison, training with manually validated data that does not contain mislabels leads to classifiers with a similar performance, but with fewer non bug issues predicted as bug at the cost of missing more bugs. Overall, the performance of the prediction models is promising and indicates that issue type prediction is likely suitable for practical use within tools and to improve data for research. We demonstrated this with two practical examples that show how the identification of bug fixing commits can be improved and that prediction in issue trackers would be comparable to the labels of developers.

The knowledge about issue type prediction is still limited, especially regarding the use as recommendation system. Therefore, we believe that future work should investigate how tools such as the issue type prediction tool by Kallis et al. (2019) are received by developers and how they can best be integrated into existing issue tracking systems. We believe that such studies should combine quantitative and qualitative aspects, e.g., to quantitatively analyze how often developers agree with predicted labels and qualitatively analyze the feedback of developers through interviews and questionnaires. The development of tools should also consider when and how prediction models are updated. For example, semi-supervised online learning could be used to continuously improve prediction models through re-training in case users actively disagree with predictions. Moreover, we believe that more validated data is the only reasonable way to overcome the problem that one must choose between models with high *recall* and models with high *precision*. A collaboration by many researchers with the research turk approach (Herbold 2020) could help to overcome the issue of the large manual effort and be suitable for the creation of a larger data set.

**Acknowledgments** This work is partially funded by DFG Grant 402774445.

**Funding** Open Access funding provided by Projekt DEAL.

**Open Access** This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons

licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

## References

- Antoniol G, Ayari K, Di Penta M, Khomh F, Guéhéneuc YG (2008) Is it a bug or an enhancement?: A text-based approach to classify change requests. In: Proceedings of the 2008 conference of the center for advanced studies on collaborative research: meeting of minds, ACM, New York, NY, USA, CASCON '08, pp 23:304–23:318. <https://doi.org/10.1145/1463788.1463819>
- Bartlett MS (1937) Properties of sufficiency and statistical tests. *Proc R Soc London A Math Phys Sci* 160(901):268–282
- Benavoli A, Corani G, Demšar J, Zaffalon M (2017) Time for a change: a tutorial for comparing multiple classifiers through bayesian analysis. *J Mach Learn Res* 18(1):2653–2688
- Chawla I, Singh SK (2015) An automated approach for bug categorization using fuzzy logic. In: Proceedings of the 8th india software engineering conference, ACM, pp 90–99
- Chawla I, Singh SK (2018) Automated labeling of issue reports using semi supervised approach. *J Comp Meth Sci Eng* 18(1):177–191
- Cohen J (1988) *Statistical power analysis for the behavioral sciences*. Lawrence Erlbaum Associates
- Demšar J (2006) Statistical comparisons of classifiers over multiple data sets. *J Mach Learn Res* 7:1–30
- Devlin J, Chang M, Lee K, Toutanova K (2018) BERT: pre-training of deep bidirectional transformers for language understanding. arXiv:1810.04805
- Dunn OJ (1961) Multiple comparisons among means. *J Am Stat Assoc* 56(293):52–64. <https://doi.org/10.1080/01621459.1961.10482090>
- Facebook AI Research (2019) fasttext - library for efficient text classification and representation learning. <https://fasttext.cc/>, [accessed 14-November-2019]
- Friedman M (1940) A comparison of alternative tests of significance for the problem of m rankings. *Ann Math Stat* 11(1):86–92
- Hall T, Beecham S, Bowes D, Gray D, Counsell S (2012) A systematic literature review on fault prediction performance in software engineering. *IEEE Trans Softw Eng* 38(6):1276–1304. <https://doi.org/10.1109/TSE.2011.103>
- Hammad M, Alzyoudi R, Otoom AF (2018) Automatic clustering of bug reports. *Int J Adv Comput Res* 8(39):313–323
- Han Z, Li X, Xing Z, Liu H, Feng Z (2017) Learning to predict severity of software vulnerability using only vulnerability description. In: 2017 IEEE International conference on software maintenance and evolution (ICSME), pp 125–136
- Herbold S (2020) With registered reports towards large scale data curation. In: Proceedings of the 2020 international conference software engineering - NIER track
- Herbold S, Trautsch A, Grabowski J (2018) A comparative study to benchmark cross-project defect prediction approaches. *IEEE Trans Softw Eng* 44(9):811–833. <https://doi.org/10.1109/TSE.2017.2724538>
- Herbold S, Trautsch A, Trautsch F (2020) Issues with szz: an empirical assessment of the state of practice of defect prediction data collection. arXiv:1911.08938
- Herzig K, Just S, Zeller A (2013) It's not a bug, it's a feature: How misclassification impacts bug prediction. In: Proceedings of the international conference on software engineering, IEEE Press, Piscataway, NJ, USA, ICSE '13, pp 392–401. <http://dl.acm.org/citation.cfm?id=2486788.2486840>
- Hosseini S, Turhan B, Gunarathna D (2017) A systematic literature review and meta-analysis on cross project defect prediction. *IEEE Trans Softw Eng* PP(99):1–1. <https://doi.org/10.1109/TSE.2017.2770124>
- Just R, Jalali D, Ernst MD (2014) Defects4j: a database of existing faults to enable controlled testing studies for java programs. In: Proceedings of the 2014 international symposium on software testing and analysis (ISSTA), ACM
- Kallis R, Di Sorbo A, Canfora G, Panichella S (2019) Ticket tagger: machine learning driven issue classification. In: IEEE International conference on software maintenance and evolution (ICSME), IEEE

- Kearns M (1998) Efficient noise-tolerant learning from statistical queries. *J ACM* 45(6):983–1006. <https://doi.org/10.1145/293347.293351>
- Limsettho N, Hata H, Matsumoto Ki (2014a) Comparing hierarchical dirichlet process with latent dirichlet allocation in bug report multiclass classification. In: 15Th IEEE/ACIS international conference on software engineering, artificial intelligence, networking and parallel/distributed computing (SNPD), IEEE, pp 1–6
- Limsettho N, Hata H, Monden A, Matsumoto K (2014b) Automatic unsupervised bug report categorization. In: 2014 6th International workshop on empirical software engineering in practice, IEEE, pp 7–12
- Limsettho N, Hata H, Monden A, Matsumoto K (2016) Unsupervised bug report categorization using clustering and labeling algorithm. *Int J Softw Eng Knowl Eng* 26(07):1027–1053
- Lukins SK, Kraft NA, Etzkorn LH (2008) Source code retrieval for bug localization using latent dirichlet allocation. In: Proceedings of the 2008 15th working conference on reverse engineering, IEEE Computer Society, USA, WCRE '08, p 155–164 <https://doi.org/10.1109/WCRE.2008.33>,
- Marcus A, Sergeev A, Rajlich V, Maletic JI (2004) An information retrieval approach to concept location in source code. In: Proceedings of the 11th working conference on reverse engineering, IEEE Computer Society, USA, WCRE '04, pp 214–223
- Mills C, Pantiuchina J, Parra E, Bavota G, Haiduc S (2018) Are bug reports enough for text retrieval-based bug localization? In: IEEE International conference on software maintenance and evolution (ICSME), pp 381–392
- Nemenyi P (1963) Distribution-free multiple comparison. PhD thesis, Princeton University
- Ortu M, Destefanis G, Adams B, Murgia A, Marchesi M, Tonelli R (2015) The jira repository dataset: Understanding social aspects of software development. In: Proceedings of the 11th international conference on predictive models and data analytics in software engineering, association for computing machinery, New York, NY, USA, PROMISE '15,. <https://doi.org/10.1145/2810146.2810147>
- Otoom AF, Al-jdaeh S, Hammad M (2019) Automated classification of software bug reports. In: Proceedings of the 9th international conference on information communication and management, ACM, pp 17–21
- Palacio DN, McCrystal D, Moran K, Bernal-Cárdenas C, Poshyvanyk D, Shenefiel C (2019) Learning to identify security-related issues using convolutional neural networks. In: 2019 IEEE International conference on software maintenance and evolution (ICSME), pp 140–144
- Pandey N, Hudait A, Sanyal DK, Sen A (2018) Automated classification of issue reports from a software issue tracker. In: Sa PK, Sahoo MN, Murugappan M, Wu Y, Majhi B (eds) Progress in intelligent computing techniques: theory, practice, and applications. Springer Singapore, Singapore, pp 423–430
- Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O, Blondel M, Prettenhofer P, Weiss R, Dubourg V, Vanderplas J, Passos A, Cournapeau D, Brucher M, Perrot M, Duchesnay E (2011) Scikit-learn: machine learning in python. *J Mach Learn Res* 12:2825–2830
- Pingclasai N, Hata H, Matsumoto K (2013) Classifying bug reports to bugs and other requests using topic modeling. In: 2013 20Th asia-pacific software engineering conference (APSEC), vol 2, pp 13–18. <https://doi.org/10.1109/APSEC.2013.105>
- Qin H, Sun X (2018) Classifying bug reports into bugs and non-bugs using lstm. In: Proceedings of the tenth asia-pacific symposium on internetware, ACM, p 20
- Rao S, Kak A (2011) Retrieval from software libraries for bug localization: a comparative study of generic and composite text models. In: Proceedings of the 8th working conference on mining software repositories, association for computing machinery, New York, NY, USA, MSR '11, p 43–52. <https://doi.org/10.1145/1985441.1985451>
- Shapiro SS, Wilk MB (1965) An analysis of variance test for normality (complete samples). *Biometrika* 52(3/4):591–611
- Śliwerski J, Zimmermann T, Zeller A (2005) When do changes induce fixes? In: Proceedings of the 2005 international workshop on mining software repositories, ACM. <https://doi.org/10.1145/1082983.1083147>
- Terdchanakul P, Hata H, Phannachitta P, Matsumoto K (2017) Bug or not? bug report classification using n-gram idf. In: IEEE International conference on software maintenance and evolution (ICSME), pp 534–538. <https://doi.org/10.1109/ICSME.2017.14>
- Trautsch A, Trautsch F, Herbold S, Ledel B, Grabowski J (2020) The smartshark ecosystem for software repository mining. In: Proceedings of the 2020 international conference software engineering - demonstrations track
- Trautsch F, Herbold S, Makedonski P, Grabowski J (2018) Addressing problems with replicability and validity of repository mining studies through a smart data platform. *Empirical Softw Engg* 23(2):1036–1083. <https://doi.org/10.1007/s10664-017-9537-x>

- Wohlin C, Runeson P, Höst M, Ohlsson MC, Regnell B, Wesslen A (2012) Experimentation in software engineering. Springer Publishing Company, Incorporated
- Zhou Y, Tong Y, Gu R, Gall H (2016) Combining text mining and data mining for bug report classification. *J Softw Evol Process* 28(3):150–176
- Zolkeply MS, Shao J (2019) Classifying software issue reports through association mining. In: Proceedings of the 34th ACM/SIGAPP symposium on applied computing, ACM, pp 1860–1863

**Publisher's note** Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.



**Steffen Herbold** is interim professor at the Karlsruhe Institute of Technology, Germany. His research is focused on the application of data science methods and their applications in software engineering as well as software engineering to data science methods.



**Alexander Trautsch** is a PhD candidate at the Institute of Computer Science of the Georg-August-Universität and works as part of the DEFECTS project on foundational issues of defect prediction research and the impact of the evolution of static analysis warnings over time on software quality.



**Fabian Trautsch** works as IT Application Specialist at the Sartorius AG. Previously, he worked as a PostDoc in the Software Engineering for Distributed Systems group at the Institute of Computer Science of the University of Goettingen. He received the BSc degree in applied computer science from the University of Goettingen in 2013, the subsequent MSc degree in 2015 and his doctorate in 2019. His research interests include mining software repositories, software evolution, software testing, and empirical software engineering.

## **E An NLP model for software engineering data**

This section contains a copy of the following publication.

J. von der Mosel, A. Trautsch, S. Herbold On the validity of pre-trained transformers for natural language processing in the software engineering domain.

© 2022 IEEE. Reprinted, with permission, from [J. von der Mosel, A. Trautsch, S. Herbold, On the validity of pre-trained transformers for natural language processing in the software engineering domain, IEEE Transactions on Software Engineering, 2022]

<https://doi.org/10.1109/TSE.2022.3178469>

# On the validity of pre-trained transformers for natural language processing in the software engineering domain

Julian von der Mosel, Alexander Trautsch, and Steffen Herbold

**Abstract**—Transformers are the current state-of-the-art of natural language processing in many domains and are using traction within software engineering research as well. Such models are pre-trained on large amounts of data, usually from the general domain. However, we only have a limited understanding regarding the validity of transformers within the software engineering domain, i.e., how good such models are at understanding words and sentences within a software engineering context and how this improves the state-of-the-art. Within this article, we shed light on this complex, but crucial issue. We compare BERT transformer models trained with software engineering data with transformers based on general domain data in multiple dimensions: their vocabulary, their ability to understand which words are missing, and their performance in classification tasks. Our results show that for tasks that require understanding of the software engineering context, pre-training with software engineering data is valuable, while general domain models are sufficient for general language understanding, also within the software engineering domain.

**Index Terms**—natural language processing, transformers, software engineering

## 1 INTRODUCTION

THE introduction of the transformer model [1] has permanently changed the field of Natural Language Processing (NLP) and paved the way for modern language representation models such as BERT [2], XLNet [3], and GPT-2 [4]. These models have in common that they use transfer learning in the form of pre-training to learn a general representation of language, which can then be fine-tuned to various downstream tasks. Pre-training is expensive and requires large text corpora. Therefore, most of the available models are pre-trained by large companies on vast amounts of general domain data such as the entire English Wikipedia or the Common Crawl News dataset [5]. While these models achieve remarkable results in a variety of NLP tasks, the learned word representations (embeddings) still reflect the general domain. This is a problem because the meaning of words varies based on context and is therefore domain dependent. Hence, there is a considerable interest in adapting language representation models to different domains. For example, SciBERT [6], BioBERT [7] and ClinicalBERT [8] are BERT models adopted to the bio-medical domain.

In Software Engineering (SE) there are many technical terms that do not exist in other domains and words that have a different meaning within the domain. For example, the word “bug” in the general domain refers to an insect, but within the SE domain it refers to a defect. Similarly, “ant” is an insect in the general domain, but a build tool within SE. Words such as “bug” and “ant” are called polysemes, i.e., words that have different meanings depending on

their context. The notion that pre-trained natural language models should be adopted for the SE domain is not new and was already considered by other, e.g., for word2vec embeddings [9] and Named Entity Recognition (NER) with BERT models [10].<sup>1</sup>

A gap within the previous research is the larger context. The work by Efstathiou *et al.* [9] only established a difference for the by now dated word2vec approach and not for transformer models. Moreover, the focus is solely on the position of a small set of terms within a vector space. This ignores the ability of transformer models to account for the context, which may be able to infer the correct meaning of polysemes. And while Tabassum *et al.* [10] successfully demonstrated that pre-training with SE data is also useful with transformer models, they used a small variant of the BERT model and did not explore why the model yielded better results. Thus, it is unclear how much better such a model is at capturing the correct meaning of words and if larger transformer models, as they are usually used within the state-of-the-art of NLP, would perform.

Within this work, we want to close this gap through an exploratory study that aims to answer the following research questions.

- RQ1:** Are large pre-trained NLP models for SE able to capture the meaning of SE vocabulary correctly?
- RQ2:** Do large pre-trained NLP models for SE outperform models pre-trained on general domain data and smaller NLP models that do not require pre-training in SE applications?

We study these research questions using *seBERT*, a BERT<sub>LARGE</sub> model we pre-trained on textual data from Stack Overflow, GitHub, and Jira Issues and BERToverflow, a

- J. von der Mosel and A. Trautsch were with the Institute of Computer Science, University of Goettingen, Germany. GA, 30332. E-mail: alexander.trautsch@cs.uni-goettingen.de
- S. Herbold was with the Institute of Software and System Engineering, TU Clausthal, Germany. E-mail: steffen.herbold@tu-clausthal.de

Manuscript received XXX, 2021; revised XXX.

1. Pre-trained models for source code are outside of our scope.

BERT<sub>BASE</sub> model that was pre-trained by Tabassum *et al.* [10] on data from Stack Overflow.

For the first research question, we evaluate the capability of the models to correctly determine the meaning of terms in three ways: the comparison of the vocabularies of the models; their capability to infer missing words in a list of curated sentences; and the capabilities to infer missing polysemes both in a SE domain corpus and a general domain corpus. Inferring missing words is a task typically referred to as Masked Language Modeling (MLM). Through this, we complement the study by Efstathiou *et al.* [9] to understand the ability of NLP models for SE to not only create valid embeddings, but even predict the correct words given the context. Such a prediction goes beyond similarity of single words and would be a strong indicator that NLP models for the SE domain should outperform general domain models because they are demonstrably better at capturing the correct meaning.

For the second research question, we study the capability of these models to improve the performance of prediction tasks. We use several classification tasks for this purpose. 1) The prediction of bug issues, similar to Herbold *et al.* [11]. This task allows us to study how well the models perform with relatively long text for a task that is, in fact, often performed inaccurately by humans [12]. 2) The identification of quality improving commits [13], i.e., a task with very short texts. 3) Sentiment mining, a task where it was already shown that general domain transformers perform well [14].

Through our study, we provide the following contributions to the literature.

- An analysis of the validity of NLP models for SE through a combination of comparison of vocabularies and the prediction of missing words. We show that BERT models trained with SE data are able to capture the correct meaning of SE terminology, at the cost of the correct modeling of some general domain concepts like geographical locations. We further show that larger models trained with more diverse data are better at capturing nuanced meanings than smaller domain specific models.
- We advance the state of issue type prediction and commit intent prediction by improving the model performance significantly with the fine-tuned SE specific BERToverflow and seBERT models. Our data indicates that the improvement is only possibly because the model was pre-trained on SE data and also that larger NLP models outperform smaller models in the SE domain similar to the differences that can be observed between general domain models.
- We found that sentiment mining was not strongly affected by domain specific pre-training and that SE domain transformers perform similar to general domain transformers for this task.
- A basic ethical evaluation of the models revealed that the SE domain models should only be used with care, because they may have some troubling properties, especially with respect to gender bias. This warrants further research and until these aspects are better understood, such models should not be used for any tasks that may be negatively affected by gender bias,

though this also warrants caution regarding other biases such as racial bias.

The remainder of this article is structured as follows. We provide some background on transformers in Section 2, followed by a discussion of related work on domain specific NLP models in Section 3. Then, we outline the creation of the SE domain model seBERT in Section 4. Based on these foundations, we introduce our method for validating transformers within the SE domain in Section 5, present the results of this validation in Section 6, and discuss these results in Section 7. Finally, we conclude in Section 8.

## 2 BACKGROUND ON TRANSFORMERS

NLP is a broad field composed of various disciplines such as computational linguistics, machine learning, artificial intelligence, computer science, and speech processing [15]. Modern NLP models have a variety of applications, including text classification, NER, machine translation, sentiment analysis, and natural language generation. In recent years, transformer models, e.g., BERT [2], BART [16], ALBERT [17], RoBERTa [18], GPT-3 [19], and Switch-C [20] established themselves at the state-of-the-art of NLP within the general domain. Since the focus of our article is on the validity of transformer models within the SE domain, we avoid an in-depth technical discussion of the underlying neural network architectures of these models and refer for this to the literature instead (e.g., [21]). Instead, we outline the concept of such models on a high level in natural language without mathematical details.

Transformers can be described in a single sentence: they are *sequence-to-sequence* networks with *self-attention*. Now we only need to understand what the highlighted aspects mean. A sequence-to-sequence network takes as input a sequence and generates a new sequence of the same length. For NLP, the input sequence is usually a tokenized text, possibly augmented with additional tokens, e.g., to represent classes. The input sequence is then encoded, such that each input token is represented by a numeric vector, which means that we have a sequence of vectors through these embeddings. This transformation relies on the self-attention, which means that the tokens are not considered in isolation, but together. Consider this paragraph: the self-attention would consider this whole paragraph at once and, for each word, model how the *contextual meaning* depends on all other words in the paragraph. This is shown in Figure 1. The figure shows, e.g., that the meaning of “bug” is strongly influenced by the words “not” and “feature”, and that the meaning of “s” is defined by the apostrophe. Thus, there is no fixed meaning of each word and the meaning depends on the context: “bug” actually means “not bug” or “feature”, “s” is only a contraction of “is”.

That the whole sequence is used as context for each word to understand the contextual meaning is the major difference to previously used NLP models. For example, word embeddings organize words such that words with similar meanings are close to each other [23]. However, while the context is taken into account when the embedding is calculated, each word only gets a single position. If words are polysemes, the meaning depends on the context, which cannot be accurately encoded with such a fixed embedding.

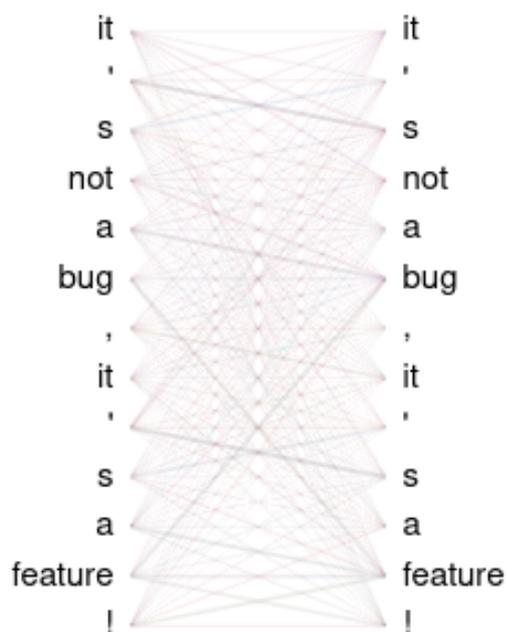


Fig. 1. Visualization of self-attention that shows how the different words in the context on the left side influence the meaning of the words on the right side. The figure shows the attention of the first transformer layer of seBERT calculated with BertViz [22].

Either one contextual meaning is lost, e.g., “bug” is only close to other insects, but not to words like “defect”. Or different concepts become similar to each other, e.g., “bug” is close to both insects and defects, which means that insects in general are now similar to defects. Moreover, since the embedding does not take the position of a word in an actual sentence into account, the interpretation of sentences based on the grammatical relationship between words is also not possible. Recurrent Neural Networks (RNN) without self-attention, such as Long-Short-Term-Memory (LSTM) networks, can also not take the complete context accurately into account. While such models can also be sequence-to-sequence networks, the relationship between a word at a specific position, and all other words, is linear. Simplified, this means that the influence on the context depends mostly on the distance to a specific word. In comparison, the self-attention learns how each word influences the context, depending on the position without a fixed influence of the distance. For our example from Figure 1, this means that the long distance between the first “it’s” and the last word “feature” does not mean that there cannot be a strong direct relationship.

Due to these differences, transformers quickly became the most powerful architecture for neural network based NLP. Models based on transformers are the current state of the art. The revolution was spear-headed by BERT [2], followed by similar models such as RoBERTa [18], GPT-3 [19] that were pre-trained with more data, used larger neural networks, and/or optimized the efficiency of the self-attention mechanisms. Such models already gained traction within the SE domain: Zhang *et al.* [14], who recently established transformers as state-of-the-art for sentiment mining within the SE domain, clearly outperforming smaller senti-

ment mining models like SentiStrength [24], including models tailored specifically to the SE domain like SentiStrength-SE [25], SentiCR [26], and Senti4SD [27]. Biswas *et al.* [28] also found a strong performance of BERT for sentiment mining.

The drawback of transformer models is the size of the neural networks. Even a relatively small model like BERT<sub>BASE</sub> [2] already has 110 million parameters. The largest currently discussed transformer architectures like GPT-3 [19] and Switch-C [20] have more than 100 billion parameters. Thus, the training of such models requires dedicated hardware and huge amounts of data. To reduce the burden, these models are *pre-trained* in a *self-supervised* setting. This means that the neural networks are trained on a large corpus of NLP data, such that the structure of text and meaning of words is known by the network. Self-supervised means that this understanding of the language is trained in a supervised way (i.e., with labels), but that the labels are generated directly from the training data. We will show how this works for the BERT pre-training in Section 4.3. Researchers and practitioners who then want to apply NLP to solve a problem, use these pre-trained models and *fine-tune* the models for a specific task on a labeled data set. This works with less data and requires only few training steps and is, therefore, usually not computationally expensive. However, this may still require dedicated hardware, due to the size of the models.

### 3 RELATED WORK ON DOMAIN-SPECIFIC NLP

In recent years there has been considerable interest in adapting pre-trained language models such as word2vec [23], ELMo [29] and BERT to different domains. However, most of the related work concerns the biomedical or financial domain. Pyysalo *et al.* [30] were among the first to provide domain-specific word2vec embeddings based on biomedical corpora. Since then, recent biomedical adaptations of word2vec and BERT include Dis2Vec [31], BioBERT [7], ClinicalBERT [8] and to some extent SciBERT [6], which has been pre-trained on biomedical and computer science papers. Models such as BioBERT and SciBERT have been shown to outperform BERT in biomedical tasks and achieve state-of-the-art results [6], [7]. The same is true for FinBERT [32] which achieved state-of-the-art results in financial sentiment analysis. However, except for SciBERT, the domain-specific BERT models are not pre-trained from scratch, but use the same vocabulary as BERT or are further pre-trained using BERTs weights.

Efstathiou *et al.* [9] pre-trained a general-purpose word representation model for the software engineering domain. They trained the so\_word2vec on 15GB of textual data from Stack Overflow posts. The authors compare the so\_word2vec model with the original Google news word2vec model and show that it performs well in capturing SE specific meanings and analogies. The main difference with our work is that we consider contextual embeddings and provide a more extensive evaluation of the validity of the resulting embeddings. In addition, our seBERT model was trained using 119GB of data from multiple sources, i.e., we also use GitHub issues and commit messages in addition to data from Stack Overflow. Ferrari and Esuli [33]

used word embeddings to identify terms that are shared between domains, but which may have different meanings. While their work was not on the difference between the SE domain and the general domain, they showed that word embeddings trained on corpora from different domains indeed yield different embeddings for polysemous word, which provides another indication that general domain models may have problems when used for domain-specific purposes.

Recently, Tabassum *et al.* [10] proposed a similar approach to ours, in which GloVe, ELMo, and BERT models are pre-trained on 152M sentences from Stack Overflow. In addition, the authors propose a novel attention based SoftNER model designed for code and NER. Their BERT model BERToverflow was pre-trained using a cased 64000 WordPiece vocabulary with the same configuration as original BERT<sub>BASE</sub> with 110 million parameters. The results show that BERToverflow clearly outperforms the other models, including BERT<sub>BASE</sub>, on NER tasks. In contrast, we pre-train seBERT with about six times more data, including data from GitHub and Jira. Moreover, seBERT uses an uncased 30522 WordPiece vocabulary and the same configuration as BERT<sub>LARGE</sub>, resulting in a much larger model with 340 million parameters. Additionally, our focus is different and not on NER tasks, but rather on the general validity of the NLP models and their usefulness for classification tasks in a pure NLP setting without considering source code.

While our focus is on pure NLP, we also want to mention similar models from the SE domain for code. Theeten *et al.* [34] proposed import2vec, a word2vec based approach for learning embeddings for software libraries. The embeddings are trained on import statements extracted from source code of Java, JavaScript and Python open source repositories. In their work, they show that their embeddings capture aspects of semantic similarity between libraries and that they can be clustered by specific domains or platform libraries. A more general source code based pre-training was conducted by Alon *et al.* [35] who proposed code2vec to represent code snippets as word embeddings. The authors show that such embeddings are a powerful tool to predict method names based on the code. Another source code based approach is CodeBERT [36], a bimodal pre-trained bidirectional transformer for natural language and programming languages. CodeBERT is pre-trained using a hybrid objective function combining the default MLM task and a Replaced Token Detection (RTD) [37] task. For the training data, the authors use both bimodal data pairs comprising a function (code) and its documentation (natural language), as well as unimodal data consisting of only of the function. The data was collected from GitHub code repositories in six different programming languages. Their results show that CodeBERT achieves state-of-the-art performance in natural language code search and code-to-document generation tasks.

## 4 seBERT

Within this section we describe how we trained seBERT, a large-scale transformer model for the SE domain that complements BERToverflow, because it is larger, was trained

on more data diverse data with a larger volume, and is uncased (see Section 3).

### 4.1 Data

We identified four data sources for our domain-specific corpus of SE textual data:

- 1) **Stack Overflow posts:** With millions of questions, answers and comments, Stack Overflow posts are a rich source of textual data from the SE field. Same as the prior work, this is one of our main data sources. The Stack Exchange Data Dump [38] contains Stack Overflow posts from 2014 to 2020 and is hosted on the Internet Archive [39]. A Stack Overflow specific mirror is available as a public dataset on the Google Cloud Platform [40]. Using BigQuery, we extracted 62.8 Gigabyte of questions, answers and comments.
- 2) **GitHub issues:** GitHub is more than a code hosting environment. For many users, GitHub issues are the first place to give feedback or report software bugs and thus provide valuable insight into the communication between users and developers. GitHub issues consist of a title, a description and comments and are available through the GitHub Archive [41]. Using Google BigQuery [42] we extracted 118.5 Gigabyte of issue descriptions and comments from the years 2015 to 2019.
- 3) **Jira issues:** Similar to GitHub issues, Jira issues provide valuable insights into software team communication regarding bug and issue tracking. In 2015, Ortu *et al.* published the Jira Repository Dataset [43], which contains 700K Jira issue reports and more than 2M Jira issue comments. With 1.4 Gigabyte of unprocessed textual data, they make up the smallest part of our corpus. However, the overall amount of data is still fairly large, and provides a perspective beyond Stack Overflow and GitHub. This includes language regarding the committing solutions to Subversion, or the submission of textual diffs as patches.
- 4) **GitHub commit messages:** Git commit messages are used to communicate the context of changes to other developers. Commit messages consist of two parts, a subject line and a message body, the latter being optional. Similar to the GitHub issues, we extracted 21.7 Gigabyte of GitHub commit messages from the GitHub Archive using BigQuery.

In total, our data set consists of 204.4 Gigabyte of unprocessed textual data. To the best of our knowledge, this the largest and most diverse corpus of textual data in the SE domain. Prior work on pre-training focused only on data from Stack Overflow. However, this ignores important aspects of the SE domain. A key aspect of natural language communication is the description of feature requests and bugs. Such data is typically not available on Stack Overflow, with the exception of users asking how they could work around a specific issue. Through the inclusion of GitHub issues and Jira issues, we enhance the corpus with such data. Moreover, commit messages are often short and on point natural language summaries of development activities and,

consequently, different from the often longer discussions within issues and on Stack Overflow.

## 4.2 Preprocessing

The textual data is mostly unstructured and needs to be preprocessed. Overall, we conducted eight different preprocessing steps.

- 1) **Basic preprocessing:** We convert all documents to lowercase, remove control characters (newline, carriage return, tab) and normalize quotation marks and whitespaces.
- 2) **English:** Since we are training a model for the English language, we use the fastText<sup>2</sup> library to remove all non-English documents.
- 3) **HTML:** We remove HTML tags and extract text using the BeautifulSoup library<sup>3</sup>.
- 4) **Markdown:** We use regular expressions to greedily remove Markdown formatting.
- 5) **Hashes:** Hashes such as SHA-1 or md5 do not provide any contextual information and should be removed. We detect hashes by checking whether alphanumeric words with a length of 7 characters or more can be cast to a hexadecimal number and replace them with [HASH] tokens.
- 6) **Code:** Source code is not natural language and should be removed. However, finding code fragments within text is a non-trivial task. We use HTML <code> tags, Markdown code blocks and other formatting to identify source code and replace it with [CODE] tokens. Code that is not within such environments is not filtered.
- 7) **User mentions:** For privacy reasons, we replace usernames and mentions (@user) with [USER] tokens.
- 8) **Special formatting:** We remove special formatting and content such as Jira specific formatting, Git sign-off, or references to SVN.

The preprocessing steps differ for the data sources. For example, removal of Markdown does not make sense for Stack Overflow posts or commit messages. Table 1 shows which steps we applied to which data and Table 2 shows the amount of data from each data source after preprocessing. After preprocessing, a total of 119.7 Gigabyte of text or 20.9 billion words remain.

## 4.3 Pre-training

The BERT implementation by Devlin *et al.* [2] uses WordPiece embeddings with a 30522 token vocabulary. We train a new SE domain-specific vocabulary of the same size using all our data and the BertWordPieceTokenizer by HuggingFace.<sup>4</sup>

An important parameter in BERT pre-training is the maximum sequence length. Training sequences shorter than the maximum sequence length are padded with [PAD] tokens, while longer sequences are truncated. Since the self-attention mechanism of BERT has a quadratic complexity

with respect to the sequence length [2], the parameter significantly affects the training time and the memory requirements. Therefore, the authors recommend training 90% of the training steps with a sequence length of 128 and 10% of the steps with a sequence length of 512 to learn longer contexts [2]. We analyzed the sequence length for all our data through histograms, which are shown in Figure 2. The sequence lengths follow exponential distribution and 96.7% of all training sequences are shorter than 256 and 90.1% are shorter than 128. Therefore, since most of our data is short-sequence data, we train with a sequence length of 128 for all steps at the cost of a possible small disadvantage with very long contexts.

BERT pre-training is self-supervised based on MLM and Next Sentence Prediction (NSP) tasks. For the NSP task, we format the data using the NLTK library<sup>5</sup> so that each line contains a sentence and documents are separated by blank lines. For the NSP task, the input data is then prepared such that each subsequent pairs of sentences makes up one input sequence of the form [CLS] sentence\_1 [SEP] sentence\_2 [SEP]. We re-use code from the original BERT to prepare the data for the MLM task.<sup>6</sup> The provided script duplicates the input sequences by a dupe factor and creates training samples by randomly masking 15% of the whole words. Whole word masking was not part of the original BERT implementation and only added later by the authors as improvement of the preprocessing. With whole word masking all tokens of a word are masked at once, which means that it is not possible that words are only partially masked. Figure 3 shows this for a document with three sentences. As a result of this data preparation, we have 2.4 Terabyte of data which we can use as input for the pre-training of BERT with TensorFlow.

We pre-train seBERT using the same configuration as BERT<sub>LARGE</sub>, i.e., with 24 layers, a hidden layer size of 1024, and 16 self-attention heads, which leads to a total of 340 million parameters. Pre-training of BERT is expensive, e.g., the original BERT<sub>LARGE</sub> was pre-trained on 16 Cloud TPUs for 3 days. Since then, several optimizations have been proposed to reduce the training time of BERT. You *et al.* [44] proposed a Layer-wise Adaptive Moments Based (LAMB) optimizer that allows the use of larger minibatches compared to the originally used ADAMW optimizer. NVIDIA then implemented their own version NVLAMB and combined it with Automatic Mixed Precision (AMP) [45], which accelerates training time and reduces memory usage by autocasting variables to 16 bit floating point numbers upon retrieval, while storing variables with the usual 32 bit precision. We used this NVIDIA implementation of the BERT pre-training to facilitate the effective usage of the system we had available with 2x24 CPU cores, 384 GB RAM and 8x NVIDIA Tesla V100 32G GPUs. The pre-training of seBERT required about 4.5 days.

## 5 VALIDATION OF SE DOMAIN MODELS

The evaluation of pre-trained language models is usually performed with benchmarks such as the GLUE benchmark [46], its successor SuperGLUE [47] or SQuAD [48].

2. <https://fasttext.cc/>

3. <https://beautiful-soup-4.readthedocs.io/>

4. <https://huggingface.co/>

5. <https://www.nltk.org/>

6. <https://github.com/google-research/bert>

Processing	GitHub issues	Commit messages	Stack Overflow	Jira issues
Basic	X	X	X	X
English	X	X		
HTML			X	
Markdown	X			
Hashes	X	X	X	X
Code	X		X	X
User mentions	X		X	
Special formatting		X		X

TABLE 1  
The applied preprocessing steps for each data source.

	GitHub	Stack Overflow	Jira
Issues	29.7 Gigabyte	Questions 10 Gigabyte	Issues 502 Megabyte
Comments	39.3 Gigabyte	Answers 10.1 Gigabyte	Comments 613 Megabyte
Commit messages	18.2 Gigabyte	Comments 11.3 Gigabyte	

TABLE 2  
The size of the data after preprocessing.

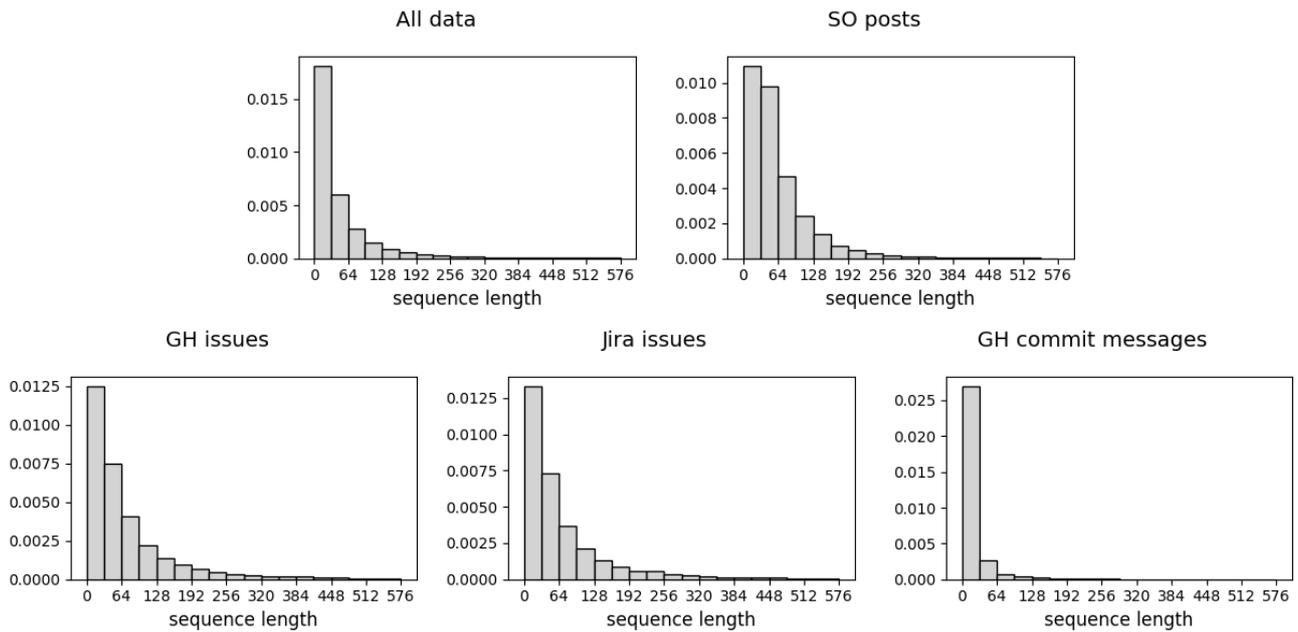


Fig. 2. Histograms of sequence length for each data source and all data combined.

Document with three sentences.	i need to run a method every 5 seconds using mono for android. is there a scheduled timer in android? i have tried this code, however, it fails to start: [CODE]
Two input sequences with sentence pairs.	[CLS] i need to run a method every 5 seconds using mono for android [SEP] is there a scheduled timer in android? [SEP] [CLS] is there a scheduled timer in android? [SEP] i have tried this [CODE], however, it fails to start: [CODE] [SEP]
Duplicates of the input sequences with 15% randomly masked words.	[CLS] i need to run a [MASK] every 5 seconds using mono [MASK] android [SEP] is there a [MASK] timer in android? [SEP] [CLS] i [MASK] to run a method every 5 seconds using [MASK] for android [SEP] is there a scheduled [MASK] in android? [SEP] [CLS] is there a scheduled [MASK] in android? [SEP] i have tried this [MASK], however, it fails to start: [CODE] [SEP] [CLS] is there a [MASK] timer in android? [SEP] i have tried this code, [MASK], it fails to start: [CODE] [SEP]

Fig. 3. Example for preparation of data for training with BERT.

However, these benchmarks require fine-tuning and are designed for the general domain and therefore not suitable for the evaluation of NLP models for the SE domain. Moreover, while the evaluation on benchmark data is useful to evaluate the overall predictive capabilities of a model, they are not suitable to understand the reasoning within a model. Therefore, we propose a different approach for the evaluation of transformer models within the SE domain that considers the validity from three different perspectives: the vocabulary, context sensitive prediction of masked words, and a fine-tuned benchmarks using a limited amounts of data for classification.

### 5.1 Vocabulary

One of the main differences between a domain-specific language model and a general domain language model should

be the vocabulary. We expect that the SE specific models should gain words from the SE domain, while lost words should be from other domains. Since alphanumeric words are more prevalent in the domain, a larger proportion of words should be alphanumeric.

We take pattern from Beltagy et al. [6] and perform a vocabulary analysis by computing the WordPiece overlap, i.e., the percentage of identical tokens between vocabularies. Additionally, we examine the vocabularies for differences in structure, tokenization, and the types of words gained or lost through a manual analysis of the differences. The manual analysis was conducted by two authors who both read the word lists and took notes about differences. The notes were then compared and discussed during a virtual meeting, which resulted in the qualitative assessment of the differences we provide in Section 6.1.

## 5.2 Contextual embeddings

Transformer models such as BERT go beyond word similarities and instead consider the complete context to infer the meaning. This goes beyond the context-free finding of similar words that was used by Efstathiou *et al.* [9] to determine if word embeddings from the SE domain are valid. Instead, we suggest to use MLM tasks to evaluate the validity of the embeddings, i.e., the evaluation of understanding how good the model is at predicting missing words in a text [2], [6]. We devise three categories of sentences suitable for validating language representation models for the software engineering domain with the goal to understand if seBERT and BERToverflow behave as expected due to the differences in training data from the original BERT. The three categories are the following.

- 1) **Positive examples:** Prediction of masked words within a software engineering context, i.e., where the missing word or context words are polysemous or from the SE domain. We expect domain-specific models to produce accurate predictions, while the original BERT may struggle because of the SE context.
- 2) **Negative examples:** Prediction of masked words outside a software engineering context, where the missing word or context words are polysemous. Given the polysemy of the negative examples, domain-specific models should produce domain-specific predictions, which are not suitable for the sentences from the general domain.
- 3) **Neutral examples:** Prediction of masked words outside a SE context, where the missing word can be inferred from general language understanding (e.g. idioms, opposites, antonyms). General language understanding should be preserved across domains, we expect all models to perform well.

Since sentences of these categories can only demonstrate the validity by way of example, we refer to them as *validation examples*. We defined ten examples per category. Due to space restrictions, we only list the sentences together with the predictions by the models in Section 6.2 in tables 4-6. For each category, we defined ten validation examples. To define these examples, one author created a list of candidate

sentences including the word that should be masked and the expectation. These candidates were prepared based on the polysemous terms from Efstathiou *et al.* [9]. Then, all authors discussed and refined these candidates together in a virtual meeting to define the ten examples per category. We can then compare how the different models complete the sentences and how this matches our expectation.

The drawback of our validation examples is that while they provide valuable insights, they are not an unbiased large sample, but rather a small and manually designed sample with a specific purpose. This raises potential issues regarding the generalizability of the effects we find to real world data. We define the following experiment with the goal to compare the differences of contextual embeddings within the SE domain on a large scale. The general idea is to conduct a MLM experiment in which polysemes are masked and then compare how good the models are at predicting the polysemes.

- Determine the overlap between the vocabularies of all involved models and discard all subwords.
- Manually extract a list of polysemes from the overlapping words, i.e., words that have a different meaning in the SE domain and the general domain (e.g., shell, bug, root).
- Collect two large corpora of text data: one from the SE domain and one from the general domain. Both corpora must not have been used during the pre-training of the models. In our case, we collected the Stack Overflow posts from 2021 as SE domain corpus (2.9 GB) and used the CC-News corpus (2.6 GB)<sup>7</sup> as general domain corpus.
- For each of the polysemes, randomly sample 100 sentences from both corpora. If there are less than 100 such sentences, drop the polysemes from the analysis.
- Mask the polyseme in each sentence and query the model for the five most likely words.

We can then evaluate the different models with respect to their capability to find the masked word. We would expect that models from the SE domain perform better on the Stack Overflow corpus and, vice versa, models from the general domain perform better on the CC-News corpus.

## 5.3 Fine-tuned Prediction Tasks

The first check regarding the validity is to analyze the pre-trained model directly. However, it is unclear if the actual use of these models in fine-tuned applications would also be affected. We propose to evaluate the validity of the models by applying them to a classification task within the SE domain, for which only a limited amount of data is available. We propose to always use (at least) three different tasks to evaluate the capabilities of the fine-tuning.

- The first type of task should have longer texts. This task is suitable to evaluate how good the model is at understanding longer texts and contexts.
- The second type of task should have shorter text, to evaluate how good the model is for short texts.

7. [https://huggingface.co/datasets/cc\\_news](https://huggingface.co/datasets/cc_news)

- The third task should be sentiment mining, as a general domain use case that is already well understood in the SE domain [49], incl. strong, already existing results that general domain transformer models are the current state-of-the-art for sentiment mining in the SE domain [14]. Since the work by Novielli *et al.* [49] suggests that SE domain models for sentiment mining do not yet work very well because their results are unstable, and Novielli *et al.* [50], Lin *et al.* [51], and Zhang *et al.* [14] showed that smaller general domain models frequently outperform SE-specific models,<sup>8</sup> this task allows us to evaluate if pre-trained SE domain transformer models are also better on SE domain data for use cases that may not require domain knowledge.

For all fine-tuning tasks, we propose to simply re-use the classification implementation for BERT from HuggingFace:<sup>9</sup> this implementation adds a single fully connected layer which produces the classification based on the output sequence of the BERT model. We then train each model for five epochs on the issue, commit, respectively, sentiment data for the fine-tuning. We use 80% of the training data for the fine-tuning and 20% as validation data to determine when the models start to overfit.

We note that we use different fine-tuning tasks than the NER task that was used by Tabassum *et al.* [10]. Our rationale for using a classification task instead is that the NER task is very specific, especially with the respect that the exploitation of casing and special characters like the underscore is a major aspect of NER within natural language models. Thus, this task does not aim to understand the meaning, but rather the inference of a specific language construct. In comparison, our tasks require the interpretation of the meaning of natural language.

In the following, we discuss how we implement and evaluate each of the fine tuning tasks in detail.

### 5.3.1 Prediction of Bug Issues

We propose to use the prediction if issues are bugs [11] as task with longer sequences. This task has several properties that make it a good candidate to evaluate the validity and benefits of transformer models. Issues are among the longer texts within SE data (see Figure 2), Moreover, there is an established baseline performance which shows that a simple neural network approach without pre-training based on fastText [52] with built-in automated hyperparameter tuning performs well [11]. Thus, we can not only compare pre-trained models to each other, but also their benefit over simpler models that can be trained in minutes without any special hardware. Second, the task is hard and often not solved correctly by humans, unless careful manual validation is used [12], [53], [54].

8. While this is not the core conclusion of Novielli *et al.* and Zhang *et al.*, their results in terms of the class imbalance insensitive macro average show the following: SentiStrength outperforms SentiStrength-SE and SentiCR on the Stack Overflow data by Novielli *et al.* [50]. SentiStrength outperforms SentiStrength-SE, Senti4SD for the API and SO data, as well as only Senti4SD on the App data by Zhang *et al.* [14]

9. [https://huggingface.co/transformers/v4.2.2/model\\_doc/bert.html](https://huggingface.co/transformers/v4.2.2/model_doc/bert.html)

We use the five projects with validated issue type data from Herzig *et al.* [12] together with the 38 validated projects from Herbold *et al.* [54] together as a single data set with 43 projects and conduct a leave-one-project-out cross validation experiment. This means that one project is used for testing and the remaining 42 projects for the training of the classifier, i.e., of the a fastText model as baseline and for the fine-tuning of the BERT models. Since there are 38,219 issues in the data and the largest project has 2,399 issues, we create 43 fine-tuned models trained with at least 35,820 issue descriptions. Following Herbold *et al.* [11], we conduct a statistical comparison based on the *F1 score* which is defined as

$$\begin{aligned} recall &= \frac{tp}{tp + fn} \\ precision &= \frac{tp}{tp + fp} \\ F1\ score &= 2 \cdot \frac{recall \cdot precision}{recall + precision} \end{aligned}$$

where *tp* are true positives, i.e., bugs that are classified as bugs, *tn* true negatives, i.e., non bugs classified as non bugs, *fp* bugs not classified as bugs and *fn* non bugs classified as bugs. Additionally, the *recall* and *precision* should be reported to allow us to understand the nature of the errors made by the models. We use a Bayesian approach for the statistical comparison following the guidelines by Benavoli *et al.* [55]. The authors suggest to use a Bayesian signed rank test for a comparison of results on multiple data sets, as is the case here. The advantage of the Bayesian approach is that the number of models that are compared does not influence the results, as this directly estimates that one approach performs better, which does not require an adjustment of the significance level [55].

### 5.3.2 Commit Intent Classification

As second fine-tuning task with shorter sequences, we use the automated classification of commits based on commit messages. Commit messages are usually very short (see Figure 2). Unfortunately, we are not aware of a solid benchmark for the machine-learning based classification of commit messages. Due to the good results for issues, we assume that fastText is also a good baseline for the classification of commit messages. As data, we use manually validated data from Trautsch *et al.* [13], i.e., we have a sample with a total size of 2533 commit messages that has as class label whether a commit is perfective or not. We follow the approach outlined by Benavoli *et al.* [55] for the comparison of classifiers on a single data set: we conduct 10x10 cross-validation and obtain 100 results for each classifier. We then use the Bayesian correlated *t*-test to compare the results. Same as above, we use the *F1 score* as foundation for the statistical comparison and report the *recall* and *precision* to understand the nature of the errors made by the models.

### 5.3.3 Sentiment Mining

As third fine-tuning task we propose sentiment mining [14], [49]. We use three data sets that were also used by Zhang *et al.* [14] in a prior benchmark: the SO [51] and API [56] data with 4522 resp. 1500 sentences from Stack Overflow posts

and the GH data [57] with 7122 sentences from GitHub pull requests and commit messages. All three data sets have three labels, i.e., positive, neutral, and negative sentiment. For each of these data sets, we use the same experimental setting as for the second fine-tuning task, i.e., we conduct 10x10 cross-validation. Since Zhang *et al.* [14] also the *F1 score*, we follow this approach and report the macro averages of the *F1 score*, *recall*, and *precision*, same as for the other use cases.

## 6 EXPERIMENTS

We conducted an experiment to evaluate the validity of the SE models seBERT and BERToverflow in comparison to the general domain BERT models. Since seBERT is based on BERT<sub>LARGE</sub> and BERT<sub>BASE</sub>, we use both of these models within our comparison. We follow the procedure outlined in Section 5. This means we first compare the vocabularies. Then, we proceed to look at the validity of the models without fine-tuning through their ability to infer the correct meaning of terms. Finally, we evaluated how the different models perform when fine-tuned with a limited amount of labeled data for a prediction task.

