Augmented Fine-Grained Defect Prediction for Code Review

Pascarella, L.

DOI
10.4233/uuid:e553e8ae-73be-4718-ab93-81f466db7347

Publication date
2020

Augmented fine-grained defect prediction for code review

Augmented fine-grained defect prediction for code review

Dissertation

for the purpose of obtaining the degree of doctor
at Delft University ofTechnology
by the authority of the Rector Magnificus prof. dr. ir. T.H.J.J. van der Hagen
chair of the Board for Doctorates
to be defended publicly on
Wednesday 2 September 2020 at 10:00 o'clock

by

Luca PASCARELLA

Master of Science in Software Engineering,
University of Sannio, Italy,
born in Benevento, Italy.

This dissertation has been approved by the promotors.

Composition of the doctoral committee:

Rector Magnificus,	chairperson
Prof. dr. A. van Deursen,	Delft University of Technology, promotor
Prof. dr. A. Bacchelli,	University of Zurich, Switzerland, promotor
Independent members:	
Prof. dr. A. Gorla,	IMDEA Software Institute, Spain
Prof. dr. D. A. Tamburri,	Eindhoven University, The Netherlands
Prof. dr. M. Nagappan,	University of Waterloo, Canada
Prof. dr. ir. D. M. van Solingen,	Delft University of Technology, The Netherlands
Prof. dr. ir. D. H. J. Epema,	Delft University of Technology, The Netherlands

The work in the thesis has been carried out under the support of the European Union's H2020 program the Marie Sklodowska-Curie grant agreement No 642954.

Keywords: Code review, defect prediction, software analytics

Printed by:

Cover:

Style: TU Delft House Style, with modifications by Moritz Beller
`https://github.com/Inventitech/phd-thesis-template`

The author set this thesis in LaTeX using the Libertinus and Inconsolata fonts.

An electronic version of this dissertation is available at
`http://repository.tudelft.nl/`.

Dedicated to Mariaclaudia who supported me during this journey

Luca Pascarella

Contents

Summary

Code review is a widely used technique to support software quality. It is a manual activity, often subject to repetitive and tedious tasks that increase the mental load of reviewers and compromise their effectiveness. The developer-centered nature of code review can represent a bottleneck that does not scale in large systems with the consequence of compromising firms' profits. This challenge has led to an entire line of research on code review improvement.

In this thesis, we present our results and remarks on the effectiveness of using fine-grained defect prediction in code review while investigating what are the information needs that lead a proper code review. We started reimplementing the state of the art of defect prediction to understand its replicability; then, we evaluated this model in a more realistic scenario that is typically considered. To improve defect prediction techniques, we come up with a fine-grained just-in-time defect prediction model that anticipates the prediction at commit time and reduces the granularity at the file level. After that, we explored how to improve further prediction performance by using alternative sources of information. We conducted a comprehensive investigation of code comments written by both open and closed source developers. Finally, to understand how to improve code review further, we explored from a reviewers' perspective what is the information that reviewers need to lead a proper code review.

Our findings show that the state of the art of defect prediction, when evaluated in a realistic scenario, cannot be directly used to support code review. Furthermore, we assessed that alternative sets of metrics, anticipated feedback, and fine-grained suggestions represent independent directions to improve prediction performance. Finally, we discovered that research must create intelligent tools that other than predict defects must satisfy actual reviewers' needs, such as expert selection, splittable changes, realtime communication, and self summarization of changes.

Samenvatting

Code reviews zijn een veelgebruikte techniek om de kwaliteit van software te meten. Het is een handmatige, tijdrovende en repetitieve taak die de mentale belasting van reviewers vergroot en hun effectiviteit nadelig beïnvloedt. De arbeidsintensieve aard van code reviews kan een knelpunt vormen dat niet schaalbaar is in grote systemen met als gevolg dat de winst van bedrijven in gevaar komt. Deze uitdaging heeft geleid tot een zelfstandig onderzoeksveld naar het verbeteren van code reviews.

1

Introduction

Software firms rely on software engineering to coordinate collaborating teams of specialists who build large software systems. To help companies safeguard their long term investments in software, researchers are investigating ways to address the inevitable obsolescence by developing techniques aimed at increasing software quality.

Code review is a technique widely used in practice and designed to support software quality. In a code review, authors and reviewers of large teams alternatively produce and inspect source code by exchanging knowledge, suggesting tips, eliminating defects, and encouraging excellence. Code review is a manual activity, often subject to repetitive and tedious tasks that increase the mental load of reviewers, compromising their effectiveness. Researchers propose alternative techniques to improve the code review process, particularly investigating how to assist developers and reduce the cognitive effort required. One solution points to support code review with smart assistants that aims at maximizing the effectiveness of reviewers without increasing the cost of their review. In this regard, a promising practice is predictive analytics, which aims at automatically predicting the areas of source code that are more likely to be problematic, thus guiding the reviewer's attention.

In this thesis, we focus on predictive analytics for code review. We evaluate the state of the art of defect prediction models in the context of code review. To improve performance, we propose two options aimed at improving the effectiveness as well as the prediction granularity. We also assess the effectiveness of using alternative software metrics as measurable properties of software in the machine learning models. Finally, we study how reviewers' needs can be addressed to further support the code review process.

1

The last decade has seen a remarkable involvement of software in our daily life [1]. Recently, software systems are growing fast by introducing new and complex functionalities to react to the frenzied demands of the market [2]. A leading strategy to address this issue pushes the increasing software complexity demand to ever-larger teams of software developers [3]. However, large and spread development teams need synchronization, synergy, and coordination to perform smoothly [4] because lack of communication, fast-paced evolution, and poorly tested changes may lead to a degradation in the maintainability of systems [5], with potentially dangerous consequences [6]. In this context, code review comes to aid software quality while increasing team collaborations [7].

Code review refers to a software engineering activity aimed at maintaining and supporting source code quality [8, 9]. In 1976, Fagan defined a formal process for performing code inspections [10] intending to catch software defects[1] [11, 12]. Fagan's inspection is performed in group meetings, and with a clear goal to catch as many defects as possible. Instead, contemporary code review, often referred to as *Modern Code Review* (MCR) [7, 13], is a lightweight variant that is (1) informal, (2) tool-based, (3) asynchronous, and (4) focused on inspecting new committed code changes rather than the whole software system [14]. While code review aims at explicitly revealing as many defects as possible, it has been reported to also implicitly share knowledge, explore alternative solutions, and reinforce team awareness [7].

In modern code review, practitioners (i.e., authors and reviewers) employ online tools to perform their tasks [15]. These online tools have the purpose of facilitating the code review practices through the use of an asynchronous communication interface [16]. Different vendors developed a variety of code review tools to lead the code review process through well-finished user interfaces [17, 18]; nonetheless, almost any vendor uses the same technological approach that implements only the basic logistics of modern code review [19] and does not offer the advantages brought by smart tools [20].

Given the lack of advanced code review tools [17, 18] and the repetitive, challenging, and time-consuming nature of code review tasks that affect the reviewers' performance [13, 21], many researchers are performing investigations aimed at supporting code review quality [16, 21, 22]. For instance, researchers aim at supporting reviewers with code review tools that can assist developers in locating defects through predictive analytics [23]. A promising defect prediction model that reduces the mental-load of reviewers by automatically catching software defects [24].

1.1 Improving Code Review With Defect Prediction

Cognitive load is a multidimensional construct that represents the load requested to the human cognitive system to perform a certain task [25]. Code review tools improve reviewers' performance by lowering the cognitive load [21]. The possibility to navigate changes, visualize differences, or add comments help reviewers in focusing on code review tasks instead of spending their cognitive resource in trivial tasks. However, elements related to

[1] A defect or informally a bug identifies a condition in which a software does not meet software requirements or end-user expectations. In other words, a defect is an error in coding that causes a software to malfunction or to produce incorrect and unexpected results. More precisely, the ISO 10303-226 refers to a software fault as an abnormal condition or defect at the component, equipment, or sub-system level, which may lead to a failure.

the invested mental load and the working memory capacity affect the reviewers' cognitive load during a code review especially in situation of overload.

Over the years, researchers have proposed and investigated many solutions to reduce the mental load during the review based on re-ordering of code changes (e.g., by relatedness [21] or by test cases [26]), reducing the size of changes [27], promptly suggesting domain experts [28], or predicting defective parts [24]. Among all those techniques, defect prediction tackles the problem at its source by estimating the defectiveness of sources inspected by developers [29].

The promises of defect prediction attracted the interest of large companies, which started to experiment with augmented code review tools [24, 30]. For example, FixCache [31], a well known academic defect prediction model, has been evaluated by Google developers in a typical working environment to support code review [24]. FixCache uses a recent developed concept of defect *locality* resulting in excellent effectiveness with *in vitro* experiments (i.e., experiments conducted in a controlled environment that reduces interactions with external factors). However, when applied to an *in vivo* setting, Google's developers found that recommendations received were too imprecise to have practical effectiveness in supporting code review. This evidence calls for further research on this topic.

1.2 Defect Prediction

Predicting the snippets of source code that contain a defect is crucial in software engineering to support software quality, plan maintenance, and allocate resources.

Defect prediction attracts a lot of interest in research, especially in the last decade [32]. The first approach dates back to 1970 with the definition of Akiyama's linear equation that relates the number of source lines with bugs [33]. While Akiyama's law represents a starting point to count the number of defects in the code, it fails in localizing defects in actual software systems [34].