All results, as well as the required scripts and link to the data for pre-training seBERT are available as part of our replication kit.<sup>10</sup> Additionally, we prepared a playground, which can be used to fill in masked words using the models.<sup>11</sup>

We restrict our comparison to BERT, as this is the most similar general domain model and, therefore, most suitable to explore the need for SE-specific models for SE domain tasks. Please note that we also exclude CodeBERT from this comparison, because CodeBERT is not natural language model, but rather a bi-modal “natural language - programming language” model, i.e., for working with source code.<sup>12</sup> Additionally, we use fastText for the comparison with respect to classification tasks, to understand if smaller NLP models may be sufficient.

### 6.1 Vocabulary Comparison

The vocabularies of BERT<sub>BASE</sub> and BERT<sub>LARGE</sub> are equal, and we refer to them as BERT vocabulary in the following. The WordPiece overlap between BERT and seBERT is 38.3%. A major difference between the vocabularies is the number of “##” sub-word WordPieces, with seBERT (7430) having almost twice as many as BERT (3285). Most of the words added to the seBERT vocabulary are from the SE domain (e.g. “bugzilla”, “jvm”, “debug”) or internet slang (e.g. “fanboy”). Lost words are mainly from the geographical (e.g. “madrid”, “switzerland”, “egypt”), religious (e.g. “jesus”, “buddha”) or political (e.g. “president”, “minister”, “clinton”) domain.

For words that occur in only one of the two vocabularies, we use the tokenizer of the other model to break them down

10. <https://github.com/smartshark/seBERT>

11. <https://smartshark2.informatik.uni-goettingen.de/sebert/index.html>

12. Nevertheless, because CodeBERT has a natural language part, we included CodeBERT in our playground for filling in masked words, for those interested in the suitability of CodeBERT for language understanding.

into their respective WordPieces. We have presented the results for a small subset of words in Table 3. As we can see, BERT tokenizes SE domain words inconsistently, e.g. “bugzilla” is tokenized into “bug ##zil ##la”, but “debug” into “de ##bu ##g”, showing that there is no WordPiece for “##bug”. In addition, in-domain abbreviations such as JVM are unknown and broken down into their individual characters. In contrast, seBERT breaks down general out-of-domain words into in-domain WordPieces, e.g., “drama” into dynamic random-access memory (DRAM) or “infantry” into “inf” and “ant”. Since the vocabularies are frequency-based, an interesting finding is that “woman” is not in the seBERT vocabulary and is tokenized into “wo ##man”. This implies that “woman” is used less frequently within the domain, highlighting the ethical issues that can arise in machine learning. The full lists of out-of-vocabulary words and their tokenizations are available as supplemental material in the seBERT replication repository.

Since the BERToverflow vocabulary is cased, a direct comparison of the overlap without further processing is not feasible. Instead, we lower case the vocabulary of BERToverflow and remove duplicates. Since the vocabulary of BERToverflow is larger than that of (se)BERT,<sup>13</sup> we still cannot directly compute the overlap, because this is a ratio with respect to the vocabulary size. The larger BERToverflow should contain more words from the other vocabularies (larger ratio), while in turn the smaller the (se)BERT vocabularies should cover a smaller ratio of terms in the BERToverflow vocabulary. Instead, we calculate i) the ratio of WordPieces of the (se)BERT vocabulary within the uncased BERToverflow vocabulary and ii) the ratio of uncased BERToverflow WordPieces within the BERT vocabulary. Thus, consider the direction of the comparison, when considering ratios.

After lower casing, there are 58,854 unique WordPiece tokens. 24,263 of these tokens are also in the seBERT vocabulary. This means seBERT covers about 40% of the uncased BERToverflow vocabulary and the uncased BERToverflow covers about 80% of the seBERT vocabulary. That BERToverflow has more tokens is not surprising, due to the larger vocabulary size. While the overlap of 80% of the seBERT tokens with BERToverflow is substantial, we would have expected an even larger overlap, given that BERToverflow has about twice the amount of uncased tokens. This should be sufficient to more or less cover the seBERT vocabulary, assuming that the textual data from Stack Overflow on which BERToverflow is pre-trained, is representative for the SE domain. A manual check revealed that many of the missing tokens are related to tools, e.g., “matplotlib”, “xenial” or “hashicorp”. We can only speculate regarding the reason for this. One possible explanation is that tools are less frequently discussed on Stack Overflow than, e.g., programming languages. However, while this may be the case to a certain degree, questions regarding the usage of specific tools are common in Stack Overflow. Another possible explanation is the casing that is used when writing tools. For example, users may write “HashiCorp”, “Hashicorp”, or “hashicorp”. Within the uncased seBERT vocabulary, this

13. We write (se)BERT to highlight that these statements are true for both BERT and seBERT, which have vocabularies of the same size.

Word	BERT Tokenization	Word	seBERT Tokenization	BERToverflow Tokenization
bugzilla	bug ##zil ##la	catholic	cat ##hol ##ic	cath ##olic
chromium	ch ##rom ##ium	drama	dram ##a	drama
debug	de ##bu ##g	infantry	inf ##ant ##ry	inf ##ant ##ry
jvm	j ##v ##m	palace	pal ##ace	pal ##ace
refactoring	ref ##act ##orin ##g	woman	wo ##man	woman

TABLE 3

Example words not included in the BERT or seBERT vocabulary broken down into their WordPieces using the respective tokenizer. ## indicates the start of a subword token.

would not make a difference and all occurrences would be counted together, possibly leading to larger importance and the inclusion in the vocabulary.

We also manually evaluated which additional terms are within the BERToverflow vocabulary that are not within the seBERT vocabulary. In addition to more terms from the general domain (see the comparison to BERT below), we noticed two additional aspects. First, there were many Unicode tokens, such as special characters only used in non-English texts. Second, there were many terms that seem to come from code snippets, such as “strftime”.

The BERT vocabulary has an overlap of 15,926 WordPiece tokens with the uncased BERToverflow vocabulary. This means that about 50% of the BERT tokens are within the BERToverflow vocabulary, which is slightly more than the overlap of 38% between seBERT and BERT. However, this increase is plausible, given the overall larger vocabulary size of BERToverflow. This is also visible in the tokenization, e.g., the terms “women” and “drama” are within BERToverflow vocabulary, meaning that it covers more general domain concepts than seBERT.

## 6.2 Contextual Comparison

Next, we use the capability of the models to predict masked words. Tables 4-6 show ten examples of sentences for the positive, negative, and neutral category.

The positive examples in Table 4 show that while the general domain BERT models are not as accurate with the identification of words from the within-domain context as the in-domain models BERToverflow and seBERT. This does not mean that BERT outright fails, but rather that the inferred are not completely unrelated, but also not directly on point. In comparison, BERToverflow and seBERT both almost always suggest reasonable completions. However, we also observe that either the larger set of training data or the larger model also allows seBERT to be more accurate in some cases. Our sensitivity analysis with a smaller version of seBERT shows that likely both play a role, as the smaller model is almost equal, but worse in some nuances, i.e., handling of emoticons and opposites. The sensitivity analysis can be found in the supplemental material. The last example gives the strongest indication of the impact of using more data: since BERToverflow was not trained on issue data, but only on Q&A data from Stack Overflow, seBERT performs better in this context. For this sentence, the general domain results from BERT are actually better than BERToverflow. Our interpretation of this is that this is caused by the context of the BERToverflow data: the criticality of issues is usually not discussed on Stack Overflow. Instead, users may ask

regarding a certain problem, with other users responding that the bug is known. Thus, the association of “This is a [MASK] bug” with “known” makes sense. However, the second part of the sentence further clarifies this context as “please address this asap”. Such a direct request to the developers does not make sense within the Q&A context and could, therefore, not be considered by BERToverflow. In comparison, the seBERT data is aware of such direct communication with a development team and correctly infers that the missing word is likely the criticality of the issue, because it should be addressed as soon as possible.

The negative examples in Table 5 highlight that models trained with SE data fail, when it comes to understanding pure general domain context. All words are interpreted within the SE context, even if this does not make sense. In comparison, the general domain models were a lot closer to the performance of the SE domain models for the positive examples. When we consider the training corpus, this makes sense: topics like dentists, actual snakes, or the weather are extremely unlikely in our SE corpus, so it is plausible that this does not work at all. On the other hand, Wikipedia also covers software engineering topics. Thus, while this is not the focus of the general domain corpus, it also contains some software engineering data.

The neutral examples in Table 6 reveal some nuanced differences between all four models. This is the first time that we clearly observe that the BERT<sub>LARGE</sub> model is better at capturing the context than the BERT<sub>BASE</sub> model. In Sentence 28, the smaller models fail to correctly understand the context, while all other models understand this perfectly. The same sentence also reveals a nuanced difference: since we often start to count from zero in computer science, this is also proposed by BERToverflow and seBERT, while BERT<sub>LARGE</sub> says to start at one. We also note that the SE-specific models sometimes misjudge these neutral concepts, e.g. with sentence 23. While the sentence clearly indicates that the context is voting, the term “raise” is so strongly associated with opinions within the SE domain that hands are not suggested. This set of sentences also contains the strangest association within our data in Sentence 30: we have no idea why seBERT believes that running out of pokemon is likely. The only explanation we can think of is the tokenization by seBERT of “poker” as “poke##r”. It is possible that the subword “poke” has a strong association with “pokemon”, which leads to this mistake.

Figure 4 shows the results of our experiments of masking polysemes in the Stack Overflow corpus and CC-News corpus. Overall, we identified 503 words, which appear

Sentence	Expectation	BERT <sub>BASE</sub>		BERT <sub>LARGE</sub>		BERT <sub>overflow</sub>		seBERT	
		Prediction	Prob.	Prediction	Prob.	Prediction	Prob.	Prediction	Prob.
1) The [MASK] is thrown when an application attempts to use null in a case where an object is required.	NullPointerException	rule	0.2407	value	0.2356	exception	0.4718	exception	0.5473
		exception	0.0742	exception	0.0804	error	0.1229	error	0.2188
		coin	0.0659	coin	0.0342	Null-	0.1161	null-	0.1158
		flag	0.0245	ball	0.0318	Pointer-		pointer-	
		penalty	0.0216	flag	0.0312	Exception		exception	
						NPE	0.0995	npe	0.0291
						Illegal-	0.0403	illegal-	0.0144
						Argument-		argument-	
						Exception		exception	
2) [MASK] is a proprietary issue tracking product developed by Atlassian that allows bug tracking and agile project management.	Jira	it	0.0382	it	0.0148	Jira	0.4914	jira	0.4954
		agile	0.0104	bug	0.0074	Redmine	0.1834	there	0.0837
		eclipse	0.0041	eclipse	0.0052	JIRA	0.099	zenhub	0.0763
		flex	0.0035	echo	0.0046	It	0.0398	it	0.0488
3) [MASK] is a software tool for automating software build processes.	build automation tools	snap	0.0031	spark	0.0035	There	0.0281	bugzilla	0.0463
		it	0.1437	it	0.0396	CMake	0.1541	jenkins	0.1907
		agile	0.0071	build	0.0294	make	0.0898	make	0.0891
		this	0.0062	eclipse	0.0107	Make	0.0644	ninja	0.0676
4) Pathlib is a python library used for handling [MASK].	paths	build	0.0055	builder	0.0075	Ant	0.0595	it	0.058
		gem	0.0055	agile	0.0067	Maven	0.0558	ant	0.0536
		applications	0.0699	applications	0.1509	paths	0.429	paths	0.8727
		systems	0.0431	software	0.1505	files	0.0789	path	0.0338
5) The solution posted by [USER] is [MASK] helpful. :)	positive adverb	programs	0.0359	programs	0.078	urls	0.0664	directories	0.0237
		software	0.0351	languages	0.0631	URLs	0.0391	filenames	0.0168
		data	0.0296	systems	0.052	pathnames	0.0384	strings	0.0088
		very	0.5632	very	0.3407	very	0.5488	very	0.5719
6) The solution posted by [USER] is [MASK] helpful. :(	negative adverb	extremely	0.0656	not	0.1755	really	0.1374	really	0.1533
		quite	0.0424	always	0.0624	also	0.0734	quite	0.0729
		always	0.0313	also	0.0593	more	0.027	also	0.0458
		more	0.0273	most	0.0429	quite	0.0237	super	0.0166
7) [MASK], is a provider of Internet hosting for software development and version control using Git.	github, gitlab	very	0.5254	very	0.2764	not	0.9182	not	0.9761
		extremely	0.0659	not	0.1767	very	0.0331	no	0.0075
		quite	0.0418	also	0.0829	really	0.0087	very	0.0038
		also	0.0349	always	0.0782	never	0.0031	really	0.0011
8) In object-oriented programming, a [MASK] is an extensible program-code-template for creating objects.	class	always	0.0338	most	0.0455	also	0.0031	never	0.0009
		microsoft	0.0158	net	0.0127	Github	0.222	github	0.3096
		net	0.0136	apache	0.011	GitHub	0.2186	gitlab	0.0397
		oracle	0.0134	parallels	0.0097	Bitbucket	0.0568	git	0.0382
9) I have to discuss this with the other [MASK].	developers	org	0.0127	foundry	0.0073	BitBucket	0.0509	sourceforge	0.0309
		apache	0.0122	radius	0.0061	Assembla	0.0413	bitbucket	0.0302
		template	0.1027	template	0.6763	constructor	0.3849	class	0.4991
		class	0.0391	prototype	0.0533	class	0.2664	constructor	0.221
10) This is a [MASK] bug, please address it asap.	critical, major	object	0.0356	module	0.0258	factory	0.1435	factory	0.0669
		model	0.0219	class	0.0136	prototype	0.038	metaclass	0.0487
		gui	0.0145	construct	0.0106	Factory	0.0182	template	0.0316
		elders	0.0703	men	0.0913	guys	0.1364	developers	0.1159
9) I have to discuss this with the other [MASK].	developers	girls	0.0677	members	0.0407	people	0.1074	team	0.1114
		men	0.0622	elders	0.0399	developers	0.0918	maintainers	0.0797
		officers	0.0526	officers	0.0265	person	0.0533	people	0.0796
		council	0.0453	people	0.024	users	0.0497	devs	0.0744
10) This is a [MASK] bug, please address it asap.	critical, major	serious	0.286	security	0.0956	known	0.5999	critical	0.6915
		new	0.1166	serious	0.0807	know	0.029	serious	0.1491
		major	0.0699	minor	0.0458	chrome	0.02	major	0.0412
		persistent	0.0473	major	0.0397	browser	0.014	big	0.0270
10) This is a [MASK] bug, please address it asap.	critical, major	big	0.031	nasty	0.0298	common	0.0118	real	0.0099

TABLE 4

Prediction of words for [MASK] tokens. Results for the *positive* category, i.e., sentences where we expect that BERT<sub>overflow</sub> and seBERT perform better than BERT.

Sentence	Expectation	BERT <sub>BASE</sub>		BERT <sub>LARGE</sub>		BERT <sub>overflow</sub>		seBERT	
		Prediction	Prob.	Prediction	Prob.	Prediction	Prob.	Prediction	Prob.
11) A [MASK] crawled across her leg, and she swiped it away.	bug	spider	0.1417	bug	0.2087	person	0.1068	man	0.1042
		tear	0.1277	spider	0.1676	car	0.0561	person	0.0518
		hand	0.0841	tear	0.1463	dog	0.0559	friend	0.0472
		bug	0.0834	flea	0.048	bird	0.0405	girl	0.0299
12) Can you open the [MASK], please? It's hot in here.	window, door	mosquito	0.0534	fly	0.0447	fox	0.0397	monkey	0.0248
		door	0.7435	door	0.8776	link	0.3264	pr	0.2829
		window	0.1227	window	0.0619	file	0.0699	issue	0.1386
		windows	0.0349	fridge	0.0058	site	0.0411	door	0.0576
13) The reticulated python is among the few [MASK] that prey on humans.	snakes	blinds	0.0171	doors	0.0057	page	0.0395	file	0.0466
		curtains	0.0113	gate	0.0055	url	0.0307	link	0.0352
		snakes	0.6433	snakes	0.803	languages	0.7376	things	0.5227
		species	0.1645	reptiles	0.0779	things	0.0836	languages	0.1625
14) "I have a [MASK] request for you." He said to the waiter.	special	reptiles	0.1055	animals	0.0696	tools	0.0087	tools	0.0632
		animals	0.0278	species	0.0223	people	0.0084	packages	0.0348
		lizards	0.0269	mammals	0.0075	programmers	0.0073	programs	0.0255
		special	0.4258	special	0.8448	new	0.115	pull	0.7016
15) He was admitting to a [MASK] he didn't commit, knowing it was somebody else who did it.	crime	business	0.0859	specific	0.0198	special	0.0361	feature	0.0432
		personal	0.0809	new	0.017	test	0.0257	change	0.0171
		new	0.0318	small	0.0166	friend	0.0224	similar	0.0146
		specific	0.0225	simple	0.0128	support	0.0213	merge	0.0139
16) It's an incurable, terminal [MASK].	disease	crime	0.7861	crime	0.9565	commit	0.1497	change	0.2475
		murder	0.1162	murder	0.0358	change	0.1223	commit	0.0974
		sin	0.0484	sin	0.005	mistake	0.0844	fix	0.0833
		lie	0.0084	felony	0.0008	fact	0.0629	file	0.0464
17) The dentist said I need to have a root [MASK].	canal	suicide	0.0056	lie	0.0005	file	0.044	code	0.0372
		disease	0.4208	disease	0.8134	error	0.0777	issue	0.0605
		condition	0.3208	illness	0.0879	operation	0.0639	problem	0.0523
		illness	0.125	condition	0.0579	command	0.0517	bug	0.0443
18) Everything was covered with a fine layer of [MASK].	dust, snow	death	0.0201	cancer	0.0203	problem	0.0336	character	0.0377
		cancer	0.0193	disorder	0.0049	)	0.0288	effect	0.0273
		canal	0.8847	canal	0.9827	account	0.1919	password	0.1476
		beer	0.0307	beer	0.01	certificate	0.0959	user	0.1431
19) There was not a single cloud in the [MASK].	sky	problem	0.0167	stop	0.0009	user	0.0825	account	0.089
		out	0.005	break	0.0005	node	0.0588	access	0.0522
		cellar	0.003	problem	0.0005	access	0.0474	certificate	0.0475
		dust	0.4895	dust	0.7656	transparency	0.0518	abstraction	0.1231
20) What does it say in your [MASK] cookie?	fortune	dirt	0.1035	snow	0.0813	abstraction	0.0417	coverage	0.095
		snow	0.0625	dirt	0.0283	code	0.0408	testing	0.0894
		paint	0.0432	ice	0.0199	complexity	0.0323	detail	0.0671
		powder	0.01	sand	0.0122	confidence	0.032	documentation	0.0636
19) There was not a single cloud in the [MASK].	sky	sky	0.9009	sky	0.9425	list	0.0797	database	0.0686
		air	0.0748	heavens	0.0104	cloud	0.0361	list	0.0663
		distance	0.003	distance	0.0076	world	0.0333	dataset	0.0391
		skies	0.0021	air	0.0041	center	0.027	file	0.0358
20) What does it say in your [MASK] cookie?	fortune	room	0.0017	east	0.0032	database	0.0263	cloud	0.0357
		fortune	0.4846	fortune	0.295	browser	0.1262	session	0.4255
		chocolate	0.0534	favorite	0.2597	session	0.083	browser	0.0994
		little	0.0289	next	0.0484	firebug	0.07	auth	0.0572
20) What does it say in your [MASK] cookie?	fortune	next	0.0207	chocolate	0.0284	firefox	0.0508	cookie	0.0318
		sugar	0.0206	sugar	0.0261	debug	0.032	login	0.0195

TABLE 5

Prediction of words for [MASK] tokens. Results for the *negative* category, i.e., sentences where we expect that BERT<sub>overflow</sub> and seBERT perform worse than BERT.

Sentence	Expectation	BERT <sub>BASE</sub>		BERT <sub>LARGE</sub>		BERT <sub>overflow</sub>		seBERT	
		Prediction	Prob.	Prediction	Prob.	Prediction	Prob.	Prediction	Prob.
21) We can [MASK] in person if you have any specific questions.	meet	talk	0.4357	meet	0.695	help	0.2649	chat	0.5492
		speak	0.207	speak	0.1173	be	0.066	talk	0.1941
		meet	0.0971	talk	0.1026	edit	0.0566	discuss	0.1025
		chat	0.0746	discuss	0.0208	assist	0.0553	meet	0.0913
		visit	0.0266	communicate	0.0057	answer	0.0477	speak	0.0064
22) [MASK] its name, vitamin D is not a vitamin. Instead, it is a hormone that promotes the absorption of calcium in the body.	despite	despite	0.999	despite	0.998	Despite	0.7482	despite	0.9396
		whatever	0.0002	unlike	0.0016	despite	0.1184	in	0.0169
		unlike	0.0002	notwithstanding	0.0002	by	0.0387	from	0.0099
		in	0.0001	like	0.0001	By	0.0212	unlike	0.0095
		notwithstanding	0.0001	whatever	0.0	In	0.01	by	0.0068
23) Would all those in favour please raise their [MASK]?	hands	hands	0.5325	hands	0.537	opinion	0.2422	opinion	0.1235
		hand	0.122	hand	0.1543	opinions	0.1803	concerns	0.0798
		voices	0.1177	voices	0.128	concerns	0.0982	priority	0.057
		arms	0.031	voice	0.11	points	0.0704	opinions	0.0531
		fists	0.0124	arms	0.0121	views	0.0444	interest	0.043
24) She surprised him with a [MASK].	something positive	smile	0.4748	smile	0.4086	surprise	0.1815	bug	0.0604
		look	0.1623	laugh	0.2211	mistake	0.0606	question	0.0485
		laugh	0.0604	kiss	0.105	message	0.0551	comment	0.0197
		kiss	0.0588	question	0.0673	bug	0.034	problem	0.0191
		grin	0.0502	grin	0.0369	warning	0.0322	joke	0.019
25) Whoever is happy will make others [MASK] too.	happy	happy	0.989	happy	0.9716	happy	0.9968	happy	0.9866
		happier	0.0025	unhappy	0.0104	unhappy	0.0003	,	0.0027
		,	0.0012	happier	0.0053	,	0.0003	complain	0.0007
		smile	0.0012	,	0.0042	pleased	0.0002	sad	0.0005
		sad	0.0007	smile	0.002	sad	0.0002	comfortable	0.0003
26) If there are night owls, are there [MASK] owls too?	day	night	0.3312	day	0.5414	day	0.1331	day	0.6346
		day	0.2063	night	0.2378	night	0.0527	evening	0.1119
		other	0.0275	morning	0.0671	power	0.0352	afternoon	0.0899
		morning	0.0175	other	0.0143	weather	0.0178	morning	0.0484
		bird	0.0158	evening	0.0085	evening	0.0149	night	0.0302
27) Never forget, always remember. Always forget, never [MASK].	remember	forget	0.979	forget	0.9271	remember	0.4836	remember	0.881
		remember	0.007	remember	0.0666	forget	0.3044	forget	0.0997
		forgot	0.0018	forgive	0.0007	trust	0.0355	recall	0.0113
		forgotten	0.0015	know	0.0005	know	0.0102	bother	0.0009
		know	0.0015	forgot	0.0004	read	0.0077	guess	0.0006
28) If you are counting things, start from [MASK].	1, one	scratch	0.5561	one	0.1389	0	0.3938	zero	0.34
		there	0.0836	here	0.1222	1	0.293	0	0.2834
		bottom	0.0511	zero	0.0731	zero	0.164	1	0.2587
		one	0.0425	three	0.064	2	0.0156	there	0.021
		here	0.0286	ten	0.0497	there	0.0102	2	0.0132
29) Soccer has really simple rules. It's not [MASK] science.	rocket	a	0.7412	rocket	0.9125	rocket	0.8597	computer	0.4413
		rocket	0.0334	about	0.0356	computer	0.0513	rocket	0.3927
		really	0.0227	a	0.032	a	0.0317	a	0.0483
		just	0.0225	even	0.0021	exact	0.0071	game	0.0146
		pure	0.0147	like	0.0015	perfect	0.0049	about	0.0132
30) He ran out of [MASK], so he had to stop playing poker.	money, time	money	0.8096	money	0.7833	cards	0.4876	pokemon	0.1655
		time	0.0285	cash	0.0579	players	0.0369	memory	0.0767
		food	0.0126	cards	0.0213	hands	0.0309	ram	0.0649
		funds	0.0113	time	0.0187	money	0.0294	money	0.0618
		cash	0.0094	patience	0.0074	memory	0.0221	mana	0.0455

TABLE 6

Prediction of words for [MASK] tokens. Results for the *neutral* category, i.e., sentences where we expect that BERT<sub>overflow</sub> and seBERT perform worse than BERT.

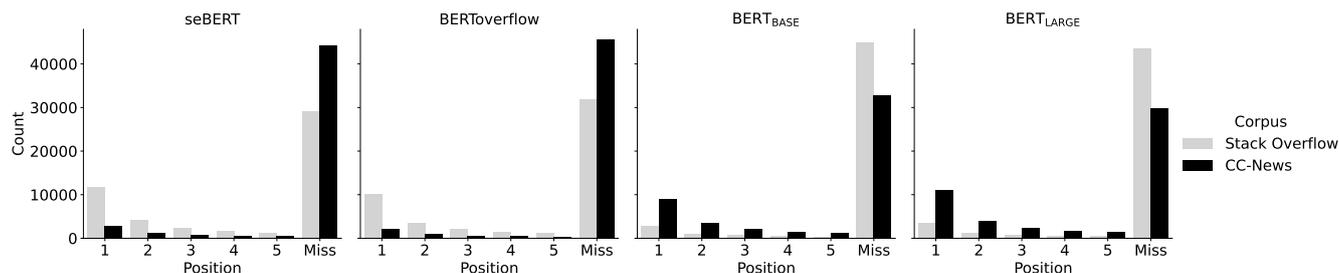


Fig. 4. Results of the MLM experiment on unknown data. The histograms show the position of the masked word in the top five predicted words of each model, or if the word was missed.

in at least 100 sentences in each corpus.<sup>14</sup> Thus, we have  $503 \cdot 100 = 50,300$  masked word predictions for each corpus. Overall, the results are in line with our expectation: seBERT and BERToverflow are better for the Stack Overflow corpus, the BERT models are better for CC-News. Moreover, the larger models (seBERT, BERTLARGE) perform a bit better than the smaller models (BERToverflow, BERTBASE). An interesting aspect of the models is that the results are symmetric: the seBERT results on Stack Overflow are similar to the BERTLARGE results on CC-News and the seBERT results on CC-News are similar to the BERTLARGE results on Stack Overflow. The same relationship can be observed between BERToverflow and BERTBASE. This could mean that the contextual understanding of the SE domain models for the general domain is about as good as that of the general domain models for the SE domain, and vice versa.

### 6.3 Prediction Task Comparison

Table 7 and figures 5-9 summarize the results of the prediction task comparison. In the following, we look at the results of the individual tasks in detail.

#### 6.3.1 Issue Type Prediction

The results show that seBERT and BERToverflow achieve the best performance for the issue type prediction tasks, outperforming both fastText and the general-domain BERT models. The improvement over fastText is very large with an about 11% higher *F1 score* for the issue type prediction. The violin plot in Figure 5 indicates that the performance improvement in *F1 score* is due to an improvement of both *recall* and *precision*, which means the models reduced both false positives and false negatives in comparison to fastText. The Bayesian signed rank test determined that this improvement of the SE-specific models over the other models is significant.<sup>15</sup> The difference between seBERT and BERToverflow is not significant. The comparison between fastText and BERTBASE shows that the general-domain models may be better than smaller text processing models without pre-training on the issue type prediction. BERTBASE significantly outperforms fastText on the issue type prediction task. The BERTLARGE model from the general domain has severe problems with this use case, i.e., there are several cases where the

models completely failed. This happened with none of the other models and may be an indication that the amount of data is too small to fine-tune such a large model from the general domain on a domain-specific corpus.

#### 6.3.2 Commit Intent Prediction

The results for the commit intent prediction are mostly in line with the results for the issue type prediction. seBERT has a 9% larger *F1 score* than the non-SE models, BERToverflow is 6% better. The violin plots in Figure 6 also indicate a stable improvement in both *recall* and *precision* and the Bayesian signed rank test determined that the improvement is significant. In comparison to the issue type prediction, seBERT is significantly better than BERToverflow with an absolute difference of about 3% in the *F1 score*. Moreover, we note that the general domain BERT models fail to outperform fastText and that BERTLARGE has the same stability issues as for the issue type prediction.

#### 6.3.3 Sentiment Mining

The results for the sentiment mining show that there are only small differences between the SE domain and general domain BERT models on all three data sets. The absolute performance on the API and GH data is almost equal. The difference on the SO data is slightly larger, where BERToverflow and seBERT outperform the BERTBASE and BERTLARGE by about 3%, with a statistically significant difference. We observe that, same as before, the BERTLARGE model is sometimes unstable, likely for the same reasons as above. The difference between the transformer models and fastText is huge for this task, i.e., at least 20% in *F1 score* on all data sets.

## 7 DISCUSSION

We now discuss our results with respect to our research questions, consider the ethical implications of our work, discuss the limitations and open issues, as well as the threats to the validity.

### 7.1 Interpretation of SE Terminology

Our results indicate that SE domain models are better at modeling natural language within the SE domain than general domain models. The vocabularies contain more tokens from the SE domain. Especially the focus of the vocabulary is interesting. Both seBERT and BERToverflow have

14. The full list of polysemes can be found in the supplemental material.

15. Posterior probabilities determined by the Bayesian signed rank test can be found in the supplemental material.

Model	Issue Type	Commit Intent	Sentiment (SO)	Sentiment (API)	Sentiment (GH)
fastText	0.69	0.75	0.44	0.37	0.47
BERT <sub>BASE</sub>	0.77	0.75	0.73	0.57	0.91
BERT <sub>LARGE</sub>	0.62	0.71	0.73	0.55	0.91
BERToverflow	0.81	0.81	0.77	0.58	0.92
seBERT	0.80	0.84	0.76	0.57	0.92

TABLE 7

Median *F1* score of the models for the fine-tuning tasks. Macro-average over the three classes for the sentiment mining.

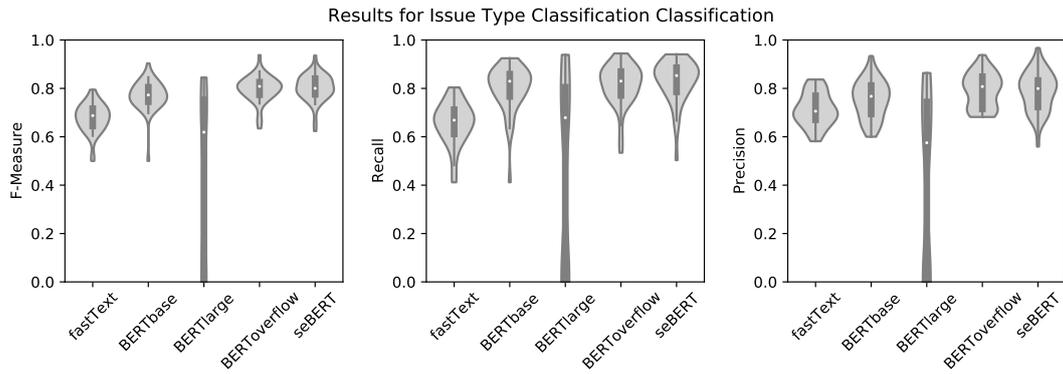


Fig. 5. Results of the prediction of bug issues.

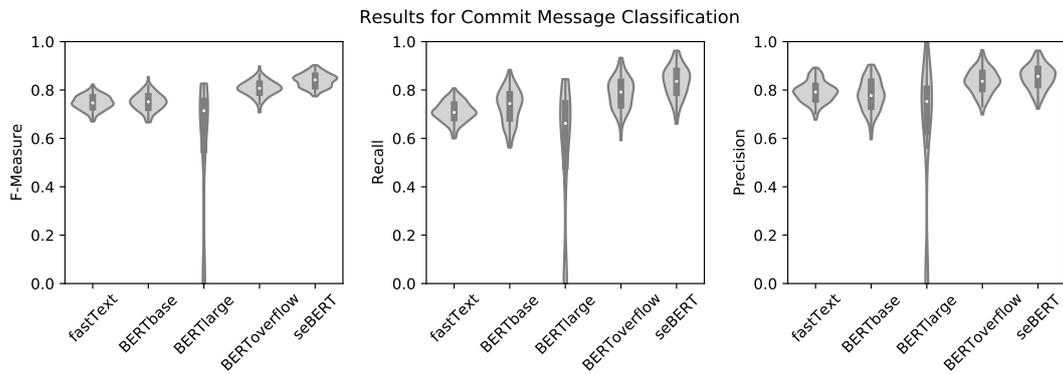


Fig. 6. Results of the prediction of quality improving commits.

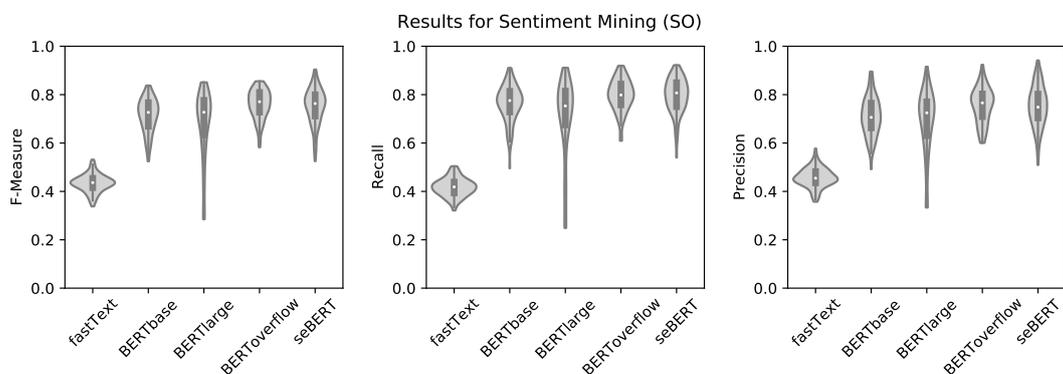


Fig. 7. Results of the sentiment mining on the SO data.

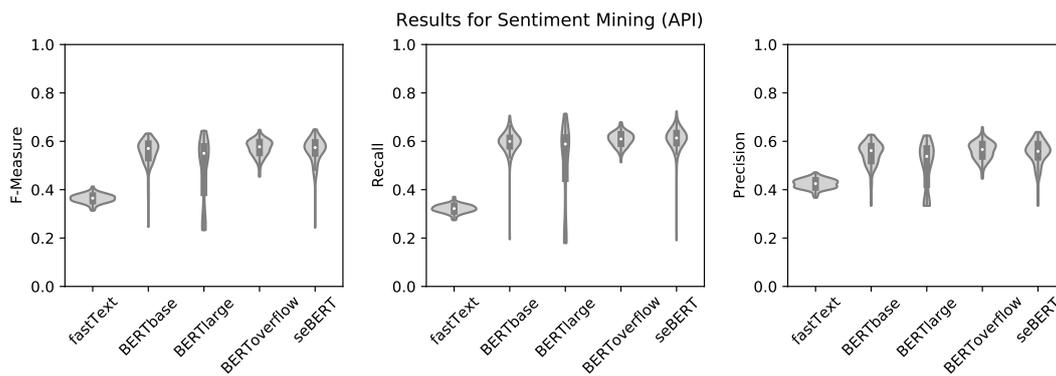


Fig. 8. Results of the sentiment mining on the API data.

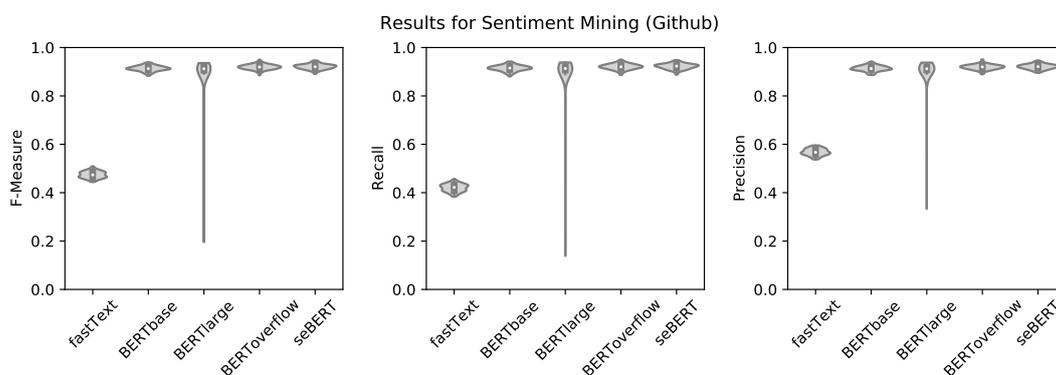


Fig. 9. Results of the sentiment mining on the GH data.

the names of many tools such as programming languages, libraries, build tools, and operating systems. Such words are mostly missing in the BERT vocabulary.

That domain specific words replace terms from other domains is not surprising. However, the magnitude of the difference between the vocabularies is larger than we would have expected: only half of the words from BERT are within the BERToverflow vocabulary, for the smaller seBERT vocabulary this drops further to 38%. Thus, less than half of the commonly used terminology from the general domain is among the commonly used terminology within the SE domain. In comparison, the lexical similarity between English and German is about 60% [58]. While this comparison is a bit unfair and likely an overstatement, because the vocabularies require exact matches of word pieces, while the lexical similarity requires only a “similarity in both form and meaning” [59], this highlights how different texts from the SE domain are from the general domain. This also demonstrates that NLP models with a fixed vocabulary should always be retrained from scratch for the SE domain to maximize the performance, instead of basing them on general domain models with additional pre-training steps, as is, e.g., done for BioBERT [7] and ClinicalBERT [8].

The MLM task demonstrates that the better representation of the SE context goes beyond the vocabulary. With SE domain statements, the general domain models often understand that they should suggest a word from the SE domain, but often do not really understand the exact meaning, which

leads to unsuitable suggestions. The SE-specific models are much better at understanding the complete context of the missing word and provide suitable suggestion. However, the examples also show that for a SE domain NLP model, data from Stack Overflow alone is not sufficient, as this only represents the domain within a Q&A context. At least some other aspects of the NLP aspects of the SE domain are not sufficiently captured by the BERToverflow model within our examples. E.g., the lack of issue handling on Stack Overflow translated into a problem with identifying the correct meaning of a missing term. This demonstrates that a larger amount of training data is beneficial and that data should be selected from a diverse range of sources. This also shows that we should be careful, when we use NLP models for tasks, where the text may be different from the pre-training texts. The additional MLM examples confirm what we expected about the SE domain models: their general language understanding is okay, but they are clearly inferior to general domain models for anything beyond the SE domain.

**Answer to RQ1:** Transformer models trained with general domain data have trouble understanding SE domain terminology. SE specific models are very good at understanding sentences within the SE context and in general language understanding, but do not perform well in contexts beyond the SE domain.

## 7.2 Impact on Applications

The results of the fine-tuned predictions show that SE domain transformer models easily are state-of-the-art for the two domain specific NLP tasks we considered, i.e., the issue type prediction and the commit intent prediction. In both tasks, seBERT and BERToverflow clearly outperform the competition and the absolute improvement is huge. Even more importantly, these models are decreasing both false negatives and false positives, i.e., they do not achieve this performance improvement through a trade-off between the errors, which indicates that this is really due to a better understanding of the language. Thus, the high validity of the domain understanding we found through the manual inspection of the vocabularies and the MLM task also translates into a better performance in relevant NLP use cases. This improvement is not explainable by the model size alone. This is demonstrated by the lower performance of the general domain BERT models. These models have the same size as our in domain-models<sup>16</sup>, but do not show the same improvement over the smaller fastText model.

Moreover, our results show that the SE domain transformers are not always better when SE data is used: we could not find notable differences between the general domain BERT models and the SE domain models for the sentiment mining tasks. Thus, SE specific pre-training only seems valuable, if the use case is specific to the SE domain and requires the interpretation of text with a SE background, which based on recent research does not seem to be the case for sentiment mining [49].

The impact of our choice to use a sequence length of only 128, in comparison to the sequence lengths of 512 from the other models is also visible in the fine-tuning. For the short commit messages of the commit intent prediction task, the seBERT model outperforms all other models, including BERToverflow. This is reasonable, because of the seBERT is the overall larger model and commit messages were used for the pre-training. We do not observe such a difference in performance for the issue type prediction task. Here, seBERT and BERToverflow are very similar to each other. This could mean that the *F1 score* of 0.8 is the performance ceiling and a better result is not possible without additional data, e.g., more training data or additional information about the issues. Alternatively, seBERT is penalized by the short sequence length of 128, which gives BERToverflow an advantage for longer issues. This could compensate the difference in model size and data for pre-training. A subgroup analysis of the predictions for issues with less than or equal to 128 tokens and longer issues did not provide conclusive evidence for this. Instead, we observed an instance of Simpson's paradox [60], i.e., seBERT outperformed BERToverflow on both subgroups, even though the overall performance is about equal. While we could consider this as an indication that the sequence length was not problematic, we cannot provide a definitive answer to this question without extending seBERT to a sequence length of 512 tokens. However, such an extension is, at this time, from our perspective not warranted due the required energy consumption (see Section 7.3) and we

16. Reminder: BERToverflow is a BERT<sub>BASE</sub> model and seBERT is a BERT<sub>LARGE</sub> model.

believe that the computational effort for this should only be spent if there is a clear expected advantage of considering longer sequences.

The difference between the BERT<sub>BASE</sub> and BERT<sub>LARGE</sub> model further shows that general-domain models – especially large general domain models – should be used with caution: often, this model converged to a trivial result where all instances were classified as negative. The logs of the training showed that this happened when the fine-tuned model was still very bad after the first epoch. In the subsequent epochs, the optimization found that a trivial model was actually better in terms of accuracy and cross-entropy than the first attempts. Unfortunately, we cannot determine the exact reason for this. However, we believe that this may be due to the combination of a small data set for fine-tuning and a lack of SE domain understanding of the general model. As a result, the training did not converge towards a reasonable solution within one epoch. Subsequently, the optimizer got stuck in the local optimum of the trivial model.<sup>17</sup> That this did not happen with the equally large seBERT provides another indication that the SE specific pre-training, in fact, the opposite happened: since the pre-training already captured the domain very well, seBERT usually achieved the optimal result within one, at most two epochs, regardless of the model size.

Together with the results from Tabassum *et al.* [10] for the NER task that showed that the BERToverflow model was also better than a BERT<sub>BASE</sub> model, we have a clear answer for RQ2.

**Answer to RQ2:** Transformer models pre-trained with SE domain data consistently outperform other models on SE use cases and should be considered as the state-of-the-art. Pre-trained models from the general domain should be used with care, especially if only a small amount of data is available for the fine-tuning. However, we also found that SE domain models perform similar to general domain models for sentiment mining on SE data, i.e., a use case that does not require SE specific knowledge.

## 7.3 Ethical Considerations

Large deep learning models for NLP, like the transformer models we consider within this work, are associated with several ethical challenges, as is, e.g., highlighted in the famous stochastic parrots paper by Bender *et al.* [61]. Due to impact of Bender *et al.* [61] and the direct relation to BERT models, we structure our consideration of ethical aspects following the four major ethical concerns highlighted in their work.

The first aspect highlighted by Bender *et al.* [61] is the energy consumption that large transformer models require. We actively considered methods to reduce the training time, e.g., by using a version of the BERT pre-training that was optimized for the hardware we were using and by restricting the sequence length to 128 tokens. Based on the

17. We cannot rule out that other training parameters (e.g., batch size, learning rate), would yield better results with BERT<sub>LARGE</sub> or any other of our models. However, this does not affect our conclusions, as we discuss in the threats to validity (see Section 7.5).

energy consumption of the system we used for training that requires between 2.5 kW and 3.5 kW when utilized fully, we estimate that we required between 270 kWh and 378 kWh for the 4.5 days of pre-training. Under load the cooling of the data center, where the compute nodes are located, has a Power Usage Effectiveness (PUE) of about 1.23. Additional overhead regarding storage and network has also to be taken into account, so we are calculating with an overhead of approximately 30%. Hence, we estimate that we consumed at most  $378 \text{ kWh} \cdot 1.30 = 491.4 \text{ kWh}$ . Based on the 366 g CO<sub>2</sub> that is generated per kWh in 2020 in Germany [62], this means the pre-training produced up to 180 kg of CO<sub>2</sub>. This is roughly the amount of the CO<sub>2</sub> for one tank filling of an average family car, which we believe is not unreasonable from an ecological perspective for a one-time effort. In comparison, Strubell *et al.* [63] estimate that they required about 1507 kWh for the training of a BERT<sub>BASE</sub> model. This means that we could train the seBERT model with only one third of the environmental impact than a normal BERT<sub>BASE</sub> model, even though we use a BERT<sub>LARGE</sub> architecture and 119.7 Gigabyte instead of 16 Gigabyte of textual data for training.

The second aspect highlighted by Bender *et al.* [61] is the quality of the training data. The authors highlight that the size of the training data does not guarantee diversity, data collected from the past can, by definition, capture how language evolves, and that there is a risk that models, therefore, capture and enforce existing biases. We did not systematically evaluate BERToverflow and seBERT for such biases. One aspect we found regardless is that “women” was not in the dictionary of seBERT. Thus, we can be quite sure, even without an in depth consideration that there is at least a severe gender bias within the data, and, consequently, within seBERT. Since BERToverflow was trained on less, but similar data, we believe that there should be a similar gender bias. We cannot comment on other biases, e.g., racial bias or similar. We also note that we cannot exclude that men is only included in the seBERT vocabulary because it is polysemous (man page). Tasks like NER or classification of bugs should not be affected by such biases and can safely be used. Tasks where bias is relevant, e.g., sentiment mining of developers, the development of chat bots or automated answering of SE questions should only be conducted after a detailed consideration of such ethical considerations. For tasks where the impact of bias is unclear, e.g., the summarizing existing texts within the SE domain, could possibly be used but we still recommend to at least conduct a basic check for such biases.

The third aspect that Bender *et al.* [61] consider is that time may be better spent than on the exploration of ever larger language models. When we transfer this to our work, this means that SE researchers should not invest too much effort into the development of NLP models for the SE domain, but rather focus on the SE data and use cases. For us, this means that we should only provide SE domain models, when machine-learning driven NLP research has major advances, instead of, e.g., trying to directly find new transformer architectures for SE benchmarks with SE data. Currently, it seems like the SE community is already using such an approach, as there are only few domain specific models and they are all re-using architectures that were

developed by researchers for the general domain (see Section 3).

The fourth issue raised by Bender *et al.* [61] is that, in the end, such large NLP models are nothing more than *stochastic parrots*, i.e., models that repeat what they have seen in the data, with some random component. This is due to our lack of understanding about the internal structure and reasoning within models with millions of parameters. This is also a property of SE domain models that should be respected for any later use of these models in an ethical way. For example, we found that the SE domain models are very good at suggesting technologies. When we use the sentence “You can use [MASK] for code coverage in java.”, seBERT and BERToverflow predict tools like Cobertura, gcov, Emma, or Jacoco. We found that this works with different languages and different kinds of tools. However, using the models within a chat bot or Q&A system to recommend suitable tools means that the past is encoded and developers of new tools would not have a chance, unless the model is retrained. In comparison, humans learn continuously about new tools, i.e., there would not be such an ethical problem. The consequence of this issue is similar to the impact of potential biases: we recommend to be careful when using the NLP models to generate responses to actual queries, unless it was determined that the possible responses are carefully validated for the given use case.

#### 7.4 Open Issues

Our work demonstrates that NLP models pre-trained with SE domain data are useful. However, there are also limitations to the understanding of transformers that we established. Since we used BERT both as general domain reference model, as well as the architecture for our models, it is unclear how the results generalize to other transformer models. While the difference to models that have a similar size like RoBERTa [18] should be relatively small, it is unclear if extremely large models like GPT-3 [19] may be able to correctly understand the meaning of texts both in the SE and general domain, same as human experts. While there is no reason that this should be the case, there is also no strong argument against this. However, currently the scale of these models is beyond almost anyone, except the largest labs and companies of the globe [64]. Therefore, even if this were the case, such improvements could currently only be harnessed by a small elite. Thus, we believe that for the majority of SE researchers and vendors who may consider building NLP capabilities into their tools, models like BERT are a more realistic, current alternative.

A related limitation is the impact of the context length on the results. Within this work, we work with a maximum context length of 512 tokens (BERT, BERToverflow) and 128 tokens (seBERT). While we argue that most SE texts in our data are shorter anyways, this is also due to the type of text we consider. If we were to, e.g., consider README files instead, we would likely have many examples of longer texts. Thus, while our results show that even the relatively short context of 128 tokens is sufficient and actually leads to the best results in both domain-dependent fine-tuning examples we consider, this may not generalize to NLP tasks on longer inputs. Especially generative tasks, like summarizing

long documents, may benefit from a longer context that is sufficient to capture the meaning of the whole text at once. In case studies find that models with shorter contexts, like BERT, do not generate suitable results for longer documents, other transformer models, like Big Bird [65], which achieves a sequence length of 4096 tokens, could be used.

Another limitation is a corollary from our statement regarding longer documents: while our corpus is already relatively diverse, with Stack Overflow, issues, and commit messages, this still does not capture the whole SE domain and, most notably, lacks examples with longer texts. Thus, even if we wanted to train a model with a longer sequence length, this could only make a difference on a small fraction of this type of SE data and the available data would likely not be sufficient to correctly model longer contextual relationships. Unfortunately, there is neither a suitable data set that could be exploited for pre-training, nor is there a benchmark task for NLP within the SE domain that requires longer texts. Thus, to advance NLP for the SE domain for tasks with long sequences, our community would first need to solve the associated data challenge, both with respect to data for pre-training, as well as through a curated data set that is suitable for the benchmarking of a fine-tuned application. These limitations are not only restricted to length, but also to tasks that actively exploit the capability of BERT models to compare two sentences by exploiting the semantics of the [SEP] token. Thus, further fine-tuning tasks that evaluate the utility of BERT for such problems, e.g., the identification of related Stack Overflow posts [66], still need to be considered. Additionally, similar fine-tuning tasks to the ones we considered to evaluate the generalization of the results to similar tasks, e.g., app review classification [67], which is similar to sentiment mining.

Finally, we already highlighted the lack of an evaluation of the SE models from an ethics perspective. While this was not within the scope of our work, future work must deal with potential biases in SE domain models, unless their usage is restricted to few and possibly uncritical use cases, as is, e.g., the case in our fine-tuned examples. We note that this does not only affect seBERT and BERToverflow, but also models like CodeBERT [68], in case this is used to generate texts, e.g., automated documentation generation. From our perspective, this is a precursor of any type of generative NLP application, i.e. NLP models that actively generate texts, e.g., to answer questions like “what does this code do” or “summarize this README file”, but also for any application like sentiment mining, which is known to encode biases [69].

## 7.5 Threats to Validity

We report the threats to the validity of our work following the classification by [70] suggested for software engineering by [71]. Additionally, we discuss the reliability as suggested by [72].

### 7.5.1 Construct Validity

The construct of our validation of NLP models may be unsuitable. The direct comparison of WordPiece vocabularies neglects to account for the internal structure of the models. For example, the internal structure could achieve

that the combination of tokens “wo##men” is the same as having the token “women” directly in the vocabulary. We address this issue by not only considering the overlap and tokenization, but also through a qualitative analysis of the non-overlapping words. Our data indicates that these are mostly domain specific terms, which makes sense given the training data and also means that an effect where these words are known by the model, regardless of them missing in the vocabulary. Similarly, our study of the contextual interpretation of sentences and words through the MLM task may be unsuitable. Our lack of consideration of different parameters for training (batch sizes, learning rates, etc.) may lead to sub-optimal models after fine-tuning, which could affect our conclusions.

### 7.5.2 Internal Validity

While we have good reason to believe that the differences between the models we observe are due to the difference in the data used for pre-training the models, we cannot rule out that there may be other reasons for these differences, due to the black box nature of the models. Most threats to the internal validity should be mitigated by our construct: a pure analysis of the vocabulary may show artificial differences, but this would not explain the differences we observe with the MLM and fine-tuning tasks. And while few random differences for the MLM and fine-tuning may be explainable by alternative hypotheses, we observe clear patterns that match our expectations, including the lack of big differences between the models when general language understanding is required. We also did not conduct an extensive hyperparameter search for the fine-tuning of the models. This means we potentially underestimate the models performance, such that differences between the models could potentially increase or decrease, with other parameters, e.g., learning rates. However, since all models use similar architectures (BERT), it is likely that they work best with similar hyperparameters for the same problem. Thus, even if our hyperparameters should be suboptimal, it is likely that the differences we observe would be preserved with different hyperparameters. Consequently, we believe that we mitigated most notable threats to the internal validity, other than the limitations of our study we acknowledge in Section 7.4.

There is one notable threat to the internal validity of our fine-tuning results, due to the way seBERT was created. The corpus used for pre-training included commit messages from Github, data from Stack Overflow, and Jira issues, some of which may be part of the test data we use for fine-tuning tasks. Since the pre-training was self-supervised based on MLM and NSP, no information about labels was part of the creation of the models. Thus, there cannot be an information leak that affects our results. However, we cannot rule out the possibility that the fine-tuning works better with data seen during the pre-training, because it is more likely that the language structure is already known by the model. However, we believe that the threat is relatively small, due to the following reasons. First, having seen the data could also be a problem as well: beyond language understanding, having seen data without the labels before can only help with memorization, not with generalization. Consequently, results on test data could actually be worse

due to this. Second, while some data may have been during the pre-training, we also have fine-tuning tasks, where none (API sentiments), or only little data was part of the pre-training (prediction of bug issues<sup>18</sup>). Third, BERToverflow used different data and can only have such an overlap for the sentiment mining on SO task, but there is no indication that this is an advantage for seBERT for the other tasks. Fourth, note that the data we used for fine-tuning is tiny (a couple of Megabytes) compared to the data used for pre-training (e.g., 2.4 Terabyte for seBERT). Thus, it is unlikely that the pre-training was significantly influenced by this data, making advantages unlikely.

### 7.5.3 Conclusion Validity

The statistical tests we used for the comparison of the fine-tuned models were suitable for the data and there is no threat to the conclusion validity of our study.

### 7.5.4 External Validity

Since our consideration of fine-tuning only considers five classification tasks, we cannot with certainty conclude that the SE domain models are also better than the general domain models for other SE tasks, e.g., summarizing content or comparing sequences. However, since our results indicate that SE-specific models are better at capturing SE specific aspects and because prior work also found a similar results regarding the NER task [10], we believe that this is unlikely. Moreover, as already discussed in Section 7.4, our results may not generalize to extremely large models, as they may be able to capture multiple domains correctly, due to their size. We note that our pre-training and evaluations are limited to data collected from open source projects. While many open source projects are developed professionally, there may still be issues due to unknown in-house terminology when these models are used within a proprietary context that requires such knowledge. Our results do not allow for conclusions regarding the question if fine-tuning of open source models would be sufficient for such use cases.

### 7.5.5 Reliability

The definition of the positive, neutral, and negative sentences for the assessment of the contextual embeddings may influence our results, i.e., other researchers would almost certainly not have selected exactly the same sentences as us and, therefore, may have a different view on the capability of the models with respect to understanding the context of statements. However, while other sentences could always lead to differences, we note that we did not conduct any cherry picking to show the most differences, but rather first defined the sentences and then applied the models. Moreover, our playground provides anyone with the ability to evaluate the differences in the predictions of masked words between the models.<sup>19</sup> Additionally, we augmented the analysis on the curated corpus with a pure empirical analysis, in which we randomly sampled sentences with words with special meaning in the SE domain and then

18. The Jira Issues only contain data until 2014 and the difference in performance of predictions is only later data is considered is small [11]

19. <https://smartsark2.informatik.uni-goettingen.de/sebert/index.html>

evaluated the ability of the models to predict the missing words. While this does not give us insights into the inner workings of the models, these empirical results provide further evidence that our results are not biased by our selection.

## 8 CONCLUSION

Within our work, we find that NLP for SE can benefit from transformer models pre-trained with SE data. While this was already known for applications that include source code, there was only little evidence for other natural language tasks. Through our work we not only explored the differences in the expected performance of applications, but also the tried to understand if the models really have a better understanding of the SE domain. For this, we manually compared the vocabularies of general domain and SE domain BERT models and compared how these models completed sentences with masked words. This analysis showed that while general domain models have a rough but imprecise understanding of the SE domain, the SE models are more precise. This improved understanding also led to significantly better results in fine-tuned within the SE domain, but not in a general task like sentiment mining applied to SE data. In conclusion, we recommend to ensure that a large amount of SE data was used for pre-training large NLP models, when these models are used for SE tasks.

## ACKNOWLEDGMENTS

The authors would like to thank the GWDG for their support regarding the usage of the GPU resources required for this article.

## REFERENCES

- [1] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.
- [2] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Minneapolis, Minnesota: Association for Computational Linguistics, Jun. 2019, pp. 4171–4186. [Online]. Available: <https://aclanthology.org/N19-1423>
- [3] Z. Yang, Z. Dai, Y. Yang, J. Carbonell, R. Salakhutdinov, and Q. V. Le, *XLNet: Generalized Autoregressive Pretraining for Language Understanding*. Red Hook, NY, USA: Curran Associates Inc., 2019.
- [4] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," 2019.
- [5] S. Nagel, "Cc-news," 2016. [Online]. Available: <https://commoncrawl.org/2016/10/news-dataset-available/>
- [6] I. Beltagy, K. Lo, and A. Cohan, "Scibert: Pretrained language model for scientific text," in *EMNLP*, 2019.
- [7] J. Lee, W. Yoon, S. Kim, D. Kim, S. Kim, C. H. So, and J. Kang, "BioBERT: a pre-trained biomedical language representation model for biomedical text mining," *Bioinformatics*, vol. 36, no. 4, pp. 1234–1240, 09 2019. [Online]. Available: <https://doi.org/10.1093/bioinformatics/btz682>
- [8] K. Huang, J. Altsosaar, and R. Ranganath, "Clinicalbert: Modeling clinical notes and predicting hospital readmission," *arXiv:1904.05342*, 2019.
- [9] V. Efstathiou, C. Chatzilenas, and D. Spinellis, "Word embeddings for the software engineering domain," in *Proceedings of the 15th International Conference on Mining Software Repositories*, ser. MSR '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 38–41. [Online]. Available: <https://doi.org/10.1145/3196398.3196448>

- [10] J. Tabassum, M. Maddela, W. Xu, and A. Ritter, "Code and named entity recognition in StackOverflow," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Online: Association for Computational Linguistics, Jul. 2020, pp. 4913–4926. [Online]. Available: <https://www.aclweb.org/anthology/2020.acl-main.443>
- [11] S. Herbold, A. Trautsch, and F. Trautsch, "On the feasibility of automated prediction of bug and non-bug issues," *Empirical Software Engineering*, vol. 25, no. 6, pp. 5333–5369, Sep. 2020. [Online]. Available: <https://doi.org/10.1007/s10664-020-09885-w>
- [12] 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*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 392–401. [Online]. Available: <http://dl.acm.org/citation.cfm?id=2486788.2486840>
- [13] A. Trautsch, J. Erbel, S. Herbold, and J. Grabowski, "On the differences between quality increasing and other changes in open source java projects," 2021.
- [14] T. Zhang, B. Xu, F. Thung, S. A. Haryono, D. Lo, and L. Jiang, "Sentiment analysis for software engineering: How far can pre-trained transformer models go?" in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2020, pp. 70–80.
- [15] J. Eisenstein, *Introduction to Natural Language Processing*, ser. Adaptive Computation and Machine Learning series. MIT Press, 2019.
- [16] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer, "BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Online: Association for Computational Linguistics, Jul. 2020, pp. 7871–7880. [Online]. Available: <https://aclanthology.org/2020.acl-main.703>
- [17] Z. Lan, M. Chen, S. Goodman, K. Gimpel, P. Sharma, and R. Soricut, "Albert: A lite bert for self-supervised learning of language representations," in *International Conference on Learning Representations*, 2020. [Online]. Available: <https://openreview.net/forum?id=H1eA7AEtvS>
- [18] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," 2019.
- [19] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," 2020.
- [20] W. Fedus, B. Zoph, and N. Shazeer, "Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity," 2021.
- [21] L. Tunstall, L. von Werra, and T. Wolf, *Natural Language Processing with Transformers*. O'Reilly Media, Inc., 2021 (early access).
- [22] J. Vig, "A multiscale visualization of attention in the transformer model," in *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, 2019, pp. 37–42.
- [23] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," 2013.
- [24] M. Thelwall, K. Buckley, G. Paltoglou, D. Cai, and A. Kappas, "Sentiment strength detection in short informal text," *Journal of the American Society for Information Science and Technology*, vol. 61, no. 12, pp. 2544–2558, 2010. [Online]. Available: <https://onlinelibrary.wiley.com/doi/abs/10.1002/asi.21416>
- [25] M. R. Islam and M. F. Zibran, "Sentistrength-se: Exploiting domain specificity for improved sentiment analysis in software engineering text," *Journal of Systems and Software*, vol. 145, pp. 125–146, 2018. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0164121218301675>
- [26] T. Ahmed, A. Bosu, A. Iqbal, and S. Rahimi, "Sentric: A customized sentiment analysis tool for code review interactions," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2017, pp. 106–111.
- [27] F. Calefato, F. Lanubile, F. Maiorano, and N. Novielli, "Sentiment polarity detection for software development," *Empirical Softw. Engg.*, vol. 23, no. 3, p. 1352–1382, jun 2018. [Online]. Available: <https://doi.org/10.1007/s10664-017-9546-9>
- [28] E. Biswas, M. E. Karabulut, L. Pollock, and K. Vijay-Shanker, "Achieving reliable sentiment analysis in the software engineering domain using bert," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2020, pp. 162–173.
- [29] M. E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer, "Deep contextualized word representations," in *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*. New Orleans, Louisiana: Association for Computational Linguistics, Jun. 2018, pp. 2227–2237. [Online]. Available: <https://aclanthology.org/N18-1202>
- [30] S. Pyysalo, F. Ginter, H. Moen, T. Salakoski, and S. Ananiadou, "Distributional semantics resources for biomedical text processing," in *Proceedings of LBM 2013*, 2013, pp. 39–44.
- [31] S. Ghosh, P. Chakraborty, E. Cohn, J. S. Brownstein, and N. Ramakrishnan, "Characterizing diseases from unstructured text: A vocabulary driven word2vec approach," 2016.
- [32] D. Araci, "Finbert: Financial sentiment analysis with pre-trained language models," *arXiv:1908.10063*, 2019.
- [33] A. Ferrari and A. Esuli, "An NLP approach for cross-domain ambiguity detection in requirements engineering," *Automated Software Engineering*, vol. 26, no. 3, pp. 559–598, Jun. 2019. [Online]. Available: <https://doi.org/10.1007/s10515-019-00261-7>
- [34] B. Theeten, F. Van deputte, and T. Van Cutsem, "Import2vec learning embeddings for software libraries," in *Proceedings of the 16th International Conference on Mining Software Repositories*, ser. MSR '19. IEEE Press, 2019, p. 18–28. [Online]. Available: <https://doi.org/10.1109/MSR.2019.00014>
- [35] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "Code2vec: Learning distributed representations of code," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, Jan. 2019. [Online]. Available: <https://doi.org/10.1145/3290353>
- [36] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "Codebert: A pre-trained model for programming and natural languages," 2020.
- [37] K. Clark, M.-T. Luong, Q. V. Le, and C. D. Manning, "Electra: Pre-training text encoders as discriminators rather than generators," 2020.
- [38] Stack Exchange, "Stack Exchange Data Dump," 2014. [Online]. Available: <https://archive.org/details/stackexchange>
- [39] Internet Archive, "Internet Archive," 2010. [Online]. Available: <https://archive.org/index.php>
- [40] Stack Exchange, "Stack Overflow public dataset," 2016. [Online]. Available: <https://console.cloud.google.com/marketplace/product/stack-exchange/stack-overflow>
- [41] I. Grigorik, "The GitHub Archive," 2012. [Online]. Available: <https://www.gharchive.org/>
- [42] Google Cloud, "BigQuery - Google Cloud Platform." 2011. [Online]. Available: <https://cloud.google.com/bigquery/>
- [43] M. Ortu, G. Destefanis, B. Adams, A. Murgia, M. Marchesi, and R. Tonelli, "The jira repository dataset: Understanding social aspects of software development," in *Proceedings of the 11th International Conference on Predictive Models and Data Analytics in Software Engineering*, ser. PROMISE '15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: <https://doi.org/10.1145/2810146.2810147>
- [44] Y. You, J. Li, S. Reddi, J. Hseu, S. Kumar, S. Bhojanapalli, X. Song, J. Demmel, K. Keutzer, and C.-J. Hsieh, "Large batch optimization for deep learning: Training bert in 76 minutes," in *International Conference on Learning Representations*, 2020. [Online]. Available: <https://openreview.net/forum?id=Syx4wnEtvH>
- [45] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, and H. Wu, "Mixed precision training," in *International Conference on Learning Representations*, 2018. [Online]. Available: <https://openreview.net/forum?id=r1gs9JgRZ>
- [46] A. Wang, A. Singh, J. Michael, F. Hill, O. Levy, and S. R. Bowman, "GLUE: A multi-task benchmark and analysis platform for natural language understanding," in *International Conference on Learning Representations*, 2019. [Online]. Available: <https://openreview.net/forum?id=rj4km2R5t7>
- [47] A. Wang, Y. Pruksachatkun, N. Nangia, A. Singh, J. Michael, F. Hill, O. Levy, and S. R. Bowman, *SuperGLUE: A Stickier Benchmark for General-Purpose Language Understanding Systems*. Red Hook, NY, USA: Curran Associates Inc., 2019.
- [48] P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang, "SQuAD: 100,000+ questions for machine comprehension of text," in

- Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*. Austin, Texas: Association for Computational Linguistics, Nov. 2016, pp. 2383–2392. [Online]. Available: <https://aclanthology.org/D16-1264>
- [49] N. Novielli, F. Calefato, F. Lanubile, and A. Serebrenik, "Assessment of off-the-shelf SE-specific sentiment analysis tools: An extended replication study," *Empirical Software Engineering*, vol. 26, no. 4, Jun. 2021. [Online]. Available: <https://doi.org/10.1007/s10664-021-09960-w>
- [50] N. Novielli, D. Girardi, and F. Lanubile, "A benchmark study on sentiment analysis for software engineering research," in *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, 2018, pp. 364–375.
- [51] B. Lin, F. Zampetti, G. Bavota, M. Di Penta, M. Lanza, and R. Oliveto, "Sentiment analysis for software engineering: How far can we go?" in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 94–104. [Online]. Available: <https://doi.org/10.1145/3180155.3180195>
- [52] Facebook AI Research, "fasttext - library for efficient text classification and representation learning," <https://fasttext.cc/>, 2019, [accessed 14-November-2019].
- [53] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y.-G. Guéhéneuc, "Is it a bug or an enhancement?: A text-based approach to classify change requests," in *Proceedings of the 2008 Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds*, ser. CASCON '08. New York, NY, USA: ACM, 2008, pp. 23:304–23:318. [Online]. Available: <http://doi.acm.org/10.1145/1463788.1463819>
- [54] S. Herbold, A. Trautsch, and F. Trautsch, "Issues with szz: An empirical assessment of the state of practice of defect prediction data collection," 2020.
- [55] A. Benavoli, G. Corani, J. Demšar, and M. Zaffalon, "Time for a change: a tutorial for comparing multiple classifiers through bayesian analysis," *Journal of Machine Learning Research*, vol. 18, no. 77, pp. 1–36, 2017. [Online]. Available: <http://jmlr.org/papers/v18/16-305.html>
- [56] G. Uddin and F. Khomh, "Automatic mining of opinions expressed about apis in stack overflow," *IEEE Transactions on Software Engineering*, vol. 47, no. 3, pp. 522–559, 2021.
- [57] N. Novielli, F. Calefato, D. Dongiovanni, D. Girardi, and F. Lanubile, *Can We Use SE-Specific Sentiment Analysis Tools in a Cross-Platform Setting?* New York, NY, USA: Association for Computing Machinery, 2020, p. 158–168. [Online]. Available: <https://doi.org/10.1145/3379597.3387446>
- [58] "Ethnologue entry for english, 24th edition," 2021. [Online]. Available: <https://www.ethnologue.com/language/eng>
- [59] "Ethnologue, 24th edition," 2021. [Online]. Available: <https://www.ethnologue.com/about/language-info>
- [60] C. R. Blyth, "On simpson's paradox and the sure-thing principle," *Journal of the American Statistical Association*, vol. 67, no. 338, pp. 364–366, 1972.
- [61] E. M. Bender, T. Gebru, A. McMillan-Major, and S. Shmitchell, "On the dangers of stochastic parrots: Can language models be too big?" in *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency*, ser. FAccT '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 610–623. [Online]. Available: <https://doi.org/10.1145/3442188.3445922>
- [62] P. Icha, T. Lauf, and G. Kuhs, "Entwicklung der spezifischen Kohlendioxid-Emissionen des deutschen Strommix in den Jahren 1990 - 2019," 2021.
- [63] E. Strubell, A. Ganesh, and A. McCallum, "Energy and policy considerations for deep learning in NLP," in *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. Florence, Italy: Association for Computational Linguistics, Jul. 2019, pp. 3645–3650. [Online]. Available: <https://www.aclweb.org/anthology/P19-1355>
- [64] V. J. Hellendoorn and A. A. Sawant, "The growing cost of deep learning for source code," *Commun. ACM*, vol. 65, no. 1, p. 31–33, dec 2021. [Online]. Available: <https://doi.org/10.1145/3501261>
- [65] M. Zaheer, G. Guruganesh, K. A. Dubey, J. Ainslie, C. Alberti, S. Ontanon, P. Pham, A. Ravula, Q. Wang, L. Yang, and A. Ahmed, "Big bird: Transformers for longer sequences," in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, Eds., vol. 33. Curran Associates, Inc., 2020, pp. 17 283–17 297. [Online]. Available: <https://proceedings.neurips.cc/paper/2020/file/c8512d142a2d849725f31a9a7a361ab9-Paper.pdf>
- [66] B. Xu, A. Shirani, D. Lo, and M. A. Alipour, "Prediction of relatedness in stack overflow: Deep learning vs. svm: A reproducibility study," in *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: <https://doi.org/10.1145/3239235.3240503>
- [67] W. Maalej, Z. Kurtanović, H. Nabil, and C. Stanik, "On the automatic classification of app reviews," *Requirements Engineering*, vol. 21, no. 3, pp. 311–331, May 2016. [Online]. Available: <https://doi.org/10.1007/s00766-016-0251-9>
- [68] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*. Online: Association for Computational Linguistics, Nov. 2020, pp. 1536–1547. [Online]. Available: <https://aclanthology.org/2020.findings-emnlp.139>
- [69] S. Kiritchenko and S. Mohammad, "Examining gender and race bias in two hundred sentiment analysis systems," in *Proceedings of the Seventh Joint Conference on Lexical and Computational Semantics*. New Orleans, Louisiana: Association for Computational Linguistics, Jun. 2018, pp. 43–53. [Online]. Available: <https://aclanthology.org/S18-2005>
- [70] T. D. Cook, D. T. Campbell, and A. Day, *Quasi-experimentation: Design & analysis issues for field settings*. Houghton Mifflin Boston, 1979, vol. 351.
- [71] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslen, *Experimentation in Software Engineering*. Springer Publishing Company, Incorporated, 2012.
- [72] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Software Engineering*, vol. 14, no. 2, pp. 131–164, 2009.



**Julian von der Mosel** Julian von der Mosel is currently working as a data engineer and received his B.Sc. degree from the University of Goettingen.



**Alexander Trautsch** Alexander Trautsch is a PhD candidate at the University of Goettingen.



**Steffen Herbold** Steffen Herbold is professor for Methods and Applications of Machine learning at the TU Clausthal.

## **F Analysis of tangled changes**

This section contains a copy of the following publication.

S. Herbold, A. Trautsch, B. Ledel, A. Aghamohammadi, T. Ahmed Ghaleb, K. Kaur Chahal, T. Bossenmaier, B. Nagaria, P. Makedonski, M. Nili Ahmadabadi, K. Szabados, H. Spieker, M. Madeja, N. Hoy, V. Lenarduzzi, S. Wang, G. Rodriguez Perez, R. Colomo-Palacios, R. Verdecchia, P. Singh, Y. Qin, D. Chakroborti, W. Davis, V. Walunj, H. Wu, D. Marcilio, O. Alam, A. Aldaej, I. Amit, B. Turhan, S. Eismann, A. Wickert, I. Malavolta, M. Sulír, F. Fard, A. Z Henley, S. Kourtzanidis, E. Tüzün, C. Treude, S. Maleki Shamasbi, I. Pashchenko, M. Wyrich, J. C. Davis, A. Serebrenik, E. Albrecht, E. Utku Aktas, D. Strüber, J. Erbel Large-Scale Manual Validation of Bug Fixing Commits: A Fine-grained Analysis of Tangling. *Empirical Software Engineering* (2022). Springer Nature

© 2022, The Author(s). Reprinted with permission.

journal doi follows after publication

preprint: <https://doi.org/10.48550/arXiv.2011.06244>

## A Fine-grained Data Set and Analysis of Tangling in Bug Fixing Commits

Steffen Herbold · Alexander Trautsch · Benjamin Ledel · Alireza Aghamohammadi · Taher Ahmed Ghaleb · Kuljit Kaur Chahal · Tim Bossenmaier · Bhaveet Nagaria · Philip Makedonski · Matin Nili Ahmadabadi · Kristof Szabados · Helge Spieker · Matej Madeja · Nathaniel Hoy · Valentina Lenarduzzi · Shangwen Wang · Gema Rodríguez-Pérez · Ricardo Colomo-Palacios · Roberto Verdecchia · Paramvir Singh · Yihao Qin · Debasish Chakroborti · Willard Davis · Vijay Walunj · Hongjun Wu · Diego Marcilio · Omar Alam · Abdullah Aldaej · Idan Amit · Burak Turhan · Simon Eismann · Anna-Katharina Wickert · Ivano Malavolta · Matúš Sulír · Fatemeh Fard · Austin Z. Henley · Stratos Kourtzanidis · Eray Tuzun · Christoph Treude · Simin Maleki Shamasbi · Ivan Pashchenko · Marvin Wyrich · James Davis · Alexander Serebrenik · Ella Albrecht · Ethem Utku Aktas · Daniel Strüber · Johannes Erbel

Received: date / Accepted: date

---

Steffen Herbold  
Institute for Software and Systems Engineering, TU Clausthal, Clausthal-Zellefeld, Germany, Germany  
E-mail: steffen.herbold@kit.edu

Alexander Trautsch  
Institute of Computer Science, University of Goettingen, Goettingen, Germany  
E-mail: alexander.trautsch@cs.uni-goettingen.de

Benjamin Ledel  
Institute for Software and Systems Engineering, TU Clausthal, Clausthal-Zellefeld, Germany, Germany  
E-mail: benjamin.ledel@tu-clausthal.de

Alireza Aghamohammadi

---

Department of Computer Engineering, Sharif University of Technology, Tehran, Iran  
E-mail: aaghamohammadi@ce.sharif.edu

Taher Ahmed Ghaleb  
School of Computing, Queen's University, Kingston, Canada  
E-mail: taher.ghaleb@queensu.ca

Kuljit Kaur Chahal, Department of Computer Science, Guru Nanak Dev University, Amritsar, India  
E-mail: kuljitahal.cse@gndu.ac.in

Tim Bossenmaier  
Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany  
E-mail: udeho@student.kit.edu

Bhaveet Nagaria  
Brunel University London, Uxbridge, United Kingdom  
E-mail: bhaveet.nagaria@brunel.ac.uk

Philip Makedonski  
Institute of Computer Science, University of Goettingen, Germany  
E-mail: makedonski@cs.uni-goettingen.de

Matin Nili Ahmadabadi  
University of Tehran, Tehran, Iran  
E-mail: matin.nili@alumni.ut.ac.ir

Kristóf Szabados  
Ericsson Hungary Ltd., Budapest, Hungary  
E-mail: kristof.szabados@ericsson.com

Helge Spieker  
Simula Research Laboratory, Fornebu, Norway  
E-mail: helge@simula.no

Matej Madeja  
Technical University of Košice, Košice, Slovakia  
E-mail: matej.madeja@tuke.sk

Nathaniel G. Hoy  
Brunel University London, Uxbridge, United Kingdom  
E-mail: nathaniel.hoy2@brunel.ac.uk

Valentina Lenarduzzi  
LUT University, Finland  
E-mail: valentina.lenarduzzi@lut.fi

Shangwen Wang  
National University of Defense Technology, Changsha, China  
E-mail: wangshangwen13@nudt.edu.cn

Gema Rodríguez-Pérez  
University of British Columbia, Kelowna, Canada  
E-mail: gema.rodriguezperez@ubc.ca

Ricardo Colomo-Palacios  
Østfold University College, Halden, Norway  
E-mail: ricardo.colomo-palacios@hiof.no

Roberto Verdecchia  
Vrije Universiteit Amsterdam, Amsterdam, The Netherlands  
E-mail: r.verdecchia@vu.nl

Paramvir Singh

---

University of Auckland, Auckland, New Zealand  
E-mail: p.singh@auckland.ac.nz

Yihao Qin  
National University of Defense Technology, Changsha, China  
E-mail: yihaoqin@nudt.edu.cn

Debasish Chakroborti  
University of Saskatchewan, Saskatoon, Canada  
E-mail: debasish.chakroborti@usask.ca

Willard Davis  
IBM, Boulder, USA  
E-mail: wdavis@us.ibm.com

Vijay Walunj  
University of Missouri-Kansas City, Kansas City, USA  
E-mail: vbwgh6@umsystem.edu

Hongjun Wu  
National University of Defense Technology, Changsha, China  
E-mail: wuhongjun15@nudt.edu.cn

Diego Marcilio  
Università della Svizzera italiana, Lugano, Switzerland  
E-mail: diego.marcilio@usi.ch

Omar Alam  
Trent University, Peterborough, Canada  
E-mail: omaram@trentu.ca

Abdullah Aldaej  
University of Maryland Baltimore County, United States and Imam Abdulrahman Bin Faisal  
University, Saudi Arabia  
E-mail: aldaej1@umbc.edu

Idan Amit  
The Hebrew University/Acumen, Jerusalem, Israel  
E-mail: idan.amit@mail.huji.ac.il

Burak Turhan  
University of Oulu, Oulu, Finland and Monash University, Melbourne, Australia  
E-mail: burak.turhan@oulu.fi

Simon Eismann  
University of Würzburg, Würzburg, Germany  
E-mail: simon.eismann@uni-wuerzburg.de

Anna-Katharina Wickert  
Technische Universität Darmstadt, Darmstadt, Germany  
E-mail: wickert@cs.tu-darmstadt.de

Ivano Malavolta  
Vrije Universiteit Amsterdam, Amsterdam, The Netherlands  
E-mail: i.malavolta@vu.nl

Matúš Sulír  
Technical University of Košice, Košice, Slovakia  
E-mail: matus.sulir@tuke.sk

Fatemeh Fard  
University of British Columbia, Kelowna, Canada  
E-mail: Fatemeh.fard@ubc.ca

**Abstract** *Context:* Tangled commits are changes to software that address multiple concerns at once. For researchers interested in bugs, tangled commits mean that they actually study not only bugs, but also other concerns irrelevant for the study of bugs.

*Objective:* We want to improve our understanding of the prevalence of tangling and the types of changes that are tangled within bug fixing commits.

*Methods:* We use a crowd sourcing approach for manual labeling to validate which changes contribute to bug fixes for each line in bug fixing commits. Each

---

Austin Z. Henley  
University of Tennessee, Knoxville, USA  
E-mail: azh@utk.edu

Stratos Kourtzanidis  
University of Macedonia, Thessaloniki, Greece  
E-mail: ekourtzanidis@uom.edu.gr

Eray Tüziin  
Department of Computer Engineering, Bilkent University, Ankara, Turkey  
E-mail: eraytuzun@cs.bilkent.edu.tr

Christoph Treude  
University of Melbourne, Melbourne, Australia  
E-mail: christoph.treude@unimelb.edu.au

Simin Maleki Shamasbi  
Independent Researcher, Tehran, Iran  
E-mail: simin.maleki@gmail.com

Ivan Pashchenko  
University of Trento, Trento, Italy  
E-mail: ivan.pashchenko@unitn.it

Marvin Wyrich  
University of Stuttgart, Stuttgart, Germany  
E-mail: marvin.wyrich@iste.uni-stuttgart.de

James C. Davis  
Purdue University, West Lafayette, IN, USA  
E-mail: davisjam@purdue.edu

Alexander Serebrenik  
Eindhoven University of Technology, Eindhoven, The Netherlands  
E-mail: a.serebrenik@tue.nl

Ella Albrecht  
Institute of Computer Science, University of Goettingen, Germany  
E-mail: ella.albrecht@cs.uni-goettingen.de

Ethem Utku Aktas  
Softtech Inc., Research and Development Center, 34947 Istanbul, Turkey  
E-mail: utku.aktas@softtech.com.tr

Daniel Strüber  
Radboud University, Nijmegen, Netherlands  
E-mail: d.strueber@cs.ru.nl

Johannes Erbel  
Institute of Computer Science, University of Goettingen, Germany  
E-mail: johannes.erbel@cs.uni-goettingen.de

line is labeled by four participants. If at least three participants agree on the same label, we have consensus.

*Results:* We estimate that between 17% and 32% of all changes in bug fixing commits modify the source code to fix the underlying problem. However, when we only consider changes to the production code files this ratio increases to 66% to 87%. We find that about 11% of lines are hard to label leading to active disagreements between participants. Due to confirmed tangling and the uncertainty in our data, we estimate that 3% to 47% of data is noisy without manual untangling, depending on the use case.

*Conclusion:* Tangled commits have a high prevalence in bug fixes and can lead to a large amount of noise in the data. Prior research indicates that this noise may alter results. As researchers, we should be skeptics and assume that unvalidated data is likely very noisy, until proven otherwise.

**Keywords** tangled changes · tangled commits · bug fix · manual validation · research turk · registered report

## 1 Introduction

Detailed and accurate information about bug fixes is important for many different domains of software engineering research, e.g., program repair (Gazzola et al., 2019), bug localization (Mills et al., 2018), and defect prediction (Hosseini et al., 2019). Such research suffers from mislabeled data, e.g., because commits are mistakenly identified as bug fixes (Herzig et al., 2013) or because not all changes within a commit are bug fixes (Herzig and Zeller, 2013). A common approach to obtain information about bug fixes is to mine software repositories for bug fixing commits and assume that all changes in the bug fixing commit are part of the bug fix, e.g., with the SZZ algorithm (Śliwerski et al., 2005). Unfortunately, prior research showed that the reality is more complex. The term *tangled commit*<sup>1</sup> was established by Herzig and Zeller (2013) to characterize the problem that commits may address multiple issues, together with the concept of untangling, i.e., the subsequent separation of these concerns. Multiple prior studies established through manual validation that tangled commits naturally occur in code bases. For example, Herzig and Zeller (2013), Nguyen et al. (2013), Kirinuki et al. (2014), Kochhar et al. (2014), Kirinuki et al. (2016), Wang et al. (2019), and Mills et al. (2020). Moreover, Nguyen et al. (2013), Kochhar et al. (2014), Herzig et al. (2016), and Mills et al. (2020) have independently shown that tangling can have a negative effect on experiments, e.g., due to noise in the training data that reduces model performance as well as due to noise in the test data which significantly affects performance estimates.

However, we identified four limitations regarding the knowledge on tangling within the current literature. The first and most common limitation of prior

---

<sup>1</sup> Herzig and Zeller (2013) actually used the term tangled change. However, we follow Rodríguez-Pérez et al. (2020) and use the term commit to reference observable groups of changes within version control systems (see Section 3.1).

work is that it either only considered a sample of commits, or that the authors failed to determine whether the commits were tangled or not for a large proportion of the data.<sup>2</sup> Second, the literature considers this problem from different perspectives: most literature considers tangling at the commit level, whereas others break this down to the file-level within commits. Some prior studies focus on all commits, while others only consider bug fixing commits. Due to these differences, in combination with limited sample sizes, the prior work is unclear regarding the prevalence of tangling. For example, Kochhar et al. (2014) and Mills et al. (2020) found a high prevalence of tangling within file changes, but their estimates for the prevalence of tangling varied substantially with 28% and 50% of file changes affected, respectively. Furthermore, how the lower boundary of 15% tangled commits that Herzig and Zeller (2013) estimated relates to the tangling of file changes is also unclear. Third, there is little work on what kind of changes are tangled. Kirinuki et al. (2014) studied the type of code changes that are often tangled with other changes and found that these are mostly logging, checking of pre-conditions, and refactorings. Nguyen et al. (2013) and Mills et al. (2020) provide estimations on the types of tangled commits on a larger scale, but their results contradict each other. For example, Mills et al. (2020) estimate six times more refactorings in tangled commits than Nguyen et al. (2013). Fourth, these studies are still relatively coarse-grained and report results at the commit and file level.

Despite the well established research on tangled commits and their impact on research results, their prevalence and content is not yet well understood, neither for commits in general, nor with respect to bug fixing commits. Due to this lack of understanding, we cannot estimate how severe the threat to the validity of experiments due to the tangling is, and how many tools developed based on tangled commits may be negatively affected.

Our lack of knowledge about tangled commits notwithstanding, researchers need data about bug fixing commits. Currently, researchers rely on three different approaches to mitigate this problem: (i) seeded bugs; (ii) no or heuristic untangling; and (iii) manual untangling. First, we have data with seeded bugs, e.g., the SIR data (Do et al., 2005), the Siemens data (Hutchins et al., 1994), and through mutation testing (Jia and Harman, 2011). While there is no risk of noise in such data, it is questionable whether the data is representative for real bugs. Moreover, applications like bug localization or defect prediction cannot be evaluated based on seeded bugs.

The second approach is to assume that commits are either not tangled or that heuristics are able to filter tangling. Examples of such data are Many-Bugs (Le Goues et al., 2015), Bugs.jar (Saha et al., 2018), as well as all defect prediction data sets (Herbold et al., 2019). The advantage of such data sets is that they are relatively easy to collect. The drawback is that the impact of noise due to tangled commits is unclear, even though modern variants of the

---

<sup>2</sup> Failing to label a proportion of the data also results in a sample, but this sample is highly biased towards changes that are simple to untangle.

SZZ algorithm can automatically filter some tangled changes like comments, whitespaces (Kim et al., 2006) or even some refactorings (Neto et al., 2018).

Third, there are also some data sets that were manually untangled. While the creation of such data is very time consuming, such data sets are the gold standard. They represent real-world defects and do not contain noise due to tangled commits. To the best of our knowledge, there are only very few such data sets, i.e., Defects4J (Just et al., 2014) with Java bugs, BugsJS (Gyimesi et al., 2019) with JavaScript bugs, and the above mentioned data by Mills et al. (2020).<sup>3</sup> Due to the effort required to manually validate changes, the gold standard data sets contain only samples of bugs from each studied project, but no data covers all reported bugs within a project, which limits the potential use cases.<sup>4</sup>

This article fills this gap in our current knowledge about tangling bug fixing commits. We provide a new large-scale data set that contains validated untangled bug fixes for the complete development history of 23 Java projects and partial data for five further projects. In comparison to prior work, we label all data on a line-level granularity. We have (i) labels for each changed line in a bug fixing commit; (ii) accurate data about which lines contribute to the semantic change of the bug fix; and (iii) the kind of change contained in the other lines, e.g., whether it is a change to tests, a refactoring, or a documentation change. This allows us not only to untangle the bug fix from all other changes, but also gives us valuable insights into the content of bug fixing commits in general and the prevalence of tangling within such commits.

Through our work, we also gain a better understanding of the limitations of manual validation for the untangling of commits. Multiple prior studies indicate that there is a high degree of uncertainty when determining whether a change is tangled or not (Herzig and Zeller, 2013; Kirinuki et al., 2014; Kirinuki et al., 2016). Therefore, we present each commit to four different participants who labeled the data independently from each other. We use the agreement among the participants to gain insights into the uncertainty involved in the labeling, while also finding lines where no consensus was achieved, i.e., that are hard for researchers to classify.

Due to the massive amount of manual effort required for this study, we employed the *research turk* approach (Herbold, 2020) to recruit participants for the labeling. The research turk is a means to motivate a large number of researchers to contribute to a common goal, by clearly specifying the complete study together with an open invitation and clear criteria for participation beforehand (Herbold et al., 2020). In comparison to surveys or other studies

---

<sup>3</sup> The cleaning performed by Mills et al. (2020) is not full untangling of the bug fixes, as all pure additions were also flagged as tangled. While this is certainly helpful for bug localization, as a file that was added as part of the bug fix cannot be localized based on an issue description, this also means that it is unclear which additions are part of the bug fix and which additions are tangled.

<sup>4</sup> We note that none of the gold standard data sets actually has the goal to contain data for all bugs of a project. This is likely also not possible due to other requirements on these data sets, e.g., the presence of failing test cases. Thus, the effort is not the only involved factor why these data sets are only samples.

where participants are recruited, participants in the research turk actively contribute to the research project, in our case by labeling data and suggesting improvements of the manuscript. As a result, 45 of the participants we recruited became co-authors of this article. Since this is, to the best of our knowledge, a new way to conduct research projects about software engineering, we also study the effectiveness of recruiting and motivating participants.

Overall, the contributions of this article are the following.

- The *Line-Labelled Tangled Commits for Java (LLTC4J)* corpus of manually validated bug fixing commits covering 2,328 bugs from 28 projects that were fixed in 3,498 commits that modified 289,904 lines. Each changed line is annotated with the type of change, i.e., whether the change modifies the source code to correct the problem causing the bug, a whitespace or documentation change, a refactoring, a change to a test, a different type of improvement unrelated to the bug fix, or whether the participants could not determine a consensus label.
- Empirical insights into the different kinds of changes within bug fixing commits which indicate that, in practice, most changes in bug fixing commits are not about the actual bug fix, but rather related changes to non-production artifacts such as tests or documentation.
- We introduce the concept of *problematic tangling* for different uses of bug data to understand the noise caused by tangled commits for different research applications.
- We find that researchers tend to unintentionally mislabel lines in about 7.9% of the cases. Moreover, we found that identifying refactorings and other unrelated changes seems to be challenging, which is shown through 14.3% of lines without agreement in production code files, most of which are due to a disagreement whether a change is part of the bug fix or unrelated.
- This is the first use of the research turk method and we showed that this is an effective research method for large-scale studies that could not be accomplished otherwise.

The remainder of this article is structured as follows. We discuss the related work in Section 2. We proceed with the description of the research protocol in Section 3. We present and discuss the results for our research questions in Section 4 and Section 5. We report the threats to the validity of our work in Section 6. Finally, we conclude in Section 7.

## 2 Related Work

We focus the discussion of the related work on the manual untangling of commits. Other aspects, such as automated untangling algorithms (e.g., Kreutzer et al., 2016; Pärtachi et al., 2020), the separation of concerns into multiple commits (e.g., Arima et al., 2018; Yamashita et al., 2020), the tangling of features with each other (Strüder et al., 2020), the identification of bug fixing or inducing commits (e.g., Rodríguez-Pérez et al., 2020), or the characterization of commits in general (e.g., Hindle et al., 2008), are out of scope.

## 2.1 Magnitude of Tangling

The tangling of commits is an important issue, especially for researchers working with software repository data, that was first characterized by Kawrykow and Robillard (2011). They analyzed how often non-essential commits occur in commit histories, i.e., commits that do not modify the logic of the source code and found that up to 15.5% of all commits are non-essential. Due to the focus on non-essential commits, the work by Kawrykow and Robillard (2011) only provides a limited picture on tangled commits in general, as tangling also occurs if logical commits for multiple concerns are mixed within a single commit.

The term *tangled change* (commit) was first used in the context of repository mining by Herzig and Zeller (2013) (extended in Herzig et al. (2016)<sup>5</sup>). The term tangling itself was already coined earlier in the context of the separation of concerns (e.g., Kiczales et al., 1997). Herzig and Zeller (2013) studied the commits of five Java projects over a period of at least 50 months and tried to manually classify which of the commits were tangled, i.e., addressed multiple concerns. Unfortunately, they could not determine if the commits are tangled for 2,498 of the 3,047 bug fixing commits in their study. Of the bug fixing commits they could classify, they found that 298 were tangled and 251 were not tangled. Because they could not label large amounts of the commits, they estimate that at least 15% of the bug fixing commits are tangled.

Nguyen et al. (2013) also studied tangling in bug fixing commits, but used the term mixed-purpose fixing commits. They studied 1296 bug fixing commits from eight projects and identified 297 tangled commits, i.e., 22% of commits that are tangled. A strength of the study is that the authors did not only label the tangled commits, but also identified which file changes in the commit were tangled. Moreover, Nguyen et al. (2013) also studied which types of changes are tangled and found that 4.9% were due to unrelated improvements, 1.1% due to refactorings, 1.8% for formatting issues, and 3.5% for documentation changes unrelated to the bug fix.<sup>6</sup> We note that it is unclear how many of the commits are affected by multiple types of tangling and, moreover, that the types of tangling were analyzed for only about half of the commits. Overall, this is among the most comprehensive studies on this topic in the literature. However, there are two important issues that are not covered by Nguyen et al. (2013). First, they only explored if commits and files are tangled, but not how much of a commit or file is tangled. Second, they do not differentiate between problematic tangling and benign tangling (see Section 4.2.2), and, most importantly, between tangling in production files and tangling in other files. Thus, we cannot estimate how strongly tangling affects different research purposes.

---

<sup>5</sup> In the following, we cite the original paper, as the extension did not provide further evidence regarding the tangling.

<sup>6</sup> These percentages are not reported in the paper. We calculated them from their Table III. We combined enhancement and annotations into unrelated improvements, to be in line with our work.

Kochhar et al. (2014) investigated the tangling of a random sample of the bug fixing commits for 100 bugs. Since they were interested in the implications of tangling on bug localization, they focused on finding out how many of the changed files actually contributed to the correction of the bug, and how many file changes are tangled, i.e., did not contribute to the bug fix. They found that 358 out of 498 file changes were corrective, the remaining 140 changes, i.e., 28% of all file changes in bug fixing commits were tangled.

Kirinuki et al. (2014) and Kirinuki et al. (2016) are a pair of studies with a similar approach: they used an automated heuristic to identify commits that may be tangled and then manually validated if the identified commits are really tangled. Kirinuki et al. (2014) identified 63 out of 2,000 commits as possibly tangled and found through manual validation that 27 of these commits were tangled and 23 were not tangled. For the remaining 13 commits, they could not decide if the commits are tangled. Kirinuki et al. (2016) identified 39 out of 1,000 commits as potentially tangled and found through manual validation that 21 of these commits were tangled, while 7 were not tangled. For the remaining 11 commits, they could not decide. A notable aspect of the work by Kirinuki et al. (2014) is that they also analyzed what kind of changes were tangled. They found that tangled changes were mostly due to logging, condition checking, or refactorings. We further note that these studies should not be seen as an indication of low prevalence of tangled commits, just because only 48 out of 3,000 commits were found to be tangled, since only 102 commits were manually validated. These commits are not from a representative random sample, but rather from a sample that was selected such that the commits should be tangled according to a heuristic. Since we have no knowledge about how good the heuristic is at identifying tangled commits, it is unclear how many of the 2,898 commits that were not manually validated are also tangled.

Tao and Kim (2015) investigated the impact of tangled commits on code review. They manually investigated 453 commits and found that 78 of these commits were tangled in order to determine ground truth data for the evaluation of their code review tool. From the description within the study, it is unclear if the sample of commits is randomly selected, if these are all commits in the four target projects within the study time frame that changed multiple lines, or if a different sampling strategy was employed. Therefore, we cannot conclude how the prevalence of tangling in about 17% of the commits generalizes beyond the sample.

Wang et al. (2019) also manually untangled commits. However, they had the goal to create a data set for the evaluation of an automated algorithm for the untangling of commits. They achieved this by identifying 50 commits that they were sure were tangled commits, e.g., because they referenced multiple issues in the commit message. They hired eight graduate students to perform the untangling in pairs. Thus, they employed an approach for the untangling that is also based on the crowd sourcing of work, but within a much more closely controlled setting and with only two participants per commit instead of four participants. However, because Wang et al. (2019) only studied tangled

commits, no conclusions regarding the prevalence of the tangling can be drawn based on their results.

Mills et al. (2020) extends Mills et al. (2018) and contains the most comprehensive analysis of tangled commits to date. They manually untangled the changes for 803 bugs from fifteen different Java projects. They found that only 1,154 of the 2,311 changes of files within bug fixing commits contributed to bug fixes, i.e., a prevalence of 50% of tangling. Moreover, they provide insights into what kinds of changes are tangled: 31% of tangled changes are due to tests, 9% due to refactorings, and 8% due to documentation. They also report that 47% of the tangled changes are due to adding source code. Unfortunately, Mills et al. (2020) flagged all additions of code as tangled changes. While removing added lines makes sense for the use case of bug localization, this also means that additions that are actually the correction of a bug would be wrongly considered to be tangled changes. Therefore, the actual estimation of prevalence of tangled changes is between 32% and 50%, depending on the number of pure additions that are actually tangled. We note that the findings by Mills et al. (2020) are not in line with prior work, i.e., the ratio of tangled file changes is larger than the estimation by Kochhar et al. (2014) and the percentages for types of changes are different from the estimations by Nguyen et al. (2013).

Dias et al. (2015) used a different approach to get insights into the tangling of commits: they integrated an interface for the untangling of commits directly within an IDE and asked the developers to untangle commits when committing their local changes to the version control system. They approached the problem of tangled commits by grouping related changes into clusters. Unfortunately, the focus of the study is on the developer acceptance, as well as on the changes that developers made to the automatically presented clusters. It is unclear if the clusters are fully untangled, and also if all related changes are within the same cluster. Consequently, we cannot reliably estimate the prevalence or contents of tangled commits based on their work.

## 2.2 Data Sets

Finally, there are data sets of bugs where the data was untangled manually, but where the focus was only on getting the untangled bug fixing commits, not on the analysis of the tangling. Just et al. (2014) did this for the Defects4J data and Gyimesi et al. (2019) for the BugsJS data. While these data sets do not contain any noise, they have several limitations that we overcome in our work. First, we do not restrict the bug fixes but allow all bugs from a project. Defects4J and BugsJS both only allow bugs that are fixed in a single commit and also require that a test that fails without the bug fix was added as part of the bug fix. While bugs that fulfill these criteria were manually untangled for Defects4J, BugJS has additional limitations. BugJS requires that bug fixes touch at most three files, modify at most 50 lines of code, and do not contain any refactorings or other improvements unrelated to the bug fix. Thus, BugsJS is rather a sample of bugs that are fixed in untangled commits than a sample

of bugs that was manually untangled. While these data sets are gold standard untangled data sets, they are not suitable to study tangling. Moreover, since both data sets only contain samples, they are not suitable for research that requires all bugs of a specific release or within a certain time frame of a project. Therefore, our data is suitable for more kinds of research than Defects4J and BugsJS, and because our focus is not solely on the creation of the data set, but also on the understanding of tangling within bug fixing commits, we also provide a major contribution to the knowledge about tangled commits.

### 2.3 Research Gap

Overall, there is a large body of work on tangling, but studies are limited in sampling strategies, inability to label all data, or because the focus was on different aspects. Thus, we cannot form conclusive estimates, neither regarding the types of tangled changes, nor regarding the general prevalence of tangling within bug fixing commits.

## 3 Research Protocol

Within this section, we discuss the research protocol, i.e., the research strategy we pre-registered (Herbold et al., 2020) to study our research questions and hypotheses. The description and section structure of our research protocol is closely aligned with the pre-registration, but contains small deviations, e.g., regarding the sampling strategy. All deviations are described as part of this section and summarized in Section 3.8.

### 3.1 Terminology

We use the term “the principal investigators” to refer to the authors of the registered report, “the manuscript” to refer to this article that resulted from the study, and “the participants” to refer to researchers and practitioners who collaborated on this project, many of whom are now co-authors of this article.

Moreover, we use the term “commit” to refer to observable changes within the version control system, which is in line with the terminology proposed by Rodríguez-Pérez et al. (2020). We use the term “change” to refer to the actual modifications within commits, i.e., what happens in each line that is modified, added, or deleted as part of a commit.

### 3.2 Research Questions and Hypotheses

Our research is driven by two research questions for which we derived three hypotheses. The first research question is the following.

**RQ1:** What percentage of changed lines in bug fixing commits contributes to the bug fix and can we identify these changes?