Over the years, researchers have introduced and evaluated a variety of defect prediction models based on the evolution [35] (e.g., number of changes), the anatomy [35] (e.g., lines of code, complexity), and the socio-technicals aspects (e.g., contribution organization) of software projects and artifacts [35]. These models have been evaluated individually or heterogeneously combining different projects [36–38].

To have a broad vision, researchers investigated what factors make software prone to be defective [39–49] proposing several unsupervised [50–52] and supervised [53–57] defect prediction models. Besides exploring different prediction granularity (e.g., package, class, or method level), researchers explored different prediction time moving from a coarse-grained release-by-release approach to a fine-grained *just-in-time* concept that localizes defect at commit time[58–64].

In this dissertation, we want to investigate whether and how defect prediction models are a feasible solution to support code review.

1.3 Research Goal and Questions

We report the goal of our thesis in form of a research statement, followed by four high-level research questions that guide and validate it.

1

This thesis is concerned with understanding how to adapt predictive analytics for code review and how reviewers' needs can be addressed to support the code review process further.

1.3.1 Research Questions

Defect prediction models are the basis for building smart assistants that aim to help developers in producing high-quality software. Therefore, the behavior of these models, when paired with code review tools, can impact software quality and practitioners' life. Hence, a synergy between defect prediction models and code review tools has the potential to speed up the development process, support software quality, and pay a better investment turnover. With our first research question, we would like to understand whether the state of the art of defect prediction models can be used realistically to support code review.

RQ$_1$. *Are current defect prediction models a feasible solution for supporting code review?*

Considering the limitations discovered in the state of the art of defect prediction models, with our second research question, we would like to understand how much effort a fine-grained just-in-time defect prediction model can save when used on a daily basis. These findings aid our primary goal of understanding whether a fine-tuned prediction model can fulfill the goal of maximizing the code review outcome.

RQ$_2$. *To what extent do fine-grain defect prediction models improve prediction performance?*

Although product and process metrics have been considered the most widely used features for defect prediction purposes, researchers have also considered the use of alternative metrics showing how a mix of sets of different parameters outperforms a single group. Whereas developing source code requires knowledge, experience, and creativity, producing high-quality software demands an extra effort. Due to the complex nature of software development, developers add code comments to their artifacts. These code comments represent an additional source of information for spreading knowledge, reporting rational or altering software behavior. Since code comments are generically bypassed by compilers and pointed only by humans, we aim at evaluating the use of code comments as an alternative source of information in defect prediction models.

With our third research question, we investigate whether the state of the art of defect prediction models in a release-by-release evaluation strategy trained with alternative sets of metrics (e.g., those derived by analyzing code comments, the presence of code smells, and developer-related factors) can improve the prediction performance by catching more software defects. This knowledge fulfills our primary goal that aims to optimize the code review outcome by employing high-quality defect prediction models.

RQ$_3$. *How can alternative features improve defect prediction performance?*

Not all information conveyed during a code review has the same significance. We like to understand the kinds of information exchanged by human inspectors during a code review. This will allow us to understand the review process better; in particular, it will ascertain what kinds of information developers need to conduct a proper code review, how often this information is given or sought, what effect produces lacking or misleading information, and how those information needs evolve during an inspection. This in-depth investigation helps us develop a broad vision of what is the role of the information needs during a code review and how to use such needs to improve code review further.

RQ$_4$. *What are reviewers' information needs and how does defect prediction fulfill them?*

After answering the four research questions mentioned above, we will have set the ground in establishing to what extent code review can benefit from defect prediction models and whether reviewers' needs can be addressed to support code review further. This sheds light on our goal and provides hints on how to design intelligent assistant tools that will aid both authors and reviewers during a code review. These tools can be used to increase the overall software quality while keeping development costs contained. However, this study leaves room for further investigations that, for example, may focus on how authors and reviewers perceive suggestions coming from automated tools. Future research may clarify developers' expectations pointing to the design of better tools that while producing useful indications to prevent defects reduce the developers' distraction choosing when and how to show in summary of each suggestion. We hope that future researchers can spread additional evidence on this direction.

1.4 Research Methodology

In this section, we describe the research methods used in this thesis.

The studies in this dissertation are conducted within the paradigm of Empirical Software Engineering that traces back to the 1970s [65]. As stated by Boehm et al., in addition to the need to engineer software was the need to understand software. Much like other science, such as physics, chemistry, and biology, software engineering needed the discipline of observation, theory formation, experimentation, and feedback [66]. In this dissertation, we relied on a mixed-method approach [67], where we combine the empirical observations on data that come from mined repositories with data collected from interviews with expert developers.

1.4.1 Mining software repositories

Empirical Software Engineering often involves the analysis of data gathered by mining software repositories [68]. The wide adoption of version control systems in both open- and closed-source projects offers to the possibility of retrieving past data for in-depth analysis [69]. Platforms such as GitHub and GitLab allow practitioners and researchers to explore the development process of several projects.

In this thesis, we rely on various mining software repositories techniques—by building *ad-hoc* tools—to collect data from both open and closed software repositories. The analysis of such data helped us to design and train defect prediction models that use the historical

1

data to predict the part of the software that can be defective in the future. In this case, we target a broad set of open-source projects to collect data regarding years of software development. Besides the possibility of training a machine learning model with historical data, an additional advantage of analyzing public repositories is that we have access to fine-grained commit information, thus allowing us to understand the rationale behind each choice of given projects. Chapters 2, 3, and 5 outline the exact tools developed to mine software repositories and extract the data that is used to train machine learning models. Furthermore, Chapter 4 uses a similar mining technique to extract code comments from both open-source and industrial software systems. Finally, to understand what are the information needed by reviewers to lead a proper code review, Chapter 6 describes how we adapted the techniques regarding mining software repositories to extract valuable information from repositories of code reviews.

1.4.2 Interviews and focus groups with developers

To complement data gathered with different mining techniques we conducted both interviews and focus groups[70, 71]. The combination of these two techniques allows us to clarify the rationale behind a choice that may not be documented through explicit written messages [72]

- **Interview.** This format is often used in exploratory investigations to understand phenomena and seek new insights [68]. The main goal of conducting interviews with domain experts helped us in improving our knowledge of actual issues while obtaining new insights that come from a different perspective. However, engaging open-source developers is not trivial because researchers often target them. Thus we complemented our study by involving experts working in a leading company of software quality. In the latter case, we adopted a different interview technique that, while stimulating in-depth discussions, optimizes invested time.

- **Focus group.** This format is particularly useful when a small number of people is available for discussing about a certain problem [70, 71]. It consists in the organization of a meeting that involves the participants and a moderator. We used this technique to enhance the interviews' results conducted with experts of open-source domain with experts working in a leading company of software quality assurance.

Nonetheless, these techniques may carry a drawback. When interviewed, people may provide socially desirable responses that may differ from reality. For mitigating these threats and have a broad vision of different contexts, we invited diverse sets of developers from both open-source and industrial systems. In doing this, we interviewed as many developers as available until we reached saturation [73] i.e., when we recorded multiple times the same responses without covering any new aspects. Successively, we relied on a lightweight technique derived from the *grounded theory* [74] that is a systematic methodology used in the social sciences that involves the construction of theories through systematic observations of data gathered from different perspectives. Recently, this model gained popularity also in computer science [75]. In particular, we adopted an open card sorting process to collect different themes in an iterative process that splits and merges topics systematically [76]. Through this inductive approach, we split interviews' transcripts

into small parts and successively assigned a label that summarizes the content of the transcripted part. Then, we merged common topics and inferred the emergent themes from all the interviews. In Chapter 6, we complemented the data coming from the mining of publicly available code reviews with topics emerged during the interviews of expert developers.

1.5 Research outline

This thesis follows a portfolio structure that comprises stand-alone scientific manuscripts. To fit the flow and goal of this dissertation, we adjusted submitted articles and, in some cases, merged them to build a single cohesive chapter. Table 1.1 summarizes the connection between chapters and research questions. Chapters follow a logical contribution order instead of a chronological one. Every article is publicly accessible as green open access and published with permanent links.

Table 1.1: Relation between research questions and chapters.

Research question	Chapters
Are current defect prediction models a feasible solution for supporting code review?	2
To what extent do fine-grain defect prediction models improve prediction performance?	3
How can additional features improve defect prediction performance?	4, 5
How can reviewers' information needs support code review tools further?	6

1.5.1 The state of the art of defect prediction models

In the first part of the thesis, we explore the state of the art of defect prediction to evaluate whether those models are a feasible solution in real working environments.

- **Chapter 2** outlines the strategy and the technique adopted to answer the first research question. In this study, we first replicate previous research on method-level defect prediction on different systems and timespans, and successively, we contemplate an evaluation strategy that approximates a real world context. Our initial results show that the performance of the method-level defect prediction model based on a mixture of product and process metrics is similar to what previously reported also for different systems/timespans, when evaluated with the same strategy. However—when evaluated with a more realistic strategy—all the models show a dramatic drop in performance, with results close to that of a random classifier. We reflect on the evaluation strategy and propose a more realistic one that better fits the context of code review.

1.5.2 Fine-grained just-in-time defect prediction

In the second part of the thesis we analyze the scale at which the prediction granularity is an actual limitation and how the change of the prediction granularity impacts on the effort required for inspecting codes likely affected by defects.

- **Chapter 3** investigates how much effort a fine-grained just-in-time defect prediction model can save with respect to a standard just-in-time model when used to catch

1

defects. To address this, we first investigate to what extent commits are partially defective—a commit can contain many changed files where only a few of them can be defective. Then, we propose a novel fine-grained just-in-time defect prediction model to predict the specific files, contained in a commit, that are defective. Finally, we evaluate our model in terms of (i) performance and (ii) the extent to which it decreases the effort required to diagnose a defect. Our study highlights that: (1) defective commits are frequently composed of a mixture of defective and non-defective files, (2) our fine-grained model can accurately predict defective files with an overall AUC-ROC up to 86% and (3) our model would allow practitioners to save inspection efforts with respect to standard just-in-time techniques. Although initial results are promising, defect prediction performance is still not perfect for supporting code review.

1.5.3 Additional code metrics for defect prediction

In the third part of the thesis, we investigate whether defect prediction models can be improved further by using additional sets of features to support code review.

- **Chapter 4** merges the three studies we performed on large scale data mined from both open and closed source Java projects. In the first study we investigate how six diverse Java OSS projects use code comments, with the aim of understanding their purpose. Through this analysis, we produce a taxonomy of source code comments; subsequently, we investigate how often each category occur by manually classifying more than 2,000 code comments from the aforementioned projects. In addition, we conduct an initial evaluation on how to automatically classify code comments at line level into our taxonomy using machine learning; initial results were promising and suggested that an accurate classification is within reach. In the second work we analyze how code comments impact on code readability and maintainability. In particular, we investigate how developers of five open-source mobile applications use code comments to document their projects. Additionally, we evaluate the performance of two machine learning models to automatically classify code comments. Initial results show marginal differences between desktop and mobile applications. Finally, in the third work we generalize these findings with an industrial study where we investigate how often each category occur by manually classifying more than 40,000 lines of code comments from the aforementioned projects. In addition, we investigate how to automatically classify code comments at line level into our taxonomy using machine learning; initial results are promising and suggest that an accurate classification is also within reach, even when training the machine learner on projects different than the target one.

- **Chapter 5** aims at improving a fine-grained defect prediction model to identify software artifacts that are more likely to be defective in future releases. We build this investigation by extending the preliminary work conducted in the paper presented in our second chapter in which we first replicate previous research on the state of the art of method-level defect-prediction, by using different systems and timespans. Afterwards, based on the limitations of existing research, we (1) re-evaluate method-level defect prediction models more realistically and (2) analyze whether alternative

features based on textual aspects, code smells, and developer-related factors can be exploited to improve method-level defect prediction. Key results of our study include that (1) the performance of the previously proposed models, tested using the same strategy but on different systems/timespans, is confirmed; but, (2) when evaluated with a more practical strategy, all the models show a dramatic drop in performance, with results close to that of a random classifier. Finally, we find that (3) the contribution of alternative features within such models is limited and unable to improve the prediction capabilities significantly. As a consequence, our replication and negative results indicate that method-level defect prediction is still an open challenge.

1.5.4 Reviewers' information needs in code review

In the last part of this thesis, we investigate the information that reviewers need to conduct a proper code review and how tool support can make reviewers more effective and efficient.

- **Chapter 6** investigates the information that reviewers need to conduct a proper code review. Previous work has provided evidence that a successful code review process is one in which reviewers and authors actively participate and collaborate. In these cases, the threads of discussions that are saved by code review tools are a precious source of information that can be later exploited for research and practice. In this work, we focus on this source of information as a way to gather reliable data on the aforementioned reviewers' information needs. We manually analyze 900 code review comments from three large open-source projects and organize them in categories by means of a card sort. Our results highlight the presence of seven high-level information needs, such as knowing the uses of methods and variables declared/modified in the code under review. Based on these results, we suggest alternative ways in which novel code review tools can be implemented to support collaborations and the reviewing task better besides supporting the localization of defects in sources.

1.5.5 Reflection

We conclude the thesis by summarizing our findings, formulating answers to the research questions, and proposing a detailed agenda of future work.

- **Chapter 7** summarizes the findings and answers the research questions of this thesis. Moreover, we discuss the implications of this thesis and elaborate on future work that can be conducted in future research.

1.6 Origin of the chapters

All chapters of this thesis have been published in the proceeding of peer-reviewed conferences or journals. As a result, each chapter is self-contained with its background, related work, and implications. Unless otherwise specified, the first author is the main contributor to each work. In the following a detailed origin of the chapters:

- *Chapter 2* was published in the paper "Re-evaluating Method-Level Bug Prediction" by Pascarella, Palomba, and Bacchelli at the International Conference on Software Analysis (SANER) 2018 reproducibility studies and negative results (RENE) track.
- *Chapter 3* was published in the paper "Fine-Grained Just-In-Time Defect Prediction" by Pascarella, Palomba, and Bacchelli in the Journal of Systems and Software (JSS) 2018. To respect the intent expressed in Section 1.8 about the study replicability, we re-implement the entire research pipeline and report the results accordingly.
- *Chapter 4* aggregates three contributions. The first contribution was published in the paper "Classifying code comments in Java open-source software systems" by Pascarella and Bacchelli at the International Conference on Mining Software Repositories (MSR) 2017. The second contribution was published in the paper "Classifying code comments in Java Mobile Applications" by Pascarella at the International Conference on Mobile Software Engineering and Systems (MOBILESoft) 2018 in the Student Research Competition track, and the last contribution was published in the paper "Classifying code comments in Java software systems" by Pascarella, Bruntink, and Bacchelli in Empirical Software Engineering (EMSE) 2019.
- *Chapter 5* extends the Chapter 2 with an novel analysis. The new contribution was published in the paper "On the Performance of Method-Level Bug Prediction: A Negative Result" by Pascarella, Palomba, and Bacchelli in the Journal of Systems and Software (JSS) 2020.
- *Chapter 6* was published in the paper "Information Needs in Contemporary Code Review" by Pascarella, Spadini, Palomba, Bruntink, and Bacchelli at the Conference on Computer-Supported Cooperative Work and Social Computing (CSCW) 2018. For this work, the first two authors contributed equally.

To ensure full replicability of the results of this thesis, we re-executed the entire pipeline of scripts for every publication. Since it was not possible to retrieve the source code for the work in Chapter 3, we re-implemented the whole research with more modern tools and recomputed the results. In Chapter 3, we report the new achievements.

1.7 Other Contributions

Beside the publications included in this dissertation, we co-authored a number of manuscripts reported in brief in the following.

- The MaLTeSQuE'18 paper "Investigating Type Declaration Mismatches in Python" [77] contains an empirical study aimed at understanding the role of code comments such as source of information to detect type declaration mismatches.
- The MSR'18 paper "A Graph-based Dataset of Commit History of Real-World Android apps" [78]
- The MSR'18 paper "How Is Video Game Development Different from Software Development in Open Source?" [79]

- The WAMA'19 paper "Healthcare Android Apps: A Tale of the Customers' Perspective" [80]

1.8 Study Replicability and Open Science

Table 1.2: Data storage locations.

Dataset	Chapter	Host
Method-Level Bug Prediction	2, 5	Zenodo [81]
Fine-grained just-in-time defect prediction	3	Zenodo [82]
Code comments for defect prediction	4	Zenodo [83]
Reviewers' information needs in code review	6	Zenodo [84]

An essential aspect of scientific research is that works must be replicable, which means that every manuscript must give detailed information on how the study can be repeated or 'replicated'. Nonetheless, industrial studies and embargoed publications make replicability not easy. In that regard, the open science movement [85] aims to turn scientific research publicly available. To promote open-science, in the Netherlands, the Dutch Funding Agency NWO requires that every scientific research conducted with the support of public funds must be publicly accessible [86]. To this aim, *pure.tudelft.nl* hosts open accessible records that respect the Green Open Access policies of Delft University of Technology[2]. This also is in line with the European Commission that requires the open accessibility of every work funded by Horizon 2020 projects.

Tools and source codes used to collect the data in the various chapters together with the instructions to execute such tools have been made publicly available. When data do not come from industrial environments, which rules do not allow us to share collected data, we also provide all the original data collected. Table 1.2 depicts where various datasets can be collected for science replicability.

[2]TU Delft Policy on Open Access Publishing - Effective from 01-05-2016

2

Re-evaluating Method-Level Bug Prediction

Bug prediction is aimed at supporting developers in the identification of code artifacts more likely to be defective. Researchers have proposed prediction models to identify bug prone methods and provided promising evidence that it is possible to operate at this level of granularity. Particularly, models based on a mixture of product and process metrics, used as independent variables, led to the best results.

In this study, we first replicate previous research on method-level bug prediction on different systems/timespans. Afterwards, we reflect on the evaluation strategy and propose a more realistic one. Key results of our study show that the performance of the method-level bug prediction model is similar to what previously reported also for different systems/timespans, when evaluated with the same strategy. However—when evaluated with a more realistic strategy—all the models show a dramatic drop in performance exhibiting results close to that of a random classifier. Our replication and negative results indicate that method-level bug prediction is still an open challenge.

This chapter is based on

L. Pascarella, F. Palomba, A. Bacchelli. Re-evaluating Method-Level Bug Prediction, SANER'18 [87]

2.1 Introduction

The last decade has seen a remarkable involvement of software artifacts in our daily life [1]. Reacting to the frenzied demands of the market, most software systems nowadays grow fast introducing new and complex functionalities [2]. While having more capabilities in a software system can bring important benefits, there is the risk that this fast-paced evolution leads to a degradation in the maintainability of the system [5], with potentially dangerous consequences [6].