We want to get a detailed understanding of both the prevalence of tangling, as well as what kind of changes are tangled within bug fixing commits. When we speak of contributing to the bug fix, we mean a direct contribution to the correction of the logical error. We derived two hypotheses related to this research question.

- H1** Fewer than 40% of changed lines in bug fixing commits contribute to the bug fix.
- H2** A label is a *consensus label* when at least three participants agree on it. Participants fail to achieve a consensus label on at least 10.5% of lines.<sup>7</sup>

We derived hypothesis H1 from the work by Mills et al. (2018), who found that 496 out of 1344 changes to files in bug fixing commits contributed to bug fixes.<sup>8</sup> We derived our expectation from Mills et al. (2018) instead of Herzig et al. (2013) due to the large degree of uncertainty due to unlabeled data in the work by Herzig et al. (2013). We derived H2 based on the assumption that there is a 10% chance that participants misclassify a line, even if they have the required knowledge for correct classification. We are not aware of any evidence regarding the probability of random mistakes in similar tasks and, therefore, used our intuition to estimate this number. Assuming a binomial distribution  $B(k|p, n)$  with the probability of random mislabels  $p = 0.1$  and  $n = 4$  participants that label each commit, we do not get a consensus for  $k \geq 2$  participants randomly mislabeling the line, i.e.,

$$\sum_{k=2}^4 B(k|0.1, 4) = \sum_{k=2}^4 \binom{4}{k} 0.1^k \cdot 0.9^{4-k} = 0.0523 \quad (1)$$

Thus, if we observe 10.5% of lines without consensus, this would be twice more than expected given the assumption of 10% random errors, indicating that lack of consensus is not only due to random errors. We augment the analysis of this hypothesis with a survey among participants, asking them how often they were unsure about the labels.

The second research question regards our crowd working approach to organize the manual labor required for our study.

**RQ2:** Can gamification motivate researchers to contribute to collaborative research projects?

- H3** The leaderboard motivates researchers to label more than the minimally required 200 commits.

<sup>7</sup> In the pre-registered protocol, we use the number of at least 16%. However, this was a mistake from the calculation of the binomial distribution, where we used a wrong value  $n = 5$  instead of  $n = 4$ . This was the result of our initial plan to use five participants per commit, which we later revised to four participants without updating the calculation and the hypothesis.

<sup>8</sup> The journal extension by Mills et al. (2020) was not published when we formulated our hypotheses as part of the pre-registration of our research protocol in January 2020.

We derived H3 from prior evidence that gamification (Werbach and Hunter, 2012) is an efficient method for the motivation of computer scientists, e.g., as demonstrated on Stack Overflow (Grant and Betts, 2013). Participants can view a nightly updated leaderboard, both to check their progress, as well as where they would currently be ranked in the author list. We believe that this has a positive effect on the amount of participation, which we observe through larger numbers of commits labeled than minimally required for co-authorship. We augment the analysis of this hypothesis with a survey among participants, asking them if they were motivated by the leaderboard and the prospect of being listed earlier in the author list.

### 3.3 Materials

This study covers bug fixing commits that we re-use from prior work (see Section 3.5.1). We use SmartSHARK to process all data (Trautsch et al., 2018). We extended SmartSHARK with the capability to annotate the changes within commits. This allowed us to manually validate which lines in a bug fixing commit are contributing to the semantic changes for fixing the bugs (Trautsch et al., 2020b).

### 3.4 Variables

We now state the variables we use as foundation for the construct of the analysis we conduct. The measurement of these variables is described in Section 3.6.1 and their analysis is described in Section 3.6.2.

For bug fixing commits, we measure the following variables as percentages of lines with respect to the total number of changed lines in the commit.

- Percentage contributing to bug fixes.
- Percentage of whitespace changes.
- Percentage of documentation changes.
- Percentage of refactorings.
- Percentage of changes to tests.
- Percentage of unrelated improvements.
- Percentage where no consensus was achieved (see Section 3.6.1).

We provide additional results regarding the lines without consensus to understand what the reasons for not achieving consensus are. These results are an extension of our registered protocol. However, we think that details regarding potential limitations of our capabilities as researchers are important for the discussion of research question RQ1. Concretely, we consider the following cases:

- In the registration, we planned to consider whitespace and documentation changes within a single variable. Now, we use separate variables for both. Our reason for this extension is to enable a better understanding of how

File type	Regular expression
Test	(^\ \/)(test tests test_long_running testing legacy-tests testdata test-framework derbyTesting unitTests java\ stubs test-lib src\ it src-lib-test src-test tests-src test-cactus test-data test-deprecated src_unitTests test-tools gateway-test-release-utils gateway-test-ldap nifi-mock)\ \/
Documentation	(^\ \/)(doc docs example examples sample samples demo tutorial helloworld userguide showcase SafeDemo)\ \/
Other	(^\ \/)(_site auxiliary-builds gen-java external nifi-external)\ \/

Table 1: Regular expressions for excluding non-production code files. These regular expressions are valid for a superset of our data and were manually validated as part of prior work Trautsch et al. (2020a).

many changes are purely cosmetic without affecting functionality (whitespace) and distinguish this from changes that modify the documentation. We note that we also used the term “comment” instead of “documentation” within the registration. Since all comments (e.g., in code) are a form of documentation, but not all documentation (e.g., sample code) is a comment, we believe that this terminology is clearer.

- Lines where all labels are either test, documentation, or whitespace. We use this case, because our labeling tool allows labeling of all lines in a file with the same label with a single click and our tutorial describes how to do this for a test file. This leads to differences between how participants approached the labeling of test files: some participants always use the button to label the whole test file as test, other participants used a more fine-grained approach and also labeled whitespace and documentation changes within test files.
- Lines that were not labeled as bug fix by any participant. For these lines, we have consensus that this is not part of the bug fix, i.e., for our key concern. Disagreements may be possible, e.g., if some participants identified a refactoring, while others marked this as unrelated improvement.

A second deviation from our pre-registered protocol is that we present the results for the consensus for two different views on the data:

- all changes, i.e., as specified in registration; and
- only changes in Java source code files that contain production code, i.e., Java files excluding changes to tests or examples.

Within the pre-registered protocol, we do not distinguish between all changes and changes to production code. We now realize that both views are important. The view of all changes is required to understand what is actually part of bug fixing commits. The view on Java production files is important, because changes to non-production code can be easily determined automatically as not contributing to the bug fix, e.g., using regular expression matching based on the file ending “.java” and the file path to exclude folders that contain tests and example code. The view on Java production files enables us to estimate

the amount of noise that cannot be automatically removed. Within this study, we used the regular expressions shown in Table 1 to identify non-production code. We note that we have some projects which also provide web applications (e.g., JSP Wiki), that also have production code in other languages than Java, e.g., JavaScript. Thus, we slightly underestimate the amount of production code, because these lines are only counted in the overall view, and not in the production code view.

Additionally, we measure variables related to the crowd working.

- Number of commits labeled per participant.
- Percentage of correctly labeled lines per participant, i.e., lines where the label of the participant agrees with the consensus.

We collected this data using nightly snapshots of the number of commits that each participant labeled, i.e., a time series per participant.

We also conducted an anonymous survey among participants who labeled at least 200 commits with a single question to gain insights into the difficulty of labeling tangled changes in commits for the evaluation of RQ1.

- *Q1*: Please estimate the percentage of lines in which you were unsure about the label you assigned.
- *A1*: One of the following categories: 0%–10%, 11%–20%, 21%–30%, 31%–40%, 41%–50%, 51–60%, 61%–70%, 71%–80%, 81%–90%, 91%–100%.

Finally, we asked all participants who labeled at least 250 commits a second question to gain insights into the effect of the gamification:

- *Q2*: Would you have labeled more than 200 commits, if the authors would have been ordered randomly instead of by the number of commits labeled?
- *A2*: One of the following categories: Yes, No, Unsure.

### 3.5 Subjects

This study has two kinds of subjects: bugs for which the lines contributing to the fix are manually validated and the participants in the labeling who were recruited using the research turk approach.

#### 3.5.1 Bugs

We use manually validated bug fixes. For this, we harness previously manually validated issue types similar to Herzig et al. (2013) and validated trace links between commits and issues for 39 Apache projects (Herbold et al., 2019). The focus of this data set is the Java programming language. The usage of manually validated data allows us to work from a foundation of ground truth and avoids noise in our results caused by the inclusion of commits that are not fixing bugs. Overall, there are 10,878 validated bug fixing commits for 6,533 fixed bugs in the data set.

Prior studies that manually validated commits limited the scope to bugs that were fixed in a single commit, and commits in which the bug was the only referenced issue (e.g., Just et al., 2014; Gyimesi et al., 2019; Mills et al., 2020). We broaden this scope in our study and also allow issues that were addressed by multiple commits, as long as the commits only fixed a single bug. This is the case for 2,283 issues which were addressed in 6,393 commits. Overall, we include 6,279 bugs fixed in 10,389 commits in this study. The remaining 254 bugs are excluded, because the validation would have to cover the additional aspect of differentiating between multiple bug fixes. Otherwise, it would be unclear to which bug(s) the change in a line would contribute, making the labels ambiguous.

Herbold et al. (2019) used a purposive sampling strategy for the determination of projects. They selected only projects from the Apache Software Foundation, which is known for the high quality of the developed software, the many contributors both from the open source community and from the industry, as well as the high standards of their development processes, especially with respect to issue tracking Bissyandé et al. (2013). Moreover, the 39 projects cover different kinds of applications, including build systems (ant-ivy), web applications (e.g., jspwiki), database frameworks (e.g., calcite), big data processing tools (e.g., kylin), and general purpose libraries (commons). Thus, our sample of bug fixes should be representative for a large proportion of Java software. Additionally, Herbold et al. (2019) defined criteria on project size and activity to exclude very small or inactive projects. Thus, while the sample is not randomized, this purposive sampling should ensure that our sample is representative for mature Java software with well-defined development processes in line with the discussion of representativeness by Baltes and Ralph (2020).

### 3.5.2 Participants

In order to only allow participants that have a sufficient amount of programming experience, each participant must fulfill one of the following criteria: 1) an undergraduate degree majoring in computer science or a closely related subject; or 2) at least one year of programming experience in Java, demonstrated either by industrial programming experience using Java or through contributions to Java open source projects.

Participants were regularly recruited, e.g., by advertising during virtual conferences, within social media, or by asking participants to invite colleagues. Interested researchers and practitioners signed up for this study via an email to the first author, who then checked if the participants are eligible. Upon registration, participants received guidance on how to proceed with the labeling of commits (see Appendix A). Participants became co-authors of this manuscript if

1. they manually labeled at least 200 commits;
2. their labels agree with the consensus (Section 3.6.1) for at least 70% of the labeled lines;

The screenshot shows the VisualSHARK tool interface. At the top, there is a legend with seven colored circles and their corresponding labels: 1 (red) bug fix, 2 (grey) whitespace, 3 (black) documentation, 4 (blue) refactoring, 5 (green) test, 6 (orange) unrelated, and 7 (white) remove current label. Below the legend, there is a text instruction: "Press the key of the color to label the current line with the belonging label, press 7 to remove the label". A navigation bar contains "Mark file as" with buttons for "whitespace", "documentation", "test", and "unrelated", along with "Jump to top", "< Previous", and "Next >". The main area displays a side-by-side diff of Java code from the file "gora-cassandra/src/main/java/org/apache/gora/cassandra/store/CassandraStore.java". The left pane shows the original code, and the right pane shows the modified code. Line 393 in the original code is highlighted in red and labeled "bug fix", while line 393 in the modified code is highlighted in green and labeled "test".

Fig. 1: Screenshot of VisualSHARK (Trautsch et al., 2020b) that was used for the labeling of the data.

3. they contributed to the manuscript by helping to review and improve the draft, including the understanding that they take full responsibility for all reported results and that they can be held accountable with respect to the correctness and integrity of the work; and
4. they were not involved in the review or decision of acceptance of the registered report.

The first criterion guarantees that each co-author provided a significant contribution to the analysis of the bug fixing commits. The second criterion ensures that participants carefully labeled the data, while still allowing for disagreements. Only participants who fulfill the first two criteria received the manuscript for review. The third criterion ensures that all co-authors agree with the reported results and the related responsibility and ethical accountability. The fourth criterion avoids conflicts of interest.<sup>9</sup>

### 3.6 Execution Plan

The execution of this research project was divided into two phases: the data collection phase and the analysis phase.

#### 3.6.1 Data Collection Phase

The primary goal of this study is to gain insights into which changed lines contribute to bug fixing commits and which additional activities are tangled

<sup>9</sup> Because the review of the registered report was blinded, the fulfillment of this criterion is checked by the editors of the Empirical Software Engineering journal.

with the correction of the bug. Participants were shown the textual differences of the source code for each bug with all related bug fixing commits. The participants then assigned one of the following labels to all changed lines:

- contributes to the bug fix;
- only changes to whitespaces;
- documentation change;
- refactoring;
- change to tests; and
- unrelated improvement not required for the bug fix.

Figure 1 shows a screenshot of the web application that we used for labeling. The web application ensured that all lines were labeled, i.e., participants could not submit incomplete labels for a commit. The web application further supported the participants with the following convenience functions:

- Buttons to mark all changes in a file with the same label. This could, e.g., be used to mark all lines in a test file with a single click as test change.
- Heuristic pre-labeling of lines as documentation changes using regular expressions.
- Heuristic pre-labeling of lines with only whitespace changes.
- Heuristic pre-labeling of lines as refactoring by automatically marking changed lines as refactorings, in case they were detected as refactorings by the RefactoringMiner 1.0 (Tsantalis et al., 2018).

All participants were instructed not to trust the pre-labels and check if these labels are correct. Moreover, we did not require differentiation between whitespaces and documentation/test changes in files that were not production code files, e.g., within test code or documentation files such as the project change log.

Each commit was shown to four participants. Consensus is achieved if at least three participants agree on the same label. If this is not the case, no consensus for the line is achieved, i.e., the participants could not clearly identify which type of change a line is.

The data collection phase started on May 16th, 2020 and ended on October 14th, 2020, registration already finished on September 30th, 2020.<sup>10</sup> The participants started by watching a tutorial video<sup>11</sup> and then labeling the same five commits that are shown in the tutorial themselves to get to know the system and to avoid mislabels due to usability problems.

Participants could always check their progress, as well as the general progress for all projects and the number of commits labeled by other participants in the leaderboard. However, due to the computational effort, the leaderboard did not provide a live view of the data, but was only updated once every day. All names in the leaderboard were anonymized, except the name of the currently logged in participant and the names of the principal investigators. The

---

<sup>10</sup> This timeframe is a deviation from the registration protocol that was necessary due to the Covid-19 pandemic.

<sup>11</sup> <https://www.youtube.com/watch?v=VWvD1q41QC0>

leaderboard is also part of a gamification element of our study, as participants can see how they rank in relation to others and may try to improve their ranks by labeling more commits.

The participants are allowed to decide for which project they want to perform the labeling. The bugs are then randomly selected from all bugs of that project, for which we do not yet have labels by four participants. We choose this approach over randomly sampling from all projects to allow participants to gain experience in a project, which may improve both the labeling speed and the quality of the results. Participants must finish labeling each bug they are shown before the next bug can be drawn. We decided for this for two reasons. First, skipping bugs could reduce the validity of the results for our second research question, i.e., how good we actually are at labeling bug fixes at this level of granularity, because the sample could be skewed towards simpler bug fixes. Second, this could lead to cherry picking, i.e., participants could skip bugs until they find particularly easy bugs. This would be unfair for the other participants. The drawback of this approach is that participants are forced to label bugs, even in case they are unsure. However, we believe that this drawback is countered by our consensus labeling that requires agreement of three participants: even if all participants are unsure about a commit, if three come to the same result, it is unlikely that they are wrong.

### *3.6.2 Analysis Phase*

The analysis took place after the data collection phase was finished on October 14th, 2020. In this phase, the principal investigators conducted the analysis as described in Section 3.7. The principal investigators also informed all participants of their consensus ratio. All participants who met the first two criteria for co-authorship received a draft of the manuscript for review. Due to the number of participants, this was conducted in multiple batches. In between, the principal investigators improved the manuscript based on the comments of the participants. All participants who reviewed the draft were added to the list of authors. Those who failed to provide a review were added to the acknowledgements, unless they specifically requested not to be added. Finally, all co-authors received a copy of the final draft one week in advance of the submission for their consideration.

### *3.6.3 Data Correction Phase*

Due to a bug in the data collection software that we only found after the data collection was finished, we required an additional phase for the correction of data. The bug led to possibly corrupt data in case line numbers in the added and deleted code overlapped, e.g., single line modifications. We computed all lines that were possibly affected by this bug and found that 28,827 lines possibly contained corrupt labels, i.e., about 10% of our data. We asked all co-authors to correct their data through manual inspection by relabeling all lines that could have been corrupted. For this, we used the same tooling as

for the initial labeling, with the difference that all labels that were not affected by the bugs were already set, only the lines affected by the bug needed to be relabeled.<sup>12</sup> The data correction phase took place from November 25th, 2020 until January 17th, 2021. We deleted all data that was not corrected by January 18th which resulted in 679 less issues for which the labeling has finished. We invited all co-authors to re-label these issues between January 18th and January 21th. Through this, the data for 636 was finished again, but we still lost the data for 43 issues as the result of this bug. The changes to the results were very minor and the analysis of the results was not affected. However, we note that the analysis of consensus ratios and participation (Section 5.1) was done prior to the correction phase. The bug affected only a small fraction of lines that should only have a negligible impact on the individual consensus ratios, as data by all participants was affected equally. We validated this intuition by comparing the consensus ratios of participants who corrected their data before and after the correction and found that there were no relevant changes. Since some participants could not participate in the data correction and we had to remove some labelled commits, the number of labeled commits per author could now be below 200. This altered the results due to outliers, caused by single very large commits. This represents an unavoidable trade-off arising from the need to fix the potentially corrupt data.

### 3.7 Analysis Plan

The analysis of data consists of four aspects: the contribution to bug fixes, the capability to label bug fixing commits, the effectiveness of gamification, and the confidence level of our statistical analysis.

#### 3.7.1 Contributions to Bug Fixes

We used the Shapiro-Wilk test (Shapiro and Wilk, 1965) to determine if the nine variables related to defects are normally distributed. Since the data is not normal, we report the median, median absolute deviation (MAD), and an estimation for the confidence interval of the median based on the approach proposed by Campbell and Gardner (1988). We reject H1 if the upper bound of the confidence interval of the median lines contributing to the bug fix in all code is greater than 40%. Within the discussion, we provide further insights about the expectations on lines within bug fixing commits based on all results, especially also within changes to production code files.

#### 3.7.2 Capability to Label Bug Fixing Commits

We use the confidence interval for the number of lines without consensus for this evaluation. We reject H2 if the lower bound of the confidence interval of

---

<sup>12</sup> Details can be found in the tutorial video: <https://www.youtube.com/watch?v=Kf6wVoo32Mc>

$\kappa$	Interpretation
$<0$	Poor agreement
0.01 – 0.20	Slight agreement
0.21 – 0.40	Fair agreement
0.41 – 0.60	Moderate agreement
0.61 – 0.80	Substantial agreement
0.81 – 1.00	Almost perfect agreement

Table 2: Interpretation of Fleiss’  $\kappa$  according to Landis and Koch (1977).

the median number of lines without consensus is less than 10.5%. Additionally, we report Fleiss’  $\kappa$  (Fleiss, 1971) to estimate the reliability of the consensus, which is defined as

$$\kappa = \frac{\bar{P} - \bar{P}_e}{1 - \bar{P}_e} \quad (2)$$

where  $\bar{P}$  is the mean agreement of the participants per line and  $\bar{P}_e$  is the sum of the squared proportions of the label assignments. We use the table from Landis and Koch (1977) for the interpretation of  $\kappa$  (see Table 2).

Additionally, we estimate the probability of random mistakes to better understand if the lines without consensus can be explained by random mislabels, i.e., mislabels that are the consequence of unintentional mistakes by participants. If we assume that all lines with consensus are labeled correctly, we can use the *minority votes* in those lines to estimate the probability of random mislabels. Specifically, we have a minority vote if three participants agreed on one label, and one participant selected a different label. We assume that random mislabels follow a binomial distribution (see Section 3.2) to estimate the probability that a single participant randomly mislabels a bug fixing line. Following Brown et al. (2001), we use the approach from Agresti and Coull (1998) to estimate the probability of a mislabel  $p$ , because we have a large sample size. Therefore, we estimate

$$p = \frac{n_1 + \frac{1}{2}z_\alpha^2}{n + z_\alpha^2} \quad (3)$$

as the probability of a mislabel of a participant with  $n_1$  the number of minority votes in lines with consensus,  $n$  the total number of individual labels in lines with consensus, and  $z_\alpha$  the  $1 - \frac{1}{2}\alpha$  quantile of the standard normal distribution, with  $\alpha$  the confidence level. We get the confidence interval for  $p$  as

$$p \pm z_\alpha \sqrt{\frac{p \cdot (1 - p)}{n + z_\alpha^2}}. \quad (4)$$

We estimate the overall probabilities of errors, as well as the probabilities of errors in production files for the different label types to get insights into the distribution of errors. Moreover, we can use the estimated probabilities of

random errors to determine how many lines without consensus are expected. If  $n_{total}$  is the number of lines, we can expect that there are

$$\begin{aligned} n_{none} &= n_{total} \cdot B(k=2|p, n=4) \\ &= n_{total} \cdot \binom{4}{2} p^2 \cdot (1-p)^2 \\ &= n_{total} \cdot 6 \cdot p^2 \cdot (1-p)^2 \end{aligned} \quad (5)$$

lines without consensus under the assumption that they are due to random mistakes. If we observe more lines without consensus, this is a strong indicator that this is not a random effect, but due to actual disagreements between participants.

We note that the calculation of the probability of mislabels and the number of expected non-consensus lines was not described in the pre-registered protocol. However, since the approach to model random mistakes as binomial distribution was already used to derive H2 as part of the registration, we believe that this is rather the reporting of an additional detail based on an already established concept from the registration and not a substantial deviation from our protocol.

In addition to the data about the line labels, we use the result of the survey among participants regarding their perceived certainty rates to give further insights into the limitations of the participants to conduct the task of manually labeling lines within commits. We report the histogram of the answers given by the participants and discuss how the perceived difficulty relates to the actual consensus that was achieved.

### 3.7.3 Effectiveness of Gamification

We evaluate the effectiveness of the gamification element of the crowd working by considering the number of commits per participant. Concretely, we use a histogram of the total number of commits per participant. The histogram tells us whether there are participants who have more than 200 commits, including how many commits were actually labeled. Moreover, we create line plots for the number of commits over days, with one line per participant that has more than 200 commits. This line plot is a visualization of the evolution of the prospective ranking of the author list. If the gamification is efficient, we should observe two behavioral patterns: 1) that participants stopped labeling right after they gained a certain rank; and 2) that participants restarted labeling after a break to increase their rank. The first observation would be in line with the results from Anderson et al. (2013) regarding user behavior after achieving badges on Stack Overflow. We cannot cite direct prior evidence for our second conjecture, other than that we believe that participants who were interested in gaining a certain rank, would also check if they still occupy the rank and then act on this.

We combine the indications from the line plot with the answer to the survey question Q2. If we can triangulate from the line plot and the survey

that the gamification was effective for at least 10% of the overall participants, we believe that having this gamification element is worthwhile and we fail to reject H3. If less than 10% were motivated by the gamification element, this means that future studies could not necessarily expect a positive benefit, due to the small percentage of participants that were motivated. We additionally quantify the effect of the gamification by estimating the additional effort that participants invested measured in the number of commits labeled greater than 200 for the subset of participants where the gamification seems to have made a difference.

#### 3.7.4 Confidence Level

We compute all confidence intervals such that we have a family-wise confidence level of 95%. We need to adjust the confidence level for the calculation of the confidence intervals, due to the number of intervals we determine. Specifically, we determine  $2 \cdot 9 = 18$  confidence intervals for the ratios of line labels within commits (see Section 3.4) and 27 confidence intervals for our estimation of the probability of random mistakes. Due to the large amount of data we have available, we decided for an extremely conservative approach for the adjustment of the confidence level (see Section 3.7.2). We use Bonferroni correction (Dunnett, 1955) for all  $18 + 27 = 45$  confidence intervals at once, even though we could possibly consider these as separate families. Consequently, we use a confidence level of  $1 - \frac{0.05}{45} = 0.99\bar{8}$  for all confidence interval calculations.<sup>13</sup>

### 3.8 Summary of Deviations from Pre-Registration

We deviated from the pre-registered research protocol in several points, mostly through the expansion on details.

- The time frame of the labeling shifted to May 16th–October 14th. Additionally, we had to correct a part of the data due to a bug between November 25th and January 21st.
- We updated **H2** with a threshold of 10.5% of lines, due to a wrong calculation in the registration (see footnote 6).
- We consider the subset of mislabels on changes to production code files, as well as mislabels with respect to all changes.
- We have additional details because we distinguish between whitespace and documentation lines.
- We have additional analysis for lines without consensus to differentiate between different reasons for no consensus.

<sup>13</sup> The pre-registration only contained correction for six confidence intervals. This increased because we provide a more detailed view on labels without consensus and differentiate between all changes and changes to production code files, and because the calculation of probabilities for mistakes was not mentioned in the registration.

Project	Timeframe	#Bugs	#Commits
ant-ivy	2005-06-16 – 2018-02-13	404 / 404	547 / 547
<i>archiva</i>	2005-11-23 – 2018-07-25	3 / 278	4 / 509
commons-bcel	2001-10-29 – 2019-03-12	33 / 33	52 / 52
commons-beanutils	2001-03-27 – 2018-11-15	47 / 47	60 / 60
commons-codec	2003-04-25 – 2018-11-15	27 / 27	58 / 58
commons-collections	2001-04-14 – 2018-11-15	48 / 48	93 / 93
commons-compress	2003-11-23 – 2018-11-15	119 / 119	205 / 205
commons-configuration	2003-12-23 – 2018-11-15	140 / 140	253 / 253
commons-dbcp	2001-04-14 – 2019-03-12	57 / 57	89 / 89
commons-digester	2001-05-03 – 2018-11-16	17 / 17	26 / 26
commons-io	2002-01-25 – 2018-11-16	71 / 72	115 / 125
commons-jcs	2002-04-07 – 2018-11-16	58 / 58	73 / 73
commons-lang	2002-07-19 – 2018-10-10	147 / 147	225 / 225
commons-math	2003-05-12 – 2018-02-15	234 / 234	391 / 391
commons-net	2002-04-03 – 2018-11-14	127 / 127	176 / 176
commons-scxml	2005-08-17 – 2018-11-16	46 / 46	67 / 67
commons-validator	2002-01-06 – 2018-11-19	57 / 57	75 / 75
commons-vfs	2002-07-16 – 2018-11-19	94 / 94	118 / 118
<i>deltaspikes</i>	2011-12-22 – 2018-08-02	6 / 146	8 / 219
<i>eagle</i>	2015-10-16 – 2019-01-29	2 / 111	2 / 121
giraph	2010-10-29 – 2018-11-21	140 / 140	146 / 146
gora	2010-10-08 – 2019-04-10	56 / 56	98 / 98
<i>jspwiki</i>	2001-07-06 – 2019-01-11	1 / 144	1 / 205
opennlp	2008-09-28 – 2018-06-18	106 / 106	151 / 151
parquet-mr	2012-08-31 – 2018-07-12	83 / 83	119 / 119
santuario-java	2001-09-28 – 2019-04-11	49 / 49	95 / 95
<i>systemml</i>	2012-01-11 – 2018-08-20	6 / 279	6 / 314
wss4j	2004-02-13 – 2018-07-13	150 / 150	245 / 245
<i>Total</i>		2328 / 3269	3498 / 4855

Table 3: Statistics about amounts of labeled data per project, i.e., data for which we have labels by four participants and can compute consensus. The columns #Bugs and #Commits list the completed data and the total data available in the time frame. The five projects marked as italic are incomplete, because we did not have enough participants to label all data.

- We extend the analysis with an estimation of the probability of random mislabels, instead of only checking the percentage of lines without consensus.

## 4 Experiments for RQ1

We now present our results and discuss their implications for RQ1 on the tangling of changes within commits.

### 4.1 Results for RQ1

In this section, we first present the data demographics of the study, e.g., the number of participants, and the amount of data that was labeled. We then

Label	All Changes		Production Code		Other Code	
Bug fix	72774	(25.1%)	71343	(49.2%)	361	(0.3%)
Test	114765	(39.6%)	8	(0.0%)	102126	(91.3%)
Documentation	40456	(14.0%)	31472	(21.7%)	749	(0.7%)
Refactoring	5297	(1.8%)	5294	(3.7%)	3	(0.0%)
Unrelated Improvement	1361	(0.5%)	824	(0.6%)	11	(0.0%)
Whitespace	11909	(4.1%)	10771	(7.4%)	781	(0.7%)
Test/Doc/Whitespace	4454	(1.5%)	0	(0.0%)	4454	(4.0%)
No Bug fix	13052	(4.5%)	4429	(3.1%)	1754	(1.6%)
No Consensus	25836	(8.9%)	20722	(14.3%)	1587	(1.4%)
<i>Total</i>	289904		144863		111826	

Table 4: Statistics of assigned line labels over all data. Production code refers all Java files that we did not determine to be part of the test suite or the examples. Other code refers to all other Java files. The labels above the line are for at least three participants selecting the same label. The labels below the line do not have consensus, but are the different categories for lines without consensus we established in Section 3.4. The eight lines labeled as test in production code are due to two test files within Apache Commons Math that were misplaced in the production code folder.

present the results of the labeling. All labeled data of the LLTC4J corpus and the analysis scripts we used to calculate our results can be found online in our replication package.<sup>14</sup>

#### 4.1.1 Data Demographics

Of 79 participants registered for this study, 15 participants dropped out without performing any labeling. The remaining 64 participants labeled data. The participants individually labeled 17,656 commits. This resulted in 1,389 commits labeled by one participant, 683 commits labeled by two participants, 303 commits that were labeled by three participants, 3,498 commits labeled by four participants, and five commits that were part of the tutorial were labeled by all participants. Table 3 summarizes the completed data for each project. Thus, we have validated all bugs for 23 projects and incomplete data about bugs for five projects. We have a value of Fleiss’  $\kappa = 0.67$ , which indicates that we have substantial agreement among the participants.

#### 4.1.2 Content of Bug Fixing Commits

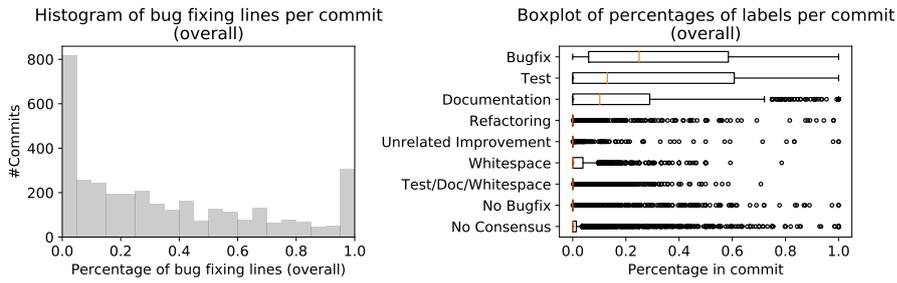
Table 4 summarizes the overall results of the commit labeling. Overall, 289,904 lines were changed as part of the bug fixing commits. Only 25.1% of the changes were part of bug fixes. The majority of changed lines were modifications of tests with 39.6%. Documentation accounts for another 14.0% of the changes.

<sup>14</sup> <https://github.com/sherbold/replication-kit-2020-line-validation>

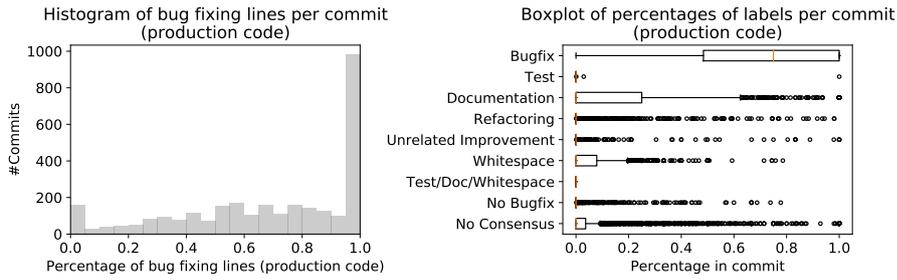
We will move the replication kit to a long-term archive on Zenodo in case of acceptance of this manuscript.

Label	Overall			Production Code			
	Med.	MAD	CI	Med.	MAD	CI	> 0
Bug fix	25.0	34.8	[22.2, 29.2]	75.0	37.1	[70.6, 79.2]	95.5
Test	13.0	19.3	[0.0, 23.4]	0.0	0.0	[0.0, 0.0]	0.1
Documentation	10.2	15.1	[7.8, 12.5]	0.0	0.0	[0.0, 5.1]	49.4
Refactoring	0.0	0.0	[0.0, 0.0]	0.0	0.0	[0.0, 0.0]	7.8
Unrelated Impr.	0.0	0.0	[0.0, 0.0]	0.0	0.0	[0.0, 0.0]	2.9
Whitespace	0.0	0.0	[0.0, 0.0]	0.0	0.0	[0.0, 1.7]	46.9
Test/Doc/Whites.	0.0	0.0	[0.0, 0.0]	0.0	0.0	[0.0, 0.0]	0.0
No Bug fix	0.0	0.0	[0.0, 0.0]	0.0	0.0	[0.0, 0.0]	5.8
No Consensus	0.0	0.0	[0.0, 0.0]	0.0	0.0	[0.0, 0.0]	29.7

(a) The median (Med.), the median absolute deviation (MAD), and the confidence interval of the median (CI) of the percentages of changes within commits. The column > 0 shows the ratio of commits that have production code file changes of the label type. The many zeros are the consequence of the fact that many commits do not contain any changes that are not of type bug fix, test, or documentation.



(b) Plots for the distribution of lines within commits for all file changes.



(c) Plots for the distribution of lines within commits for changes to production code.

Fig. 2: Data about changes within commits.

For 8.9% of the lines there was no consensus among the participants and at least one participant labeled the line as bug fix. For an additional 1.5% of lines, the participants marked the line as either documentation, whitespace change or test change, but did not achieve consensus. We believe this is the result of different labeling strategies for test files (see Section 3.4). For 4.5% of the lines no participant selected bug fix, but at least one participant selected refactoring or unrelated improvement. When we investigated these lines, we found that the majority of cases were due to different labeling strategies: some

participants labeled updates to test data as unrelated improvements, others labeled them as tests. How this affected production code files is discussed separately in Section 4.1.3.

256,689 of the changed lines were Java code, with 144,863 lines in production code files and 11,826 lines in other code files. The other code is almost exclusively test code. Within the production code files, 49.2% of the changed lines contributed to bug fixes and 21.7% were documentation. Refactorings and unrelated improvements only represent 4.3% of the lines. In 14.3% of the lines, the participants did not achieve consensus with at least one participant labeling the line as bug fix.

Figure 2 summarizes the results of labeling per commit, i.e., the percentages of each label in bug fixing commits. We found that a median of 25.0% of all changed lines contribute to a bug fix. When we restrict this to production code files, this ratio increases to 75.0%. We note that while the median for all changes is roughly similar to the overall percentage of lines that are bug fixing, this is not the case for production code files. The median of 75.0% per commit is much higher than the 49.2% of all production lines that are bug fixing. The histograms provide evidence regarding the reason for this effect. With respect to all changed lines, Figure 2(b) shows that there are many commits with a relatively small percentage of bug fixing lines close to zero, i.e., we observe a peak on the left side of the histogram. When we focus the analysis on the production code files, Figure 2(c) shows that we instead observe that there are many commits with a large percentage of bug fixing lines (close to 100%), i.e., we observe a peak on the right side of the histogram, but still a long tail of lower percentages. This tail of lower percentages influences the ratio of lines more strongly in the median per commit, because the ratio is not robust against outliers.

The most common change to be tangled with bug fixes are test changes and documentation changes with a median of 13.0% and 10.2% of lines per commit, respectively. When we restrict the analysis to production code files, all medians other than bug fix drop to zero. For test changes, this is expected because they are, by definition, not in production code files. For other changes, this is due to the extremeness of the data which makes the statistical analysis of most label percentages within commits difficult. What we observe is that while many commits are not pure bug fixes, the type of changes differs between commits, which leads to commits having percentages of exactly zero for all labels other than bug fix. This leads to extremely skewed data, as the median, MAD, and CI often become exactly zero such that the non-zero values are – from a statistical point of view – outliers. The last column of the table in Figure 2(a) shows for how many commits the values are greater than zero. We can also use the boxplots in Figure 2(b) and (c) to gain insights into the ratios of lines, which we analyze through the outliers.

The boxplots reveal that documentation changes are common in production code files, the upper quartile is at around 30% of changed lines in a commit. Unrelated improvements are tangled with 2.9% of the commits and are usually less than about 50% of the changes, refactorings are tangled with 7.8% of the

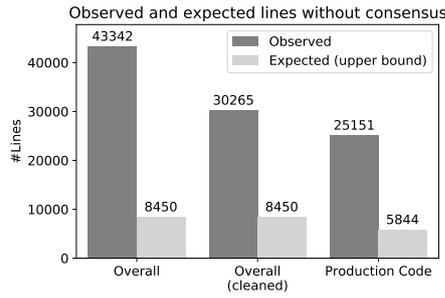


Fig. 3: Number of expected lines without consensus versus the actual lines without consensus. Overall (cleaned) counts lines with the labels Test/Doc/Whitespace and No Bug fix as consensus.

commits and usually less than about 60% of the changes. Whitespace changes are more common, i.e., 46.9% of the commits are tangled with some formatting. However, the ratio is usually below 40% and there are no commits that contain only whitespace changes, i.e., pure reformatting of code. The distribution of lines without consensus shows that while we have full consensus for 71.3% of the commits, the consensus ratios for the remaining commits are distributed over the complete range up to no consensus at all.

As a side note, we also found that the pre-labeling of lines with the RefactoringMiner was not always correct. Sometimes logical changes were marked as refactoring, e.g., because side effects were ignored when variables were extracted or code was reordered. Overall, 21.6% of the 23,682 lines marked by RefactoringMiner have a consensus label of bug fix. However, the focus of our study is not the evaluation of RefactoringMiner and we further note that we used RefactoringMiner 1.0 (Tsantalis et al., 2018) and not the recently released version 2.0 (Tsantalis et al., 2020), which may have resolved some of these issues.

#### 4.1.3 Analysis of Disagreements

Based on the minority votes in lines with consensus, we estimate the probability of random mistakes in all changes as  $7.5\% \pm 0.0$ , when we restrict this to production code files we estimate the probability as  $9.0\% \pm 0.0$ . We note that the confidence intervals are extremely small, due to the very large number of lines in our data. Figure 3 shows the expected number of lines without consensus given these probabilities versus the observed lines without consensus. For all changes, we additionally report a cleaned version of the observed data. With the cleaned version, we take the test/doc/whitespace and no bug fix labels into account, i.e., lines where there is consensus that the line is not part of the bug fix. The results indicate that there are more lines without consensus than could be expected under the assumption that all mislabels in our data are the result of random mislabels.

	Bug fix	Doc.	Refactoring	Unrelated	Whitespace	Test
Bug fix	-	1	508	705	23	2
Documentation	189	-	11	47	17	2
Refactoring	446	2	-	35	3	0
Unrelated	110	0	2	-	1	0
Whitespace	103	1	49	34	-	1
Total Expected:	847	3	570	821	42	5
Total Observed:	12837	3987	6700	8251	3951	252

Table 5: A more detailed resolution of the number of expected mislabels per label type. The rows represent the correct labels, the columns represent the number of two expected mislabels of that type. The sum of the observed values is not equal to the sum of the observed lines without consensus, because it is possible that a line without consensus has two labels of two types each and we cannot know which one is the mislabel. Hence, we must count these lines twice here. The size of the confidence intervals is less than one line in each case, which is why we report only the upper bound of the confidence intervals, instead of the confidence intervals themselves. There is no row for test, because there are no correct test labels in production code files.

Table 5 provides additional details regarding the expectation of the mislabels per type in production code files. The data in this table is modeled using a more fine-grained view on random mistakes. This view models the probabilities for all labels separately, both with respect to expected mislabels, as well as the mislabel that occurs. The estimated probabilities are reported in Table 7 in the appendix. Using the probabilities, we calculated the expected number of two random mistakes for each label type, based on the distribution of consensus lines. Table 5 confirms that we do not only observe more lines without consensus, but also more pairs of mislabels than could be expected given a random distribution of mistakes for all labels. However, the mislabels are more often of type bug fix, unrelated improvement, or refactoring than of the other label types. Table 8 in the appendix shows a full resolution of the individual labels we observed in lines without consensus and confirms that disagreements between participants whether a line is part of a bug fix, an unrelated improvement, or a refactoring, are the main driver of lines without consensus in production code files.

In addition to the view on disagreements through the labels, we also asked the participants that labeled more than 200 commits how confident they were in the correctness of their labels. Figure 4 shows the results of this survey. 55% of the participants estimate that they were not sure in 11%–20% of lines, another 20% estimate 21%–30% of lines. Only one participant estimates a higher uncertainty of 51%–60%. This indicates that participants have a good intuition about the difficulty of the labeling. If we consider that we have about 8% random mistakes and 12% lines without consensus, this indicates that about 20% of the lines should have been problematic for participants. We note that there are also three that selected between 71%–100% of lines. While we cannot know for sure, we believe that these three participants misread the

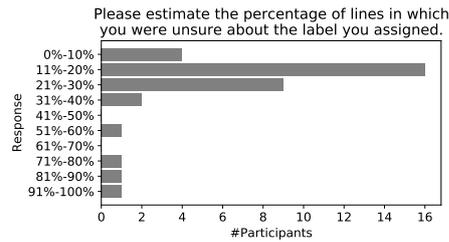


Fig. 4: Answers of the participants regarding their labeling confidence. Participants were not informed about the distribution of consensus ratios of participants or their individual consensus ratio before answering the survey to avoid bias. 35 out of 48 participants with at least 200 commits answered this question.

question, which is in line with up to 10% of participants found by Podsakoff et al. (2003).

## 4.2 Discussion of RQ1

We now discuss our results for RQ1 with respect to our hypotheses to gain insights into the research question, put our results in the context of related work, and identify consequences for researchers who analyze bugs.

### 4.2.1 Prevalence of Tangled Commits

Tangling is common in bug fixing commits. We estimate that the average number of bug fixing lines per commit is between 22% and 38%. The 22% of lines are due to the lower bound of the confidence interval for the median bug fixing lines per commit. The 38% of lines assume that all 8.9% of the lines without consensus would be bug fixing and that the median would be at the upper bound of the confidence interval. However, we observe that a large part of the tangling is not within production code files, but rather due to changes in documentation and test files. Only half of the changes within bug fixing commits are to production code files. We estimate that the average number of bug fixing lines in production code files per commit is between 60% and 93%.

We fail to reject the hypothesis H1 that less than 40% of changes in bug fixing commits contribute to the bug fix and estimate that the true value is between 22% and 38% of lines. However, this is only true if all changes are considered. If only changes to production code files are considered, the average number of bug fixing lines is between 69% and 93%.

### 4.2.2 Impact of Tangling

The impact of tangling on software engineering research depends on the type of tangling as well as the research application. Moreover, tangling is not, by definition, bad. For example, adding new tests as part of a bug fix is actually a best practice. In the following, we use the term *benign tangling* to refer to tangled commits that do not negatively affect research and *problematic tangling* to refer to tangling that results in noise within the data without manual intervention. Specifically, we discuss problematic tangling for the three research topics we already mentioned in the motivation, i.e., program repair, bug localization, and defect prediction.

If bug fixes are used for program repair, only changes to production code are relevant, i.e., all other tangling is benign as the tangling can easily be ignored using regular expressions, same as we did in this article. In production code files, the tangling of whitespaces and documentation changes is also benign. Refactorings and unrelated improvements are problematic, as they are not required for the bug fix and needlessly complicate the problem of program repair. If bug fixes are used as training data for a program repair approach (e.g., Li et al., 2020), this may lead to models that are too complex as they would mix the repair with additional changes. If bug fixes are used to evaluate the correctness of automatically generated fixes (e.g., Martinez et al., 2016), this comparison may be more difficult or noisy, due to the tangling. Unrelated improvements are especially problematic if they add new features. This sort of general program synthesis is usually out of scope of program repair and rather considered as a different problem, e.g., as neural machine translation (Tufano et al., 2019) and, therefore, introduces noise in dedicated program repair analysis tasks as described above.

For bug localization, tangling outside of production code files is also irrelevant, as these changes are ignored for the bug localization anyways. Within production code files, all tangling is irrelevant, as long as a single line in a file is bug fixing, assuming that the bug localization is at the file level.

For defect prediction, we assume that a state-of-the-art SZZ variant that accounts for whitespaces, documentations, and automatically detectable refactorings is used (Neto et al., 2018) to determine the bug fixing and bug inducing commits. Additionally, we assume that a heuristic is used to exclude non-production changes, which means that changes to non-production code files are not problematic for determining bug labels. For the bug fixes, we have the same situation as with bug localization: labeling of files would not be affected by tangling, as long as a single line is bug fixing. However, all unrelated improvements and non-automatically detectable refactorings may lead to the labeling of additional commits as inducing commits and are, therefore, potentially problematic. Finally, defect prediction features can also be affected by tangling. For example, the popular metrics by Kamei et al. (2013) for just-in-time defect prediction compute change metrics for commits as a whole, i.e., not based on changes to production code only, but rather based on all changes. Consequently, all tangling is problematic with respect to these metrics. Ac-

Research Topic	Bugs	File Changes
Program repair	12%–35%	9%–32%
Bug localization	9%–23%	7%–21%
Defect prediction (bug fix)	3%–23%	2%–21%
Defect prediction (inducing)	5%–24%	3%–18%
Defect prediction (total)	8%–47%	5%–39%

Table 6: The ratio of problematic tangling within our data with respect to bugs and production file changes. The values for the inducing commits in defect prediction are only the commits that are affected in addition to the bug fixing commits.

According to the histogram for overall tangling in Figure 2, this affects most commits.

Table 6 summarizes the presence of problematic tangling within our data. We report ranges, due to the uncertainty caused by lines without consensus. The lower bound assumes that all lines without consensus are bug fixing and that all refactorings could be automatically detected, whereas the upper bound assumes that the lines without consensus are unrelated improvements and refactorings could not be automatically detected. We observe that all use cases are affected differently, because other types of tangled changes cause problems. Program repair has the highest lower bound, because any refactoring or unrelated improvement is problematic. Bug localization is less affected than defect prediction, because only the bug fixing commits are relevant and the bug inducing commits do not matter. If bug localization data sets would also adopt automated refactoring detection techniques, the numbers would be the same as for defect prediction bug fix labels. However, we want to note that the results from this article indicate that the automated detection of refactorings may be unreliable and could lead to wrong filtering of lines that actually contribute to the bug fix. Overall, we observe that the noise varies between applications and also between our best case and worst case assumptions. In the best case, we observe as little as 2% problematically tangled bugs (file changes for bug fixes in defect prediction), whereas in the worst case this increases to 47% (total number of bugs with noise for defect prediction). Since prior work already established that problematic tangling may lead to wrong performance estimations (Kochhar et al., 2014; Herzig et al., 2016; Mills et al., 2020) as well as the degradation of performance of machine learning models (Nguyen et al., 2013), this is a severe threat to the validity of past studies.

Tangled commits are often *problematic*, i.e., lead to noise within data sets that cannot be cleaned using heuristics. The amount of noise varies between research topics and is between an optimistic 2% and a pessimistic value of 47%. As researchers, we should be skeptical and assume that unvalidated data is likely very noisy and a severe threat to the validity of experiments, until proven otherwise.

### 4.2.3 Reliability of Labels

Participants have a probability of random mistakes of about 7.5% overall and 9.0% in production code files. Due to these numbers, we find that our approach to use multiple labels per commit was effective. Using the binomial distribution to determine the likelihood of at least three random mislabels, we expect that 111 lines with a consensus label are wrong, i.e., an expected error rate of 0.09% in consensus labels. This error rate is lower than, e.g., using two people that would have to agree. In this case, the random error rate would grow to about 0.37%. We fail to achieve consensus on 8.9% of all lines and 14.3% of lines in production code files. Our data indicates that most lines without consensus are not due to random mistakes, but rather due to disagreements between participants whether a line contributes to the bug fix or not. This indicates that the lines without consensus are indeed hard to label. Our participant survey supports our empirical analysis of the labeled data. A possible problem with the reliability of the work could also be participant dependent “default behavior” in case they were uncertain. For example, participants could have labeled lines as bug fixing in case they were not sure, which would reduce the reliability of our work and also possibly bias the results towards more bug fixing lines. However, we have no data regarding default behavior and believe that this should be studied in an independent and controlled setting.

We reject H2 that participants fail to achieve consensus on at least 10.5% of lines and find that this is not true when we consider all changes. However, we observe that this is the case for the labeling of production code files with 14.3% of lines without consensus. Our data indicates that these lines are hard to label by researchers with active disagreement instead of random mistakes. Nevertheless, the results with consensus are reliable and should be close to the ground truth.

### 4.2.4 Comparison to Prior Work

Due to the differences in approaches, we cannot directly compare our results with those of prior studies that quantified tangling. However, the fine-grained labeling of our data allows us to restrict our results to mimic settings from prior work. From the description in their paper, Herzig and Zeller (2013) flagged only commits as tangled, that contained source code changes for multiple issues or clean up of the code. Thus, we assume that they did not consider documentation changes or whitespace changes as tangling. This definition is similar to our definition of problematic tangling for program repair. The 12%–35% affected bugs that we estimate includes the lower bound of 15% on tangled commits. Thus, our work seems to replicate the findings from Herzig and Zeller (2013), even though our methodologies are completely different. Whether the same holds true for the 22% of tangled commits reported by Nguyen et al. (2013) is unclear, because they do not specify how they handle non-production

code. Assuming they ignore non-production code, our results would replicate those by Nguyen et al. (2013) as well.

Kochhar et al. (2014) and Mills et al. (2020) both consider tangling for bug localization, but have conflicting results. Both studies defined tangling similar to our definition of problematic tangling for bug localization, except that they did not restrict the analysis to production code files, but rather to all Java files. When we use the same criteria, we have between 39% and 48% problematic tangling in Java file changes. Similar to what we discussed in Section 4.2.2, the range is the result of assuming lines with consensus either as bug fixes or as unrelated improvements. Thus, our results replicate the work by Mills et al. (2020) who found between 32% and 50% tangled file changes. We could not confirm the results by Kochhar et al. (2014) who found that 28% of file changes are problematic for bug localization.