Maintaining an evolving software structure becomes more complex over time [88]. Since time and manpower are typically limited, software projects must strategically manage their resources to deal with this increasing complexity. To assist this problem, researchers have been conducting several studies on how to advise and optimize the limited project resources. One broadly investigated idea, known as *bug prediction* [32], consists in determining non-trivial areas of systems subjected to a higher quantity of bugs, to assign them more resources.

Researchers have introduced and evaluated a variety of bug prediction models based on the evolution [89] (e.g., number of changes), the anatomy [90] (e.g., lines of code, complexity), and the socio-technicals aspects (e.g., contribution organization) of software projects and artifacts [91]. These models have been evaluated individually or heterogeneously combining different projects [36–38].

Even though several proposed approaches achieved remarkable prediction performance [92], the practical relevance of bug prediction research has been largely criticized as not capable of addressing a real developer's need [24, 93, 94]. One of the criticisms regards the *granularity* at which bugs are found; in fact, most of the presented models predict bugs at a coarse-grained level, such as modules or files. This granularity is deemed not informative enough for practitioners, because files and modules can be arbitrarily large, thus requiring a significant amount of files to be examined [35]. In addition, considering that large classes tend to be more bug-prone [44, 46], the effort required to identify the defective part is even more substantial [90, 95, 96].

Menzies et al. [97] and Tosun et al. [98] introduced the first investigations exploring a finer granularity: function-level. Successively, Giger et al. [35] and Hata et al. [99] delved into finer granularity investigating the method-level bug prediction. Giger et al. found that product and process metrics contribute to the identification of buggy methods and their combination achieves promising performance [35]. Hata et al. found that method-level bug prediction saves more effort than both file-level and package-level prediction [99].

In this work, we replicate the investigations on bug prediction at method-level, focusing on the study by Giger et al. [35]. We use the same features and classifiers as the reference work, but on a different dataset to test the generalizability of their findings. Then we reflect on the evaluation strategy and propose a more realistic one. That is, instead of both taking change history and predicted bugs from the same time frame and of using cross-validation, we estimate the performance using data from subsequent releases (as done by the most recent studies, but at a coarser granularity [100]).

Our results—computed on different systems/timeframes than the reference work—corroborate the generalizability of the performance of the proposed method-level models, when estimated using the previous evaluation strategy. However, when evaluated with a release-by-release strategy, all the estimated models present lower performance, close to

that of a random classifier. As a consequence, even though we could replicate the reference work, we found that its realistic evaluation leads to negative results. This suggests that method-level bug prediction is still not a solved problem and the research community has the chance to devote more effort in devising more effective models that better assist software engineers in practice.

2.2 Background and Related Work

Bug prediction has been extensively studied by our research community in the last decade [32]. Researchers have investigated what makes source code more bug-prone (e.g., [39–49]), and have proposed several unsupervised (e.g., [50–52]) as well as supervised (e.g., [53–57]) bug prediction techniques. More recently, researchers have started investigating the concept of *just-in-time* bug prediction, which has been proposed with the aim of providing developers with recommendations at commit-level (e.g., [58–64]).

Our current paper focuses on investigating how well supervised approaches can identify bug-prone methods. For this reason, we firstly describe related work on predicting bug-prone classes, then we detail the earlier work on predicting bug-prone methods and how our work investigates its limitations and re-evaluates it.

2.2.1 Class-level Bug Prediction

The approaches in this category differ from each other mainly for the underlying prediction algorithm and for the considered features, i.e., *product metrics* (e.g., lines of code) and/or *process metrics* (e.g., number of changes to a class).

Product metrics. Basili et al. [90] found that five of the CK metrics [101] can help determining buggy classes and that Coupling Between Objects (CBO) is that mostly related to bugs. These results were later re-confirmed [95, 102, 103].

Ohisson et al. [104] focused on design metrics (e.g., 'number of nodes') to identify bug-prone modules, revealing the applicability of such metrics for the identification of buggy modules. Nagappan and Ball [105] exploited two static analysis tools to predict the pre-release bug density for Windows Server, showing good performance. Nagappan et al. [106] experimented with code metrics for predicting buggy components across five Microsoft projects, finding that there is no single universally best metric. Zimmerman et al. [57] investigated complexity metrics for bug prediction reporting a positive correlation between code complexity and bugs. Finally, Nikora et al. [107] showed that measurements of a system's structural evolution (e.g., 'number of executable statements') can serve as bug predictors.

Process metrics. Graves et al. [108] experimented both product and process metrics for bug prediction, finding that product metrics are poor predictors of bugs. Khoshgoftaar et al. [109] assessed the role of debug churns (i.e., the number of lines of code changed to fix bugs) in an empirical study, showing that modules having a large number of debug churns are likely to be defective.

To further investigate the role played by product and process metrics, Moser et al. [110, 111] performed two comparative studies, which highlighted the superiority of process metrics in predicting buggy code components. Later on, D'Ambros et al. [112] performed an extensive comparison of bug prediction approaches relying on both the sources of

information, finding that no technique works better in all contexts. A complementary approach is the use of developer-related factors for bug prediction. For example, Hassan investigated a technique based on the entropy of code changes by developers [89], reporting that it has better performance than models based on code components changes. Ostrand et al. [113, 114] proposed the use of the number of developers who modified a code component as a bug-proneness predictor: however, the performance of the resulting model was poorly improved with respect to existing models. Finally, Di Nucci et al. [91] defined a bug prediction model based on a mixture of code, process, and developer-based metrics outperforming the performance of existing models.

Despite the aforementioned promising results, developers consider class/module level bug prediction too coarse-grained for practical usage [93]. Hence, the need for a more fine-grained prediction, such as *method-level*. This target adjustment does not negate the value of the preceding work but calls for a re-evaluation of the effectiveness of the proposed methods and, possibly, a work of adaptation.

2.2.2 Method-level Bug Prediction

So far, only Giger et al. [35] and Hata et al. [99] independently and almost contemporaneously targeted the prediction of bugs at method-level. Overall they defined a set of metrics (Hata et al. mostly process metrics, while Giger et al. also considered product metrics) and evaluated their bug prediction capabilities. Giger et al. found that both product and process metrics contribute to the identification of buggy methods and their combination achieves promising performance (i.e., F-Measure=86%) [35]. Hata et al. found that using method-level bug prediction one saves more effort (measured in number of LOC to be analyzed) than both file-level and package-level prediction [99]. The data collection approach used by both sets of researchers is very similar, here we detail that used by Giger et al. [35], as an exemplification.

To produce the dataset used in their evaluation, Giger et al. conducted the following steps [35]: they (1) took a large time frame in the history of 22 Java OSS systems, (2) considered the methods present at the end of the time frame, (3) computed product metrics for each method at the end of the time frame, (4) computed process metrics (e.g., number of changes) for each method throughout the time frame, and (5) counted the number of bugs for each method throughout the time frame, relying on bug fixing commits. Finally, they used 10-fold cross-validation [115] to evaluate three models (only process metrics, only product metrics, and both combined), considering the presence/absence of bug(s) in a method as the dependent binary variable.

In the work presented in this paper, we replicate the same methodology of Giger et al. and Hata et al. on an overlapping sets of projects to see whether we are able to reach similar results for other contexts. For simplicity and because the methodological details are more extensive, we follow more closely the case of Giger et al. [35].

2.3 Research Goal and Subjects

This section defines the goal of our empirical study in terms of research questions and the subject systems we consider.

2.3.1 Research Questions

The first *goal* in our study is to replicate bug-prediction work at method level, by using the research method employed by Giger et al. [35] on a partially overlapping set of software systems in different moments in time, with the *purpose* of understanding the extent to which their results are actually generalizable. This leads to our first research question:

> **RQ1.** *How effective are existing method-level bug prediction approaches when tested on new systems/timespans?*

While replicating the methodology proposed by Giger et al. [35], we found some limitations with the validation approach that they followed to assess the effectiveness of the prediction methods. In fact, although reasonable for an initial validation, the type of validation followed by Giger et al. has the following limitations: (1) it uses 10-fold cross-validation, which is at the risk of producing biased estimates in certain circumstances [116], (2) product metrics are considered only at the end of the time frame (while bugs are found *within* the time frame), (3) the number of changes and the number of bugs were both considered in the same time frame (this *time-insensitive* validation strategy may have led to biased results).

In the second part of our study we try to overcome the aforementioned limitations by re-evaluating the performance using data from subsequent releases (i.e., a release-by-release validation). Release-by-release validation better models a real-case scenario where a prediction model is updated as soon as new information is available. Our expectation is that the performance is going to be weaker in this setting, but we hope still promising. This leads to our second research question:

> **RQ2.** *How effective are existing method-level bug prediction models when validated with a release-by-release validation strategy?*

Table 2.1: Overview of the subject projects investigated in this study

Projects	LOC	Developers	Releases	Methods	All Buggy Methods	Last Buggy Methods
Ant	213k	15	4	42k	2.3k	567
Checkstyle	235k	76	6	31k	4.1k	670
Cloudstack	1.16M	90	2	85k	13.4k	6.8K
Eclipse JDT	1.55M	22	33	810k	3.3k	96
Eclipse Platform	229k	19	3	7k	2.7k	932
Emf Compare	3.71M	14	2	9k	0.7k	444
Gradle	803k	106	4	73k	4.6k	1.1k
Guava	489k	104	17	262k	1.2k	71
Guice	19k	32	4	9k	0.5k	145
Hadoop	2.46M	93	5	179k	5.8k	1.3k
Lucene-solr	586k	59	7	213k	8.7k	962
Vaadin	7.06M	133	2	43k	11.3k	7.7K
Wicket	328k	19	2	30k	4.9k	2.2K
Overall	**19M**	**782**	**91**	**1.8M**	**63.4k**	**22.9k**

2.3.2 Subject Systems

The *context* of our work consists of the 13 software systems whose characteristics are reported in Table 2.1. For each system, the table reports its size (in KLOCs), number of contributors, releases, methods, and number of buggy methods over the entire change history, and number of buggy methods contained in its last release. In particular, we focus on systems implemented in Java (i.e., one of the most popular programming languages [117]), since the metrics previously used/defined by both Giger et al. [35] and Hata et al. [99] mainly target this programming language. In addition, we choose projects whose source code is publicly available (i.e., open-source software projects) and are developed using Git as version control system, in order to enable the extraction of product and process metrics. Hence, starting from the list of open-source projects available on Github,[1] we randomly selected 13 systems that have a change history composed of at least 1,000 commits and more than 5,000 methods. Our dataset is numerically smaller than the one by Giger et al., but comprises larger systems composed of a much larger number of both methods (1.8M vs 112,058) and bugs (63,400 vs 23,762); this allows us to test the effectiveness of method-level bug prediction on software systems of a different kind of size.

2.4 RQ_1 - Replicating Method-Level Bug Prediction

The work regarding RQ_1 aims at replicating the study conducted by Giger et al. [35] on a different set of systems and time spans and relies on the method-level bug prediction technique.

2.4.1 RQ_1 - Research Method

To answer our first research question, we (i) build a method-level bug prediction model using the same features as Giger et al. [35] and (ii) evaluate its performance applying it to our projects. To this aim, we follow a set of methodological steps such as (i) creation of an oracle reporting buggy methods in each of the projects considered, i.e., the dependent variable to predict (ii) definition of the independent variables, i.e., the metrics on which the model relies on, (iii) testing of the performance of different machine learning algorithms, and (iv) definition of the validation methodology to test the performance of the model.

Extraction of Buggy Data. For each system we need to detect the buggy methods contained at the end of the time frame, i.e., in the *last* release R_{last}, to do so we use a methodology in line with that followed by Giger et al. [35]. Given the tagged issues available in the issue tracking systems (i.e., Bugzilla or Jira) of the subject systems, we firstly use ReLink [118] to identify links between issues and commits. ReLink considers several constraints, i.e., (i) a match exists between the committer and the contributor who created the issue in the issue tracking system, (ii) the time interval between the commit and the last comment posted by the same contributor in the issue tracker is less than seven days, and (iii) the cosine similarity between the commit note and the last comment referred above, computed using the Vector Space Model (VSM) [119], is greater than 0.7. Afterwards, we consider as buggy all the methods actually changed in the buggy commits detected by ReLink and referring to the time period between the R_{last-1} and R_{last}, i.e., the ones

[1] https://github.com

introduced during the final time frame. We filtered out test cases, which might be modified with the production code, but might not directly be implicated in a bug.

Independent variables. As for the metrics to characterize source code methods, we compute the set of 9 product and 15 process features defined by Giger et al. [35].

Table 2.2: List of method-level product metrics used in this study

Metric name	Description (applies to method-level)
FanIN	# of methods that reference a given method
FanOUT	# of methods referenced by a given method
LocalVar	# of local variables in the body of a method
Parameters	# of parameters in the declaration
CommentToCodeRatio	Ratio of comments to source code (line based)
CountPath	# of possible paths in the body of a method
Complexity	McCabe Cyclomatic complexity of a method
execStmt	# of executable source code statements
maxNesting	Maximum nested depth of all control structures

Table 2.3: List of method-level process metrics used in this study

Metric name	Description (applies to method level)
MethodHistories	# of times a method was changed
Authors	# of distinct authors that changed a method
StmtAdded	Sum of all source code statements added
MaxStmtAdded	Maximum StmtAdded
AvgStmtAdded	Average of StmtAdded
StmtDeleted	Sum of all source code statements deleted
MaxStmtDeleted	Maximum of StmtDeleted
AvgStmtDeleted	Average of StmtDeleted
Churn	Sum of stmtAdded - stmtDeleted
MaxChurn	Maximum churn for all method histories
AvgChurn	Average churn per method history
Decl	# of method declaration changes
Cond	# of condition changes over all revisions
ElseAdded	# of added else-parts over all revisions
ElseDeleted	# of deleted else-parts over all revisions

- *Product Metrics:* Existing literature demonstrated how such set of features might be effective to characterize the extent to which a source code method is difficult to maintain, possibly indicating the presence of defects [90, 101, 104, 112]. Giger et al. [35] proposed the use of the metrics reported in Table 2.2. The features cover different method characteristics, e.g., number of parameters or McCabe's cyclomatic complexity [120]. We re-implement all of the metrics due to the lack of available tools.

- *Process Metrics:* According to previous literature [109, 111], process features effectively complement the capabilities of product predictors for bug prediction. For this reason, Giger et al. [35] relied on the change-based metrics described in Table 2.3 and that widely characterize the life of source code methods, e.g., by considering how many statements were added over time or the number of developers that touched the method.

  Also in this case, we re-implement the proposed process metrics defined at method-level by Giger et al. [35].

Similarly to Giger et al. [35], in the context of RQ_1, we build three different method-level bug prediction models relying on (i) *only* product metrics, (ii) *only* process metrics, and (ii) *both* product and process metrics.

Training Data Preprocessing. Once we have the dataset containing (i) product and process metrics (i.e., the independent variables) and (ii) buggy methods (i.e., the dependent variable), we start the method-level bug prediction process. As first step, we take into account two common problems that may affect machine learning algorithms, namely (i) data unbalance [121] and (ii) multi-collinearity [122].

The former represents a frequent issue in bug prediction occurring when the number of instances that refer to buggy resources (in our case, source code methods) is drastically smaller than the number of non-buggy instances. We address this problem by applying the Random Over-Sampling algorithm [123] implemented as a supervised filter in the Weka toolkit.[2] The filter re-weights the instances in the dataset to give them the same total weight for each class maintaining unchanged the total sum of weights across all instances.

The second problem comes from the use of multiple metrics. These independent variables may have a high correlation causing collinearity that negatively impacts the performance of bug prediction models [124]. To cope with this problem, we preprocess our dataset filtering out the unwanted features. Specifically, we apply the *Correlation-based Feature Selection* [125] algorithm implemented as a filter in the Weka toolkit: It evaluates the worth of a subset of attributes by considering the individual predictive ability of each feature along with their degree of redundancy.

Machine Learner. Once preprocessed the training data, we need to select a classifier that leverages the independent variables to predict buggy methods [126]. To this aim, we exploit the four classifiers used by Giger et al., which are all available in Weka toolkit: *Random Forest*, *Support Vector Machine*, *Bayesian Network*, and *J48*. Afterwards, we compare the different classification algorithms using validation strategy and metrics we describe later.

Evaluation Strategy. The final step to answer RQ_1 consists of the validation of the prediction models. As done in the reference work, we adopt the 10-fold cross-validation strategy [115, 127]. This strategy randomly partitions the original set of data into 10 equal sized subset. Of the 10 subsets, one is retained as test set, while the remaining 9 are used as training set. The cross-validation is then repeated 10 times, allowing each of the 10 subsets to be the test set exactly once.

Evaluation Metrics. Once we had run the experimented models over the considered systems, we measure their performance using the same metrics proposed by Giger et al. [35]

[2] https://www.cs.waikato.ac.nz/ml/weka/

to allow for comparison: *precision* and *recall* [119]. Precision is defined as $precision = \frac{|TP|}{|TP+FP|}$ where TP (True Positives) are methods that are correctly retrieved by a prediction model and FP (False Positives) are methods that are wrongly classified by a prediction model. Recall is defined as $recall = \frac{|TP|}{|TP+FN|}$, where FN (False Negatives) are methods that are not retrieved by a prediction model (i.e., buggy methods misclassified as non-buggy by a model). We also compute F-Measure [119], which combines precision and recall in a single metric: $F-Measure = 2 \cdot \frac{Precision \cdot Recall}{Precision+Recall}$.

In addition to the aforementioned metrics, we also compute the *Area Under the Receiver Operation Characteristic* curve (AUC-ROC) [128]. In fact, the classification chosen by the machine learning algorithms is based on a threshold (e.g., all the method whose predicted value is above the threshold 0.5 are classified as buggy), which can greatly affect the overall results [116]; precision and recall alone are not able to capture this aspect. ROC plots the true positive rates against the false positive rates for all possible thresholds between 0 and 1; the diagonal represents the expected performance of a random classifier. AUC computes the area below the ROC and allows us to have a comprehensive measure for comparing different ROCs: An area of 1 represents a perfect classifier (all the defective methods are recognized without any error), whereas for a random classifier an area close 0.5 is expected (since the ROC for a random classifier tends to the diagonal).

2.4.2 RQ$_1$ - Results

Table 2.4: Median classification results of method-level bug prediction models when validated using 10-fold cross validation.