Regarding the types of changes, we can only compare our work with the data from Nguyen et al. (2013) and Mills et al. (2020). The percentages are hard to compare, due to the different levels of abstraction we consider. However, we can compare the trends in our data with those of prior work. The results regarding tangled test changes are similar to Mills et al. (2020). For the other types of changes, the ratios we observe are more similar to the results reported by Mills et al. (2020) than those of Nguyen et al. (2013). We observe only few unrelated improvements, while this is as common as tangled documentation changes in the work by Nguyen et al. (2013). Similarly, Mills et al. (2020) observed a similar amount of refactoring as documentation changes. In our data, documentation changes are much more common than refactorings, both in all changes as well as in changes to production code files. A possible explanation for this are the differences in the methodology: refactorings and unrelated improvement labels are quite common in the lines without consensus. Thus, detecting such changes is common in the difficult lines. This could mean that we underestimate the number of tangled refactorings and unrelated improvements, because many of them are hidden in the lines without consensus. However, Nguyen et al. (2013) and Mills et al. (2020) may also overestimate the number of unrelated improvements and refactorings, because they had too few labelers to have the active disagreements we observed. Regarding the other contradictions between the trends observed by Nguyen et al. (2013) and Mills et al. (2020), our results indicate that the findings by Mills et al. (2020) generalize to our data: we also find more refactorings than other unrelated improvements.

Our results confirm prior studies by Herzig and Zeller (2013), Nguyen et al. (2013), and Mills et al. (2020) regarding the prevalence of tangling. We find that the differences in estimations are due to the study designs, because the authors considered different types of tangling, which we identify as different types of problematic tangling. Our results for tangled change types are similar to the results by Mills et al. (2020), but indicate also a disagreement regarding the prevalence of refactorings and

unrelated improvements, likely caused by the uncertainty caused by lines that are difficult to label.

#### 4.2.5 Anecdotal Evidence on Corner Cases

Due to the scope of our analysis, we also identified some interesting corner cases that should be considered in future work on bug fixes. Sometimes, only tests were added as part of a bug fixing commit, i.e., no production code was changed. Usually, this indicates that the reported issue was indeed not a bug and that the developers added the tests to demonstrate that the case works as expected. In this case, we have a false positive bug fixing commit, due to a wrong issue label. However, sometimes it also happened, that the label was correct, but the bug fixing commit still only added tests. An example for this is the issue IO-466,<sup>15</sup> where the bug was present, but was already fixed as a side effect of the correction of IO-423. This means we have an issue that is indeed a bug and a correct link from the version control system to the issue tracking system, but we still have a false positive for the bug fixing commit. This shows that even if the issue types and issue links are manually validated, there may be false positives in the bug fixing commits, if there is no check that production code is modified.

Unfortunately, it is not as simple as that. We also found some cases, where the linked commit only contained the modification of the change log, but no change to production code files. An example for this is NET-270.<sup>16</sup> This is an instance of missing links: the actual bug fix is in the parent of the linked commit, which did not reference NET-270. Again, we have a correct issue type, correct issue link, but no bug fix in the commit, because no production code file was changed. However, a directly related fix exists and can be identified through manual analysis.

The question is how should we deal with such cases in future work? We do not have clear answers. Identifying that these commits do not contribute to the bug fix is trivial, because they do not change production code files. However, always discarding such commits as not bug fixing may remove valuable information, depending on the use case. For example, if we study test changes as part of bug fixing, the test case is still important. If you apply a manual correction, or possibly even identify a reliable heuristic, to find fixes in parent commits, the second case is still important. From our perspective, these examples show that even the best heuristics will always produce noise and that even with manual validations, there are not always clear decisions.

An even bigger problem is that there are cases where it is hard to decide which modifications are part of the bug fix, even if you understand the complete logic of the correction. A common case of this is the “new block problem” that is depicted in Figure 5. The problem is obvious: the unchecked use of `a.foo()` can cause a `NullPointerException`, the fix is the addition of a

<sup>15</sup> <https://issues.apache.org/jira/browse/IO-466>

<sup>16</sup> <https://issues.apache.org/jira/browse/NET-270>

Bug:	Fix 1:	Fix 2:
<pre>public void foo() {   a.foo(); }</pre>	<pre>public void foo() {   if(a!=null) {     a.foo();   } }</pre>	<pre>public void foo() {   if(a==null) {     return;   }   a.foo(); }</pre>
	Diff for Fix 1: <pre>+ if(a!=null) { - a.foo() + a.foo() + }</pre>	Diff for Fix 2: <pre>+ if(a==null) { +   return; + }</pre>

Fig. 5: Example for a bug fix, where a new condition is added. In Fix 1, the line `a.foo()` is modified by adding whitespaces and part of the textual difference. In Fix 2, `a.foo()` is not part of the diff.

null-check. Depending on how the bug is fixed, `a.foo()` is either part of the diff, or not. This leads to the question: is moving `a.foo()` to the new block part of the bug fix or is this “just a whitespace change”? The associated question is, whether the call to `a.foo()` is the bug or if the lack of the null-check is the bug. What are the implications that the line with `a.foo()` may once be labeled as part of the bug fix (Fix 1), and once not (Fix 2)? We checked in the data, and most participants labeled all lines in both fixes as contributing the bug fix in the cases that we found. There are important implications for heuristics here: 1) pure whitespace changes can still be part of the bug fix, if the whitespace changes indicates that the statement moved to a different block; 2) bugs can sometimes be fixed as modification (Fix 1), but also as pure addition (Fix 2), which leads to different data. The first implication is potentially severe for heuristics that ignore whitespace changes. In this case, textual differences may not be a suitable representation for reasoning about the changes and other approaches, such as differences in abstract syntax trees (Yang, 1991), are better suited. The second is contrary to the approach by Mills et al. (2020) for untangling for bug localization, i.e., removing all pure additions. In this case the addition could also be a modification, which means that the bug could be localized and that ignoring this addition would be a false negative. Similarly, the SZZ approach to identify the bug inducing commits cannot find inducing commits for pure additions. Hence, SZZ could blame the modification of the line `a.foo()` from Fix 1 to find the bug inducing commit, which would be impossible for Fix 2.

#### 4.2.6 Implications for Researchers

While we have discussed many interesting insights above, two aspects stand out and are, to our mind, vital for future work that deals with bugs.

- *Heuristics are effective!* Most tangling can be automatically identified by identifying non-production code files (e.g., tests), and changes to whitespaces and documentation within production code files.<sup>17</sup> Any analysis that should target production code but does not carefully remove tests cannot be trusted, because test changes are more common than bug fixing changes.
- *Heuristics are imperfect!* Depending on the use case and the uncertainty in our data, up to 47% could still be affected by tangling, regardless of the heuristics used to clean the data. We suggest that researchers should carefully assess which types of tangling are problematic for their work and use our data to assess how much problematic tangling they could expect. Depending on this estimation, they could either estimate the threat to the validity of their work or plan other means to minimize the impact of tangling on their work.

#### 4.2.7 Summary for RQ1

In summary, we have the following result for RQ1.

We estimate that only between 22% and 38% of changed lines within bug fixing commits contribute to the functional correction of the code. However, much of this additional effort seems to be focused on changes to non-production code, like tests or documentation files, which is to be expected in bug fixing commits. For production code, we estimate that between 69% and 93% of the changes contribute to the bug fix. We further found that researchers are able to reliably label most data, but that multiple raters should be used as there is otherwise a high likelihood of noise within the data.

## 5 Experiments for RQ2

We now present our results and discuss the implications for RQ2 on the effect of gamification for our study.

### 5.1 Results for RQ2

Figure 6 shows how many commits were labeled by each participant and how labeling progressed over time. We observe that most participants labeled around 200 commits and that the labeling started slowly and accelerated at the end of August.<sup>18</sup> Moreover, all participants with more than 200 commits achieved at least 70% consensus and most participants were clearly above this

<sup>17</sup> For example, the pycoSHARK contains methods for checking if code is Java Production code for our projects and all standard paths for tests and documentation. The VisualSHARK and the inducingSHARK are both able to find whitespace and comment-only changes. All tools can be found on GitHub: <https://github.com/smartshark>

<sup>18</sup> Details about when we recruited participants are provided in Appendix C.

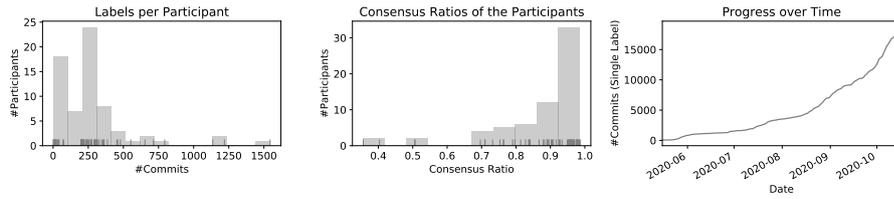


Fig. 6: Number of commits labeled per participant and number of commit labels over time.

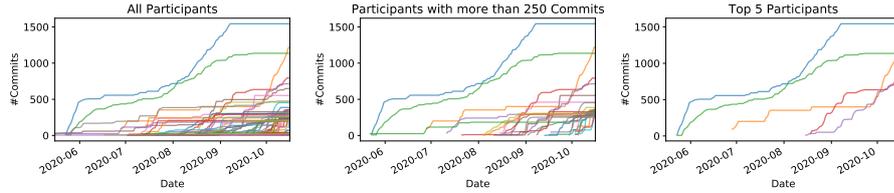


Fig. 7: Lines labeled per participants over time. Each line is one participant. The plot on the left shows the data for all participants. The plot in the middle shows only participants who labeled more than 250 commits, excluding Steffen Herbold, Alexander Trautsch, and Benjamin Ledel because their position in the author list is not affected by the number of labeled commits. The plot on the right shows only the top 5 participants.

lower boundary. Five participants that each labeled only few lines before dropping out are below 70%. We manually checked, and found that the low ratio was driven by single mistakes and that each of these participants have participated in the labeling of less than 243 lines with consensus. This is in line with our substantial agreement measured with Fleiss'  $\kappa$ .

Figure 7 shows how many lines were labeled by each participant over time. The plot on the left confirms that most participants started labeling data relatively late in the study. The plot in the middle is restricted to the participants that have labeled substantially more lines than required, which resulted in being mentioned earlier in the list of authors. We observe that many participants are relatively densely located in the area around 250 commits. These participants did not stop at 200 commits, but continued a bit longer before stopping. Seven participants stopped when they finished 200 bugs, i.e., they mixed up the columns for bugs finished with commits finished in the leaderboard, which may explain some of the data. We could see no indication that these participants stopped because they achieved a certain rank. The plot on the right shows the top five participants that labeled most data. We observe that the first and third ranked participant both joined early and consistently labeled data over time. We note that the activity of the first ranked participant decreased, until the third ranked participant got close, and then accelerated again. The second ranked participant had two active phases of labeling, both

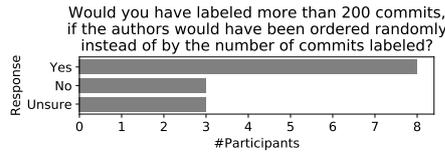


Fig. 8: Histogram of the answers to the question Q2: Would you have labeled more than 200 commits, if the authors would have been ordered randomly instead of by the number of commits labeled? 14 participants out of 26 with at least 250 commits answered this question.

of which have over 500 labeled commits. From the data, we believe that achieving the second rank over all may have been the motivation for labeling here. The most obvious example of the potential impact of the gamification on activity are the fourth and fifth ranked participants. The fourth ranked stopped labeling, until they were overtaken by the fifth ranked participant, then both changed places a couple of times, before the fourth ranked participant prevailed and stayed on the fourth rank. Overall, we believe that the line plot indicates these five participants were motivated by the gamification to label substantially more. Overall, these five participants labeled 5,467 commits, i.e., produced 5.4 times more data than was minimally required by them.

Figure 8 shows the results of our question if participants with at least 250 commits would have labeled more than 200 commits, if the author order would have been random. The survey indicates that most participants did not care about the gamification element, which is in line with the label data we observed. However, we also see that three participants answered with “No”, i.e., they were definitively motivated by the gamification and would have labeled less data otherwise. Another three answered with “Unsure”, i.e., they were also motivated by the gamification, but would possibly still have labeled the same amount of data otherwise. Thus, the gamification increased the amount of effort invested in this study for three to six participants, i.e., participants were motivated by the gamification. This means that between 7% and 13% of the recruited participants<sup>19</sup> who fulfilled the criteria for co-authorship were motivated by the gamification.

## 5.2 Discussion of RQ2

Our data yields relatively clear results. While only between 7% and 13% of participants labeled more data due to the gamification element, the impact of this was still strong. The five participants that invested the most effort labeled 5,467 commits, which is roughly the same amount of data as the 24 participants who labeled at least 200 commits and are closest to 200 commits.

<sup>19</sup> Not counting the three principal investigators Steffen Herbold, Alexander Trautsch and Benjamin Ledel

A risk of this gamification is that this could negatively affect the reliability of our results, e.g., the participants who labeled most data would be sloppy. However, this was not the case, i.e., the consensus ratios are consistently high and, hence, independent of the amount of commits that were labeled. While we have no data to support this, our requirements of a minimal consensus ratio for co-authors may have prevented a potential negative impact of the gamification, because the penalty for sloppiness would have been severe. In summary, we have the following results for RQ2.

We fail to reject H3 and believe that motivating researchers to invest more effort through gamification is worthwhile, even if only a minority is motivated.

## 6 Threats to Validity

We report the threats to the validity of our work following the classification by Cook et al. (1979) suggested for software engineering by Wohlin et al. (2012). Additionally, we discuss the reliability as suggested by Runeson and Höst (2009).

### 6.1 Construct Validity

There are several threats to the validity of our research construct. We differentiate not only between bug fixing lines and other lines, but use a more detailed differentiation of changes into whitespaces only, documentation changes, test changes, refactorings, and unrelated improvements. This could have influenced our results. Most notably, this could have inflated the number of lines without consensus. We counter this threat by differentiating between lines without consensus with at least one bug fix label, and other lines without consensus.

Another potential problem is our assumption that minority votes are random mistakes and that random mistakes follow a binomial distribution. However, we believe that unless the mistakes follow a binomial distribution, i.e., are the result of repeated independent Bernoulli experiments, they are not truly random. Our data indicates that there is support for this hypothesis, i.e., that there are also many active disagreements, but that these are unlikely to be in cases where we have consensus.

We also ignore a potential learning curve of participants for our evaluation of mistakes and, therefore, cannot rule out that the amount of mistakes decreases over time. We mitigate these issues through the five tutorial bugs that all participants had to label at the beginning, which provides not only a training for the tool usage, but possibly also flattens the learning curve for the bug labeling.

Furthermore, the participants may have accepted pre-labeled lines without checking if the heuristic pre-labeling was correct. We mitigated this bias by advising all participants to be skeptical of the pre-labels and carefully check

them, as they are only hints. The disagreements with the refactoring pre-labels indicate that participants were indeed skeptical and did not accept pre-labeled lines without checking. While all projects use Java as the main programming language, some projects also use additional languages, which means we may slightly underestimate the amount of production code.

The results regarding the gamification may be untrustworthy, because the participants were fully aware that this would be analyzed. Thus, a participant could have, e.g., avoided to label more data, if they wanted to show that gamification is ineffective, or labeled more data, if they wanted to show that gamification is effective. We have no way to counter this threat, since not revealing this aspect of the analysis as part of the pre-registered study protocol would not have been in line with our open science approach<sup>20</sup> and be ethically questionable. Instead, we rely on our participants, who are mostly researchers, to act ethical and not intentionally modify our data through such actions. Additionally, one may argue that authorship order is not really gamification, because the reward is not a game element, but rather an impact on the real world. Gamification elements that would have no tangible real world benefit, e.g., badges that could be earned while labeling, may lead to different results.

Similarly, the participants who answered the questions regarding the estimation of the lines where they were unsure or if they would have labeled more than 200 commits if the author order would have been randomized were effectively answered by the authors of this article. Thus, we could have constructed the results in a way to support our findings. Again, we can only rely on the ethical behavior of all participants but want to note that, in the end, any anonymous survey has similar risks. Moreover, we see no benefit for anybody from answering the questions incorrectly.

Finally, there were several opportunities for additional surveys among the participants, e.g., to directly ask specific participants if the gamification was a motivating factor or if they followed the provided guidance in specific aspects. We decided against this because this was not part of the pre-registered protocol and such questions could put participants into ethically problematic situations, e.g., because they may feel that stating that they were driven by curiosity instead of benefit would be beneficial. This is just hypothetical, but we still want to avoid such situations.

## 6.2 Conclusion Validity

There are no threats to the conclusion validity of our study. All statistical tests are suitable for our data, we correct our results to err on the side of caution, and the pre-registration ensured that we did not tailor our statistical analysis in a way to artificially find coincidental results, e.g., through subgroup analysis.

---

<sup>20</sup> Pre-registration, all tools are open source, all data publicly available.

### 6.3 Internal Validity

The main threat to the internal validity of our work is that our analysis of percentages of change types per commit could be misleading, because the distribution of the percentages is not mainly driven by tangling, but by a different factor. The only such factor that we consider to be a major threat is the size: larger commits could be more difficult to label and, hence, have a different distribution of mislabels. Moreover, larger commits could indicate that not only the bug was fixed, but that there are many tangled changes. Similar arguments can be used for the number of commits used to fix a bug. Figure 9 explores the relationship between the number of changed lines, as well as the number of commits and the percentage of bug fixing lines, respectively the lines without consensus. Both plots reveal that there is no pattern within the data that is related to the size and that no clear correlation is visible. We find very weak linear correlations that are significant between the number of lines changed and the ratio of bug fixing lines. However, given the structure of the data, this rather looks like a random effect than a clear pattern. The significance is not relevant in our case, as even weak correlations are significant given our amount of data. Another factor could also be that our results are project dependent. However, since this would be a case of subgroup analysis, we cannot simply extend our study protocol to evaluate this without adding a risk of finding random effects. Moreover, adding this analysis would invalidate one major aspect of pre-registrations, which is to avoid unplanned subgroup analysis.

Our decision to label data on a line level and not the character or statement level may affect the validity of our conclusions. In case changes are tangled within the same line, this is not detected by our labeling, since we advised participants to label lines as bug fixing in that case.<sup>21</sup> However, such changes are very rare, as this requires not only lines with multiple concerns, but also that these different concerns are modified within the same commit, such that at least one modification is not part of the bug fix. While we do not have data on how often exactly that was the case, our participants reported that this happened only rarely. Thus, any noise we missed this way is likely to have only a negligible effect on our results.

Another threat to the internal validity is that we may underestimate the ability of researchers to achieve consensus on tangling. More training, e.g., on how to use the labeling tool or how to deal with special cases could improve the consensus among participants in future studies. Moreover, our study design does not include a disagreement resolution phase, where participants can discuss to resolve disagreements. We opted against such a phase due to the scale of our analysis, but hope that one of the aspects of future work is to gain further insights into disagreements, including their potential resolution.

Participants may have missed the gamification aspect in the study protocol and, hence, were not aware of how the author ranking would be determined.

---

<sup>21</sup> Within the FAQ on [https://smartshark.github.io/msr20\\_registered\\_report/](https://smartshark.github.io/msr20_registered_report/)

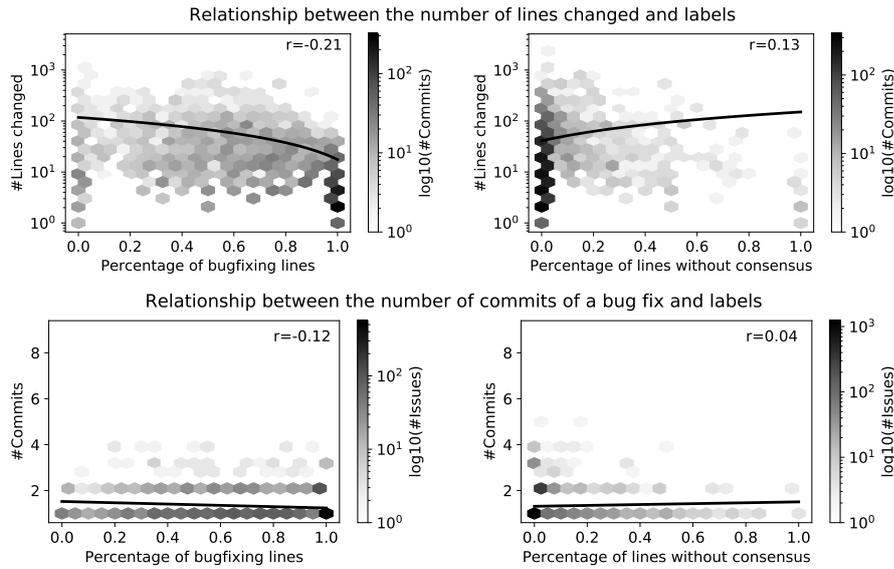


Fig. 9: Relationship between the number of lines changed in production code files and the percentage of bug fixing lines and lines without consensus. The line indicates the regression line for a linear relationship between the variables, the  $r$ -value is Pearson’s correlation coefficient Kirch (2008). The correlations are significant with  $p < 0.01$  for all values. The regression line has a curve in the upper plots, due to the logarithmic scale of the y-axis.

This could affect our conclusions regarding the impact of the gamification. However, missing the gamification aspect should only lead to underestimating the impact of the gamification, which means that our results should at least be reliable as a lower boundary. Moreover, participants may have labeled more than 200 commits, because the leaderboard was only updated once a day and they did not notice they passed 200 commits until the next day. However, our data indicates that if this was the case, this only led to a small amount of additional effort, i.e., at most 50 commits.

#### 6.4 External Validity

Due to our restriction to mature Java software, our results may not generalize to other languages or software with less mature development processes than those of the Apache projects. Moreover, the sample of projects may not be representative for mature Java projects as well, as we did not conduct random sampling, but re-used data from Herbold et al. (2019) who used a purposive sampling approach (Patton, 2014). However, Baltes and Ralph (2020) argue that well-done purposive sampling should also yield generalizable results. Moreover, our analysis was done on open source projects. While there

are many industrial contributors in the projects we studied, our results may not generalize to closed source software.

## 6.5 Reliability

We cannot rule out that the results of our study are due to the participants we recruited as fellow researchers. However, our participants are very diverse: they are working on all continents except Africa and Antarctica, the experience ranges from graduate students to full professors, and from a couple of years of development to several decades. Regardless of this, the overall consensus ratios are relatively high for all participants and Fleiss'  $\kappa$  indicates substantial agreement among our participants.

## 7 Conclusion

We now conclude our article with a short summary and an outlook on possible future work on the topic.

### 7.1 Summary

Our study shows that tangled commits have a high prevalence. While we found that most tangling is benign and can be identified by current heuristics, there is also problematic tangling that could lead to up to 47% of noise in data sets without manual intervention. We found that the identification of tangled commits is possible in general, but also that random errors have to be accounted for through multiple labels. We also found that there are about 14% of lines in production code files where participants actively disagree with each other, which we see as an indication that these lines are especially difficult to label. Overall, our study showed that tangling is a serious threat for research on bugs based on repository mining that should always be considered, at least to assess the noise which may be caused by the tangling.

### 7.2 Future Work

There are many opportunities for future work, many of them enabled directly by the data we generated: the data can be used as accurate data for program repair, bug localization, defect prediction, but also other aspects like the assessment of the relation between static analysis warnings or technical debt and bugs. Moreover, many studies were conducted with tangled data. If and how this affected the research results requires further attention. Based on prior work by Herzig et al. (2016) and Mills et al. (2020) there are already indications that tangling may have a significant impact on results. These are just

some examples of use cases for this data and we fully expect the community to have many more good ideas.

Moreover, while we have advanced our understanding of tangled commits, there are still important issues which we do not yet understand and that are out of scope of our study. Some of these aspects can be analyzed with the help of our data. For example, lines without consensus could undergo further inspection to understand why no consensus was achieved. Through this, we may not only get a better understanding of tangling, but possibly also about program comprehension. For example, small bug fixes without consensus could be an indicator that even small changes may be hard to understand, which in turn could indicate that this may be because the concept of atoms of confusion does not only apply to C/C++ (Gopstein et al., 2018), but also to Java. Benign tangling could also undergo further scrutiny: how many tangled documentation or test changes are directly related to the bug fix and how many are completely unrelated. This could also be correlated with other factors, e.g., to understand when code documentation must be updated as part of bug fixes to avoid outdated documentation.

However, there are also aspects of tangling, which should not or cannot be studied on our data. For example, how the prevalence of tangling commits varies between projects and which factors influence the degree of tangling in a project. While our data could certainly be used to derive hypothesis for this, independent data must be used to evaluate these hypotheses as this would be a case of post-hoc subgroup analysis that could lead to overinterpretation of results that are actually non-reproducible random effects. Another gap in our knowledge is tangling beyond bug fixing commits, for which we still only have limited knowledge. Moreover, while we do not see any reason why tangling should be different in other programming languages, we also have no data to support that our results generalize beyond Java. Similarly, it is unclear how our results translate to closed source development, that possibly follows stricter rules regarding the use of issue tracking systems and the implementation of changes, which could reduce the tangling.

Finally, we note that the research turk was a suitable research method for this study and we look forward to future studies that follow a similar approach. Since even the very basic gamification approach we used was effective, future studies should consider to use more game elements (e.g., badges) and evaluate if they are effective at increasing the participation.

## Acknowledgments

Alexander Trautsch and Benjamin Ledel and the development of the infrastructure required for this research project were funded by DFG Grant 402774445. Ivan Pashchenko was partially funded by the H2020 AssureMOSS project (Grant No. 952647). We thank the cloud team of the GWDG (<https://gwdg.de>) that helped us migrate the VisualSHARK to a stronger machine within

a day, to meet the demands of this project once many participants started to label at the same time.

## Author Contributions

Conceptualization: Steffen Herbold; Methodology: Steffen Herbold, Alexander Trautsch, Benjamin Ledel; Formal analysis and investigation: *all authors*; Writing - original draft preparation: Steffen Herbold; Writing - review and editing: *all authors*; Funding acquisition: Steffen Herbold; Resources: Steffen Herbold, Alexander Trautsch, Benjamin Ledel; Supervision: Steffen Herbold; Project administration: Steffen Herbold, Alexander Trautsch; Software: Steffen Herbold, Alexander Trautsch, Benjamin Ledel.

## References

- Agresti A, Coull BA (1998) Approximate is better than "exact" for interval estimation of binomial proportions. *The American Statistician* 52(2):119–126
- Anderson A, Huttenlocher D, Kleinberg J, Leskovec J (2013) Steering user behavior with badges. In: *Proceedings of the 22nd International Conference on World Wide Web*, Association for Computing Machinery, New York, NY, USA, WWW '13, p 95–106, DOI 10.1145/2488388.2488398
- Arima R, Higo Y, Kusumoto S (2018) A study on inappropriately partitioned commits: How much and what kinds of ip commits in java projects? In: *Proceedings of the 15th International Conference on Mining Software Repositories*, Association for Computing Machinery, New York, NY, USA, MSR '18, p 336–340, DOI 10.1145/3196398.3196406
- Baltes S, Ralph P (2020) Sampling in software engineering research: A critical review and guidelines. *arXiv preprint arXiv:200207764*
- Bissyandé TF, Thung F, Wang S, Lo D, Jiang L, Réveillère L (2013) Empirical evaluation of bug linking. In: *2013 17th European Conference on Software Maintenance and Reengineering*, pp 89–98, DOI 10.1109/CSMR.2013.19
- Brown LD, Cai TT, DasGupta A (2001) Interval estimation for a binomial proportion. *Statistical Science* 16(2):101–133, DOI 10.1214/ss/1009213286
- Campbell MJ, Gardner MJ (1988) Calculating confidence intervals for some non-parametric analyses. *Br Med J (Clin Res Ed)* 296(6634):1454–1456
- Cook TD, Campbell DT, Day A (1979) *Quasi-experimentation: Design & analysis issues for field settings*, vol 351. Houghton Mifflin Boston
- Dias M, Bacchelli A, Gousios G, Cassou D, Ducasse S (2015) Untangling fine-grained code changes. In: *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pp 341–350
- Do H, Elbaum S, Rothermel G (2005) Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering* 10(4):405–435

- Dunnett C (1955) A multiple comparison procedure for comparing several treatments with a control. *Journal of the American Statistical Association* 50(272):1096–1121
- Fleiss JL (1971) Measuring nominal scale agreement among many raters. *Psychological bulletin* 76(5):378
- Gazzola L, Micucci D, Mariani L (2019) Automatic software repair: A survey. *IEEE Trans Softw Eng* 45(1):34–67
- Gopstein D, Zhou HH, Frankl P, Cappos J (2018) Prevalence of confusing code in software projects: Atoms of confusion in the wild. In: 2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR), pp 281–291
- Grant S, Betts B (2013) Encouraging user behaviour with achievements: An empirical study. In: Proceedings of the 10th Working Conference on Mining Software Repositories (MSR), IEEE, pp 65–68
- Gyimesi P, Vancsics B, Stocco A, Mazinanian D, Beszedes A, Ferenc R, Mesbah A (2019) Bugsjs: a benchmark of javascript bugs. In: 2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST), pp 90–101
- Herbold S (2020) With registered reports towards large scale data curation. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results, Association for Computing Machinery, New York, NY, USA, ICSE-NIER '20, p 93–96, DOI 10.1145/3377816.3381721
- Herbold S, Trautsch A, Trautsch F, Ledel B (2019) Issues with SZZ: An empirical assessment of the state of practice of defect prediction data collection. 1911.08938
- Herbold S, Trautsch A, Ledel B (2020) Large-scale manual validation of bugfixing changes. DOI 10.17605/OSF.IO/ACNWK, URL <https://osf.io/acnwk>
- Herzig K, Zeller A (2013) The impact of tangled code changes. In: Proceedings of the 10th Working Conference on Mining Software Repositories, IEEE Press, MSR '13, p 121–130
- Herzig K, Just S, Zeller A (2013) It's not a bug, it's a feature: How misclassification impacts bug prediction. In: Proceedings of the 2013 International Conference on Software Engineering (ICSE), IEEE
- Herzig K, Just S, Zeller A (2016) The impact of tangled code changes on defect prediction models. *Empirical Software Engineering* 21(2):303–336
- Hindle A, German DM, Holt R (2008) What do large commits tell us? a taxonomical study of large commits. In: Proceedings of the 2008 International Working Conference on Mining Software Repositories, Association for Computing Machinery, New York, NY, USA, MSR '08, p 99–108, DOI 10.1145/1370750.1370773
- Hosseini S, Turhan B, Gunarathna D (2019) A systematic literature review and meta-analysis on cross project defect prediction. *IEEE Transactions on Software Engineering* 45(2):111–147
- Hutchins M, Foster H, Goradia T, Ostrand T (1994) Experiments on the effectiveness of dataflow- and control-flow-based test adequacy criteria. In:

- Proceedings of 16th International Conference on Software Engineering, pp 191–200
- Jia Y, Harman M (2011) An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering* 37(5):649–678
- Just R, Jalali D, Ernst MD (2014) Defects4j: A database of existing faults to enable controlled testing studies for java programs. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis, Association for Computing Machinery, New York, NY, USA, ISSTA 2014, p 437–440, DOI 10.1145/2610384.2628055
- Kamei Y, Shihab E, Adams B, Hassan AE, Mockus A, Sinha A, Ubayashi N (2013) A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering* 39(6):757–773
- Kawrykow D, Robillard MP (2011) Non-essential changes in version histories. In: Proceedings of the 2011 33rd International Conference on Software Engineering (ICSE), pp 351–360
- Kiczales G, Lamping J, Mendhekar A, Maeda C, Lopes C, Loingtier JM, Irwin J (1997) Aspect-oriented programming. In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Springer, pp 220–242
- Kim S, Zimmermann T, Pan K, Jr Whitehead EJ (2006) Automatic identification of bug-introducing changes. In: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE), pp 81–90, DOI 10.1109/ASE.2006.23
- Kirch W (ed) (2008) *Pearson’s Correlation Coefficient*, Springer Netherlands, Dordrecht, pp 1090–1091. DOI 10.1007/978-1-4020-5614-7\_2569
- Kirinuki H, Higo Y, Hotta K, Kusumoto S (2014) Hey! are you committing tangled changes? In: Proceedings of the 22nd International Conference on Program Comprehension, Association for Computing Machinery, New York, NY, USA, ICPC 2014, p 262–265, DOI 10.1145/2597008.2597798
- Kirinuki H, Higo Y, Hotta K, Kusumoto S (2016) Splitting commits via past code changes. In: Proceedings of the 2016 23rd Asia-Pacific Software Engineering Conference (APSEC), pp 129–136
- Kochhar PS, Tian Y, Lo D (2014) Potential biases in bug localization: Do they matter? In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, Association for Computing Machinery, New York, NY, USA, ASE ’14, p 803–814, DOI 10.1145/2642937.2642997
- Kreutzer P, Dotzler G, Ring M, Eskofier BM, Philippsen M (2016) Automatic clustering of code changes. In: Proceedings of the 2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR), pp 61–72
- Landis JR, Koch GG (1977) The measurement of observer agreement for categorical data. *Biometrics* 33(1):159–174
- Le Goues C, Holtschulte N, Smith EK, Brun Y, Devanbu P, Forrest S, Weimer W (2015) The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs. *IEEE Transactions on Software Engineering* 41(12):1236–1256

- Li Y, Wang S, Nguyen TN (2020) Dlfix: Context-based code transformation learning for automated program repair. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, Association for Computing Machinery, New York, NY, USA, ICSE '20, p 602–614, DOI 10.1145/3377811.3380345, URL <https://doi.org/10.1145/3377811.3380345>
- Martinez M, Durieux T, Sommerard R, Xuan J, Monperrus M (2016) Automatic repair of real bugs in java: a large-scale experiment on the defects4j dataset. *Empirical Software Engineering* 22(4):1936–1964, DOI 10.1007/s10664-016-9470-4, URL <https://doi.org/10.1007/s10664-016-9470-4>
- Mills C, Pantiuchina J, Parra E, Bavota G, Haiduc S (2018) Are bug reports enough for text retrieval-based bug localization? In: Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME), pp 381–392
- Mills C, Parra E, Pantiuchina J, Bavota G, Haiduc S (2020) On the relationship between bug reports and queries for text retrieval-based bug localization. *Empirical Software Engineering* pp 1–42
- Neto EC, da Costa DA, Kulesza U (2018) The impact of refactoring changes on the SZZ algorithm: An empirical study. In: Proceedings of the 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp 380–390, DOI 10.1109/SANER.2018.8330225
- Nguyen HA, Nguyen AT, Nguyen TN (2013) Filtering noise in mixed-purpose fixing commits to improve defect prediction and localization. In: Proceedings of the 2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE), IEEE, pp 138–147
- Pártachi P, Dash S, Allamanis M, Barr E (2020) Flexeme: Untangling commits using lexical flows. In: Proceedings of the 2020 ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ACM, ESEC/FSE 2020
- Patton MQ (2014) *Qualitative research & evaluation methods: Integrating theory and practice*. Sage publications
- Podsakoff PM, MacKenzie SB, Lee JY, Podsakoff NP (2003) Common method biases in behavioral research: a critical review of the literature and recommended remedies. *Journal of applied psychology* 88(5):879
- Rodríguez-Pérez G, Robles G, Serebrenik A, Zaidman A, Germán DM, Gonzalez-Barahona JM (2020) How bugs are born: a model to identify how bugs are introduced in software components. *Empirical Software Engineering* 25(2):1294–1340, DOI 10.1007/s10664-019-09781-y
- Runeson P, Höst M (2009) Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering* 14(2):131–164
- Saha R, Lyu Y, Lam W, Yoshida H, Prasad M (2018) Bugs.jar: A large-scale, diverse dataset of real-world java bugs. In: Proceedings of the 2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR), pp 10–13
- Shapiro SS, Wilk MB (1965) An analysis of variance test for normality (complete samples). *Biometrika* 52(3/4):591–611

- Śliwierski J, Zimmermann T, Zeller A (2005) When do changes induce fixes? In: Proceedings of the 2005 Int. Workshop on Mining Software Repositories (MSR), ACM, p 1–5
- Strüder S, Mukelabai M, Strüber D, Berger T (2020) Feature-oriented defect prediction. In: Proceedings of the International Systems and Software Product Line Conference (SPLC), ACM, pp 21:1–21:12, DOI 10.1145/3382025.3414960
- Tao Y, Kim S (2015) Partitioning composite code changes to facilitate code review. In: Proceedings of the 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories, pp 180–190
- Trautsch A, Herbold S, Grabowski J (2020a) A longitudinal study of static analysis warning evolution and the effects of PMD on software quality in apache open source projects. *Empirical Software Engineering* DOI 10.1007/s10664-020-09880-1
- Trautsch A, Trautsch F, Herbold S, Ledel B, Grabowski J (2020b) The smartshark ecosystem for software repository mining. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings, Association for Computing Machinery, New York, NY, USA, ICSE '20, p 25–28, DOI 10.1145/3377812.3382139
- Trautsch F, Herbold S, Makedonski P, Grabowski J (2018) Addressing problems with replicability and validity of repository mining studies through a smart data platform. *Empirical Software Engineering* 23(2):1036–1083
- Tsantalis N, Mansouri M, Eshkevari LM, Mazinanian D, Dig D (2018) Accurate and efficient refactoring detection in commit history. In: Proceedings of the 40th International Conference on Software Engineering (ICSE), ACM, pp 483–494
- Tsantalis N, Ketkar A, Dig D (2020) Refactoringminer 2.0. *IEEE Transactions on Software Engineering* DOI 10.1109/TSE.2020.3007722
- Tufano M, Pantiuchina J, Watson C, Bavota G, Poshyvanyk D (2019) On learning meaningful code changes via neural machine translation. In: Proceedings of the 41st International Conference on Software Engineering, IEEE Press, ICSE '19, p 25–36, DOI 10.1109/ICSE.2019.00021, URL <https://doi.org/10.1109/ICSE.2019.00021>
- Wang M, Lin Z, Zou Y, Xie B (2019) Cora: Decomposing and describing tangled code changes for reviewer. In: Proceedings of the 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp 1050–1061
- Werbach K, Hunter D (2012) For the win: How game thinking can revolutionize your business. Wharton Digital Press
- Wohlin C, Runeson P, Höst M, Ohlsson MC, Regnell B, Wesslen A (2012) *Experimentation in Software Engineering*. Springer Publishing Company, Incorporated
- Yamashita S, Hayashi S, Saeki M (2020) Changebeadsthreader: An interactive environment for tailoring automatically untangled changes. In: Proceedings of the 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp 657–661, DOI 10.1109/

SANER48275.2020.9054861

Yang W (1991) Identifying syntactic differences between two programs. *Software: Practice and Experience* 21(7):739–755, DOI 10.1002/spe.4380210706

## Appendix

### A Email with Instructions

Dear FIRST\_NAME,

Thank you for joining us! You will get your account soon (at most a couple of days, usually a lot faster) from Alexander via Email.

Once you have your credentials, you can access the labeling system (1).

Some more things, that are not in the tutorial video, but also important:

- The first five bugs that you are shown are scripted, i.e., everybody will have to work on these five bugs. They overlap with the tutorial. This should help you get comfortable with the system and avoid mislabels that could happen while get used to the system.
- We use the RefactoringMiner (2) to automatically label lines that are potentially refactorings to help you identify such lines. Please check this carefully. We already found several instances, where RefactoringMiner was wrong, e.g., because it overlooked side effects due to method calls.
- You can select the project you want to work on in the dropdown menu on the right side of the page. You may have to open the menu first by clicking on the three bars in the top-right corner.
- If you are not sure which project you want to work on, check the leaderboard. The leaderboard not only lists how much the other participants have labeled, but also the progress for the individual projects. Ideally, work on projects where others have also started to label issues.
- If no bugs are shown for the project you selected, the reason could also be that the labeling for this project is already finished. You can check this in the leaderboard.
- While we manually validated if the issues are really bugs and the links between the issues and the commits, there is always a chance that we made a mistake or missed something. If you think that there is a wrong link to a bug fixing commit or that an issue is not really a bug, write me an email with the bug id (e.g., IO-111) and the problem you found. We will then check this and possibly correct this in the database. All such corrections will be recorded.
- Finally, if you find any bugs in our system, either file an issue on GitHub (3) or just write me an email.

Best,

Steffen

(1) <https://visualshark.informatik.uni-goettingen.de/>

(2) <https://github.com/tsantalis/RefactoringMiner>

(3) <https://github.com/smartshark/visualSHARK>

### B Detailed Results for Lines without Consensus

	Bug fix	Doc.	Refactoring	Unrelated	Whitespace	Test
Bug fix	-	$0.1 \pm 0.0$	$3.6 \pm 0.0$	$4.2 \pm 0.0$	$0.7 \pm 0.0$	$0.2 \pm 0.0$
Documentation	$3.3 \pm 0.0$	-	$0.8 \pm 0.0$	$1.6 \pm 0.0$	$0.9 \pm 0.0$	$0.3 \pm 0.0$
Refactoring	$13.7 \pm 0.0$	$0.7 \pm 0.0$	-	$3.4 \pm 0.0$	$0.9 \pm 0.0$	$0.0 \pm 0.0$
Unrelated	$18.5 \pm 0.0$	$0.8 \pm 0.0$	$1.9 \pm 0.0$	-	$1.2 \pm 0.0$	$0.0 \pm 0.0$
Whitespace	$4.2 \pm 0.0$	$0.3 \pm 0.0$	$2.8 \pm 0.0$	$2.4 \pm 0.0$	-	$0.5 \pm 0.0$

Table 7: Estimated probabilities of random mislabels in production code files. The rows are the correct labels, the columns are the mislabels.

Table 8: Observed label combinations for lines without consensus.

Observed Labels	#Lines	Percentage
2 bug fix, 2 refactoring	3680	14.6%
2 bug fix, 2 unrelated	3516	14.0%
1 refactoring, 1 unrelated, 2 bug fix	1971	7.8%
1 bug fix, 1 whitespace, 2 unrelated	1245	5.0%
1 bug fix, 1 unrelated, 2 documentation	1130	4.5%
1 bug fix, 1 documentation, 2 unrelated	1114	4.4%
1 bug fix, 1 refactoring, 2 unrelated	1028	4.1%
1 unrelated, 1 whitespace, 2 bug fix	902	3.6%
2 bug fix, 2 documentation	811	3.2%
2 refactoring, 2 whitespace	767	3.0%
1 bug fix, 1 unrelated, 2 whitespace	719	2.9%
2 bug fix, 2 whitespace	688	2.7%
1 bug fix, 1 unrelated, 2 refactoring	655	2.6%
2 documentation, 2 refactoring	633	2.5%
2 documentation, 2 unrelated	537	2.1%
2 unrelated, 2 whitespace	494	2.0%
1 bug fix, 1 refactoring, 1 unrelated, 1 whitespace	490	1.9%
1 refactoring, 1 whitespace, 2 bug fix	489	1.9%
1 documentation, 1 unrelated, 2 bug fix	435	1.7%
1 bug fix, 1 whitespace, 2 refactoring	415	1.7%
1 refactoring, 1 unrelated, 2 whitespace	402	1.6%
1 bug fix, 1 refactoring, 2 whitespace	355	1.4%
1 refactoring, 1 unrelated, 2 documentation	297	1.2%
1 bug fix, 1 refactoring, 2 documentation	175	0.7%
1 bug fix, 1 documentation, 2 refactoring	166	0.7%
1 documentation, 1 refactoring, 2 whitespace	156	0.6%
1 bug fix, 1 documentation, 1 refactoring, 1 unrelated	148	0.6%
2 refactoring, 2 unrelated	140	0.6%
1 refactoring, 1 test, 2 whitespace	139	0.6%
1 documentation, 1 refactoring, 2 bug fix	129	0.5%
1 unrelated, 1 whitespace, 2 refactoring	116	0.5%
1 refactoring, 1 whitespace, 2 unrelated	97	0.4%
2 documentation, 2 whitespace	95	0.4%
1 refactoring, 1 whitespace, 2 documentation	92	0.4%
1 documentation, 1 test, 2 whitespace	89	0.4%
2 bug fix, 2 test	85	0.3%
1 test, 1 unrelated, 2 bug fix	77	0.3%
2 documentation, 2 test	69	0.3%
1 documentation, 1 unrelated, 2 refactoring	65	0.3%
1 bug fix, 1 test, 2 unrelated	61	0.2%
1 test, 1 whitespace, 2 documentation	44	0.2%

Continued on next page

Table 8 – continued from previous page

Observed Labels	#Lines	Percentage
1 unrelated, 1 whitespace, 2 documentation	43	0.2%
1 refactoring, 1 test, 2 documentation	31	0.1%
1 bug fix, 1 documentation, 1 unrelated, 1 white...	30	0.1%
1 test, 1 whitespace, 2 bug fix	29	0.1%
2 refactoring, 2 test	26	0.1%
1 bug fix, 1 refactoring, 2 test	26	0.1%
1 bug fix, 1 documentation, 1 refactoring, 1 whitespace	26	0.1%
1 documentation, 1 whitespace, 2 refactoring	23	0.1%
1 documentation, 1 whitespace, 2 bug fix	19	0.1%
1 refactoring, 1 unrelated, 2 test	16	0.1%
1 bug fix, 1 refactoring, 1 test, 1 unrelated	16	0.1%
1 bug fix, 1 documentation, 2 whitespace	15	0.1%
1 bug fix, 1 whitespace, 2 test	15	0.1%
1 test, 1 unrelated, 2 whitespace	12	<0.1%
1 bug fix, 1 test, 2 documentation	12	<0.1%
1 bug fix, 1 test, 2 refactoring	10	<0.1%
1 bug fix, 1 test, 2 whitespace	10	<0.1%
1 bug fix, 1 whitespace, 2 documentation	10	<0.1%
1 documentation, 1 refactoring, 2 unrelated	9	<0.1%
1 test, 1 unrelated, 2 documentation	8	<0.1%
1 documentation, 1 test, 2 unrelated	7	<0.1%
1 bug fix, 1 unrelated, 2 test	7	<0.1%
2 test, 2 whitespace	6	<0.1%
1 refactoring, 1 test, 2 bug fix	6	<0.1%
1 documentation, 1 refactoring, 1 unrelated, 1 whitespace	5	<0.1%
1 documentation, 1 unrelated, 2 whitespace	4	<0.1%
1 test, 1 whitespace, 2 refactoring	4	<0.1%
1 bug fix, 1 test, 1 unrelated, 1 whitespace	3	<0.1%
1 refactoring, 1 test, 2 unrelated	3	<0.1%
1 bug fix, 1 documentation, 2 test	2	<0.1%
1 bug fix, 1 documentation, 1 test, 1 unrelated	1	<0.1%
1 bug fix, 1 documentation, 1 test, 1 whitespace	1	<0.1%

## C Recruitment of Participants

Figure 10 shows that the number of registered participants increased throughout the timespan of the data collection phase. At the beginning, the increase was relatively slow, but recruitment picked up once we presented the registered report at the MSR 2020<sup>22</sup> and the idea about the research turk at the ICSE 2020<sup>23</sup>. Moreover, we actively advertised for the study at both conferences in the provided virtual communication media, which also helped to attract participants. Next, we see a sharp rise in the number of participants at the beginning of August. This is likely due to an announcement of the study on Twitter that we combined with asking all participants registered so far to share this and also to invite colleagues. The increase before the ASE 2020<sup>24</sup> is the result of advertising for this study in software engineering related Facebook groups. The rise during the ASE is partially attributed to our advertisement at the ASE, but also due to many participants from a large research group joining at the same time, independent of the ASE. A final push for more participants at the SCAM and ICSME 2020<sup>25</sup> was also effective and resulted in additional registrations.

<sup>22</sup> <https://2020.msrconf.org/>

<sup>23</sup> <https://conf.researchr.org/home/icse-2020>

<sup>24</sup> <https://conf.researchr.org/home/ase-2020>

<sup>25</sup> <https://icsme2020.github.io/>

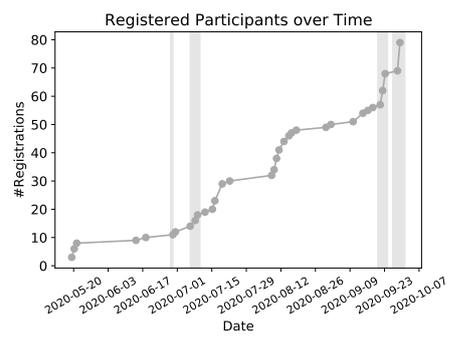


Fig. 10: Number of participants over time. The shaded areas show the time at which major software engineering conferences, during which we advertised for this project, took place. From left to right these dates are the MSR, the ICSE, the ASE, and the SCAM/ICSME.

## **G Changes in quality increasing commits**

This section contains a copy of the following publication.

A. Trautsch, J. Erbel, S. Herbold, J. Grabowski What really changes when developers intend to improve their source code: A commit-level study of static metric value and static analysis warning changes.

© 2022, The Author(s). Reprinted with permission.

journal doi follows after publication

preprint: <https://doi.org/10.48550/arXiv.2109.03544>

## What really changes when developers intend to improve their source code:

### A commit-level study of static metric value and static analysis warning changes

Alexander Trautsch · Johannes Erbel ·  
Steffen Herbold · Jens Grabowski

Received: date / Accepted: date

**Abstract** Many software metrics are designed to measure aspects that are believed to be related to software quality. Static software metrics, e.g., size, complexity and coupling are used in defect prediction research as well as software quality models to evaluate software quality. Static analysis tools also include boundary values for complexity and size that generate warnings for developers. While this indicates a relationship between quality and software metrics, the extent of it is not well understood. Moreover, recent studies found that complexity metrics may be unreliable indicators for understandability of the source code. To explore this relationship, we leverage the intent of developers about what constitutes a quality improvement in their own code base. We manually classify a randomized sample of 2,533 commits from 54 Java open source projects as quality improving depending on the intent of the developer by inspecting the commit message. We distinguish between perfective and corrective maintenance via predefined guidelines and use this data as ground truth for the fine-tuning of a state-of-the-art deep learning model for natural language processing. The benchmark we provide with our ground truth indicates that the deep learning model can be confidently used for commit intent classification. We use the model to increase our data set to 125,482 commits.

---

Alexander Trautsch  
Institute of Software and Systems Engineering, TU Clausthal, Germany  
E-mail: alexander.trautsch@tu-clausthal.de

Johannes Erbel  
Institute of Computer Science, University of Goettingen, Germany  
E-mail: johannes.erbel@cs.uni-goettingen.de

Steffen Herbold  
Institute of Software and Systems Engineering, TU Clausthal, Germany  
E-mail: steffen.herbold@tu-clausthal.de

Jens Grabowski  
Institute of Computer Science, University of Goettingen, Germany  
E-mail: grabowski@cs.uni-goettingen.de

Based on the resulting data set, we investigate the differences in size and 14 static source code metrics between changes that increase quality, as indicated by the developer, and other changes. In addition, we investigate which files are targets of quality improvements. We find that quality improving commits are smaller than other commits. Perfective changes have a positive impact on static source code metrics while corrective changes do tend to add complexity. Furthermore, we find that files which are the target of perfective maintenance already have a lower median complexity than other files. Our study results provide empirical evidence for which static source code metrics capture quality improvement from the developers point of view. This has implications for program understanding as well as code smell detection and recommender systems.

**Keywords** Static code analysis · Quality evolution · Software metrics · Software quality

## 1 Introduction

Software quality is notoriously hard to measure (Kitchenham and Pfleeger, 1996). The main reason is that quality is subjective and that it consists of multiple factors. This idea was formalized by Boehm and McCall in the 70s (Boehm et al., 1976; McCall et al., 1977). Both introduced a layered approach where software quality consists of multiple factors. The standard ISO/IEC (2001) and successor ISO/IEC (2011) also approach software quality in this fashion.

All these ideas contain abstract quality factors. However, the question remains what concrete measurements can we perform to evaluate the abstract factors of which software quality consists, i.e., how do we measure software quality. Some software quality models recommend concrete measurements, e.g., ColumbusQM (Bakota et al., 2011) and Quamoco (Wagner et al., 2012). Defect prediction researchers also try to build (machine learning) models to find a function that can map measurable metrics to the number of defects in the source code. This can also be thought of as software quality evaluation, that tries to map internal software quality, measured by code or process metrics, to external software quality measured by defects (Fenton and Bieman, 2014). The internal and external quality categories can also be mapped to perfective and corrective maintenance categories after Swanson (1976). Perfective maintenance should increase internal quality while corrective maintenance should increase external quality. Both categories should increase the overall quality of the software. To ease the readability, we adopt the perfective and corrective terms defined by Swanson for the rest of the paper when referring to the categories. For general assumptions, we adopt the internal and external quality terms. Internal quality represents what the developer sees, e.g., structure, size, and complexity while external quality what the user sees, e.g., defects.

Software quality models and defect prediction models use static source code metrics as a proxy for quality (Hosseini et al., 2017). The intuition is

that complex code, as measured by static source code metrics, is harder to reason about and, therefore, is more prone to errors. However, recent research by Peitek et al. (2021) showed that measured code complexity is perceived very differently between developers and does not translate well to code understanding. A similar result was found by Scalabrino et al. (2021) although their work is focused on readability measured in a static way. Both studies, due to their nature, observe developers in a controlled experiment with code snippets. To supplement these results, it would be interesting to measure what developers change in their code “in the wild” to improve software quality and if their intent matches what we can measure, e.g., if complexity is reduced in a change that intends to improve quality.

While there are multiple publications on maintenance or change classification after Swanson (1976), e.g., Mockus and Votta (2000), Mauczka et al. (2012), Levin and Yehudai (2017) and Hönel et al. (2019), we are not aware of publications that investigate differences between multiple software metrics for corrective and perfective maintenance as well as other changes. The inclusion of other changes results in computational effort as we need every metric for every file in every commit. However, we are able to provide this data via the SmartSHARK ecosystem (Trautsch et al., 2017, 2020b). This additional effort allows us to infer if categories of changes are different when regarding all changes of a software project. Most recent work focuses on certain aspects instead of a generic overview, e.g., how software metric values change when code smells are removed (Bavota et al., 2015) or refactorings are applied (Bavota et al., 2015; Alshayeb, 2009; Pantiuchina et al., 2020). However, we believe that taking a step back from focused approaches and investigating generic quality improvements is worthwhile. A generic overview has the advantage of mitigating possible problems that can occur for narrow meaning keywords of topically focused approaches while at the same time providing a cohesive overview. Moreover, it allows for generic statements about software quality evolution based on this information and can complement focused approaches.

In this work, we find changes that increase the quality, while we measure current, previous and delta of common source code metric values used in a current version (Bakota et al., 2014) of the Columbus quality model (Bakota et al., 2011). We use the commit message contained in each change to find commits where the intent of the developer is to improve software quality. This provides us with a view of corrective and perfective maintenance commits.

Within our study, we first classify the commit intent for a sample of 2,533 commits from 54 open source projects manually. The manual classification is provided by two researchers according to predefined guidelines. According to the overview of previous research in this area provided by AlOmar et al. (2021) our study would be the largest manual classification study of commits. We use this data as ground truth to fine-tune a state-of-the-art deep learning model for natural language processing that was pre-trained exclusively on software engineering data (von der Mosel et al., 2021). After we determine the performance of the model, we classify all commits, increasing our data to 125,482 commits.

We use the automatically classified data to conduct a two part study. The first part is a confirmatory study into the expected behavior of metric values for quality increasing changes. Expected behaviour, e.g., complexity is reduced in quality increasing changes is derived as hypothesis from existing quality models and the related literature.

In case our data matches the expected behavior from the literature, we can confirm the postulated theories and provide evidence in favor of using the measurements. Otherwise, we try to establish which metrics may be unsuitable for quality estimation, including the potential reasons. Even further, we determine whether metrics used in software quality models are impacted by quality increasing maintenance, therefore providing an evaluation for software quality measurement metrics.

The second part of our study is of exploratory nature. We investigate which files are the target of quality improvements by the developers. We explore whether only complex files are receiving perfective changes and which metric values are indicative of corrective changes. This provides us with data for practitioners and static analysis tool vendors for boundary values which are likely to have a positive impact on the quality of source code from the perspective of the developers.

Overall, our work provides the following contributions:

- A large data set of manual classifications of commit intents with improving internal and external quality categories.
- A confirmatory study of size and complexity metric value changes for quality improvements.
- An exploratory study of size and complexity metric values of files that are the target of quality improvements.
- A fine-tuned state-of-the-art deep learning model for automatic classification of commit intents.

The main findings of our study are the following:

- We confirm previous work that quality increasing commits are smaller than other changes.
- While perfective changes have a positive impact on most static source code metric values, corrective changes have a negative impact on size and complexity.
- The files that are the target of perfective changes are already less complex and smaller than other files.
- The files that are the target of corrective changes are more complex and larger than other files.

The remainder of this paper is structured as follows. In Section 2, we define our research questions and hypotheses. In Section 3, we discuss the previous work related to our study. Section 4 contains our case study design with descriptions for subject selection as well as data sources and analysis procedure. In Section 5, we present the results of our case study and discuss them in Section 6. Section 7 lists our identified threats to validity and Section 8 closes with a conclusion of our work.

## 2 Research Questions and Hypotheses

In our study, we answer two research questions.

- **RQ1:** *Does developer intent to improve internal or external quality have a positive impact on software metric values?*

Previous work provides us with certain indications about the impact on software metric values. This is part of our confirmatory study, and we derive two hypotheses from previous work regarding how size and software metric values should change for different types of quality improvement. We formulate our assumptions as hypothesis and test these in our case study.

- **H1: Intended quality improvements are smaller than other changes.**

Mockus and Votta (2000) found that corrective changes modify fewer lines while perfective changes delete more lines. Purushothaman and Perry (2005) also observed more deletions for perfective maintenance and an overall smaller size of perfective and corrective maintenance. Both studies provide measurements we base our hypothesis on. While they are using the same closed source project we will be able to see if our assumption holds for our multiple Java open source projects.

Hönel et al. (2019) used size-based metrics as additional features for an automated approach to classify maintenance types. They found that the size-based metric values increased the classification performance. Moreover, just-in-time quality assurance (Kamei et al., 2013) builds on the assumption that changes and metrics derived from these changes can predict bug introduction, meaning there should be a difference. Therefore, we hypothesize that corrective as well as perfective maintenance consist of smaller changes. Addition of features should be larger than both, and therefore we assume that the categories we are interested in, perfective and corrective, are smaller than other changes.

- **H2: Intended quality improvements impact software quality metric values in a positive way.**

In this paper, we focus on metrics used in the Columbus Quality Model (Bakota et al., 2011; Bakota et al., 2014). The metrics are specifically chosen for a quality model so they should provide different measurements based on their maintenance category. Prior research, e.g., Ch’avez et al. (2017) and Stroggylos and Spinellis (2007) found that refactorings, which are part of our classification, have a measurable impact on software metric values. We hypothesize that an improvement consciously applied by a developer via a perfective commit has a measurable, positive impact on software metric values. Positive means that we expect a value change direction of the metric value, e.g., complexity is reduced. We note our expected direction for each metric together with a description in Table 4.

Defect prediction research assumes a connection between software metrics and external software quality in the form of bugs. While most publications in defect prediction are not investigating the impact of single

bug fixing changes the most common datasets all contain coupling, size and complexity metrics as independent variables, e.g., (Jureczko and Madeyski, 2010; NASA, 2004; D’Ambros et al., 2012), see also the systematic literature review by Hosseini et al. (2017). We hypothesize that fixing bugs via corrective commits has a measurable, positive impact on software metric values. While a bug fix may add complexity, our study compares bug fix changes with all other changes including feature additions. Therefore, we do not hypothesize that bug fixing decreases complexity generally, but that it is decreasing complexity in comparison to all other changes. In contrast to **H1** we are not able to compare our results to concrete studies as we are not aware of a study that investigates metric value changes of perfective and corrective changes and compares them against all other changes. We are instead trying to validate the assumption that quality improvements should have a positive impact on software quality metrics as they are found to improve detection of defects (Gyimothy et al., 2005).

Our second research question is exploratory in nature.

- **RQ2:** *What kind of files are the target of internal or external quality improvements?*

The first part of our study provides us with information about metric value changes for quality increasing commits. In this part, we are exploring which files are the target of quality increasing commits. We are interested in how complex, e.g., via cyclomatic complexity, a file is on average that receives perfective maintenance. Moreover, on the external quality side we are interested in which files are receiving corrective changes. Due to the exploratory nature of this research question, we do not derive hypotheses.

### 3 Related Work

We separate the discussion of the related work into publications on the classification of changes and publications on the relation between quality improvements and software metrics.

Most prior work that follows a similar approach to ours is concerned with specific types of quality improving changes, e.g., refactoring and removal of code smells. We note that some code smell detection is based on internal software quality metrics, which we use in our study.

We first present previous research related to the first phase of our study, i.e., classification of changes with respect to maintenance types. Mockus and Votta (2000) study changes in a large system and identified reasons for changes. They find that a textual description of the change can be used to identify the type of change with a keyword based approach which they validated with a developer survey. The authors classified changes to Swansons maintenance types. They find that corrective and perfective changes are smaller and that perfective changes delete more lines than other changes. Mauczka et al. (2012)

present an automatic keyword based approach for classification into Swansons maintenance types. They evaluate their approach and provide a keyword list for each maintenance type together with a weight.

Fu et al. (2015) present an approach for change classification that uses latent dirichlet allocation. They study five open source projects and classify changes into Swansons maintenance types together with a *not sure* type. The keyword list of their study is based on Mauczka et al. (2012).

Mauczka et al. (2015) collect developer classifications for three different classification schemes. Their data contains 967 commits from six open source projects. While the developers themselves are the best source of information, we believe that within the guidelines of our approach our classifications are similar to those of the developers. We evaluate this assumption in Section 4.2.

Yan et al. (2016) use discriminative topic modeling also based on the keyword list by Mauczka et al. (2012). They focus on changes with multiple categories. Levin and Yehudai (2017) improve maintenance type classification by utilizing source code in addition to keywords. This is an indication that metric values which are computed from source code are impacted by different maintenance types.

Hönel et al. (2019) use size metrics as additional features for automated classification of changes. In our study, we first classify the change and then look at how this impacts size and spread of the change. However, the differences we found in our study support the assumption that size-based features can be used to distinguish change categories.

More recently, Wang et al. (2021) also analyze developer intents from the commit messages. They focus on large review effort code changes instead of quality changes or maintenance types. They also use a keyword based heuristic for the classification. They do not, however, include a perfective maintenance classification.

Ghadhab et al. (2021) also use a deep learning model to classify commits. They use word embeddings from the deep learning model in combination with fine-grained code changes to classify into Swansons maintenance categories. In contrast to Ghadhab et al., we do not include code changes in our automatic classifications and focus on the commit message.

The classification of changes for the ground truth in our study is based on manual inspection by two researchers instead of a keyword list. We specify guidelines for the classification procedure which enable other researchers to replicate our work. To accept or reject our hypotheses, we only inspect internal and external quality improvements which would correspond to the perfective and corrective maintenance types by Swanson. In contrast to the previous studies, we relate our classified changes also to a set of static software metrics.

We now present research related to our second phase of our study, the relation between intended quality improvements and software metrics. Stroggylos and Spinellis (2007) found changes where the developers intended a refactoring via the commit message. The authors then measured several source code metrics to evaluate the quality change. In contrast to the work of Stroggylos and Spinellis (2007), we do not focus on refactoring keywords. Instead, we con-

sider refactoring as a part of our classification guidelines. Moreover, our aim is to investigate whether the metrics most commonly used as internal quality metrics (see also Al Dallal and Abdin (2018)) are the ones that are changing if developers perform quality improving changes including refactoring.

Fakhoury et al. (2019) investigated the practical impact of software evolution with developer perceived readability improvements on existing readability models. After finding target commits via commit message filtering they applied state-of-the-art readability models before and after the change and investigated the impact of the change on the resulting readability score.

Pantiuchina et al. (2018) analyze commit messages to extract the intent of the developer to improve certain static source code metrics related to software quality. In contrast to their work, we are not extracting the intent to improve certain static code metrics but instead focus on overall improvement to measure the delta of a multitude of metrics between the improving commit and its parents. Developers may not use the terminology Pantiuchina et al. base their keywords on, e.g., instead of writing reduce coupling or increase cohesion the developer may simply write refactoring or simplify code.

In contrast to the previous studies, we relate developer intents to improve the quality either by perfective maintenance or by corrective maintenance to change size metrics and static source code metrics. In addition, we also look at mean static source code metrics per file which are the target of quality improvements.

## 4 Case Study Design

The goal of our case study is to gather empirical data about what changes when a developer intends to improve the quality of the code base in comparison to other changes. To achieve this, we first sample a number of commits from our selected study subjects. This sample is classified by two researchers into two categories of quality improving and other changes. The classification into categories is only done via the commit message as it expresses the intent of the developer on what the change should achieve.

This data is then used to train a model that can confidently classify the rest of our commit messages. The classified commits are then used to investigate the static source code metric value changes to accept or reject our hypotheses in the confirmatory part of our study. After that, we investigate the metric values before the change is applied in the exploratory part of our study.

### 4.1 Data and Study Subject Selection

The data used in our study is a SmartSHARK (Trautsch et al., 2017) database taken from Trautsch et al. (2020a). We use all projects and commits in the database. However, only commits that change production code and which are not empty are considered. For each change in our data, we extract a list of

Table 1: Case study subjects with time frame and distribution of commits. All considered commits (#C), sample size (#S), sample perfective commits (#SP), sample corrective commits (#SC), all perfective commits (#AP), all corrective commits (#AC)

Project	Timeframe	#C	#S	#SP	#SC	#AP	#AC
archiva	2005-2018	3,914	79	35	17	1,478	1,005
calcite	2012-2018	1,987	40	8	14	565	665
cayenne	2007-2018	3,738	75	31	14	1,470	1,007
commons-bcel	2001-2019	884	18	9	6	588	171
commons-beanutils	2001-2018	577	12	5	2	317	130
commons-codec	2003-2018	828	17	12	1	619	76
commons-collections	2001-2018	1,827	37	27	3	1,185	200
commons-compress	2003-2018	1,598	32	17	6	873	317
commons-configuration	2003-2018	2,075	42	23	7	1,027	253
commons-dbcp	2001-2019	1,034	21	15	3	672	211
commons-digester	2001-2017	1,256	26	16	0	744	113
commons-imaging	2007-2018	682	14	10	2	476	96
commons-io	2002-2018	1,036	21	15	3	613	171
commons-jcs	2002-2018	788	16	10	1	400	162
commons-jexl	2002-2018	1,469	30	20	1	873	199
commons-lang	2002-2018	3,261	66	50	6	2,182	420
commons-math	2003-2018	4,675	94	66	10	2,981	574
commons-net	2002-2018	1,092	22	13	5	585	246
commons-rdf	2014-2018	529	11	9	0	341	35
commons-scxml	2005-2018	479	10	6	2	256	76
commons-validator	2002-2018	1,573	32	18	6	900	296
commons-vfs	2002-2018	1,136	23	11	8	628	207
eagle	2015-2018	582	12	5	4	104	199
falcon	2011-2018	1,547	31	7	13	255	676
flume	2011-2018	1,489	30	5	14	266	591
giraph	2010-2018	854	18	4	6	201	281
gora	2010-2019	569	12	3	4	182	141
helix	2011-2019	2,199	44	8	9	552	580
httpcomponents-client	2005-2019	2,399	48	22	16	1,113	639
httpcomponents-core	2005-2019	2,598	52	25	12	1,326	544
jena	2002-2019	8,698	174	88	34	4,163	1,424
jspwiki	2001-2018	4,326	87	32	25	1,523	941
knox	2012-2018	1,131	23	3	10	266	306
kylin	2014-2018	6,789	136	40	40	1,904	2,163
lens	2013-2018	1,370	28	9	9	321	479
mahout	2008-2018	2,075	42	16	15	836	467
manifoldcf	2010-2019	2,867	58	10	21	602	1,164
mina-sshd	2008-2019	1,281	26	10	6	381	396
nifi	2014-2018	3,299	66	12	18	592	1,052
opennlp	2008-2018	1,763	36	22	6	805	275
parquet-mr	2012-2018	1,228	25	7	9	439	316
pdfbox	2008-2018	8,256	166	81	69	3,934	2,904
phoenix	2014-2019	7,835	157	23	83	828	4,545
ranger	2014-2018	2,213	45	10	20	434	908
roller	2005-2019	2,435	49	15	13	869	723
santuario-java	2001-2019	1,455	30	14	5	627	406
storm	2011-2018	2,839	57	24	9	987	716
streams	2012-2019	911	19	7	2	264	196
struts	2006-2018	2,945	59	21	18	1,191	682
systemml	2012-2018	3,860	78	21	25	921	1,416
tez	2013-2018	2,359	48	8	27	443	1,223
tika	2007-2018	2,581	52	11	10	705	740
wss4j	2004-2018	2,455	50	22	10	712	702
zeppelin	2013-2018	1,836	37	11	6	333	699
		125,482	2,533	1,022	685	47,852	35,124

changed files, the number of changed lines, the number of hunks<sup>1</sup>, and the delta as well as the previous and current value of source code metrics from the changed files between the parent and the current commit. To create our ground truth sample, we randomly sample 2% of commits per project rounded up for manual classification.

The data consists of Java open source projects under the umbrella of the Apache Software Foundation<sup>2</sup>. All projects use an issue tracking system and were still active when the data was collected. Each project consist of at least 100 files and 1000 commits and is at least two years old. Table 1 shows every project, the number of commits and the years of data we consider for sampling. In addition, we include the number of perfective and corrective commits for our ground truth and final classification.

#### 4.2 Change Type Classification Guidelines

As we are not relying on a keyword based approach and there is no existing guideline for this kind of classification, we created a guideline based on Herzig et al. (2013). Our ground truth consists of a sample of changes which we manually classified into perfective, corrective, and other changes. We do not consider adaptive changes as separate a category. Instead, we include them in the other changes. The reason is that we focus on internal and external quality improvements and map perfective to internal quality and corrective to external quality. Every commit message is inspected independently by two researchers with software development experience. The inspection is using a graphical frontend that loads the sample and displays the commit message which can then be assigned a label by each researcher independently. If the commit message does not provide enough information, we inspect additional linked information in the form of bug reports or the change itself. In case of a link between the commit message and the issue tracking system, we inspect the bug report and determine if it is a bug according to the guidelines by Herzig et al. (2013). We perform this step because the reporter of a bug sometimes assigns a wrong type. We defined the guidelines listed in Table 2 used by both researchers for the classification of changes. The deep learning model for our final classification of intents only receives the commit messages. This is a conscious trade-off. On the one hand we want the ground truth to be as exact as possible, on the other hand we want to keep the automatic intent classification as simple as possible. The results of our fine-tuning evaluation (Table 3) show that the model does not need the additional data from changes and issue reports to perform well.

Both researchers achieve a substantial inter-rater agreement (Landis and Koch, 1977) with a Kappa score of 0.66 (Cohen, 1960). Disagreements are discussed and assigned a label both researchers agree upon after discussion.

---

<sup>1</sup> An area within a file that is changed.

<sup>2</sup> <https://www.apache.org>

The disagreement front end shows both prior labels anonymized in random order.

In contrast to the classification by Mauczka et al. (2015) and Hattori and Lanza (2008), we do not categorize release tagging, license or copyright corrections as perfective. Our rationale is that these changes are not related to the code quality, which is our main interest in this study.

In Mauczka et al. (2015) the researchers selected six projects and seven developers with personal commitment and provided the developers with the commit messages that they then labeled according to different classification schemes. One of which is the Swanson classification which matches our study. Each developer labeled a sample of commit messages from their respective project. As we are focused on Java we also use the Java projects of the Mauczka et al. (2015) dataset to validate our guidelines.

Two authors of this paper re-classified the Java projects from Mauczka et al. (2015): Deltaspike, Mylyn-reviews and Tapiji. The commit messages were classified separately first. Disagreements were then resolved together in a separate session. In the first session both authors achieve a substantial inter-rater agreement (Landis and Koch, 1977) with a Kappa score of 0.62 (Cohen, 1960).

Aside from the classification differences regarding release tagging, license or copyright changes, we noticed further differences. Several commits contain some variation of “minor bugfixes” which are classified as perfective maintenance by the developers or both corrective and perfective, whereas we classify them as corrective. Additionally, code removal or test additions were not classified as perfective changes by the developers, but rather as corrective changes. This reveals a difference of perspective between researchers and developers. We consider pure code removal and test additions as perfective instead of corrective as we think of corrective changes as improving external quality, e.g., by fixing a customer facing bug. The data also contains clean-up and removal messages without a hint of an underlying bug which are classified as corrective by the developers. Based on the information available to us, we cannot decide if these are misclassifications by the developers, the result of differences in the classification guidelines, or misclassifications by us due to lack of in-depth knowledge about the projects.

The authors achieve a substantial inter-rater agreement (Landis and Koch, 1977) with the developers yielding a Kappa score of 0.63 (Cohen, 1960).

### 4.3 Deep Learning For Commit Intent Classification

In order to use all available data, we use a deep learning model that classifies all data which is not manually classified into perfective, corrective or other. Due to the size of state-of-the-art deep learning models and the computing requirements for training them, a current best practice is to use a pre-trained model which was trained unsupervised on a large data set. The model is then fine-tuned on labeled data for a specific task.

Table 2: Classification rules and examples, footnotes denote different commit messages from our data.

---

A change is classified as *perfective* if . . .

1. the commit message says code is removed or marked as deprecated.
2. code is moved to new packages.
3. generics are introduced, new Java features are used, existing code is switched to collections, or class members are switched to final.
4. documentation is improved or example code is updated.
5. static analysis warnings are fixed even though no related bug is reported.
6. code is reformatted or the readability is otherwise improved (e.g. whitespace fixes or tabs to spaces).
7. existing code is cleaned up, simplified, or its efficiency improved.
8. dependencies are updated.
9. developer tooling is improved, e.g., build scripts or logging facilities.
10. the repository layout is cleaned, e.g., by removing compiled code or maintaining .gitignore files.
11. tests are improved or added.

**Examples:** *Eliminated unused private field. JIRA: DBCP-255*<sup>3</sup> Because of other null checks it was already impossible to use the field. Thus, this is clean up. *[CODEC-127] Non-ascii characters in source files*<sup>4</sup> While the linked issue is a bug, it only affects IDEs for developers and not the compiled code. Thus, this is an improvement of developer tooling. *JEXL-240: Javadoc*<sup>5</sup> The message indicates that this commit only improved the code comments. Therefore, it is classified as perfective.

---

A change is classified as *corrective* if . . .

1. the commit message mentions bug fixes.
2. the commit message or the linked issue mentions that a wrong behaviour is fixed.
3. the commit message or the linked issue mentions that a NullPointerException is fixed.
4. a bug report is linked via the commit message that is of type bug and is not just a feature request in disguise (see Herzig et al. (2013)).

**Examples:** *KYLIN-940 ,fix NPE in monitor module ,apply patch from Xiaoyu Wang*<sup>6</sup> This fixes a NullPointerException that is visible to the end user. *owl syntax checker (bug fixes)*<sup>7</sup> Fixes a wrong behavior.

---

A change is classified as *other* if . . .

1. the commit message mentions feature or functionality addition.
2. the commit message mentions license information or copyrights changes.
3. the commit message mentions repository related information with unclear purpose, e.g., merges of branches without information, tagging of releases.
4. the commit message mentions that a release is prepared.
5. an issue is linked via the commit message that requests a feature.
6. any of the 1-5 are tangled with a perfective or corrective classification.

**Examples:** *KYLIN-715 fix license issue*<sup>8</sup> License changes or additions are not direct improvements of source code. *Support the alpha channel for PAM files. Fix the alpha channel order when reading and writing. Add various tests.*<sup>9</sup> This change adds support for a new feature, fixes something and adds tests, it is therefore highly tangled and we do not classify it as either or both.

---

Table 3: Change classification model performance comparison.

Model	Acc.	F1	MCC	Description
von der Mosel et al. (2021)	0.80	0.79	0.70	BERT model pre-trained on software engineering data, fine-tuned with only commit messages
Ghadhab et al. (2021)	0.78	0.80	-	BERT model pre-trained on natural language, includes code changes.
Gharbi et al. (2019)	-	0.46	-	Multi-label active learning, only commit message
Levin and Yehudai (2017)	0.76	-	-	Keywords and code changes, Random Forest model
Hönel et al. (2019)	0.80	-	-	LogitBoost model, includes code density.

To achieve a high performance, we use seBERT (von der Mosel et al., 2021), a model that is pre-trained on textual software engineering data in two common Natural Language Processing (NLP) tasks. Masked Language Model (MLM) and Next Sentence Prediction (NSP) which predict randomly masked words in a sentence and the next sentence respectively. Combined, this allows the model to learn a contextual understanding of the language. While von der Mosel et al. (2021) include a similar benchmark based on our ground truth data, it only used the perfective label, i.e., a binary classification to demonstrate text classification for software engineering data. In our study, we measure performance of the multi-class case with all three labels, perfective, corrective and other. Within this study, we first use our ground truth data to evaluate the multi-class performance of the model. We perform a 10x10 cross-validation which splits our data into 10 parts and uses 9 for fine-tuning the model and one for evaluating the performance. The fine-tuning itself splits the data into 80% training and 20% validation. The model is then fine-tuned and evaluated on the validation data for each epoch. At the end the best epoch is chosen to classify the test data of the fold. This is repeated 10 times for every fold which yields 100 performance measurements.

Our experiment shows sufficient performance comparable to other state-of-the-art models for commit classification. We provide the final fine-tuned model as well as the fine-tuning code as part of our replication kit for other researchers. Performance wise our model is comparable to Ghadhab et al. (2021) and improves performance compared other studies, e.g., Gharbi et al. (2019); Levin and Yehudai (2017). However, we note that we fine-tuned the model with only the labels used in our study, i.e., perfective, corrective and other. Therefore, it cannot be used or directly compared with models that support other commit classification labels. This would require the same data and labels, we can only compare the given model performance metrics, which we do in Table 3. If we look at the overview of commit classification studies by AlOmar et al. (2021) we can see that our model outperforms the other models for comparable tasks where accuracy or F-measure is given. While this is

Table 4: Static source code metrics and static analysis warning severities used in this study including the expected direction of their values in quality increasing commits.

Name and Description	Abbrev	↕
Cyclomatic Complexity (McCabe, 1976) The number of independent control-flow paths.	McCC	↓
Logical Lines of Code Number of lines in a file without comments and empty lines.	LLOC	↓
Nesting Level else-if Maximum of nesting level in a file.	NLE	↓
Number of parameters in a method The sum of all parameters of all methods in a file.	NUMPAR	↓
Clone Coverage Ratio of code covered by duplicates.	CC	↓
Comment lines of code Sum of commented lines.	CLOC	↑
Comment density Ratio of CLOC to LLOC.	CD	↑
API Documentation Number of documented public methods, +1 if class is documented.	AD	↑
Number of Ancestors Number of classes, interfaces, enums from which the class is inherited.	NOA	↓
Coupling between object classes Number of used classes (inheritance, function call, type reference).	CBO	↓
Number of Incoming Invocations Other methods that call the current class.	NII	↓
Minor static analysis warnings E.g., brace rules, naming conventions.	Minor	↓
Major static analysis warnings E.g., type resolution rules, unnecessary/unused code rules.	Major	↓
Critical static analysis warnings E.g., equals for string comparison, catching null pointer exceptions.	Critical	↓

evidence that our model can perform our required commit intent classification a throughout comparison of different commit intent classification approaches is not within the scope of this study.

#### 4.4 Metric Selection

The metric selection is based on the Columbus software quality model by Bakota et al. (2011). The metrics are selected from the current version of the model also in use as QualityGate (Bakota et al., 2014). The current model consists of 14 static source code metrics related to size, complexity, documentation, re-usability and fault-proneness. While the quality model provides us

with a selection of metrics, we do not use it directly as it requires a baseline of projects before estimating quality of a candidate project.

Table 4 shows the metrics utilized in this study, a short description, and the direction which we assume they change in quality improving commits. As most of the metrics are size and complexity metrics, we expect that their values decrease in comparison to all other commits. The metrics we expect to increase in quality improving commits are commented lines of code, comment density, and API documentation, as added documentation should increase these metrics. The three bottom rules consist of static analysis warnings from PMD<sup>10</sup> aggregated by severity for every file. We are of the opinion that this selection strikes a good balance of size, complexity, documentation, clone, and coupling based metrics.

As we are interested in static source code metrics in a commit granularity, we sum the metrics values for all files that are changed within a commit. In addition, we extract meta information about each change. The static source code metrics are provided by a SmartSHARK plugin using the OpenStaticAnalyzer<sup>11</sup>. To answer our research question, we provide the delta of the metric value changes as well as their current and previous value.

#### 4.5 Analysis Procedure

For our confirmatory study as part of **RQ1**, we compare the difference between two samples. To choose a valid statistical test of whether there is a difference between both samples, we first perform the Shapiro-Wilk test (Wilk and Shapiro, 1965) to test for normality of each sample. Since we found that the data is non-normal, we perform the Mann-Whitney U-test (Mann and Whitney, 1947) to evaluate if the metric values of one population dominates the other. Since we have an expectation about the direction of metric changes, we perform a one-sided Mann-Whitney U test. The  $H_0$  hypothesis is that both samples are the same, the alternative hypothesis is that one sample contains lower or higher values depending on our expectation. The expected direction of the metric value change is noted in the last column of Table 4.

As our data contains a large number of metrics, we cannot assume a statistical test with  $p < 0.05$  is a valid rejection of a  $H_0$  hypothesis. To mitigate the problem posed by a high number of statistical tests, we perform Bonferroni correction (Abdi, 2007). We choose a significance level of  $\alpha = 0.05$  with Bonferroni correction for 192 statistical tests. They consist of four size metrics with two groups and three statistical tests as well as 14 source code metrics with two groups and three statistical tests (normality tests for two samples and Mann-Whitney U for difference between samples). The second part is repeated for **RQ2**. We reject the  $H_0$  hypothesis that there is no difference between samples at  $p < 0.00026$ .

---

<sup>10</sup> <https://pmd.github.io/>

<sup>11</sup> <https://openstaticanalyzer.github.io/>

To calculate the effect size of the Mann-Whitney U test, we use Cliff’s  $d$  (Cliff, 1993) as a non-parametric effect size measure. We follow a common interpretation of  $d$  values Griessom and Kim (2005):  $d < 0.10$  is negligible,  $0.10 \leq d < 0.33$  is small,  $0.33 \leq d < 0.474$  is medium and  $d \geq 0.474$  is large. We provide the effect size for every difference that is statistically significant.

We report the results visually with box plots. The box plots shows three groups: all, perfective and corrective, this allows us to show the values for each metric for each group and serves to highlight the differences. Additionally, we report the differences between each group and its counterpart, e.g., perfective and not perfective in the tables where we report the statistical differences.

A more detailed description of the procedure for each hypothesis follows. For **H1**, we compare the structure of quality improving changes with every other change. We compare the size (changed lines) and diffusion (number of hunks, number of changed files) to evaluate the hypothesis. We visualize the results with box plots and report results for statistical tests to determine if the difference in samples is statistically significant.

For **H2**, we also visualize the results via box plots. As most of the differences hover around zero, we transform the data before plotting via  $sign(x) \cdot \log(abs(x + 1))$ . As we are interested in the differences between changes of metric values, we also require  $x \neq 0 : \forall x \in X$  where  $X$  is the complete, non-transformed data set for the visualizations. Due to the difference in changes, we provide our data size corrected, e.g., the delta of McCC is divided by the modified lines. Additionally, we report the percentage of data that is non-zero to indicate how often the measurements are changing in our data. In addition to the visualization, we provide a table with differences between the samples and statistical test results.

As part of our exploratory study for answering **RQ2**, we also provide box plots of our metric values. Instead of transformed delta values, we provide the raw averages per file in a change before the change was applied. In addition, we provide the median values of all of our metrics before the change was applied. In this part, we apply a two-sided Mann-Whitney U test as we have no expectation of the direction the metrics change into for the categories. To complement the visualization, we also provide density plots for both categories. They show the overlap between the perfective and corrective changes.

## 4.6 Replication Kit

All data and source code can be found in our replication kit (Trautsch et al., 2021). In addition, we provide a small website for this publication that contains all information and where the fine-tuned model can be tested live<sup>12</sup>.

<sup>12</sup> [https://user.informatik.uni-goettingen.de/~trautsch2/emse\\_2021/](https://user.informatik.uni-goettingen.de/~trautsch2/emse_2021/)

## 5 Results

In this section, we first present the results for evaluating our hypotheses of our first research question. After that, we describe the results of the exploratory part of our study for our second research question.

### 5.1 Confirmatory Study

We first present the results of our confirmatory study and evaluate our hypotheses. These results answer our first research question: Does developer intent to improve internal or external quality have a positive impact on software metric values?

Table 5: Statistical test results for perfective and corrective commits, Mann-Whitney U test  $p$ -values ( $p$ -value) and effect size ( $d$ ) with category,  $n$  is negligible,  $s$  is small. Statistically significant  $p$ -values are bolded.

Metric	Perfective		Corrective	
	$p$ -value	$d$	$p$ -value	$d$
#lines added	< <b>0.0001</b>	0.20 (s)	< <b>0.0001</b>	0.21 (s)
#lines deleted	< <b>0.0001</b>	0.15 (s)	< <b>0.0001</b>	0.16 (s)
#files modified	0.2081	-	< <b>0.0001</b>	0.22 (s)
#hunks	< <b>0.0001</b>	0.01 (n)	< <b>0.0001</b>	0.22 (s)

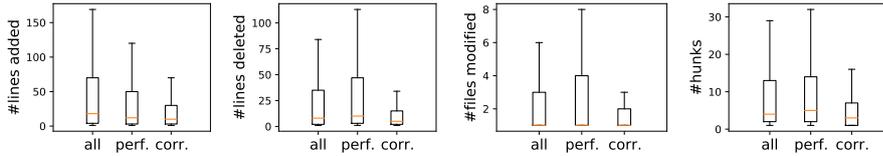


Fig. 1: Commit size distribution over all projects for all, perfective and corrective commits. Fliers are omitted.

### 5.1.1 Results H1

Figure 1 shows the distribution of sizes between perfective, corrective, and all commits. Table 5 shows the statistical test results for the differences. We can see that perfective commits tend to add fewer lines but instead remove more lines as the other commits. When we calculate a median delta between all commits and perfective commits, we find a difference of 28 for added lines and -2 for deleted lines. While the effect sizes are negligible to small, we can see this difference also in Figure 1. The diffusion of the change over files is also different, however for the number of modified files the difference is not significant for perfective commits.

Corrective commits also tend to add less code, while they do not delete as much, the difference in added and deleted lines is also statistically significant. While the effect size is small, we can see the difference in Figure 1. For corrective commits, we can also see a difference in the number of files changed and the number of hunks modified. This diffusion of the change via the number of files and hunks is also statistically significant although, again, with a small effect size.

We can conclude, that perfective commits tend to remove more lines, and are generally adding fewer lines to the repository. Corrective commits delete fewer lines and add fewer lines than other commits. Corrective commits are also distributed over less hunks and less files than other commits.

Table 6: Percentage of commits where the metric value is not zero on all commits (%nz), perfective commits (%nz p) and corrective commits (%nz c).

Metric	%NZ	%NZ P	%NZ C
McCC	51.03	31.01	57.70
LLOC	74.69	60.93	77.99
NLE	36.76	23.92	34.28
NUMPAR	35.93	24.44	24.98
CC	49.41	37.81	55.14
CLOC	51.56	46.52	42.51
CD	76.07	66.48	77.35
AD	27.19	20.63	15.82
NOA	10.51	6.96	3.62
CBO	30.89	22.52	22.22
NII	27.08	17.78	21.09
Minor	36.15	27.02	29.77
Major	19.87	13.23	14.77
Critical	7.23	4.20	4.95

We **accept H1** that intended quality improvements are smaller than other changes. Perfective and corrective commits tend to add fewer lines, perfective commits remove more lines. The effect size is negligible to small in all cases.

### 5.1.2 Results H2

We first note that no metric value changes for each instance of our data. This can be seen in Table 6, which shows the percentages for each metric value for perfective, corrective, and all changes. We can see some differences between changes, e.g., critical PMD warnings only change in about 7% of commits while LLOC changes in about 75%. Some differences are also between categories, e.g., McCC changes in 31% of perfective changes and in 57% of corrective changes.

To evaluate **H2**, we present the differences in all changes visually as box plots in Figure 2.

In addition, we provide Table 7 which shows the Mann-Whitney U test (Mann and Whitney, 1947) p-values, and effect sizes for differences between the types of commits. We can see that most metric values are different depending on whether they are measured in perfective, corrective, or all other commits. In the following, we discuss the differences for each measured metric value. A description for each metric and the expected direction of metric value change is shown in Table 4.

**McCC**: the cyclomatic complexity of perfective changes is smaller than for other changes. Even when we do not account for the size of the change. This is expected as some perfective commits mention simplification of code. For perfective commits the effect size is medium. Corrective commits however have higher McCC than other commits. This can be seen in Figure 2.

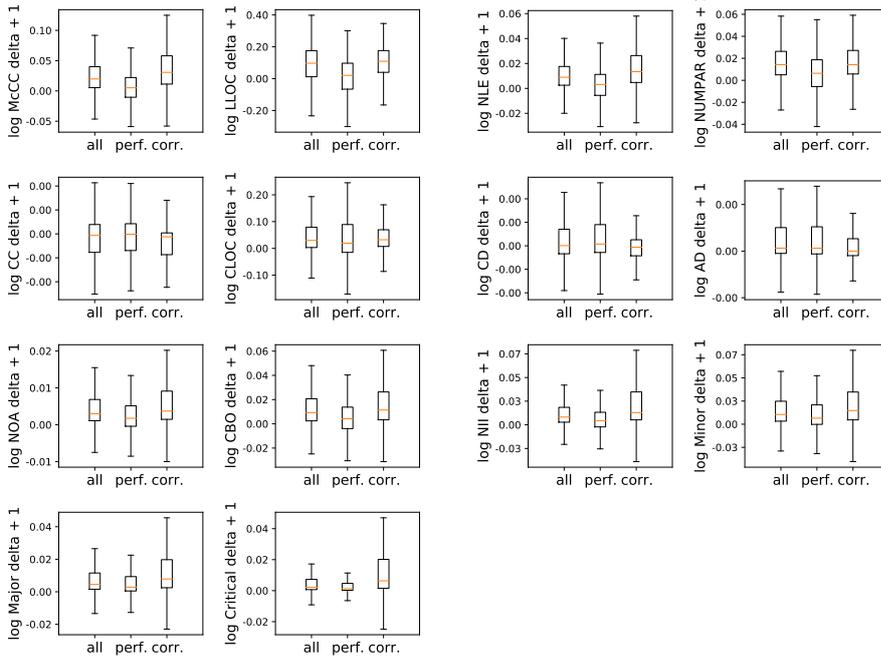


Fig. 2: Static source code metric value changes in all, perfective and corrective commits divided by changed lines. Fliers are omitted.

Table 7: Statistical test results for perfective and corrective commits, Mann-Whitney U test  $p$ -values ( $p$ -value) and effect size ( $d$ ) with category,  $n$  is negligible,  $s$  is small,  $m$  is medium. Statistically significant  $p$ -values are bolded. All values are normalized for changed lines.

Metric	Perfective		Corrective	
	$p$ -val	$d$	$p$ -val	$d$
McCC	<b>&lt;0.0001</b>	0.39 (m)	1.0000	-
LLOC	<b>&lt;0.0001</b>	0.45 (m)	1.0000	-
NLE	<b>&lt;0.0001</b>	0.27 (s)	1.0000	-
NUMPAR	<b>&lt;0.0001</b>	0.25 (s)	<b>&lt;0.0001</b>	0.09 (n)
CC	1.0000	-	<b>&lt;0.0001</b>	0.12 (s)
CLOC	<b>&lt;0.0001</b>	0.16 (s)	<b>&lt;0.0001</b>	0.05 (n)
CD	1.0000	-	<b>&lt;0.0001</b>	0.16 (s)
AD	<b>&lt;0.0001</b>	0.02 (n)	<b>&lt;0.0001</b>	0.08 (n)
NOA	<b>&lt;0.0001</b>	0.08 (n)	<b>&lt;0.0001</b>	0.07 (n)
CBO	<b>&lt;0.0001</b>	0.19 (s)	<b>&lt;0.0001</b>	0.06 (n)
NII	<b>&lt;0.0001</b>	0.19 (s)	<b>&lt;0.0001</b>	0.02 (n)
Minor	<b>&lt;0.0001</b>	0.19 (s)	<b>&lt;0.0001</b>	0.05 (n)
Major	<b>&lt;0.0001</b>	0.12 (s)	<b>&lt;0.0001</b>	0.05 (n)
Critical	<b>&lt;0.0001</b>	0.05 (n)	<b>&lt;0.0001</b>	0.03 (n)

The median of corrective commits is higher than for the other commits. Our assumption about McCC being lower in all quality improving commits is not met in this case. While it makes sense that corrective commits add complexity, the comparison here is one of stochastic dominance between all other commits and only corrective commits, not if corrective commits remove or add McCC. Thus, this means that changes in corrective commits are more complex than those of other changes.

**LLOC:** the difference of LLOC is the most pronounced in our data. We find that even when we do not correct for size of the change the difference between perfective and other changes in LLOC is the most pronounced. While manually classifying the commits, we found that often code is removed because it was marked as deprecated before or it was no longer needed due to other reasons. The effect size for perfective commits is medium. For corrective commits, we can see the same result as for McCC. While we assumed that bug fixes usually add code, we did not expect them to dominate all other commits including feature additions.

**NLE:** the nesting level if-else is smaller in perfective commits. We expect this is due to simplification and removal of complex code. When we look at the box plot in Figure 2 it shows a noticeable difference. This means simplification is a high priority when improving code quality in perfective commits. For corrective commits, we can see the same effect as previously seen for McCC and LLOC. The NLE is not lower but higher for corrective commits. This is more evidence for the fact that bug fixes add more complex code. There may be a timing factor involved, e.g., if bug fixes are quick fixes, they would add more complex code without a more complex refactoring which would decrease the complexity again.

**NUMPAR:** the number of parameters in a method is also different for perfective commits. This may be a hint of the type of perfective maintenance performed the most in perfective commits. The manual classification showed a lot of commit messages that claimed a simplification of the changed code. This metric would also be impacted by a simplification or refactoring operations. Corrective commits also show less additions in this metric, while it only has a negligible effect size it is still statistically significant. Fixing bugs seems to include some code reduction or at least less addition of parameters for methods.

**CC:** the clone coverage is not different for perfective commits. We would have expected that it is decreasing in perfective commits. However, it seems that clone removal is not a big part of perfective maintenance in our study subjects, which contradicts our expectation. Corrective commits contain a lower clone coverage, however. This could either be because corrective commits introduce fewer new clones than other commits or because they remove more. A possible reason for clone removal may be the correction of copy and paste related bugs.

**CLOC:** the comment lines of code show a difference for perfective commits and corrective commits. While we expected the CLOC to increase in both types of quality improving commits the effect size is higher in perfective commits.

It seems that bug fixing operations do not add enough comment lines to show a larger difference here for corrective commits.

**CD:** the comment density of perfective commits is not statistically significantly different from other commits. We would have expected a difference here because perfective maintenance should include additional comments on new or previously uncommented code. We can see a difference for corrective commits here. This shows that the density of comments is also improving in bug fixing operations probably due to clarifications for parts of the code that were fixed.

**AD:** the API documentation metric does change in perfective and corrective commits compared to other commits. A reason could be that perfective commits do add API documentation to make the difference significant. Corrective changes that introduce code in our study subjects seem to almost always include API documentation, therefore we can see a difference here. However, the effect size is negligible in both cases.

**NOA:** the number of ancestors is lower in perfective commits as expected. This metric would be affected in simplification and clean up maintenance operations. For corrective commits we can also see a lower value, this hints at some clean up operations happening during bug fixing.

**CBO:** the coupling between objects is lower after perfective commits. This is expected due to class removal and subsequent decoupling of classes. For corrective commits we can also see a difference. While the effect size is negligible, there is some code clean up happening during bug fixes, e.g., NOA and CC are also lower in corrective than in other commits.

**NI:** the number of incoming invocations is lower in both perfective and corrective commits. However, the effect size is small in perfective and negligible in corrective commits. It seems reasonable to see a difference in this metric, because in the case of perfective commits, we have lots of source code removal. However, there are also maintenance activities which are decoupling classes which would also impact this metric. Corrective maintenance seems to involve only limited decoupling operations, also seen in CBO.

**Minor:** The PMD warnings of minor severity are different in both types of changes. However, we can see that the effect size is larger for perfective changes which makes sense as those warnings can be part of perfective maintenance.

**Major:** The PMD warnings of major severity are also different in both types of changes. We can see the difference in effect size again and we expect the reason is the same as for Minor.

**Critical:** The PMD warnings of critical severity are different for both types of changes. Here, the effect size is negligible for both types. However, as they are only changed in about 7% of our commits, they are not changing often regardless of commit type.

There are significant differences between perfective and corrective changes. We **reject H2** that intended quality improvements have a positive impact on quality metric values.

## 5.2 Summary RQ1

In summary, we have the following results for RQ1.

### **RQ1 Summary**

While intended quality improvements by developers yield measurable differences in almost all metrics we find that not all metric values are changing in the expected direction.

### **Perfective changes**

Perfective commits have a positive effect on metric values that measure code complexity through the size, conditional statements, number of parameters, and coupling. For two metrics we do not find the expected difference to other commits. Code clones and comment density metric values are not statistically significantly different in perfective commits.

### **Corrective changes**

Only for two metrics, we observe a non-negligible and statistically significant change that we predicted. For LLOC, McCC and NLE, we observe the opposite of the expectation, which indicates that bug fixes add complex code.

## 5.3 Exploratory Study

To answer our **RQ2**: What kind of files are the target of internal or external quality improvements? We conduct an exploratory study. We present the results which files are changed in which change category with respect to their metric values. The extracted metrics are considered on a per-change basis, i.e., we divide the metrics by the number of changed files to get an average metric value per file.

Figure 3 shows box plots for the metric values of files before the change is applied. We can see that, perfective changes are not necessarily applied to complex files. If we compare the median values in Table 8 we can see that perfective changes are applied to smaller, simpler files than the average or corrective change. McCC, LLOC, NLE, NUMPAR and CBO are lower for the files which receive perfective changes, while CLOC, CD, AD are higher. This means that less complex and well documented files are often the target of perfective changes. If we look at corrective changes we see that they are more complex and usually larger files. McCC, LLOC, NLE, NUMPAR, CBO, NII as well as Minor, Major and Critical are higher than other, or perfective changes. As we consider the metric values before the change is applied they can be considered pre-bugfix. However, when we consider our results for **RQ1** the corrective changes usually increase the complexity even further.

Table 9 show the results of our statistical tests. Analogous to **RQ1** we compare the difference between perfective and non-perfective as well as corrective

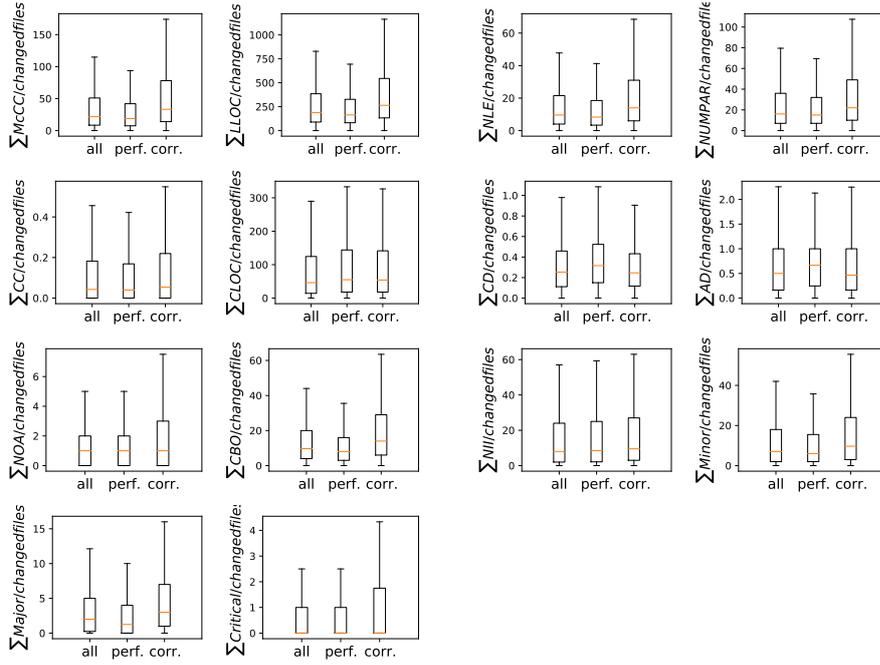


Fig. 3: Static source code metrics before the change is applied. Fliers are omitted.

Table 8: Median metric values per file before the change is applied.