π = Product	Precision			Recall			F-measure			AUC-ROC		
Π = Process	π	Π	π&Π	S	Π	π&Π	π	Π	π&Π	π	Π	π&Π
Bayesian Network	0.71	0.77	0.77	0.46	0.68	0.70	0.56	0.72	0.72	0.60	0.72	0.72
J48	0.73	0.82	0.84	0.60	0.84	0.83	0.65	0.83	0.83	0.60	0.79	0.80
Random Forest	0.72	0.85	**0.86**	0.64	**0.86**	0.86	0.68	0.85	**0.86**	0.66	0.84	**0.86**
Support Vector Machines	0.66	0.74	0.74	0.09	0.80	0.79	0.16	0.77	0.76	0.50	0.51	0.51
Overall	0.71	0.80	0.80	0.44	0.80	0.80	0.51	0.80	0.80	0.59	0.72	0.73

Table 2.4 reports the median precision, recall, F-measure, and AUC-ROC achieved by models based on (i) only product, (ii) only process, and (iii) both product and process features when using different classifiers. A detailed report of the performance achieved by the single classifiers over all the considered systems is available in our online appendix [129]. Overall, the obtained results are in line with those by Giger et al., yet we achieve values that are 10 percentage points lower on average.

The model based on product metrics achieves the lowest results. For instance, the overall precision is 0.71, meaning that a software engineer using this model has to needlessly analyze almost 39% of the recommendations it outputs. This result is in line with the findings provided by Giger et al., who already showed that the model only trained on product metrics offers generally lower performance.

Secondly, our results confirm that process metrics are stronger indicator of bug-proneness of source code methods (overall F-Measure=0.80). Also in this case, this finding is in line with the previous results achieved by the research community that report the superiority of process metrics with respect to product ones [100, 111]. Our results also

confirm another finding by Giger et al.: The combination of product and process metrics does not improve dramatically the prediction capabilities: Results are—at most—two points percentage higher than the model with process metrics only. We find this surprising, since both set of metrics have values in the prediction and we expected that the use of these orthogonal predictors would improve the overall performance of the approach.

As for the different classifiers experimented, *Support Vector Machines* gives the worst results; likely, this is due to the extreme sensitivity of the classifier to the configuration [130]. In fact, as shown in previous research [130, 131], the use of the default configuration might lead to significantly worsen the overall performance of the machine learner. Future studies could be setup and conducted to investigate the impact of the configuration on SVM for method-level bug prediction.

Other classifiers provide more stable results. *Random Forest* and *J48* obtain the best prediction accuracy considering all the evaluation metrics. The differences are particularly evident when considering the AUC-ROC values, which are 36% and 29% higher than VSM, respectively. Our results confirm what was reported by Giger et al. on the capabilities of *Random Forest*, and more in general on the performance of this classifier in the context of bug prediction [54, 132].

To test the statistical significance of the results discussed so far, we compared the AUC-ROC values of the experimented models over the different systems using the Scott-Knott Effect Size Difference (ESD) test [133], which is effect-size aware variant of the Scott-Knott test [134] that is recommended in case of comparisons of multiple models over multiple datasets [133]. As a result, process-based models built using *Random Forest* and *J48* are considered statistically better than product-based ones, while they work similarly to the combined ones. Detailed statistical results are in our online appendix [129].

Result 1: Our results, computed with the same evaluation strategy but on a different set of systems/timespans, confirm the findings by Giger et al.: Method-level bug prediction models based on process metrics perform better than those based on product metrics. Our results are 10 percentage points lower than those of Giger et al., yet far better than random. The combination of predictors of different nature does not dramatically improve the prediction capabilities.

2.5 Reflecting on the Evaluation Strategy

By replicating the work by Giger et al., we had the chance to reflect on the evaluation strategy. Figure 2.1 shows an exemplification of the history of a system and how the training and testing are done in the approach by Giger et al. (named '10-fold overall evaluation' in the figure and depicted using red lines and text) and in the one we propose in this work (named 'release-by-release' and depicted in blue).

The system in Figure 2.1 has four methods (i.e., M_a, M_b, M_c, M_d) that were changed several times throughout the history of the system. The changes sometimes were related to a bug (i.e., the method was involved in a bug fix; purple dot), sometimes not (i.e., green dot); for example method M_a was changed four times, two of which involving a bug fix. This system had at least three releases (i.e., R_x throughout R_{x+1}).

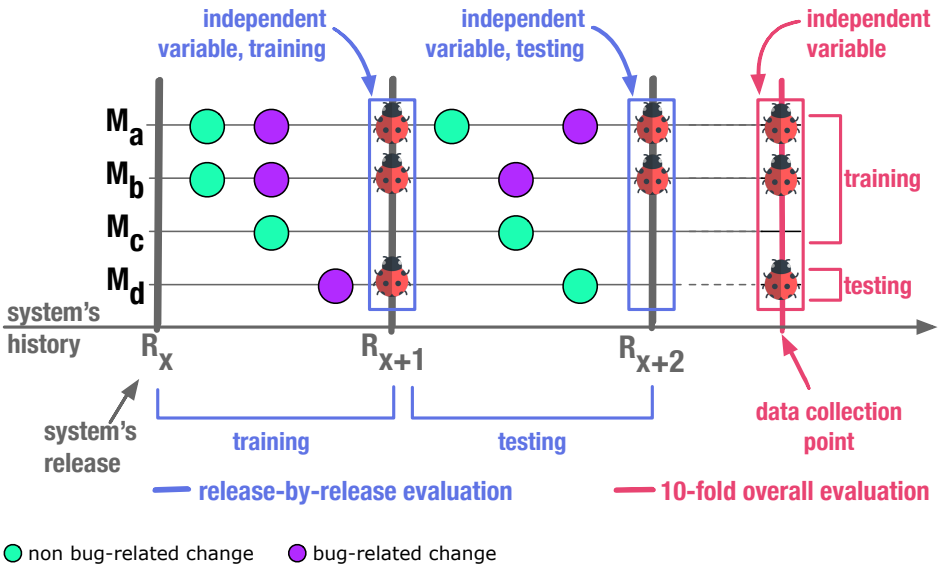

Figure 2.1: Training and testing strategies for method-level bug prediction.

The approach applied by Giger et al. collects all the available information until the 'data collection point', then marks a method as 'buggy' whenever the method was involved in a bug fix (hence it was buggy before the bugfix) in the entire history of the system. Then, each method would be considered as an instance to classify, where the independent variable is whether the method was marked as 'buggy' or not. In this case, the validation would be done "vertically": 10-fold cross validation ensures that the classifier is trained on a subset of methods (e.g., M_a, M_b, M_c in Figure 2.1) that is different from that used for the testing (e.g., M_d).

The limitation of this approach is that it uses dependent variables (such as most of the process metrics, including 'number of changes') (1) whose value could not be known at prediction time in a real-world scenario (i.e., one would try to predict bugs that still have to occur, not that already happened) and that (2) seem to be highly correlated to the independent variable (for each bug fix there has to be a change). Moreover, there are moments in which the methods were not buggy, but if they have been buggy at least once in the lifetime of the system, they are considered as buggy.

Although reasonable for an initial validation, the approach followed by Giger et al. may lead to unrealistic results. For this reason, we propose a release-by-release strategy, similar to one adopted by Kpodjedo et al. [135]. We train and test "horizontally" instead of "vertically": We assume to be in the moment of a release (e.g., R_{x+1} in Figure 2.1) and we train on all the information available from the previous release to this moment (e.g., from R_x); in this case the independent variable is whether or not a method has been buggy during the considered release. Then, we consider the next release (e.g., R_{x+2}) and try to predict which methods will be buggy in the course of the development of this release; yet, we do not consider any information available from the current release to the next, because

this would not be available in real life. With this strategy we answer RQ_2.

An addition to the release-by-release strategy would be to consider the SZZ algorithm [136] and consider as buggy only the methods in which a bug was introduced before the release (regardless of when the fix happened). We decided *not* to follow this path for three reasons: (1) SZZ could give information that is not available at prediction time (e.g., when the bug fix happens after the considered release, but the bug inducing commit happens before the release), (2) SZZ has been proven to be not reliable [137], and (3) we want to reduce at a minimum the differences from the work of Giger et al. we are replicating, so that the obtained results are not due to unconsidered causes.

2.6 RQ_2 - Re-evaluating Method-Level Bug Prediction

Our RQ_2 seeks to evaluate the performance of method-level bug prediction models in a more realistic setting.

2.6.1 RQ_2 - Methodology

To answer RQ_2, we need to (i) extract all the releases of the considered projects, (ii) identify the buggy methods occurring in each of them, and (iii) build the three bug prediction models considered in the context of RQ_1.

Extraction of The Major Releases. The first step to test the performance of method-level bug prediction models consists in the identification of the major releases of the considered systems. To this purpose, we automatically extract them from the list of releases declared on the Github repository of the subject systems. To discriminate major releases from the others we rely on a heuristic based on naming conventions: if the version name ends with the patterns 0 or 0.0 (e.g., versions 3.0 or 3.0.0), then a major release is identified. We manually verified the performance of this heuristic on one of the subject systems: We verified that all the major releases of Lucene-Solr were correctly caught, thus quantifying the actual performance of this approach.

Extraction of Buggy Data. Differently from what we have done in RQ_1, in this research question we need to extract the buggy data for all the considered releases. For each release pair $ri-1$ and r_i, we (i) run ReLink and (ii) consider as buggy all the methods actually changed in the buggy commits detected by ReLink and referring to the time frame between $ri-1$ and r_i. We filtered out test cases.

Bug Prediction Models: Setup. As done for RQ_1, we test the performance of three bug prediction models, i.e., the ones relying on (i) product metrics *only*, (ii) process metrics *only*, and (ii) *both* product and process metrics, built using the same set of machine learning approaches, i.e., *Random Forest*, *Support Vector Machine*, *Bayesian Network*, and *J48*. Also in this case, the training data is preprocessed to avoid (i) data unbalance and (ii) multicollinearity by using the same set of techniques previously exploited, i.e., *Random Over-Sampling* algorithm [123] and *Correlation-based Feature Selection* [125], respectively.