Metric	All	Perfective	Corrective
McCC	21.78	18.78	33.23
LLOC	186.98	163.75	264.18
NLE	9.60	8.33	14.00
NUMPAR	16.06	15.00	22.00
CC	0.04	0.04	0.05
CLOC	46.25	55.00	54.00
CD	0.25	0.32	0.25
AD	0.50	0.67	0.46
NOA	1.00	1.00	1.00
CBO	9.67	8.00	14.00
NII	8.00	8.50	9.50
Minor	7.00	6.00	9.67
Major	2.00	1.25	3.00
Critical	0.00	0.00	0.00

and non-corrective. While most metric differences are statistically significant, we observe only some small effect sizes for the comment related metrics while the rest is negligible.

Figure 4 shows another perspective on our data in the form of a direct comparison of the density between perfective and corrective changes. We can

Table 9: Statistical test results for perfective and corrective commits regarding their average metrics before the change, Mann-Whitney U test p-values ( $p$ -value) and effect size ( $d$ ) with category,  $n$  is negligible,  $s$  is small,  $m$  is medium. Statistically significant  $p$ -values are bolded.

Metric	Perfective		Corrective	
	$p$ -val	$d$	$p$ -val	$d$
McCC	< <b>0.0001</b>	0.05 (n)	< <b>0.0001</b>	0.08 (n)
LLOC	< <b>0.0001</b>	0.05 (n)	< <b>0.0001</b>	0.05 (n)
NLE	< <b>0.0001</b>	0.04 (n)	< <b>0.0001</b>	0.07 (n)
NUMPAR	0.6367	-	0.0218	-
CC	< <b>0.0001</b>	0.01 (n)	0.0011	-
CLOC	< <b>0.0001</b>	0.12 (s)	< <b>0.0001</b>	0.06 (n)
CD	< <b>0.0001</b>	0.15 (s)	< <b>0.0001</b>	0.15 (s)
AD	< <b>0.0001</b>	0.17 (s)	< <b>0.0001</b>	0.15 (s)
NOA	0.5109	-	< <b>0.0001</b>	0.02 (n)
CBO	< <b>0.0001</b>	0.09 (n)	< <b>0.0001</b>	0.07 (n)
NII	< <b>0.0001</b>	0.05 (n)	< <b>0.0001</b>	0.04 (n)
Minor	< <b>0.0001</b>	0.04 (n)	< <b>0.0001</b>	0.02 (n)
Major	< <b>0.0001</b>	0.09 (n)	< <b>0.0001</b>	0.04 (n)
Critical	< <b>0.0001</b>	0.05 (n)	< <b>0.0001</b>	0.03 (n)

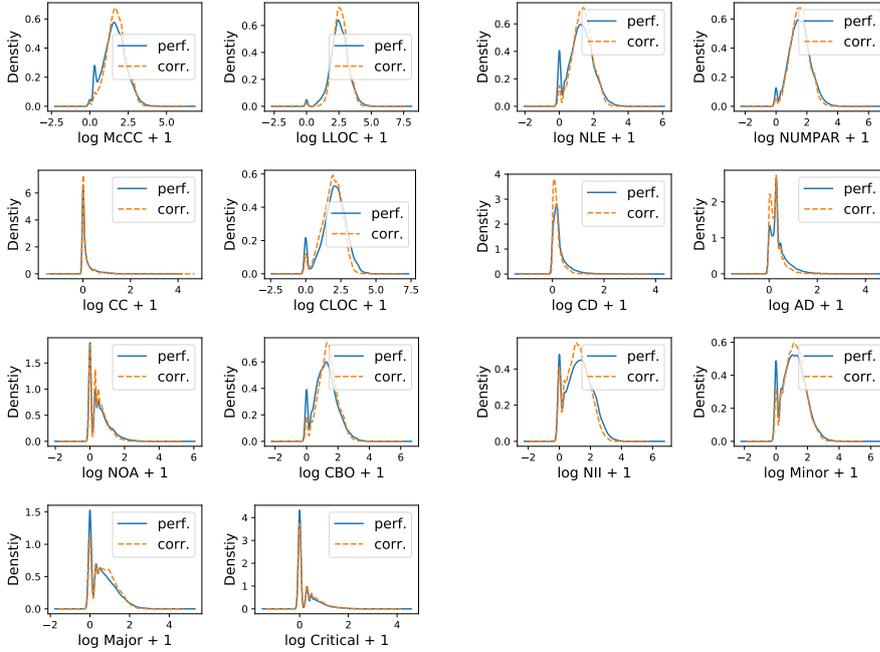


Fig. 4: Density plot of metric values for perfective and corrective categories before the change.

see that McCC, NLE, LLOC, NUMPAR, CD, CBO, NII and Minor have a lower density for perfective than for corrective. While the differences are small they are noticeable.

### RQ2 Summary

The files that are targets of perfective changes are in median not large and complex even before the change is applied. Corrective changes are applied to files which are in median already complex and large. The differences are statistically significant for most metrics, however the effect sizes are negligible to small.

## 6 Discussion

Our results show that size is different in both types of commits in **H1**. The size difference between all commits and perfective as well as corrective commits shows that both tend to be smaller than other commits. In case of perfective commits, code is statistically significantly more often deleted.

The differences in change size as well as the increased number of deletions for perfective commits we found for **H1** confirms previous research. The studies by Mockus and Votta (2000), Purushothaman and Perry (2005) and Alali et al. (2008) found that perfective maintenance activities are usually smaller. Mockus and Votta (2000) as well as Purushothaman and Perry (2005) found that corrective maintenance is also smaller and that perfective maintenance deletes more code. Another indication that size between maintenance types is different can be seen in the work by Hönel et al. (2019), which used size based metrics as predictors for maintenance types and showed that it improved the performance of classification models.

Our results for **H2** show statistically significant differences in metric measurements between perfective commits and other commits. This result indicates a confirmation of the measurements used by quality models, as the majority of metrics change as expected when developers actively improve the internal code quality. This empirical confirmation of the connection between quality metrics and developer intent is one of our main contributions and was, to the best of our knowledge, not part of any prior study. However, there are several examples of prior work that assumed this relationship.

The publications by McCabe (1976) and Chidamber and Kemerer (1994) assume that reducing complexity and coupling metrics increases software quality which is in line with our developer intents. While all metrics are included in a current ColumbusQM version (Bakota et al., 2014) because we used it as a basis, the CBO, McCC, LLOC, NOA metrics are also part of the SQUALE model (Mordal-Manet et al., 2009) AD, NLE, McCC, and PMD warnings are also part of Quamoco (Wagner et al., 2012). It seems that developers and the Columbus quality model agree with their view on software quality. We find that most of the metrics used in the quality model change when developers perceive their change as quality increasing. This is also true for most of

the metrics shared with the SQUALE model and with the Quamoco quality model. However, the implementation for the metrics may differ between the models. Our work establishes that all these quality models are directly related to intended improvements of the internal code quality by the developers.

Surprisingly, we found only few statistically significant and non-negligible differences for corrective commits. Not all software metric values are changing into the expected direction for corrective commits. For example, we can see that McCC, LLOC and NLE are increasing in corrective changes compared to other commits. While we are not expecting them to decrease for every corrective commit, we assumed that in comparison to all other commits they would be decreasing. Even when considering software aging (Parnas, 2001) we would expect the aging to impact all kinds of changes not just corrective changes. When we look at popular data sets used in the defect prediction domain we often find coupling, size and complexity software metrics (Herbold et al., 2021). For example, the popular (as per the literature review from Hosseini et al. (2017)) data set by Jureczko and Madeyski (2010) uses such features, but they are also common in more recent data sets, e.g., by Ferenc et al. (2020) or Yatish et al. (2019).

That the most significant difference is in the size of changes could explain various recent findings from the literature, in which size was found to be a very good indicator both for release level defect prediction (Zhou et al., 2018) and just-in-time defect prediction (Huang et al., 2017). This could also be an explanation for possible ceiling effects (Menzies et al., 2008) when such criteria are used, as the difference to other changes are relatively small. We believe that these aspects should be further considered by the defect prediction community and believe that more research is required to establish causal relationships between features and defectiveness.

While the work by Peitek et al. (2021) indicates that cyclomatic complexity may not be as indicative of code understandability as expected, we show within our work that it often changes in quality increasing commits. It seems that developers associate overall complexity as measured by McCC, NLE, NUMPAR with code that needs quality improvement. However, as we can see in the exploratory part of our study the most complex files are usually not targeted for quality increasing changes.

Our exploratory study to answer **RQ2** about files that are the target of quality increasing commits reveals additional interesting data. We show that perfective maintenance does not necessarily target files that are in need of it due to high complexity in comparison to other changes. In fact, low complexity files as measured by McCC and NLE are more often part of additional quality increasing work by the developers. This may hint at problems regarding the prioritization of quality improvements in the source code. Maybe errors could have been avoided when perfective changes would have targeted more complex files. There could also be effects of different developers or a bias for perfective changes towards simpler code, this warrants future investigation. Corrective changes, in contrast to perfective changes, are applied to files which are large and complex. This was expected, however combined with the results of **RQ1**

this means that bugs are fixed in complex and large files and then the files get, on average, even more complex and even larger.

Future work could investigate boundary values according to our data. When we compare the median values of our measurements in Table 8 with current boundary values from PMD<sup>13</sup>, we may think that the PMD warning value of 80 McCC per file may be too high. A PMD warning triggered at 34 McCC per file would have warned about at least 50% of the files that were in need of a bug fix. However, lowering the boundary will also result in more warnings for files that were not target of corrective changes.

## 6.1 Implications for Researchers

Our results for **H1** increase the validity of previous research by confirming previous results in our study on a larger data set of different projects. Our confirmation that quality increasing changes are smaller than other changes shows that researchers developing a change classification approach can benefit from including size based metrics.

Our results for **H2** show that perfective changes reduce size and complexity metrics in comparison to all other changes. Previous studies investigating refactorings also found an impact on size and complexity metrics. We are able to generalize this finding by providing results of a superset of refactoring operations, namely perfective changes. This indicates that perfective changes generally reduce size and complexity metrics. This also indicates that software quality models that use the affected metrics in their code quality estimations agree with the developers on what impacts code quality.

Increasing the external quality by fixing bugs, i.e., corrective changes, decreases the internal quality, i.e., complexity metric values. Defect prediction models may assign a higher risk to parts of the code that contained a bug before as there is an assumption of latent bugs still existing (Kim et al., 2007; Rahman et al., 2011). Our data provides a fine grained perspective by providing empirical data which shows that the code quality as measured by static source code metrics is actually decreasing.

This also has implications for researchers developing and deploying defect prediction models in practice. The fact that fixing a bug increases the risk of the file can lead to problems regarding the acceptance of the model by practitioners as they have no way of reducing the risk (Lewis et al., 2013). The results of our study could help to explain the reasons to developers. We can empirically show that fixing a bug is a complex operation that introduces even more complexity than other changes, even feature additions. According to our results, the main driver of complexity in a project are bug fixes and the only way to combat the rising complexity is perfective maintenance which should especially target large and complex files.

In our results for **RQ2** we see a difference between files before corrective changes are applied, and before other changes are applied. This difference is one

<sup>13</sup> [https://pmd.github.io/pmd/pmd\\_rules\\_java\\_design.html#cyclomaticcomplexity](https://pmd.github.io/pmd/pmd_rules_java_design.html#cyclomaticcomplexity)

of the sources of the predictive power of defect prediction models. However, the difference is smaller than expected. Incorporating metrics that have a larger difference in our data, e.g., comment density and API documentation into defect prediction models, may increase their prediction performance.

## 6.2 Implications for Practitioners

Our results for **H2** suggest that, for the most part, software quality models match the expectations of the developers. If practitioners select a software quality model which uses static source code metrics that show a difference in our data they can expect that the model matches their intuition.

In combination with **RQ2**, our results indicate that bug fixing is the main driver of complexity in a software project and perfective changes are the main reducer of complexity. This has implications for developers. If more complex files were targeted for perfective maintenance bugs could possibly have been prevented. As fixing bugs does not decrease complexity, perfective maintenance is the best way to reduce it and combat rising complexity of the project as a whole. However, given the results for **RQ2**, we see that large and complex files are not the main target of perfective maintenance. This is an opportunity for improvement by shifting priorities for perfective maintenance to large and complex files. Moreover, our results indicate that a bug fix should be treated similar to technical debt regarding its negative impact on complexity metrics. To mitigate this, practitioners should be aware that it would be beneficial to clean up and simplify the code that is introduced as part of the bug fix.

## 7 Threats to validity

In this section, we discuss the threats to validity we identified for our work. We discuss four basic types of validity separately as suggested by Wohlin et al. (2000) and include reliability due to our manual classification approach.

### 7.1 Reliability

We classify changes to a software retroactively and without the developers. This may introduce a researcher bias to the data and subsequently the results. However, this is a necessity given the size of the data and the unrestricted time frame for the sample and full data because it would not be feasible to ask developers about a couple of commits from years ago. To mitigate this threat, we perform the classification labeling according to guidelines and every change is independently classified by two researchers. We also compare our differences with a sample of changes classified by the developers themselves from Mauczka et al. (2015) and confirm that we are agreeing on most changes. In addition, we measure the inter-rater agreement between the researchers and find that it is substantial.

## 7.2 Construct Validity

Our definition of quality improving may be too broad. We aggregate different types of quality improvement together, e.g., improving error messages, structure of the code or readability. This may influence the changes we observe within our metric values. While these differences should be studied as well, we believe that a broad overview of generic quality improvements independent of their type has advantages. We avoid the risk of being focused only on structural improvements, i.e., due to use of generics or new Java features without missing bigger changes due to simplification of method code.

## 7.3 Conclusion Validity

We are reporting differences in metric value changes between perfective and corrective changes of the software development history of our study subjects. We find a difference for perfective commits and only some non-negligible, statistically significant difference for corrective commits. This could be an effect of our sample used as ground truth, however we chose to draw randomly from a list of commits in our study subjects so that our sample should be representative.

We use a deep learning model to classify all of our commits based on the ground truth we provide. This can introduce a bias or errors in the classification. We note however, that the non-negligible effect sizes for our results do not change. The quality metric evaluation of only the ground truth data is included in the appendix and shows similar results. We note that for the small effect sizes we observe, a large number of observations are needed to show a significant difference as is demonstrated by the results in this article when compared to the ground truth.

## 7.4 Internal Validity

A possible threat could be tangled commits which improve quality and at the same time add a feature. We mitigate this in our ground truth, by manual inspection of the commit message of every change considered. We excluded tangled commits if it was possible to determine this by the commit message. As no automatic untangling approach is available to us and available approaches to label tangled commits already use the commit message to find tangled commits we determine that tangled commits which are not identifiable from the commit message are a minor threat.

Another threat could be a lower number of feature additions in our study subjects. Maybe feature additions happen too infrequently to influence the results, therefore, corrective commits are seen as adding more complex code than other commits. While we include some projects that are in development for a long period of time, we believe this threat is mitigated by the unrestricted time frame of our study.

Bots which commit code (Dey et al., 2020) could be a possible threat to our study. We mitigate this threat by matching our author data against the bot data set provided by Dey et al. (2020). We did not find matches for bots in our data. We were able to detect a Jenkins bot only when dropping the restriction of our case study data that a commit has to change non-test code. We also implemented the detection mechanism by Dey et al. (2020) which uses the username and email of the author of the commit, as used by Dey et al. to create their bot data set. This also yielded no bots in our data. Manual inspection of the author data yielded two bot-like accounts which turned out to be from a previous cvs2svn conversion as well as asf-sync-process which allows user patches without an account. However, the content of changes by the accounts we found are created by developers. We determine that the threat of bots in our data is low.

## 7.5 External Validity

We focus on a convenience sample of data consisting of Java Open Source projects under the umbrella of the Apache Software Foundation. We consider this a minor threat to external validity. The reason is that although we are limited to one organization, we still have a wide variety of different types of software in our data. We believe that this mitigates the missing variety of project patronage.

Furthermore, we only include Java projects. However, Java is used in a wide variety of projects and remains a popular language. Its age provides us with a long history of data we can utilize in this study. However, we note that this study may not generalize to all Java projects much less all software projects in other languages.

## 8 Conclusion

Numerous quality measurements exist, and numerous software quality models try to connect concrete quality metrics with abstract quality factors and sub factors. Although it seems clear that some static source code metrics influence software quality factors, the question of which and how much remains. Instead of relying on necessarily limited developer and expert evaluations of source code or changes we extract metrics from past changes where developers intended to increase the quality extracted from the commit message.

Within this work, we performed a manual classification of developer intents on a sample of 2,533 commits from 54 Java open source projects by two researchers independently and guided by classification guidelines. We classify the commits into three categories, perfective maintenance, corrective maintenance, or neither. We further evaluate our classification guidelines by re-classifying of a developer labeled sample. We use the manually labeled data as ground truth to evaluate and then fine tune a state-of-the-art deep learning model for

text classification. The fine-tuned model is then used to classify all available commits into our categories increasing our data size to 125,482 commits. We extract static source code metrics and static analysis warnings for all 125,482 commits which allows us to investigate the impact of changes and the distribution of metric values before the changes are applied. Based on the literature, we hypothesize that certain metric values change in a certain direction, e.g., perfective changes reduce complexity. We find that perfective commits are more often removing code and generally add fewer lines. Regarding the metric measurements, we find that most metric value changes of perfective commits are significantly different to other commits and have a positive, non-negligible impact on the majority of metric values.

Surprisingly, we found that corrective changes are more complex and larger than other changes. It seems that fixing a bug increases the size, but also the complexity measured via McCC and NLE. As we compare against all other changes, we were expecting less addition of complexity as e.g., feature additions. We conclude that the process of performing a bug fix tends to add more complex code than other changes.

We find that complex files are not necessarily the primary target for quality increasing work by developers, including refactoring. To the contrary, we find that perfective quality changes are applied to files that are already less complex than files changed in other or corrective commits. Files contained in corrective changes on the other hand are more complex and usually larger than both perfective and other files. In combination with our first result this shows that corrective changes are applied to files which are already complex and get even more complex after the change is applied.

While we explored a limited number of metrics and commits we think that this approach can be used to evaluate more metrics connected with software quality in a meaningful way and help practitioners and researchers with additional empirical data.

## Declarations

This work was partly funded by the German Research Foundation (DFG) through the project DEFECTS, grant 402774445.

The authors have no competing interests to declare that are relevant to the content of this article.

## Acknowledgements

We want to thank the GWDG Göttingen<sup>14</sup> for providing us with computing resources within their HPC-Cluster.

---

<sup>14</sup> <https://www.gwdg.de>

## References

- Abdi H (2007) Bonferroni and Sidak corrections for multiple comparisons. In: Encyclopedia of Measurement and Statistics, Sage, Thousand Oaks, CA, pp 103–107
- Al Dallal J, Abdin A (2018) Empirical evaluation of the impact of object-oriented code refactoring on quality attributes: A systematic literature review. *IEEE Transactions on Software Engineering* 44(1):44–69, DOI 10.1109/TSE.2017.2658573
- Alali A, Kagdi H, Maletic JI (2008) What’s a typical commit? a characterization of open source software repositories. In: 2008 16th IEEE International Conference on Program Comprehension, pp 182–191, DOI 10.1109/ICPC.2008.24
- AlOmar EA, Mkaouer MW, Ouni A (2021) Toward the automatic classification of self-affirmed refactoring. *Journal of Systems and Software* 171:110821, DOI 10.1016/j.jss.2020.110821, URL <http://www.sciencedirect.com/science/article/pii/S016412122030217X>
- Alshayeb M (2009) Empirical investigation of refactoring effect on software quality. *Information and Software Technology* 51(9):1319 – 1326, DOI 10.1016/j.infsof.2009.04.002, URL <http://www.sciencedirect.com/science/article/pii/S095058490900038X>
- Bakota T, Hegedűs P, Körtvélyesi P, Ferenc R, Gyimóthy T (2011) A probabilistic software quality model. In: 2011 27th IEEE International Conference on Software Maintenance (ICSM), pp 243–252, DOI 10.1109/ICSM.2011.6080791
- Bakota T, Hegedűs P, Siket I, Ladányi G, Ferenc R (2014) Qualitygate sourceaudit: A tool for assessing the technical quality of software. In: 2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE), pp 440–445, DOI 10.1109/CSMR-WCRE.2014.6747214
- Bavota G, De Lucia A, Di Penta M, Oliveto R, Palomba F (2015) An experimental investigation on the innate relationship between quality and refactoring. *Journal of Systems and Software* 107:1 – 14, DOI <https://doi.org/10.1016/j.jss.2015.05.024>, URL <http://www.sciencedirect.com/science/article/pii/S0164121215001053>
- Boehm BW, Brown JR, Lipow M (1976) Quantitative evaluation of software quality. In: Proceedings of the 2Nd International Conference on Software Engineering, IEEE Computer Society Press, Los Alamitos, CA, USA, ICSE ’76, pp 592–605, URL <http://dl.acm.org/citation.cfm?id=800253.807736>
- Ch’avez A, Ferreira I, Fernandes E, Cedrim D, Garcia A (2017) How does refactoring affect internal quality attributes? a multi-project study. In: Proceedings of the 31st Brazilian Symposium on Software Engineering, Association for Computing Machinery, New York, NY, USA, SBES’17, p 74–83, DOI 10.1145/3131151.3131171
- Chidamber SR, Kemerer CF (1994) A metrics suite for object oriented design. *IEEE Transactions on Software Engineering* 20(6):476–493, DOI 10.1109/

32.295895

- Cliff N (1993) Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological Bulletin*
- Cohen J (1960) A coefficient of agreement for nominal scales. *Educational and Psychological Measurement* 20(1):37–46, DOI 10.1177/001316446002000104
- D’Ambros M, Lanza M, Robbes R (2012) Evaluating defect prediction approaches: A benchmark and an extensive comparison. *Empirical Softw Engg* 17(4-5):531–577, DOI 10.1007/s10664-011-9173-9
- Dey T, Mousavi S, Ponce E, Fry T, Vasilescu B, Filippova A, Mockus A (2020) Detecting and characterizing bots that commit code. In: *Proceedings of the 17th International Conference on Mining Software Repositories*, Association for Computing Machinery, New York, NY, USA, p 209–219, URL <https://doi.org/10.1145/3379597.3387478>
- Fakhoury S, Roy D, Hassan A, Arnaoudova V (2019) Improving source code readability: Theory and practice. In: *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pp 2–12, DOI 10.1109/ICPC.2019.00014
- Fenton N, Bieman J (2014) *Software Metrics: A Rigorous and Practical Approach*, Third Edition, 3rd edn. CRC Press, Inc., Boca Raton, FL, USA
- Ferenc R, Gyimesi P, Gyimesi G, Tóth Z, Gyimóthy T (2020) An automatically created novel bug dataset and its validation in bug prediction. *Journal of Systems and Software* 169:110691, DOI 10.1016/j.jss.2020.110691, URL <http://www.sciencedirect.com/science/article/pii/S0164121220301436>
- Fu Y, Yan M, Zhang X, Xu L, Yang D, Kymer JD (2015) Automated classification of software change messages by semi-supervised latent dirichlet allocation. *Information and Software Technology* 57:369 – 377, DOI 10.1016/j.infsof.2014.05.017, URL <http://www.sciencedirect.com/science/article/pii/S0950584914001347>
- Ghadhab L, Jenhani I, Mkaouer MW, Ben Messaoud M (2021) Augmenting commit classification by using fine-grained source code changes and a pre-trained deep neural language model. *Information and Software Technology* 135:106566, DOI 10.1016/j.infsof.2021.106566, URL <https://www.sciencedirect.com/science/article/pii/S0950584921000495>
- Gharbi S, Mkaouer MW, Jenhani I, Messaoud MB (2019) On the classification of software change messages using multi-label active learning. In: *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, Association for Computing Machinery, New York, NY, USA, SAC ’19, p 1760–1767, DOI 10.1145/3297280.3297452
- Griessom RJ, Kim JJ (2005) *Effect sizes for research: A broad practical approach*. Lawrence Erlbaum Associates Publishers
- Gyimothy T, Ferenc R, Siket I (2005) Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering* 31(10):897–910, DOI 10.1109/TSE.2005.112
- Hattori LP, Lanza M (2008) On the nature of commits. In: *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software En-*

- gineering, IEEE Press, Piscataway, NJ, USA, ASE'08, pp III-63-III-71, DOI 10.1109/ASEW.2008.4686322
- Herbold S, Trautsch A, Trautsch F, Ledel B (2021) Issues with szz: An empirical assessment of the state of practice of defect prediction data collection, URL <http://arxiv.org/abs/1911.08938>, recently Accepted at Empirical Software Engineering
- Herzig K, Just S, Zeller A (2013) It's not a bug, it's a feature: How misclassification impacts bug prediction. In: Proceedings of the 2013 International Conference on Software Engineering, IEEE Press, ICSE '13, p 392-401
- Hosseini S, Turhan B, Gunarathna D (2017) A systematic literature review and meta-analysis on cross project defect prediction. IEEE Transactions on Software Engineering PP(99):1-1, DOI 10.1109/TSE.2017.2770124
- Huang Q, Xia X, Lo D (2017) Supervised vs unsupervised models: A holistic look at effort-aware just-in-time defect prediction. In: 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp 159-170, DOI 10.1109/ICSME.2017.51
- Hönel S, Ericsson M, Löwe W, Wingkvist A (2019) Importance and aptitude of source code density for commit classification into maintenance activities. In: 2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS), pp 109-120, DOI 10.1109/QRS.2019.00027
- ISO/IEC (2001) Iso/iec 9126. software engineering – product quality
- ISO/IEC (2011) ISO/IEC 25010:2011, systems and software engineering — systems and software quality requirements and evaluation (square) — system and software quality models
- Jureczko M, Madeyski L (2010) Towards identifying software project clusters with regard to defect prediction. In: Proceedings of the 6th International Conference on Predictive Models in Software Engineering, Association for Computing Machinery, New York, NY, USA, PROMISE '10, DOI 10.1145/1868328.1868342
- Kamei Y, Shihab E, Adams B, Hassan AE, Mockus A, Sinha A, Ubayashi N (2013) A large-scale empirical study of just-in-time quality assurance. IEEE Transactions on Software Engineering 39(6):757-773, DOI 10.1109/TSE.2012.70
- Kim S, Zimmermann T, Whitehead Jr EJ, Zeller A (2007) Predicting faults from cached history. In: 29th International Conference on Software Engineering (ICSE'07), pp 489-498, DOI 10.1109/ICSE.2007.66
- Kitchenham B, Pfleeger SL (1996) Software quality: the elusive target [special issues section]. IEEE Software 13(1):12-21, DOI 10.1109/52.476281
- Landis JR, Koch GG (1977) An application of hierarchical kappa-type statistics in the assessment of majority agreement among multiple observers. Biometrics 33(2):363-374, URL <http://www.jstor.org/stable/2529786>
- Levin S, Yehudai A (2017) Boosting automatic commit classification into maintenance activities by utilizing source code changes. In: Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering, Association for Computing Machinery, New York, NY, USA, PROMISE, p 97-106, DOI 10.1145/3127005.3127016

- Lewis C, Lin Z, Sadowski C, Zhu X, Ou R, Whitehead EJ (2013) Does bug prediction support human developers? findings from a google case study. In: 2013 35th International Conference on Software Engineering (ICSE), pp 372–381, DOI 10.1109/ICSE.2013.6606583
- Mann HB, Whitney DR (1947) On a test of whether one of two random variables is stochastically larger than the other. *Annals of Mathematical Statistics* 18(1):50–60
- Mauczka A, Huber M, Schanes C, Schramm W, Bernhart M, Grechenig T (2012) Tracing your maintenance work — a cross-project validation of an automated classification dictionary for commit messages. In: Proceedings of the 15th International Conference on Fundamental Approaches to Software Engineering, Springer-Verlag, Berlin, Heidelberg, FASE’12, p 301–315, DOI 10.1007/978-3-642-28872-2\_21
- Mauczka A, Brosch F, Schanes C, Grechenig T (2015) Dataset of developer-labeled commit messages. In: Proceedings of the 12th Working Conference on Mining Software Repositories, IEEE Press, Piscataway, NJ, USA, MSR ’15, pp 490–493, URL <http://dl.acm.org/citation.cfm?id=2820518.2820595>
- McCabe TJ (1976) A complexity measure. *IEEE Trans Softw Eng* 2(4):308–320, DOI 10.1109/TSE.1976.233837
- McCall JA, Richards PK, Walters GF (1977) Factors in software quality: concept and definitions of software quality. Rome Air Development Center, Air Force Systems Command, Griffiss Air Force Base, New York 1(3)
- Menzies T, Turhan B, Bener A, Gay G, Cukic B, Jiang Y (2008) Implications of ceiling effects in defect predictors. In: Proceedings of the 4th International Workshop on Predictor Models in Software Engineering, Association for Computing Machinery, New York, NY, USA, PROMISE ’08, p 47–54, DOI 10.1145/1370788.1370801
- Mockus, Votta (2000) Identifying reasons for software changes using historic databases. In: Proceedings 2000 International Conference on Software Maintenance, pp 120–130, DOI 10.1109/ICSM.2000.883028
- Mordal-Manet K, Balmas F, Denier S, Ducasse S, Wertz H, Laval J, Bellingard F, Vaillergues P (2009) The squal model — a practice-based industrial quality model. In: 2009 IEEE International Conference on Software Maintenance, pp 531–534, DOI 10.1109/ICSM.2009.5306381
- von der Mosel J, Trautsch A, Herbold S (2021) On the validity of pre-trained transformers for natural language processing in the software engineering domain, URL <https://arxiv.org/abs/2109.04738>, currently in a minor revision to *Transactions on Software Engineering*
- NASA (2004) Nasa IV & V facility metrics data program. URL <http://mdp.ivv.nasa.gov/repository.html>
- Pantiuchina J, Lanza M, Bavota G (2018) Improving code: The (mis) perception of quality metrics. In: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp 80–91, DOI 10.1109/ICSME.2018.00017

- Pantiuchina J, Zampetti F, Scalabrino S, Piantadosi V, Oliveto R, Bavota G, Penta MD (2020) Why developers refactor source code: A mining-based study. *ACM Trans Softw Eng Methodol* 29(4), DOI 10.1145/3408302
- Parnas DL (2001) *Software Aging*, Addison-Wesley Longman Publishing Co., Inc., USA, p 551–567
- Peitek N, Apel S, Parnin C, Brechmann A, Siegmund J (2021) Program comprehension and code complexity metrics: An fmri study. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), pp 524–536, DOI 10.1109/ICSE43902.2021.00056
- Purushothaman R, Perry DE (2005) Toward understanding the rhetoric of small source code changes. *IEEE Transactions on Software Engineering* 31(6):511–526, DOI 10.1109/TSE.2005.74
- Rahman F, Posnett D, Hindle A, Barr E, Devanbu P (2011) Bugcache for inspections: Hit or miss? In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, Association for Computing Machinery, New York, NY, USA, ESEC/FSE '11, p 322–331, DOI 10.1145/2025113.2025157
- Scalabrino S, Bavota G, Vendome C, Linares-Vásquez M, Poshyvanyk D, Oliveto R (2021) Automatically assessing code understandability. *IEEE Transactions on Software Engineering* 47(3):595–613, DOI 10.1109/TSE.2019.2901468
- Stroggylos K, Spinellis D (2007) Refactoring—does it improve software quality? In: Fifth International Workshop on Software Quality (WoSQ'07: ICSE Workshops 2007), pp 10–10, DOI 10.1109/WOSQ.2007.11
- Swanson EB (1976) The dimensions of maintenance. In: Proceedings of the 2nd International Conference on Software Engineering, IEEE Computer Society Press, Washington, DC, USA, ICSE '76, p 492–497
- Trautsch A, Herbold S, Grabowski J (2020a) A Longitudinal Study of Static Analysis Warning Evolution and the Effects of PMD on Software Quality in Apache Open Source Projects. *Empirical Software Engineering* DOI 10.1007/s10664-020-09880-1
- Trautsch A, Trautsch F, Herbold S, Ledel B, Grabowski J (2020b) The smartshark ecosystem for software repository mining. In: Proceedings of the 42st International Conference on Software Engineering - Demonstrations, ACM
- Trautsch A, Erbel J, Herbold S, Grabowski J (2021) Replication kit. URL [https://github.com/atrautsch/emse2021\\_replication](https://github.com/atrautsch/emse2021_replication)
- Trautsch F, Herbold S, Makedonski P, Grabowski J (2017) Addressing problems with replicability and validity of repository mining studies through a smart data platform. *Empirical Software Engineering* DOI 10.1007/s10664-017-9537-x
- Wagner S, Lochmann K, Heinemann L, Kläs M, Trendowicz A, Plösch R, Seidl A, Goeb A, Streit J (2012) The quamoco product quality modelling and assessment approach. In: Proceedings of the 34th International Conference on Software Engineering, IEEE Press, Piscataway, NJ, USA, ICSE '12, pp 1133–1142, URL <http://dl.acm.org/citation.cfm?id=2337223>.

2337372

- Wang S, Bansal C, Nagappan N (2021) Large-scale intent analysis for identifying large-review-effort code changes. *Information and Software Technology* 130:106408, URL <http://www.sciencedirect.com/science/article/pii/S0950584920300033>
- Wilk MB, Shapiro SS (1965) An analysis of variance test for normality (complete samples)†. *Biometrika* 52(3-4):591–611, DOI 10.1093/biomet/52.3-4.591
- Wohlin C, Runeson P, Höst M, Ohlsson MC, Regnell B, Wesslén A (2000) *Experimentation in Software Engineering: An Introduction*. Kluwer Academic Publishers, Norwell, MA, USA
- Yan M, Fu Y, Zhang X, Yang D, Xu L, Kymer JD (2016) Automatically classifying software changes via discriminative topic model: Supporting multi-category and cross-project. *Journal of Systems and Software* 113:296 – 308, DOI 10.1016/j.jss.2015.12.019, URL <http://www.sciencedirect.com/science/article/pii/S016412121500285X>
- Yatish S, Jiarpakdee J, Thongtanunam P, Tantithamthavorn C (2019) Mining software defects: Should we consider affected releases? In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), pp 654–665, DOI 10.1109/ICSE.2019.00075
- Zhou Y, Yang Y, Lu H, Chen L, Li Y, Zhao Y, Qian J, Xu B (2018) How far we have progressed in the journey? an examination of cross-project defect prediction. *ACM Trans Softw Eng Methodol* 27(1), DOI 10.1145/3183339

## Appendix A Ground truth only results

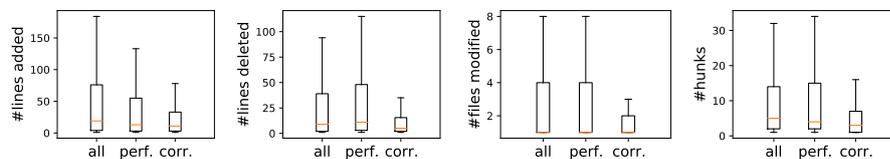


Fig. 5: Ground truth only. Commit size distribution over all projects for all, perfective and corrective commits. Fliers are omitted.

Table 10: Ground truth only. Statistical test results for perfective and corrective commits, Mann-Whitney U test  $p$ -values ( $p$ -value) and effect size ( $d$ ) with category  $n$  is negligible,  $s$  is small. Statistically significant  $p$ -values are bolded.

Metric	Perfective		Corrective	
	$p$ -value	$d$	$p$ -value	$d$
#lines added	< <b>0.0001</b>	0.20 (s)	< <b>0.0001</b>	0.20 (s)
#lines deleted	< <b>0.0001</b>	0.13 (s)	< <b>0.0001</b>	0.17 (s)
#files modified	0.2829	-	< <b>0.0001</b>	0.22 (s)
#hunks	0.7009	-	< <b>0.0001</b>	0.21 (s)

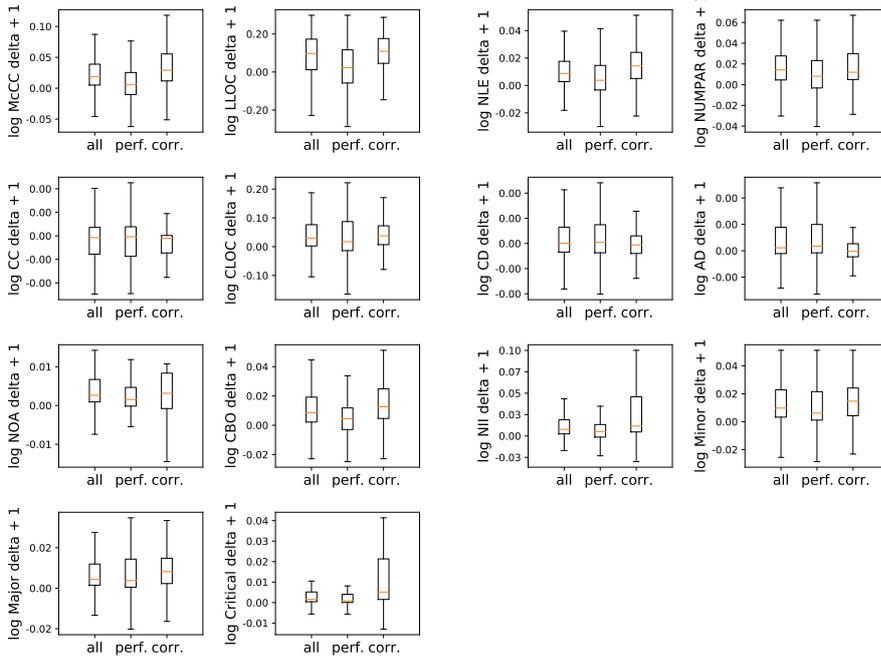


Fig. 6: Ground truth only. Static source code metrics changes in all, perfective and corrective commits divided by changed lines. Fliers are omitted.

Table 11: Ground truth only. Statistical test results for perfective and corrective commits, Mann-Whitney U test  $p$ -values ( $p$ -value) and effect size ( $d$ ) with category,  $n$  is negligible,  $s$  is small,  $m$  is medium. Statistically significant  $p$ -values are bolded. All values are normalized for changed lines.

Metric	Perfective		Corrective	
	$p$ -val	$d$	$p$ -val	$d$
McCC	<b>&lt;0.0001</b>	0.37 (m)	1.0000	-
LLOC	<b>&lt;0.0001</b>	0.42 (m)	1.0000	-
NLE	<b>&lt;0.0001</b>	0.26 (s)	0.9577	-
NUMPAR	<b>&lt;0.0001</b>	0.24 (s)	<b>&lt;0.0001</b>	0.09 (n)
CC	1.0000	-	<b>&lt;0.0001</b>	0.12 (s)
CLOC	<b>&lt;0.0001</b>	0.19 (s)	0.1906	-
CD	0.9303	-	<b>&lt;0.0001</b>	0.15 (s)
AD	0.1556	-	<b>&lt;0.0001</b>	0.10 (s)
NOA	<b>&lt;0.0001</b>	0.08 (n)	<b>&lt;0.0001</b>	0.09 (n)
CBO	<b>&lt;0.0001</b>	0.18 (s)	0.0145	-
NII	<b>&lt;0.0001</b>	0.19 (s)	0.0620	-
Minor	<b>&lt;0.0001</b>	0.18 (s)	0.0005	-
Major	<b>&lt;0.0001</b>	0.10 (s)	<b>0.0002</b>	0.06 (n)
Critical	<b>&lt;0.0001</b>	0.06 (n)	0.1111	-

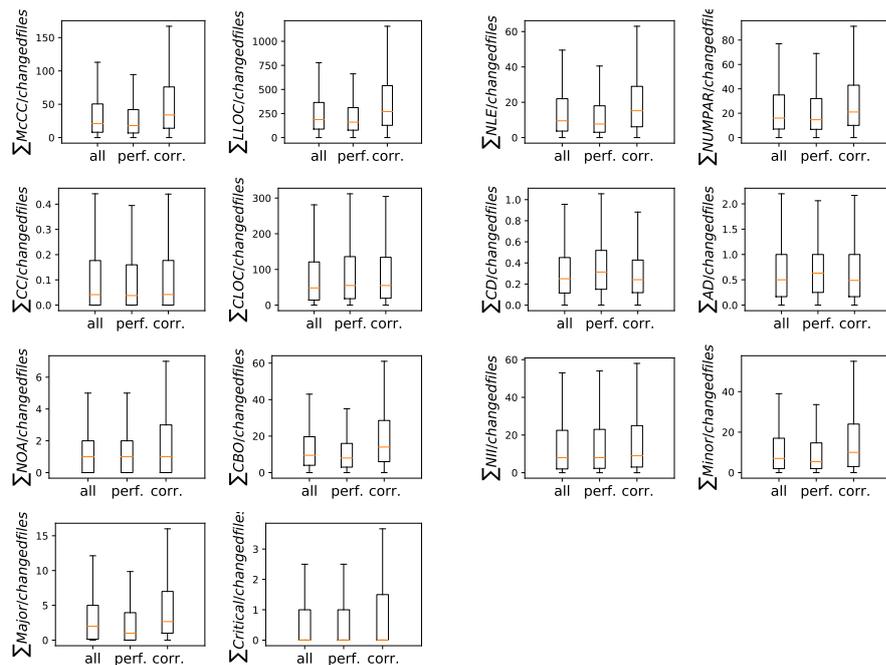


Fig. 7: Ground truth only. Static source code metrics before the change is applied. Fliers are omitted.

Table 12: Median metric values before the change is applied.

Metric	All	Perfective	Corrective
McCC	21.00	18.00	34.00
LLOC	187.22	160.38	270.00
NLE	9.50	7.67	15.20
NUMPAR	16.00	14.67	21.00
CC	0.04	0.04	0.04
CLOC	48.22	55.00	55.00
CD	0.25	0.31	0.24
AD	0.50	0.63	0.49
NOA	1.00	1.00	1.00
CBO	9.50	8.00	14.00
NII	8.00	8.00	9.00
Minor	7.00	5.43	10.00
Major	2.00	1.00	2.67
Critical	0.00	0.00	0.00

Table 13: Ground truth only. Statistical test results for perfective and corrective commits regarding their average metrics before the change, Mann-Whitney U test  $p$ -values ( $p$ -value) and effect size ( $d$ ) with category,  $n$  is negligible,  $s$  is small,  $m$  is medium. Statistically significant  $p$ -values are bolded.

Metric	Perfective		Corrective	
	$p$ -val	$d$	$p$ -val	$d$
McCC	0.0003	-	0.0016	-
LLOC	0.0005	-	0.1138	-
NLE	0.0003	-	0.0072	-
NUMPAR	0.5344	-	0.4704	-
CC	0.4142	-	0.0210	-
CLOC	<b>&lt;0.0001</b>	0.10 (n)	0.0111	-
CD	<b>&lt;0.0001</b>	0.15 (s)	<b>&lt;0.0001</b>	0.16 (s)
AD	<b>&lt;0.0001</b>	0.15 (s)	<b>&lt;0.0001</b>	0.15 (s)
NOA	0.6847	-	0.2103	-
CBO	<b>&lt;0.0001</b>	0.11 (s)	0.0190	-
NII	0.0510	-	0.0105	-
Minor	0.0006	-	0.6288	-
Major	<b>&lt;0.0001</b>	0.12 (s)	0.0852	-
Critical	0.0179	-	0.5730	-

Table 14: Detailed statistical tests results for metric changes. Accompanies Table 7. SHA is Shapiro-Wilk, MWU is Mann-Whitney U test, the number of samples for not perfective is 77,630, for perfective the number is 47,852. The number of samples for not corrective is 90,258 and for corrective 35,124. For both samples the median, Shapiro-Wilk test statistic and p-value are given comma separated.

Perfective changes				
Metric	MWU Statistic	Median	SHA Statistic	SHA p-val
McCC	2579401012.5	0.02,0.00	0.55,0.27	<0.0001,<0.0001
LLOC	2691844899.5	0.25,0.00	0.57,0.20	<0.0001,<0.0001
NLE	2351847133.0	0.00,0.00	0.58,0.20	<0.0001,<0.0001
NUMPAR	2328626543.0	0.00,0.00	0.39,0.05	<0.0001,<0.0001
CC	1666541612.5	0.00,0.00	0.03,0.01	<0.0001,<0.0001
CLOC	2158356261.5	0.00,0.00	0.32,0.34	<0.0001,<0.0001
CD	1715608163.5	0.00,0.00	0.41,0.21	<0.0001,<0.0001
AD	1899339427.0	0.00,0.00	0.25,0.13	<0.0001,<0.0001
NOA	1997259809.5	0.00,0.00	0.06,0.01	<0.0001,<0.0001
CBO	2208901912.0	0.00,0.00	0.21,0.04	<0.0001,<0.0001
NII	2210463268.0	0.00,0.00	0.09,0.07	<0.0001,<0.0001
Minor	2201853734.0	0.00,0.00	0.04,0.01	<0.0001,<0.0001
Major	2077680338.5	0.00,0.00	0.04,0.00	<0.0001,<0.0001
Critical	1952568002.5	0.00,0.00	0.05,0.05	<0.0001,<0.0001
Corrective changes				
McCC	1319862052.5	0.00,0.00	0.36,0.36	<0.0001,<0.0001
LLOC	1406100592.5	0.07,0.18	0.36,0.36	<0.0001,<0.0001
NLE	1538986445.5	0.00,0.00	0.35,0.35	<0.0001,<0.0001
NUMPAR	1736495605.5	0.00,0.00	0.14,0.14	<0.0001,<0.0001
CC	1781604826.0	0.00,0.00	0.01,0.01	<0.0001,<0.0001
CLOC	1665288104.5	0.00,0.00	0.38,0.38	<0.0001,<0.0001
CD	1833218654.0	0.00,0.00	0.28,0.28	<0.0001,<0.0001
AD	1719709796.5	0.00,0.00	0.19,0.19	<0.0001,<0.0001
NOA	1700427713.0	0.00,0.00	0.03,0.03	<0.0001,<0.0001
CBO	1687001103.5	0.00,0.00	0.09,0.09	<0.0001,<0.0001
NII	1621472694.0	0.00,0.00	0.11,0.11	<0.0001,<0.0001
Minor	1664776380.0	0.00,0.00	0.01,0.01	<0.0001,<0.0001
Major	1667877088.0	0.00,0.00	0.01,0.01	<0.0001,<0.0001
Critical	1631274846.5	0.00,0.00	0.07,0.07	<0.0001,<0.0001

Table 15: Detailed statistical tests results for metrics before the change is applied. Accompanies Table 9. SHA is Shapiro-Wilk, MWU is Mann-Whitney U test, the number of samples for not perfective is 77,630, for perfective the number is 47,852. The number of samples for not corrective is 90,258 and for corrective 35,124. For both samples the median, Shapiro-Wilk test statistic and p-value are given comma separated.

Perfective changes				
Metric	MWU Statistic	Median	SHA Statistic	SHA p-val
McCC	1946723702.0	47.00,39.00	0.27,0.21	<0.0001,<0.0001
LLOC	1946637361.5	397.00,335.00	0.26,0.21	<0.0001,<0.0001
NLE	1934702498.0	21.00,18.00	0.28,0.24	<0.0001,<0.0001
NUMPAR	1860319211.0	34.00,32.00	0.25,0.20	<0.0001,<0.0001
CC	1881849087.0	0.09,0.08	0.07,0.10	<0.0001,<0.0001
CLOC	1642584608.5	84.00,118.00	0.18,0.24	<0.0001,<0.0001
CD	1570226793.0	0.40,0.54	0.08,0.14	<0.0001,<0.0001
AD	1548982847.0	0.83,1.00	0.11,0.14	<0.0001,<0.0001
NOA	1861405551.0	2.00,2.00	0.13,0.09	<0.0001,<0.0001
CBO	2023896520.5	21.00,15.00	0.24,0.16	<0.0001,<0.0001
NII	1756171669.5	15.00,18.00	0.25,0.21	<0.0001,<0.0001
Minor	1926916681.5	15.00,13.00	0.15,0.13	<0.0001,<0.0001
Major	2025070328.5	4.00,3.00	0.23,0.17	<0.0001,<0.0001
Critical	1949852017.5	0.00,0.00	0.19,0.12	<0.0001,<0.0001
Corrective changes				
McCC	1455448657.0	41.00,50.00	0.23,0.23	<0.0001,<0.0001
LLOC	1506999970.0	361.00,399.00	0.23,0.23	<0.0001,<0.0001
NLE	1477296467.5	18.00,22.00	0.25,0.25	<0.0001,<0.0001
NUMPAR	1573653093.5	33.00,33.00	0.22,0.22	<0.0001,<0.0001
CC	1605078855.0	0.09,0.08	0.09,0.09	<0.0001,<0.0001
CLOC	1683529860.5	101.00,83.00	0.22,0.22	<0.0001,<0.0001
CD	1832629967.5	0.51,0.35	0.12,0.12	<0.0001,<0.0001
AD	1822953682.5	1.00,0.75	0.13,0.13	<0.0001,<0.0001
NOA	1616227506.0	2.00,2.00	0.10,0.10	<0.0001,<0.0001
CBO	1476937730.0	17.00,21.00	0.19,0.19	<0.0001,<0.0001
NII	1655048856.5	17.00,14.00	0.22,0.22	<0.0001,<0.0001
Minor	1557520234.5	14.00,14.00	0.14,0.14	<0.0001,<0.0001
Major	1516141644.5	3.00,4.00	0.19,0.19	<0.0001,<0.0001
Critical	1546362144.0	0.00,0.00	0.15,0.15	<0.0001,<0.0001



## **H ASAT warning density in defect inducing changes**

This section contains a copy of the following publication.

A. Trautsch, S. Herbold, J. Grabowski Are automated static analysis tools worth it?  
An investigation into relative warning density and external software quality.  
© 2022, The Author(s). Reprinted with permission.

journal doi follows after publication

preprint: <https://doi.org/10.48550/arXiv.2111.09188>

# Are automated static analysis tools worth it? An investigation into relative warning density and external software quality

Alexander Trautsch · Steffen Herbold ·  
Jens Grabowski

Received: date / Accepted: date

**Abstract** Automated Static Analysis Tools (ASATs) are part of software development best practices. ASATs are able to warn developers about potential problems in the code. On the one hand, ASATs are based on best practices so there should be a noticeable effect on software quality. On the other hand, ASATs suffer from false positive warnings, which developers have to inspect and then ignore or mark as invalid. In this article, we ask the question if ASATs have a measurable impact on external software quality, using the example of PMD for Java. We investigate the relationship between ASAT warnings emitted by PMD on defects per change and per file. Our case study includes data for the history of each file as well as the differences between changed files and the project in which they are contained. We investigate whether files that induce a defect have more static analysis warnings than the rest of the project. Moreover, we investigate the impact of two different sets of ASAT rules. We find that, bug inducing files contain less static analysis warnings than other files of the project at that point in time. However, this can be explained by the overall decreasing warning density. When compared with all other changes, we find a statistically significant difference in one metric for all rules and two metrics for a subset of rules. However, the effect size is negligible in all cases, showing that the actual difference in warning density between bug inducing changes and other changes is small at best.