Bug Prediction Models: Validation. As a final step to answer the second research question, we test the performance of the prediction models by applying an *inter-release* validation procedure, i.e., we trained the prediction models using the release r_{i-1} and tested it on r_i. This technique implies that the first release of each system could not be used as

testing set as well as the last release could not be used as training. To measure the accuracy of such models, we computed the same set of metrics previously exploited, i.e., precision, recall, F-Measure, and AUC-ROC.

2.6.2 RQ_2 - Results

Table 2.5: Median classification results of method-level bug prediction models when validated using a release-by-release strategy.

S = Product	Precision			Recall			F-measure			AUC-ROC		
H = Process	S	H	S&H	S	H	S&H	S	H	S&H	S	H	S&H
Bayesian Network	0.72	0.70	0.70	0.58	0.64	**0.65**	0.59	0.60	0.61	0.53	0.52	0.53
J48	0.71	0.71	0.71	0.59	0.59	0.59	0.62	0.62	0.63	0.51	0.51	0.51
Random Forest	0.72	0.70	0.72	0.63	0.60	0.63	**0.64**	0.61	0.63	0.52	0.51	0.52
Support Vector Machines	0.72	**0.73**	0.72	0.59	0.57	0.60	0.62	0.58	0.62	0.53	0.53	**0.53**
Overall	0.71	0.71	0.71	0.59	0.60	0.60	0.62	0.60	0.61	0.52	0.52	0.53

Table 2.5 reports the median *precision*, *recall*, *F-measure*, and *AUC-ROC* achieved by models based on (i) only product, (ii) only process, and (iii) both product and process metrics when using different classifiers and the release-by-release strategy. For sake of space limitation, we report the results aggregated using the median operator, however, detailed reports are available in our appendix [138].

The performance achieved by all the prediction models experimented is significantly lower than those found in the replication presented in RQ_1. We observe a limited decrease between the highest/lowest values and overall in each of the subject systems in our dataset.

In this evaluation scenario, the use of code metrics as predictors only slightly *improves* the capabilities of method-level bug prediction models. This is in contrast with past literature reporting the superiority of process metrics for bug prediction [100, 111]. We hypothesize that this result may be caused both by the different granularity of the experimented models and by the different validation strategy with respect to the one used in RQ_1. In particular, while the historical information computed at class-level could better characterize the complexity of the development process followed by developers while implementing changes in an entire class [89], it is reasonable to think that the bugginess of source code methods may be better expressed by the methods' current code quality. An additional possible cause that refutes the observation of previous studies [100, 111] comes from the irregular distribution of the length of the time frames for the considered releases. In our analyzed projects, these intervals stretch between a few months to a couple of years and the distribution of the releases is strictly correlated to the needs and the approach adopted by developers in a given historical moment. The higher prediction capabilities of code metrics are confirmed also when looking at other indicators, i.e., precision, recall, AUC-ROC. Moreover, this result holds for all the classifiers considered.

Finally, we observe that the performance of different classifiers is similar and there is no clear winner. To some extent, this result confirms previous findings in the field [139, 140] showing that different classifiers achieve similar performance. This result potentially highlights the possibility to further study the orthogonality of classifiers for method-level bug prediction with the aim of exploiting ensemble methodologies [54, 132].

Result 2: All the experimented method-level bug prediction models resulted in dramatically lower performance (up to 20 points percentage less in terms of AUC-ROC) when evaluated with the more realistic release-by-release evaluation strategy, instead of 10-fold cross validation. The achieved AUC-ROC scores achieved by all the models, regardless of the machine learning approach, are close to the results that a random classifier would provide.

2.7 Threats to Validity

In this section, we describe the factors that might have affected the validity of our empirical study.

Threats to Construct Validity. A first factor influencing the relationship between theory and observation is related to the dataset exploited. In our study, we rely on the same methodology previously adopted by Giger et al. [35] to build our own repository of buggy methods, i.e., we first retrieve bug-fixing commits using the textual-based technique proposed by Fisher et al. [69] and then consider as buggy the methods changed in that commits. To understand possible imprecisions and/or incompleteness of the data used in this study, we manually validate a statistically significant sample of 275 buggy methods detected on the Lucene-Solr system. Such a set represents a 95% statistically significant stratified sample with a 5% confidence interval of the 962 total buggy methods detected in the last release of the project. The validation was conducted by the first two authors of this paper. Based on (i) the description of the bug reported on the issue tracker system, (ii) the source code of a method detected as buggy in a commit c_i, and (iii) the list of modifications to the method between c_i and its predecessor c_{i-1} (extracted using the `diff` unix command), each author checked *independently* whether the changes applied between the two revisions might have actually introduced the bug reported on the issue tracker. After the first round, the two inspectors started a discussion on the independent classifications made to reach consensus. The level of agreement between the inspectors is computed using the krippendorff's α [141], finding it to be 0.84, which is higher than the 0.80 used as standard reference score [142]. As for the accuracy of the linking methodology, we found that it correctly captured the bugginess of 85% of methods. Thus, at the end of this process we can claim that the oracle built is accurate enough for our purposes.

A threat to the validity of our replication is that we had to re-implement the product and process metrics used to build the experimented models, due to the lack of a publicly available tool. When re-implementing such metrics we faithfully followed the descriptions by Giger et al. [35].

Although we test the performance of the models using the same machine learning classifiers used by Giger et al. [35] to closely replicate their study, the use of different classifiers may produce different results. A more detailed analysis of the impact of other classifiers on our results is part of our future agenda. A complete overview of this analysis is available in our online appendix [129]. Moreover, all the tested classifiers use the default parameters, since finding the best configurations would have been too expensive [143].

Threats to Conclusion Validity. To ensure that the results would not have been biased by confounding effects such as data unbalance [121] or multi-collinearity [124], we

adopt formal procedures aimed at (i) over-sampling the training sets [121] and (ii) removing non-relevant independent variables through feature selection [125].

Threats to External Validity. This category refers to the generalizability of our findings. While in the context of this work we analyze software projects having different size and scope, we limit our focus to Java systems because some of the tools exploited to compute the independent and dependent variables mainly target this programming language. Thus, the generalizability with respect to systems written in different languages as well as to projects belonging to industrial environments is limited.

2.8 Conclusion

In this paper, we investigated (i) the practical benefits of different classes of method-level bug prediction models when applied in a real-case scenario and (ii) the contribution of textual features to existing bug prediction models.

The main contributions made by this work are:

1. A validation aimed at understanding the applicability of a method-level bug prediction model in a real-case scenario, by predicting the bug-proneness of methods in a release-by-release shape. The results highlight that they achieve an overall AUC-ROC of 53% and an overall F-measure up to 63%, i.e., lower than what previously found in literature when evaluated in a less realistic scenario.

2. An empirical analysis of how the performance of existing method-level bug prediction models can be improved by considering a set of 8 textual features. Our results reveal that the overall prediction capabilities can be improved up to 10%, by including these metrics.

3. A fine-grained empirical analysis of the gain provided by each textual feature in method-level bug prediction. The results show a reduction of the entropy up to 35% for three textual features: *Readability*, *Textual Coherence*, and *Notice*.

4. An online appendix [138] that reports the dataset and all the additional analyses performed in the work described in this paper.

Based on the results achieved so far, our future agenda includes (i) the replication of our study on a larger set of systems along with a study aimed to measure the capabilities of ensemble methods [54, 132] and (ii) an *in-vivo* analysis of the capabilities of method-level bug prediction models, done by involving practitioners during their daily activities [94].

3

Fine-Grained Just-In-Time Defect Prediction

Defect prediction models focus on identifying defect-prone code elements, for example to allow practitioners to allocate testing resources on specific subsystems and to provide assistance during code reviews. While the research community has been highly active in proposing metrics and methods to predict defects on long-term *periods (i.e., at release time), a recent trend is represented by the so-called* short-term *defect prediction (i.e., at commit-level). Indeed, this strategy represents an effective alternative in terms of effort required to inspect files likely affected by defects. Nevertheless, the granularity considered by such models might be still too coarse. Indeed, existing commit-level models highlight an* entire *commit as defective even in cases where only* specific *files actually contain defects.*

In this work, we first investigate to what extent commits are partially defective; then, we propose a novel fine-grained just-in-time *defect prediction model to predict the specific files, contained in a commit, that are defective. Finally, we evaluate our model in terms of (i) performance and (ii) the extent to which it decreases the effort required to diagnose a defect. Our study highlights that: (1) defective commits are frequently composed of a mixture of defective and non-defective files, (2) our fine-grained model can accurately predict defective files with an overall AUC-ROC up to 86% and (3) our model would allow practitioners to save inspection efforts with respect to standard just-in-time techniques.*

This chapter is based on

L. Pascarella, F. Palomba, A. Bacchelli. Fine-Grained Just-In-Time Defect Prediction, JSS'18 [144]

The results have been re-evaluated (and the text updated accordingly) to satisfy the replicability of this study, as described in Sections 1.6 and 1.8.

3.1 Introduction

During software maintenance and evolution, developers constantly modify the source code to introduce new features or fix defects [145]. These modifications, however, may lead to the introduction of new defects [58], thus developers must carefully verify that the performed modifications do not introduce new defects in the code. This task is usually performed directly during development (e.g., by running test cases) [146] or when changes are reviewed [7]. An efficient way to allocate inspection and testing resources to the portion of source code more likely to be defective is represented by *defect prediction* [32], which involves the construction of statistical models to predict the defect-proneness of software artifacts, by mostly exploiting information regarding the source code or the development process.