---

Alexander Trautsch  
Institute of Computer Science, University of Goettingen, Germany  
E-mail: alexander.trautsch@cs.uni-goettingen.de

Steffen Herbold  
Institute of Software and Systems Engineering, TU Clausthal, Germany  
E-mail: steffen.herbold@tu-clausthal.de

Jens Grabowski  
Institute of Computer Science, University of Goettingen, Germany  
E-mail: grabowski@cs.uni-goettingen.de

**Keywords** Static code analysis · Quality evolution · Software metrics · Software quality

## 1 Introduction

Automated Static Analysis Tools (ASATs) or linters are programs that perform rule matching of source code via different representations, e.g., Abstract Syntax Trees (ASTs), call graphs or bytecode to find potential problems. Rules are predefined by the ASAT and based on common coding mistakes and best practices. If a rule is matched, a warning is generated for the developer who can then inspect the given file, line number and rule. Common coding best practices involve ASATs use at different contexts (Vassallo et al., 2020), e.g., as part of Continuous Integration (CI), within IDEs, or to support code reviews. Developers also think of these tools as quality improving when used correctly (Christakis and Bird, 2016; Vassallo et al., 2020; Devanbu et al., 2016; Querel and Rigby, 2021). However, due to their rule matching nature, ASATs are prone to false positives, i.e., warnings about code that is not problematic (Johnson et al., 2013). This hinders the adoption of these tools and their usefulness, as developers have to inspect every warning that is generated whether it is a false positive or not. As a result, research into classifying ASAT warnings into true and false positives or actionable warnings is conducted, e.g., Heckman and Williams (2009), Kim and Ernst (2007) and Koc et al. (2017). Due to these two aspects, ASATs are perceived as quality improving while at the same time require manual oversight and corrections.

Due to this manual effort, we want to have a closer look on the impact of ASATs on measurable quality. Previous research regarding the impact on quality can be divided into direct measures which target bug fixing commits, e.g., Vetro et al. (2011), Thung et al. (2012); Habib and Pradel (2018) and indirect measures which use ASAT warnings as part of predictive models, e.g., Nagappan and Ball (2005), Plosch et al. (2008), Rahman et al. (2014), Querel and Rigby (2018), Lenarduzzi et al. (2020), Trautsch et al. (2020b) and Querel and Rigby (2021). Both approaches usually suffer from data validation issues. ASAT warnings measured directly in bug fixing commits are usually validated via manual inspection of the warnings either by students (Vetro et al., 2011) or by the researchers themselves (Thung et al., 2012; Habib and Pradel, 2018). Predictive models that include ASAT warnings as features includes bugs and bug fixing commits which introduces new data validity problems, e.g., with noisy issue types as reported by Antoniol et al. (2008) and Herzig et al. (2013a), links from bug reports to commits, or the SZZ (Śliwerski et al., 2005) implementation (Fan et al., 2019). Recently Rosa et al. (2021) demonstrated in their investigation that finding bug inducing information is challenging, when using common or even state-of-the art SZZ approaches. In this article, we mitigate this by using manually validated bug fixing data from a large-scale validation study (Herbold et al., 2021). While this reduces the data available

for the study, we believe that this is a worthwhile trade off to be able to get more signal with less noise from the available data.

Between direct and indirect impact studies is a missing piece however. Direct impact studies make sense for security focused ASATs (Flawfinder, RATS) or bug focused ASATs (FindBugs, SpotBugs), but not necessarily for general purpose ASATs (PMD, SonarQube). General purpose ASATs are able to uncover readability related issues, e.g., style or naming issues in addition to common programming errors or violations of best practices. These issues, while possibly not directly responsible for a bug, may introduce bugs later, or, in other parts of the file due to the reduced understandability of the code. Indirect studies that rely on ASAT warnings as features for defect prediction may be too indirect to measure the actual relation between the ASATs and defects, as this is by necessity done in relation to prediction models using other features of changes. Moreover, this approach ignores problems due to differences in warning density between projects or correlations with size or time.

In order to make general statements, differencing techniques can be used, e.g., differences between bug inducing commits and previous commits as is done in the work by Lenarduzzi et al. (2020). What is missing however, is a more general view of the differences between warning densities of the files and the rest of the project at each time step. Such a view would allow us to explore whether files that contain more static analysis warnings than the rest of the project also induce more bugs.

In prior work (Trautsch et al., 2020a) we investigated trends of ASAT warnings and investigated whether usage of an ASAT has an influence on software quality measured via defect density (Fenton and Bieman, 2014). However, our analysis with regards to bugs via defect density was coarse grained. We address this limitation in this article. We adopt a recently introduced approach for fine-grained just-in-time defect prediction by Pascarella et al. (2019) to perform a more targeted investigation of the impact of static analysis warnings on quality via introduced bugs as a proxy for software quality instead of the more coarse grained defect density for one year. The approach by Pascarella et al. (2019) provides the advantage of handling files within commits and not only commits which makes it suitable for us to study the impact of static analysis over the history of all files within our study subjects.

In our previous work, we found that static analysis warnings are evolving over time and that we can not just use the density or the sum of warnings in our case study (Trautsch et al., 2020a). Therefore, we use an approach that is able to produce a current snapshot view of the files we are interested in by measuring the file that induces a bug and, at the same time, all other files of the project. This ensures that we are able to produce time and project independent measurements. The drawback of this approach is that it requires a large computational effort, as we have to run the ASAT under study on every file in every revision of all study subjects. However, the resulting empirical data yields insights for researchers and practitioners.

The research question that we answer in our exploratory study is:

- Do bug inducing files contain more static analysis warnings than other files?

We apply a modified fine-grained just-in-time defect prediction data collection method to extract software evolution data including bug inducing file changes and static analysis warnings from a general purpose ASAT. We chose PMD<sup>1</sup> as the general purpose ASAT as it has been available for a long time and provides a good mix of available rules. Using this data and a warning density based metric calculation, we investigate the differences between bug inducing files and the rest of the studied system at the point in time when the bug is introduced. In summary, this article contains the following contributions.

- A unique and simple approach to measure impact of ASATs that is independent of differences between projects, size and time.
- Complete static analysis data for PMD for 23 open source projects for every file in every commit.
- An investigation into relative warning density differences within bug inducing changes.

The main findings of our exploratory study are:

- Bug inducing files do not contain higher warning density than the rest of the project at the time when the bug is introduced.
- When comparing bug inducing warning density with all other changes we can measure higher warning density on a subset of PMD warnings that is a popular default for two metrics and for all available rules for one metric.

The rest of this article is structured as follows. Section 2 lists previous research related to this article and discusses the differences. Section 3 describes the case study setup, methodology, analysis procedure and the results. Section 4 discusses the results of our case study and relates them to the literature. Section 5 lists and discusses threats to validity we identified for our study. Section 6 concludes the article with a short summary and provides a short outlook.

## 2 Related Work

In this article, we explore a more general view of ASATs and the warning density differences of bug inducing changes. This can be seen as a mix of a direct and indirect impact study. Therefore we describe related work for both direct and indirect impact studies within this section.

The direct impact is often evaluated by exploring if bugs that are detected in a project are fixed by removing ASAT warnings, i.e., did the warning really indicate a bug that needed to be fixed later.

---

<sup>1</sup> <https://pmd.github.io/>

Thung et al. (2012) investigated bug fixes of three open source projects and three ASATs: PMD, JLint, and FindBugs. The authors look at how many defects are found fully and partially by changed lines and how many are missed by the ASATs. Moreover, the authors describe the challenges of this approach: not every line that is changed is really a fix for the bug, therefore the authors perform manual investigation on a per-line level to identify the lines. They were able to find all lines responsible for 200 of 439 bugs. In addition, the authors find that PMD and FindBugs perform best, however their warnings are often very generic.

Habib and Pradel (2018) perform an investigation of the capabilities to find real world bugs via ASATs. The authors used the Defects4J dataset by Just et al. (2014) with an extension<sup>2</sup> to investigate the number of bugs found by three static analysis tools, SpotBugs, Infer and error-prone. The authors show that 27 of 594 bugs are found by at least one of the ASATs.

In contrast to Thung et al. (2012) and Habib and Pradel (2018), we only perform an investigation of PMD. However, due to our usage of SmartSHARK (Trautsch et al., 2017), we are able to investigate 1,723 bugs for which at least three researchers achieved consensus on the lines responsible for the bug. Moreover, as PMD includes many rules related to readability and maintainability, we build on the assumption that while they are not directly indicating a bug, resolving these warnings improves the quality of the code and may prevent future bugs. This extends previous work by taking possible long term effects of ASAT warnings into account.

Indirect impact is explored by using ASAT warnings as features for predictive models and providing a correlation measure of ASAT warnings to bugs.

Nagappan and Ball (2005) explore the ability of ASAT warnings to predict defect density in modules. The authors found in a case study with Microsoft, that static analysis warnings can be used to predict defect density, therefore they can be used to focus quality assurance efforts on modules that show a high number of static analysis warnings. In contrast to Nagappan and Ball (2005), we are exploring open source projects. Moreover, we explore warning density differences between files and the project they are contained in.

Rahman et al. (2014) compare static analysis and statistical defect prediction. They find that FindBugs is able to outperform statistical defect prediction, while PMD does not. Within our study, we focus on PMD as a general purpose ASAT. Instead of a comparison with statistical defect prediction we explore, whether we can measure a difference of ASAT warnings between bug inducing changes and other changes.

Plosch et al. (2008) explores a correlation between ASAT warnings as features for a predictive model and the dependent variable, i.e., bugs. They found that static analysis warnings may improve the performance of predictive models and that they are correlated with bugs. In contrast to Plosch et al. (2008), we are not building a predictive model. We are exploring whether we can

---

<sup>2</sup> <https://github.com/rjust/defects4j/pull/112>

find an effect of static analysis tools without a predictive model with multiple features, instead we strive to keep the approach as simple as possible.

Querel and Rigby (2018) improve the just-in-time defect prediction based commit guru (Rosen et al., 2015) by adding ASAT warnings to the predictive model. The authors show, that just-in-time defect prediction can be improved by adding static analysis warnings. This means that there should be a connection between external quality in the form of bugs and static analysis warnings. In a follow up study (Querel and Rigby, 2021) the authors found that while there is an effect of ASAT warnings the effect is likely small. In our study, we explore a different view on the data. We explore warning density differences between bug inducing files and the rest of the project.

Lenarduzzi et al. (2020) investigated SonarQube as an ASAT and if the reported warnings can be used as features to detect reported bugs. The authors are combining direct with indirect impact but are more focused on predictive model performance measures. In contrast to Lenarduzzi et al. (2020), we are mainly interested in the differences in warning density between bug inducing files and the rest of the project. We are also investigating an influence, but in contrast to Lenarduzzi et al., we are comparing our results for bug fixing changes to all other changes to determine whether what we see is really part of the bug fixing change and not a general trend of all changes.

### 3 Case Study

The goal of the case study is to find evidence if usage of ASATs have a positive impact on the external software quality of our case study subjects. In this section, we explain the approach and ASAT choice. Moreover, we explain our study subject selection and describe the methodology and analysis procedure. At the end of this section we present the results.

#### 3.1 Static analysis

Static analysis is a programming best practice. ASATs scan source code or byte code and match against a predefined set of rules. When a rule matches, the tool creates a warning for the part of the code that matches the rule.

There are different tools for performing static analysis of source code. For Java these would be, e.g., Checkstyle, FindBugs/SpotBugs, PMD, or SonarQube. In this article, we focus on Java as a programming language because it is widely used in different domains and has been in use for a long time. The static analysis tool we use is PMD. There are multiple reasons for this. PMD does not require the code to be compiled first as, e.g., FindBugs does. This is an advantage especially with older code that might not compile anymore due to missing dependencies (Tufano et al., 2017). PMD supports a wide range of warnings of different categories, e.g., naming and brace rules as well as common coding mistakes. This is an advantage over, e.g., Checkstyle which

mostly deals with coding style related rules. This enables PMD to give a better overview of the quality of a given file instead of giving only probable bugs within it. The relation to software quality that we expect of PMD stems directly from its rules. The rules are designed to make the code more readable, less error prone and overall more maintainable.

### 3.2 Just-in-time defect prediction

The idea behind just-in-time defect prediction is to assess the risk of a change to an existing software project (Kamei et al., 2013). Previous changes are extracted from the version control system of the project and, as they are in the past, it is known whether the change induced a bug. This can be observed by subsequent removal or alteration of the change as part of a bug fixing commit. If the change was indeed removed or altered as part of a bug fixing operation it is traced back to its previous file and change and labeled as bug inducing, i.e., it introduced a bug that needed to be fixed later. In addition to these labels, certain characteristics of the change are extracted as features, e.g., lines added or the experience of the author to later train a model to predict the labels correctly for the commits. The result of the model is then a label or probability whether the change introduces a bug, i.e., the risk of the change.

However, ASATs are working on a file basis and we also want to investigate longer-term effects of ASATs. This means we need to track a file over its evolution in a software project. To achieve this, we are building on previous work by Pascarella et al. (2019) which introduced fine-grained just-in-time defect prediction. In a previous study, we improved the concept by including better labels and static analysis warnings as well as static code metrics as features (Trautsch et al., 2020b). Similar to Pascarella et al. (2019), we are building upon PyDriller (Spadini et al., 2018). In this article, we build upon our previous work and include not only counts of static analysis warnings but relations between the files, e.g., how different is the number of static analysis warnings in one file from the rest of the project. We also include aggregations of warnings with and without a decay over time.

### 3.3 Study Subjects

Our study subjects consist of 23 Java projects under the umbrella of the Apache Software Foundation<sup>3</sup> previously collected by (Herbold et al., 2020). Table 1 contains the list of our study subjects. We only use projects which contain fully validated bug fixing on a line-by-line level collected in a crowd sourcing study (Herbold et al., 2021). Every line in our data was labeled by four researchers. We only consider bug fixing lines for which at least three researchers agree that it fixes the considered bug. This naturally restricts the

---

<sup>3</sup> <https://www.apache.org>

Table 1: Study subjects in our case study

Project	#commits	#file changes	#issues	Time frame
ant-ivy	1,647	7,860	296	2005-2017
commons-bcel	850	9,604	27	2001-2017
commons-beanutils	561	2,648	28	2001-2017
commons-codec	810	2,062	21	2003-2017
commons-collections	1,687	11,296	32	2001-2017
commons-compress	1,401	3,566	87	2003-2017
commons-configuration	1,659	4,177	97	2003-2017
commons-dbcp	729	2,211	39	2001-2017
commons-digester	1,131	3,750	11	2001-2017
commons-io	985	2,781	51	2002-2017
commons-jcs	774	7,775	37	2002-2017
commons-lang	3,028	6,312	109	2002-2017
commons-math	4,135	21,440	190	2003-2017
commons-net	1,076	4,666	96	2002-2017
commons-scxml	469	1,774	39	2005-2017
commons-validator	557	1,324	37	2002-2017
commons-vfs	1,098	7,209	67	2002-2017
giraph	819	7,715	109	2010-2017
gora	464	2,256	38	2010-2017
opennlp	1,166	9,679	82	2010-2017
parquet-mr	1,053	5,957	69	2012-2017
santuario-java	1,177	8,503	41	2001-2017
wss4j	1,711	12,218	120	2004-2017
Sum	28,987	146,783	1,723	

number of available projects but improves the noise to signal ratio of the data. We now give a short overview what the potential problems are and how we mitigate them. When we look at external quality, we want to extract data about defects. However, there are several additional restrictions we want to apply. First, we want to extract defects from the Issue Tracking System (ITS) of the project and link them to commits in the Version Control System (VCS) to determine bug fixing changes. Several data validity considerations need to be taken in to account here. The ITS usually has a kind of label or type to distinguish bugs from other issues, e.g., feature requests. However, research shows that this label is often incorrect, e.g., Antoniol et al. (2008), Herzig et al. (2013b) and Herbold et al. (2020). Moreover, with this kind of software evolution research, we are interested in bugs existing in the software and not bugs which occur because of external factors, e.g., new environments or dependency upgrades. Therefore, we are only considering intrinsic bugs (Rodriguez-Pérez et al., 2020).

The next step is the linking between the issue from the ITS and the commit from the VCS. This is achieved via a mention of the issue in the commit message, e.g., fixes JIRA-123. While this seems straightforward there are certain cases where this can be problematic. The simplest one being that there is a typo in the project key, e.g., JRIA-123.

Moreover, not all changes within bug fixing commits contribute to bug fixes. Unrelated changes can be tangled with the bug fix. The restriction of all data to only changes that directly contribute to the bug fix further reduces noise in the data. We are only interested in the lines of the changes that contribute to the bug fix. This is probably the hardest to manually validate.

This was achieved in a prior publication (Herbold et al., 2020) which served as the base for the publication which data we use in this article (Herbold et al., 2021). In (Herbold et al., 2021) a detailed untangling is performed by four different persons for each change that fixes a bug that meets our criteria.

### 3.4 Replication Kit

We provide all data and scripts as part of a replication kit<sup>4</sup>.

### 3.5 Methodology

To answer our research question, we extract information about the history of our study subjects including bugs and the evolution of static analysis warnings. While the bulk of the data is based on (Herbold et al., 2021) we include several additions necessary for answering our research question.

To maximize the relevant information within our data we include as much information from the project source code repository as possible. After extracting the bug inducing changes, we build a commit graph of all commits of the project and then find the current main branch, usually master. After that, we find all orphan commits, i.e., all commits without parents. Then we discard all orphans that do not have a path to the last commit on the main branch, this discards separate paths in the graph, e.g., gh-pages<sup>5</sup> for documentation. As we also want to capture data on release branches which are never merged back into the main branch, we add all other branches that have a path to one of our left over orphan commits. The end result is a connected graph which we traverse via a modified breadth first search. We take the date of the commit into account while we traverse the graph.

The traversal is an improved version of previous work (Trautsch et al., 2020b). In addition to the previously described noise reduction via manual labeling, we additionally restrict all files to production code. One of the results of Herbold et al. (2021) is that non-production code is often tangled with bug fixing changes. Therefore we only add files that are production files to our final data analogous to Trautsch et al. (2020a). This also helps us to provide a clearer picture of warning density based features as production code may have a different evolution of warning density than, e.g., test or example code.

In our previous study (Trautsch et al., 2020a) we found that static analysis warnings are correlated to Logical Lines of Code (LLOC). This is not surprising

---

<sup>4</sup> [https://github.com/atrautsch/emse2021a\\_replication](https://github.com/atrautsch/emse2021a_replication)

<sup>5</sup> <https://docs.github.com/en/pages>

as we are observing large portions of our study subjects code history. Large files that are added and removed have an impact on the number of static analysis warnings. While we do not want to discard this information we also want to avoid the problem of large changes overshadowing information in our data. Therefore, like in our previous study we are using warning density as a base metric in this study analogous to prior studies, e.g., Aloraini et al. (2019) and Penta et al. (2009).

Warning density ( $wd$ ) is the ratio of the number of warnings and the size of the analyzed part of the code.

$$wd = \frac{\text{Number of static analysis warnings}}{\text{Product size}} \quad (1)$$

Product size is measured in LLOC. If we measure the warning density of a system  $wd(s)$ , we sum warnings and LLOC for each file. If we measure the warning density of a file  $wd(f)$ , we restrict the number of warnings and the LLOC to that file.

While this measure provides a size independent metric, we also need to take differences between projects into account. Warning density can be different between projects and even more so for different points in time for each project. To be able to use all available data we account for these differences by using differences in warning density between the files of interest and the rest of the project under study (the system) at the specific point in time.

We calculate the warning density difference between the file and the system  $fd(f_t)$ .

$$fd(f_t) = wd(f_t) - wd(s_t) \quad (2)$$

If the file  $f$  at time  $t$  contains less static analysis warnings per LLOC than the system  $s$  at time  $t$  the value is negative and if it contains more it is positive. We can use this metric to investigate bug inducing commits and determine whether the files responsible for bugs contain less or more static analysis warnings per LLOC than the system they belong to.

While this yields information corrected for size, project differences, and time of the change we also want to incorporate the history of each file. Therefore, we also sum this difference in warning density for all changes to the file. We assume that recent changes are more important than old changes. Therefore, we introduce a decay in our warning density derived features.

$$dfd(f_t) = \sum_{j=1}^{j=t} \frac{wd(f_j) - wd(s_j)}{t - j + 1} \quad (3)$$

For the decayed file system warning density delta  $dfd(f_t)$  we compute the decayed, cumulative sum of the difference between the warning density of the file ( $wd(f_t)$ ) and the warning density of the system ( $wd(s_t)$ ). The rationale is that if a file is constantly better, with regards to static analysis warnings, than the mean of the rest of the system this should have a positive effect. As the static analysis rules are diverse this can be improved readability, maintainability or

robustness due to additional null checks. Within our study, we explore if this effect has a measurable effect on buggyness, i.e., the lower this value is the less often the file should be part of bug inducing commits.

Instead of using all warnings for warning density we can also restrict these warnings to a smaller set to see if this has an effect. While we do not want to choose multiple subsets to avoid false positive findings, we have to investigate whether our approach to use all available warnings just waters down the ability to indicate files which may contain bugs. To this end, we also investigate the warning density consisting only of PMD warnings that are enabled by default by the maven-pmd plugin<sup>6</sup> which we denote as *default* rules. This restricts the number of warnings that are the basis of the warning density calculation to a subset of 49 warnings that are generally considered relevant in comparison to the total number of 314 warnings. Their use as default warnings serves to restrict this subset to generally accepted important warnings.

To answer our research question we compare the warning density for each bug inducing file against the project at the time before and after the bug inducing change. If the difference is positive this means that the file had a higher warning density than the rest of the project and negative vice versa. We plot the difference in warning density in a box plot for all bug inducing files to provide an overview over all our data.

As this is influenced by a continuously improving warning density we also measure the differences between bug inducing file changes and all other file changes. We first perform a normality test and find that the data is not normal in all cases. Thus, we apply a Mann-Whitney U test (Mann and Whitney, 1947) with  $H_0$  that there is no difference between both populations and  $H_1$  that bug inducing files have a different warning density. We set a significance level of 0.05. Additionally, we perform a Bonferroni (Abdi, 2007) correction for 8 normality tests and 4 Mann-Whitney U tests. Therefore we reject  $H_0$  at  $p < 0.0042$ . If the difference is statistically significant we calculate the effect size with Cliff's  $\delta$  (Cliff, 1993).

### 3.6 Results

We now present the results of our study and the answer to our research question whether bug inducing files contain more static analysis warnings than other files. For this, we divide the results into three parts. First, we look at the warning density via  $fd(f)$  at the time before and after a bug is induced and  $dfd(f)$  after a bug is induced<sup>7</sup>. Second, we look at the differences between our study subjects and the prior number of changes for bug inducing file changes. Third, we compare bug inducing file changes with all other changes and determine if they are different.

---

<sup>6</sup> <https://maven.apache.org/plugins/maven-pmd-plugin/>

<sup>7</sup> before is already part of the formula

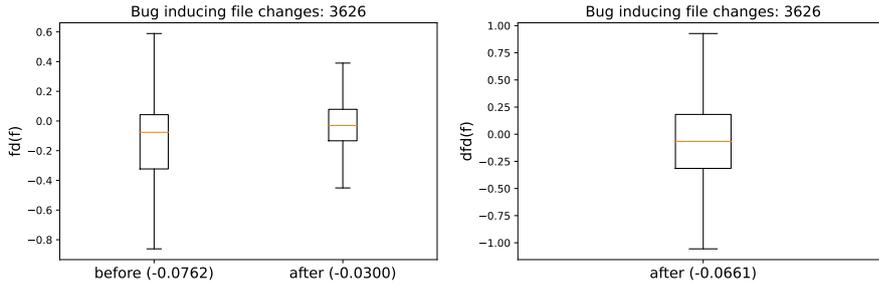


Fig. 1: Box plot of  $fd(f)$  for all bug inducing files before and after the bug inducing change and  $dfd(f)$  for all bug inducing files after the bug inducing change, median value in parentheses. Fliers are omitted.

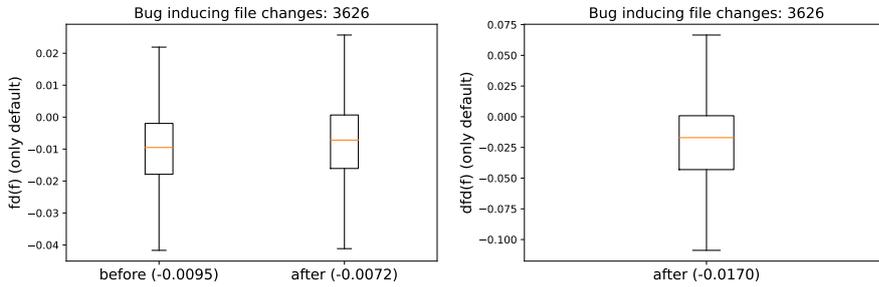


Fig. 2: Box plot of  $fd(f)$  for only default warnings of all bug inducing files before and after the bug inducing change, median value in parentheses. Fliers are omitted.

### 3.6.1 Differences of warning density before and after the bug inducing change

Figure 1 shows the difference in warning density between the each bug inducing file and the rest of the system at the point in time before inducing the bug and after. Surprisingly, we see a negative warning density median difference for  $fd(f)$ . This means that the warning density of the files in which bugs are induced is lower than the rest of the project. The drop in warning density shows that the code before the change had less warning density than after the bug inducing change. This means that code that on average contains more static analysis warnings was introduced as part of the bug inducing change.

Now, we are also interested if the history of preceding differences in warning density makes a difference. Instead of using the warning density difference at the point in time of the bug inducing change we use a decayed sum of the warning density differences leading up to the considered bug inducing change.

Figure 1 shows a negative median for  $dfd(f)$  as well. The accumulated warning density differences between the file and the rest of the project are therefore also negative. Figure 2 shows  $fd(f)$  and  $dfd(f)$  for bug inducing

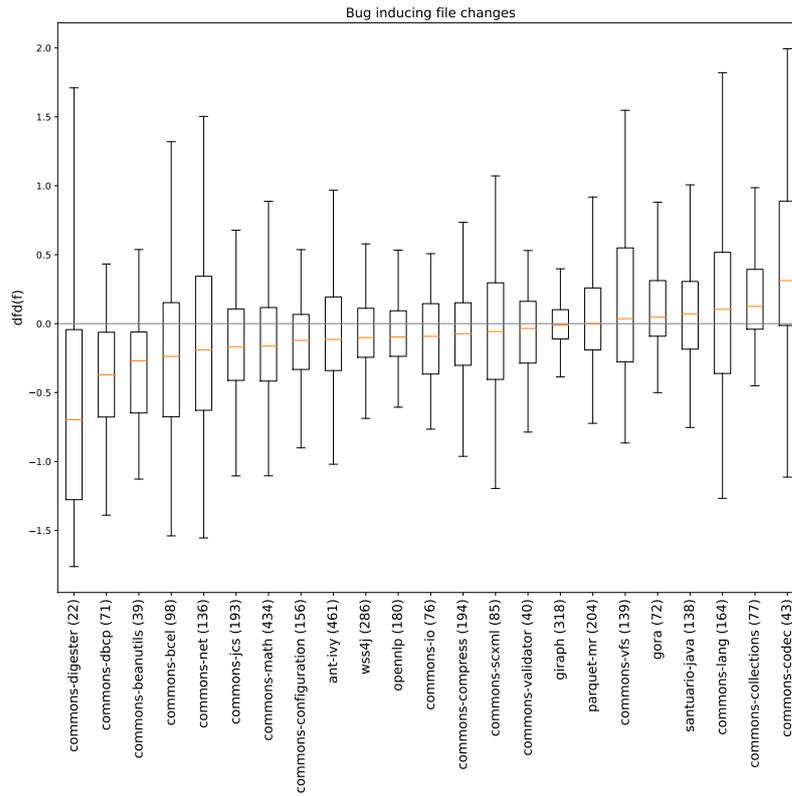


Fig. 3: Box plots of  $dfd(f)$  separately for all study subjects. The number of bug inducing file changes are in parentheses, median value in parentheses. Fliers are omitted.

changes restricted to default rules. We can see, that the warning density for default only is much lower due to the lower number of warnings that are considered. We can also see, that the same negative median is visible when we restrict the set of ASAT rules to default. Overall, bug inducing changes have lower warning density than the other files of the project at the time the bug was induced. However, as we will see later, this is an effect of overall decreasing warning density of our study subjects.

### 3.6.2 Differences between projects and number of changes

Instead of looking at all files combined we can also look at each project on its own. We provide this data in Figure 3. However, we note that the number of bug inducing files is low in some projects. Such projects may be influenced by few changes with extreme values. Hence, the results of single projects should be interpreted with caution. Instead we consider trends visible in the data. While

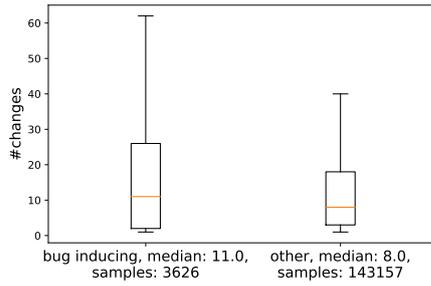


Fig. 4: Number of changes for bug inducing files and other files. Fliers are omitted.

we can combine all our data due to our chosen method of metric calculation we still want to provide an overview of the per project values. This is shown in Figure 3 for  $dfd(f)$ . Figure 3 also demonstrates the difference between projects. For example, the median  $dfd(f)$  for comons-codec is positive, i.e., files which induce bugs contain more warnings. The opposite is the case for, e.g., commons-digester, where the median is negative.

Overall, Figure 3 shows that the median  $dfd(f)$  is negative for 16 of 23 projects. This means that bug inducing changes have less warning density than the rest of the project for most study subjects. A possible explanation for this could be that files which have a lower warning density are changed more often and those are the same that could be inducing bugs. If we look at the number of changes a file has in Figure 4, we can see that bug inducing files have a bit more changes. However, the sample sizes for both are vastly different.

### 3.6.3 Comparison with all other changes

We now take a look at how warning density metrics differ in bug inducing changes from all other changes. We notice that the median is below zero in all cases. This is due to the effect that warning density usually decreases over time (Trautsch et al., 2020a). Therefore, we provide a comparison of bug inducing changes with all other changes.

Figure 5 shows  $fd(f)$  for bug inducing and other changes for both all rules and only the default rules. We can see that bug inducing changes have a slightly higher warning density than other changes. If we apply only default rules we see that bug inducing changes are also slightly higher.

Figure 6 shows the same comparison for  $dfd(f)$ . The difference for all rules is very small. However, the median for bug inducing changes is slightly higher. In contrast, we can see that for default rules the bug inducing changes have a slightly higher warning density than other changes. Table 2 shows the results of the statistical tests for differences between the values for Figure 5 and Figure 6.

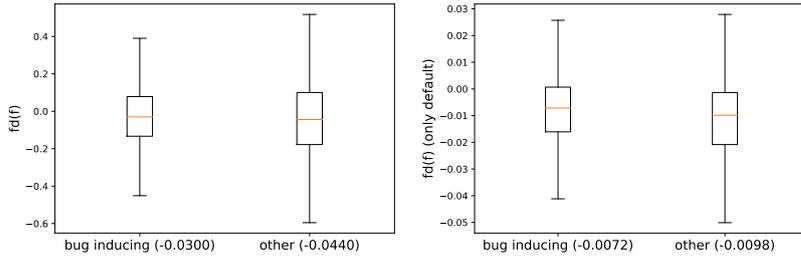


Fig. 5: Box plots of  $fd(f)$  before the bug inducing change for all and default only rules for bug inducing and other file changes, median value in parentheses. Fliers are omitted.

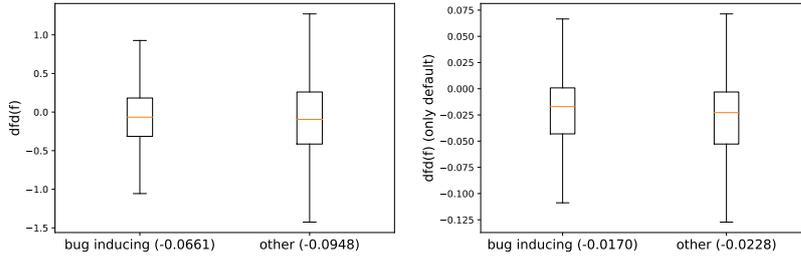


Fig. 6: Box plots of  $dfd(f)$  for all and default only rules bug inducing and other file changes, median value in parentheses. Fliers are omitted.

Table 2: Median values, Mann-Whitney U test p-values and effect sizes for all warning density metrics.

WD Metric	Median other	Median bug inducing	P-value	Effect size
$fd(f)$	-0.0440	-0.0300	<0.0001	0.05 (n)
$fd(f)$ (default)	-0.0098	-0.0072	<0.0001	0.10 (n)
$dfd(f)$	-0.0948	-0.0661	0.0247	-
$dfd(f)$ (default)	-0.0228	-0.0170	<0.0001	0.07 (n)

We can see that for all rules  $fd(f)$  there is a statistically significant difference. This shows that bug inducing file changes have a higher warning density than other changes.

Overall, we see that there is a significant difference with a negligible effect size for  $fd(f)$  and  $dfd(f)$  for default rules between bug inducing and other changes. The data shows that in these cases the bug inducing file changes have a higher warning density than other changes. Together with Figure 5, Figure 6 and Table 2 we can conclude, that bug inducing file changes contain more static analysis warnings than other file changes. Restricting the rules to the default set increases the effect size slightly. However, the effect sizes are still negligible in all cases.

### 3.6.4 Summary

In summary, we have the following results for our research question.

**RQ Summary:** Do bug inducing files contain more static analysis warnings than other files?

We find that bug inducing files contain less static analysis warnings than other files at bug inducing time. However, this is not because these files have a higher quality, but rather because the warning density decreases over time, i.e., most files that are changed have less warnings than the rest of the project.

When we compare the differences between bug inducing and other changes, we find that it depends on the applied rules. If we apply all rules we find a statistically significant difference in  $fd(f)$ . If we apply only default rules we find a statistically significant difference in  $fd(f)$  and  $dfd(f)$ . This indicates that bug inducing files have slightly more static analysis warnings than other files, although the effect size in all cases is negligible.

## 4 Discussion

We found that the bug inducing change itself increased the warning density of the code in comparison to the rest of the project as shown in Figure 1. This means that the actual change in warning density is as we expected, i.e., the change that induces the bug is increasing the warning density in comparison to the rest of the project. This is an indication that warning density related metrics can be of use in just-in-time defect prediction scenarios, i.e., change based scenarios, as also shown by Querel and Rigby (2021) and in our previous work Trautsch et al. (2020b). However, the effect is negligible in our data. This was also the case for predictive models by Querel and Rigby (2021). Thus, any gain in prediction models due to general static analysis warnings is likely very small.

However, when we look at the median difference between bug inducing files and the rest of the project at that point in time we see that bug inducing files contain less static analysis warnings. This counter intuitive result can be fully explained by the overall decreasing warning density over time we found in our previous study (Trautsch et al., 2020a). This finding is highly relevant for researchers, because this shows the importance of accounting for time as confounding factor for the evaluation of the effectiveness of methods. Without the careful consideration of the change over time, we would now try to explain why bug inducing files have less warnings and other researchers may built on this faulty conclusion. Therefore, this part of our results should also be a cautionary tale for other researchers that investigate the effectiveness of tools

and methods: if the complete project is used as a baseline, it should always be considered when source code was actually worked on. If parts of the source code have been stable for a long time, they are not suitable for a comparison with recently changed code, without accounting for general changes, e.g., in coding or testing practices, over time.

However, we did find that code with more PMD warnings leads to more bugs when changed. When looking into the differences between bug inducing file changes and all other file changes we find significant differences in 3 of 4 cases. While the effect size is negligible, in all cases using only the default rules yields a higher effect size. These rules were hand-picked by the Maven developers, arguably because of their importance for the internal quality. For practitioners, this finding is of particular importance: not only does it reduce the number of alerts to carefully select ASAT warnings from a large set of candidates, it also helps to reduce general issues that are associated with bugs.

This also has implications for researchers when including warning density based metrics into predictive models. Our data shows that the model might be improved by choosing an appropriate subset of the possible warnings of an ASAT. Using all warnings without considering their potential relation to defects is not a good strategy. Our data also shows that a good starting point might be a commonly used default, e.g., for PMD the maven-pmd-plugin default rules.

## 5 Threats to validity

In this section, we discuss the threats to validity we identified for our work. To structure this section we discuss four basic types of validity separately, as suggested by Wohlin et al. (2000).

### 5.1 Construct validity

A threat to the relation between theory and observation may occur in our study from the measurement of warning density. We restrict the data to production code to mitigate effects test code has on warning density as it is often much simpler than production code.

### 5.2 Internal validity

A general threat to internal validity would be a selection of static analysis warnings. We mitigate this by measuring the warning density for all warnings and for only default warnings as a common subset for Java projects. Due to the nature of our approach, we mitigate differences between projects regarding the handling of warnings as well as the impact of size.

### 5.3 External validity

Due to the usage of manually validated data in our study, our study subjects are restricted to those for which we have this kind of data. This is a threat to the generalizability of our findings, e.g., to all Java projects or to all open source projects. Still, as we argue in (Herbold et al., 2021), our data should be representative for mature Java open source projects.

Moreover, we observe only one static analysis tool (PMD). While this may also restrict the generalizability of our study, we believe that due to the large range of rules of this ASAT our results should generalize to ASATs that are broad in scope. ASATs of a different focus, e.g., on coding style (Checkstyle) or directly finding bugs (FindBugs, SpotBugs) may result in different results.

### 5.4 Conclusion validity

We report lower warning density for bug inducing files in comparison to the rest of the project at that point in time. While this reflects the difference in warning density between the file and the project, it can be influenced by constantly decreasing warning density. We mitigate this by also including a comparison between bug inducing changes and all other changes.

## 6 Conclusion

In this article we provide evidence for a common assumption in software engineering, i.e., that static analysis tools provide a net-benefit to software quality even though they suffer from problems with false positives. We use an improved state-of-the-art approach used for fine-grained just-in-time defect prediction to establish a link between files within commits that induce bugs and measure warning density related features which we aggregate over the evolution of our study subjects. This approach runs on data which allows us to remove several noise factors from our data, wrong issue types, wrong issue links to commits and tangled bug fixes. The analysis approach allows us to merge the available data as it mitigates differences between projects, sizes and to some extent the evolution of warnings over time.

We find that bugs are induced in files which have a comparably low warning density, i.e., less static analysis warnings than the files of the rest of the project at the time the bug was induced. However, this difference can be explained by the fact that the warning density decreases over time. When we compare the bug inducing changes with all other changes, we do find a significant higher warning density when using all PMD rules in one of two metrics. However, the effect size is negligible. When we use a small rule set that restricts the 314 PMD warnings to the 49 warnings hand-picked by the Maven developers as default warnings, we find that bug inducing changes have a significant but also negligible larger warning density. However, the effect size increases for the

default rule set. Assuming that the smaller rule set was crafted with the intent to single out the most important rules for the quality, this indicates that there is indeed a (weak) relationship between general ASAT tools and bugs.

This is also direct evidence for a common best practice in the use of static analysis tools: Appropriate rules for ASATs should be chosen for the project. This not only reduces the number of alarms, which is important for the acceptance by developers, but also has a better relationship with the external quality of the software measured through bugs.

## Declarations

This work was partly funded by the German Research Foundation (DFG) through the project DEFECTS, grant 402774445.

The authors have no competing interests to declare that are relevant to the content of this article.

## References

- Abdi H (2007) Bonferroni and Sidak corrections for multiple comparisons. In: Encyclopedia of Measurement and Statistics, Sage, Thousand Oaks, CA, pp 103–107
- Aloraini B, Nagappan M, German DM, Hayashi S, Higo Y (2019) An empirical study of security warnings from static application security testing tools. *Journal of Systems and Software* 158:110427, DOI 10.1016/j.jss.2019.110427, URL <http://www.sciencedirect.com/science/article/pii/S0164121219302018>
- Antoniol G, Ayari K, Di Penta M, Khomh F, Guéhéneuc YG (2008) Is it a bug or an enhancement? a text-based approach to classify change requests. In: Proceedings of the 2008 Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds, Association for Computing Machinery, New York, NY, USA, CASCON '08, DOI 10.1145/1463788.1463819
- Christakis M, Bird C (2016) What developers want and need from program analysis: An empirical study. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ACM, New York, NY, USA, ASE 2016, pp 332–343, DOI 10.1145/2970276.2970347, URL <http://doi.acm.org/10.1145/2970276.2970347>
- Cliff N (1993) Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological Bulletin*
- Devanbu P, Zimmermann T, Bird C (2016) Belief evidence in empirical software engineering. In: 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), pp 108–119, DOI 10.1145/2884781.2884812
- Fan Y, Xia X, Alencar da Costa D, Lo D, Hassan AE, Li S (2019) The impact of changes mislabeled by szz on just-in-time defect prediction. *IEEE Transactions on Software Engineering* pp 1–1, DOI 10.1109/TSE.2019.2929761

- Fenton N, Bieman J (2014) *Software Metrics: A Rigorous and Practical Approach*, Third Edition, 3rd edn. CRC Press, Inc., Boca Raton, FL, USA
- Habib A, Pradel M (2018) How many of all bugs do we find? a study of static bug detectors. In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ACM, New York, NY, USA, ASE 2018, pp 317–328, DOI 10.1145/3238147.3238213, URL <http://doi.acm.org/10.1145/3238147.3238213>
- Heckman S, Williams L (2009) A model building process for identifying actionable static analysis alerts. In: *2009 International Conference on Software Testing Verification and Validation*, pp 161–170, DOI 10.1109/ICST.2009.45
- Herbold S, Trautsch A, Trautsch F (2020) Issues with szz: An empirical assessment of the state of practice of defect prediction data collection, URL <http://arxiv.org/abs/1911.08938>, article was recently accepted at *Empirical Software Engineering*
- Herbold S, Trautsch A, Ledel B, Aghamohammadi A, Ghaleb TA, Chahal KK, Bossenmaier T, Nagaria B, Makedonski P, Ahmadabadi MN, Szabados K, Spieker H, Madeja M, Hoy N, Lenarduzzi V, Wang S, Rodríguez-Pérez G, Colomo-Palacios R, Verdecchia R, Singh P, Qin Y, Chakroborti D, Davis W, Walunj V, Wu H, Marcilio D, Alam O, Aldaej A, Amit I, Turhan B, Eismann S, Wickert AK, Malavolta I, Sulir M, Fard F, Henley AZ, Kourtzanidis S, Tuzun E, Treude C, Shamasbi SM, Pashchenko I, Wyrich M, Davis J, Serebrenik A, Albrecht E, Aktas EU, Strüber D, Erbel J (2021) Large-scale manual validation of bug fixing commits: A fine-grained analysis of tangling. URL <https://arxiv.org/abs/2011.06244>, article was recently accepted at *Empirical Software Engineering*, 2011.06244
- Herzig K, Just S, Zeller A (2013a) It’s not a bug, it’s a feature: How misclassification impacts bug prediction. In: *Proceedings of the International Conference on Software Engineering*, IEEE Press, Piscataway, NJ, USA, ICSE ’13, pp 392–401, URL <http://dl.acm.org/citation.cfm?id=2486788.2486840>
- Herzig K, Just S, Zeller A (2013b) It’s not a bug, it’s a feature: How misclassification impacts bug prediction. In: *Proceedings of the 2013 International Conference on Software Engineering*, IEEE Press, ICSE ’13, p 392–401
- Johnson B, Song Y, Murphy-Hill E, Bowdidge R (2013) Why don’t software developers use static analysis tools to find bugs? In: *Proceedings of the 2013 International Conference on Software Engineering*, IEEE Press, Piscataway, NJ, USA, ICSE ’13, pp 672–681, URL <http://dl.acm.org/citation.cfm?id=2486788.2486877>
- Just R, Jalali D, Ernst MD (2014) Defects4j: A database of existing faults to enable controlled testing studies for java programs. In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, Association for Computing Machinery, New York, NY, USA, ISSTA 2014, p 437–440, DOI 10.1145/2610384.2628055
- Kamei Y, Shihab E, Adams B, Hassan AE, Mockus A, Sinha A, Ubayashi N (2013) A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering* 39(6):757–773, DOI 10.1109/

TSE.2012.70

- Kim S, Ernst MD (2007) Which warnings should i fix first? In: Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, ACM, New York, NY, USA, ESEC-FSE '07, pp 45–54, DOI 10.1145/1287624.1287633
- Koc U, Saadatpanah P, Foster JS, Porter AA (2017) Learning a classifier for false positive error reports emitted by static code analysis tools. In: Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, pp 35–42, DOI 10.1145/3088525.3088675
- Lenarduzzi V, Lomio F, Huttunen H, Taibi D (2020) Are sonarqube rules inducing bugs? 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER) pp 501–511
- Mann HB, Whitney DR (1947) On a test of whether one of two random variables is stochastically larger than the other. *Annals of Mathematical Statistics* 18(1):50–60
- Nagappan N, Ball T (2005) Static analysis tools as early indicators of pre-release defect density. In: Proceedings of the 27th International Conference on Software Engineering, ACM, New York, NY, USA, ICSE '05, pp 580–586, DOI 10.1145/1062455.1062558, URL <http://doi.acm.org/10.1145/1062455.1062558>
- Pascarella L, Palomba F, Bacchelli A (2019) Fine-grained just-in-time defect prediction. *Journal of Systems and Software* 150:22 – 36, DOI 10.1016/j.jss.2018.12.001, URL <http://www.sciencedirect.com/science/article/pii/S0164121218302656>
- Penta MD, Cerulo L, Aversano L (2009) The life and death of statically detected vulnerabilities: An empirical study. *Information and Software Technology* 51(10):1469 – 1484, DOI 10.1016/j.infsof.2009.04.013, URL <http://www.sciencedirect.com/science/article/pii/S0950584909000500>, source Code Analysis and Manipulation, SCAM 2008
- Plosch R, Gruber H, Hentschel A, Pomberger G, Schiffer S (2008) On the relation between external software quality and static code analysis. In: 2008 32nd Annual IEEE Software Engineering Workshop, pp 169–174, DOI 10.1109/SEW.2008.17
- Querel L, Rigby PC (2021) Warning-introducing commits vs bug-introducing commits: A tool, statistical models, and a preliminary user study. In: 29th IEEE/ACM International Conference on Program Comprehension, ICPC 2021, Madrid, Spain, May 20–21, 2021, IEEE, pp 433–443, DOI 10.1109/ICPC52881.2021.00051
- Querel LP, Rigby PC (2018) Warningsguru: Integrating statistical bug models with static analysis to provide timely and specific bug warnings. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Association for Computing Machinery, New York, NY, USA, ESEC/FSE 2018, p 892–895, DOI 10.1145/3236024.3264599

- Rahman F, Khatri S, Barr ET, Devanbu P (2014) Comparing static bug finders and statistical prediction. In: Proceedings of the 36th International Conference on Software Engineering, ACM, New York, NY, USA, ICSE 2014, pp 424–434, DOI 10.1145/2568225.2568269, URL <http://doi.acm.org/10.1145/2568225.2568269>
- Rodriguez-Pérez G, Nagappan M, Robles G (2020) Watch out for extrinsic bugs! a case study of their impact in just-in-time bug prediction models on the openstack project. *IEEE Transactions on Software Engineering* pp 1–1, DOI 10.1109/TSE.2020.3021380
- Rosa G, Pascarella L, Scalabrino S, Tufano R, Bavota G, Lanza M, Oliveto R (2021) Evaluating SZZ implementations through a developer-informed oracle. In: 43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021, pp 436–447, DOI 10.1109/ICSE43902.2021.00049
- Rosen C, Grawi B, Shihab E (2015) Commit guru: Analytics and risk prediction of software commits. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, Association for Computing Machinery, New York, NY, USA, ESEC/FSE 2015, p 966–969, DOI 10.1145/2786805.2803183
- Śliwinski J, Zimmermann T, Zeller A (2005) When do changes induce fixes? *SIGSOFT Softw Eng Notes* 30(4):1–5, DOI 10.1145/1082983.1083147
- Spadini D, Aniche M, Bacchelli A (2018) PyDriller: Python framework for mining software repositories. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018, ACM Press, New York, New York, USA, pp 908–911, DOI 10.1145/3236024.3264598, URL <http://dl.acm.org/citation.cfm?doid=3236024.3264598>
- Thung F, Lucia, Lo D, Jiang L, Rahman F, Devanbu PT (2012) To what extent could we detect field defects? an empirical study of false negatives in static bug finding tools. In: 2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, pp 50–59, DOI 10.1145/2351676.2351685
- Trautsch A, Herbold S, Grabowski J (2020a) A Longitudinal Study of Static Analysis Warning Evolution and the Effects of PMD on Software Quality in Apache Open Source Projects. *Empirical Software Engineering* DOI 10.1007/s10664-020-09880-1
- Trautsch A, Herbold S, Grabowski J (2020b) Static source code metrics and static analysis warnings for fine-grained just-in-time defect prediction. In: 36th International Conference on Software Maintenance and Evolution (IC-SME 2020)
- Trautsch F, Herbold S, Makedonski P, Grabowski J (2017) Addressing problems with replicability and validity of repository mining studies through a smart data platform. *Empirical Software Engineering* DOI 10.1007/s10664-017-9537-x
- Tufano M, Palomba F, Bavota G, Penta MD, Oliveto R, Lucia AD, Poshyvanyk D (2017) There and back again: Can you compile that snapshot? *Journal of*

- Software: Evolution and Process 29(4), URL <http://dblp.uni-trier.de/db/journals/smr/smr29.html#TufanoPBP0LP17>
- Vassallo C, Panichella S, Palomba F, Proksch S, Gall HC, Zaidman A (2020) How developers engage with static analysis tools in different contexts. *Empirical Software Engineering* 25, DOI 10.1007/s10664-019-09750-5
- Vetro A, Morisio M, Torchiano M (2011) An empirical validation of find-bugs issues related to defects. In: 15th Annual Conference on Evaluation Assessment in Software Engineering (EASE 2011), pp 144–153, DOI 10.1049/ic.2011.0018
- Wohlin C, Runeson P, Höst M, Ohlsson MC, Regnell B, Wesslén A (2000) *Experimentation in Software Engineering: An Introduction*. Kluwer Academic Publishers, Norwell, MA, USA