The problem of defect prediction has attracted the attention of many researchers in the past decade, who tried to address it by (i) conducting empirical studies on the factors making artifacts more defect-prone (e.g., [44, 46, 47, 49, 90, 147–149]) and (ii) proposing novel prediction models aimed at accurately predicting the defect-proneness of the source code (e.g., [56, 89, 91, 100, 113, 150]).

Most of the existing techniques evaluate the defectiveness of software artifacts perform *long-term* predictions. Analyzing the information accumulated in previous software releases, these models predict which artifacts are going be more prone to defect *in future releases*. For instance, Basili et al. investigated the effectiveness of Object-Oriented metrics [101] in predicting post-release defects [90], while other approaches consider process metrics (e.g., the entropy of changes [89]) or developer-related factors [91, 113] for the same purpose.

Kamei et al. reported that these long-term defect prediction models—despite their good accuracy—may have a limited usefulness in practice because they do not provide developers with immediate feedback [59], thus not avoid the introduction of defects during the commit of artifacts on the repository. To overcome this limitation, a recent trend is the investigation of *just-in-time* prediction models, i.e., techniques exploiting the characteristics of a commit to perform *short-term* predictions of vthe likelihood of a commit introducing a defect. With this solution, a developer can limit the effort required to diagnose problems since s/he focuses on the committed artifacts only [59]. Among the studies investigating *just-in-time* prediction models, Kamei et al. [59, 151] defined 14 metrics characterizing a commit under five perspectives, demonstrating how such metrics can be successfully exploited for predicting defective commits either in the case the model is trained using previous data of the project [59] and in the case the training information come from different projects [151]. Other approaches proposed the use of deep-learning [62], textual analysis [152], and unsupervised methodologies [153].

It is reasonable to think that, in a real-world scenario, a commit may be *partially* defective, i.e., it may be composed of both defective and non-defective files. In this case, despite the advantages provided by *just-in-time* defect prediction, a developer might still need to spend a considerable effort to locate the files of a commit that are actually defective. For instance, during a Modern Code Review (MCR) the reviewers iterate several times over the proposed set of changes and the amount of time spent finding a subset of defective files might substantially increase [7]. In this work, we aim at making a further step ahead in the context of *just-in-time* defect prediction by investigating the original problem at a finer

granularity. Particularly, our goal is to investigate the prominence of partially defective commits and, should they be a significant amount, devise a defect prediction model to identify the *defective files within a commit*.

To this aim, we firstly performed an exploratory study to characterize defective commits and evaluate whether fine-grained solutions are actually needed. In the second place, we built a *fine-grained just-in-time* defect prediction model adapting 24 basic features previously defined in the papers by Kamei et al. [59] and Rahman and Devanbu [100]. We assessed the performance of the model in terms of (i) accuracy of the predictions and (ii) effort developers can save using our model with respect to state-of-the-art *just-in-time* prediction models. The study was conducted considering 10 major open source systems and 164,000 commits created by 2,000 developers. Key findings of our investigations revealed that (i) up to 55% of defective commits are composed of a mixture of both defective and non-defective resources, (ii) the devised *fine-grained* model obtained an AUC-ROC 86% on average when locating defective files in a commit, and (iii) our model is more cost-effective than the state of the art *just-in-time* model. For reproducibility and replicability, we release all tools and scripts needed to reproduce the results presented in this work [82].

Structure of the Chapter. Section 3.2 reports background, related work, and a concept of the envisioned solution. Section 3.3 reports the methodology used to address our research goal as well as possible threats that might influence our findings, while Section 3.4 presents the results of the study. Finally, Section 3.5 concludes the Chapter.

3.2 Background and Related Work

In this section we introduce the terminology used through the paper, discuss the related work, and motivate our study.

3.2.1 Terminology

Throughout the paper, we frequently refer to the following five concepts:

Defect. To define a condition in which a software system does not meet its requirements, we use the term *defect*, among all the possible terms (e.g., *bug* and *fault* [32]).

Defect-Inducing/-Fixing Change. We identify two events in the life of a defect: (i) the *defect-inducing change*, i.e., the code change that inserts the defect into a project and (ii) the *defect-fixing change*, i.e., the code change that fixes the defect.

Commit. In most modern collaborative software projects, authors develop code relying on version system control tools such as Git.[1] Such tools track changes as *commits*, which are documented changes that involve one or more files.

Non-/Partially/Fully Defective Commit. We define three classes of commits: *non-defective commits* (when all the committed files are changed without introducing any defect), *fully defective commits* (when all the committed files are changed introducing defects), and *partially defective commits* (when a subset of the committed files are changed introducing a defect). The top part of Figure 3.1 depicts a part of the history of an example

[1] https://git-scm.com/

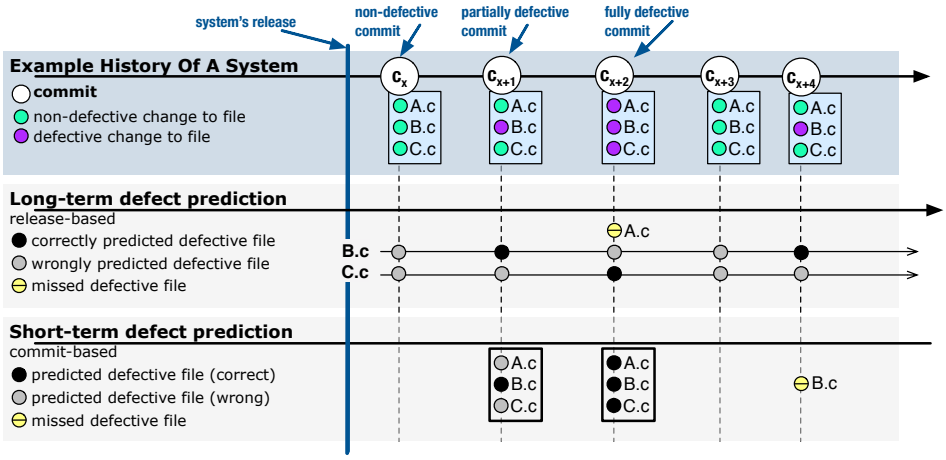

Figure 3.1: An example set of (defective) commits after release to files in a software system.

software system, with the activities made on the versioning system after a system's release. A set of commits $C = \{c_x, ..., c_{x+4}\}$ are performed by developers to evolve the system; all the commits change the same three files (A.c, B.c, and C.c). In Figure 3.1, we see examples of non-defective commits (c_x and c_{x+3}), partially defective commits (c_{x+1}, c_{x+4}), and a fully defective commit (c_{x+2}).

3.2.2 Related Work

In this section, we discuss the related work that inspired and guided this study, considering long- and short-term defect prediction.

Long-term Defect Prediction

Long-term defect prediction pertains to models able to classify defect-prone files in future releases of a software project. Several studies addressed this problem in the recent years (a relatively recent survey has been compiled by Hall et al. [32]). Basically, the proposed models differ for the source of information used to predict the defectiveness of a class: the main distinction is between static, product information and historical, process data.

Product Information. Structural data are computed with metrics such as the McCabe's cyclomatic complexity [120] or the Chidamber and Kemerer (CK) [101] metrics. These product metrics have been investigated in several studies [54, 95, 102, 108, 154–157] and researchers have shown how such metrics can provide useful contribution in the prediction of defective classes. For instance, Nagappan et al. [157] found that a model based on code metrics may achieve up to 83% of accuracy in the identification of defect-prone classes.

Process Information. Historical data are computed with metrics such as relative code churn, entropy of changes, or developer-related factors [32, 89, 91, 108, 111, 158]. Also in this case, researchers provided empirical evidence on the value of such metrics for defect prediction. For instance, Moser et al. [111] performed a comparative study analyzing

static- and historical-based predictors, concluding that metrics computed over the history of projects are better predictors and can significantly improve the performance of defect prediction models.

Combining Information. Later on, D'Ambros et al. [112] found that combined techniques work better than models based on single unique set of metrics. On the basis of this result, di Nucci et al. [91] defined a combined model based on a mixture of static and historical metrics able to outperform the prediction capabilities of single models. Finally, Menzies et al. [150] introduced the concept of *local* defect prediction, an approach in which classes that will be used for training the classifier are firstly clustered into homogeneous groups to reduce the differences among such classes and obtain higher prediction accuracy.

We build on top of this line of work, by considering the features that are better able to predict defects at file-level—a key attribute that we include in our model.

Short-term Defect Prediction

Short-term defect prediction refers to models able to classify defect-prone at commit time. Previous work motivated the introduction of this new strategy with the need of having tools able to locate defects in the shortest possible time [159]. While Madeyski et al. [160] proposed the idea of continuous defect prediction, Mockus et al. [159] addressed the problem by proposing a model based on the change-proneness of files to predict defects at commit-level.

Characteristics of Defect-Introducing Changes. Other studies (e.g., Sliwerski et al. [136] and Eyolfson et al. [161]) tried to localize defect-introducing changes in open source projects by means of correlation between the defectiveness of a commit and the experience of developers. Sliwerski et al. [136] also discovered that defect-introducing changes are generally a part of large transactions. The "unnaturalness" of defective code was subsequently confirmed by Ray and Hellendoorn et al. [48], who discovered that source code presenting defects is characterized by higher entropy than non-defective code. They also found that source code entropy might be a valid and simpler way to complement the effectiveness of static analysis tools (e.g., CheckStyle[2]) in recommending to developers the areas of source code where to focus inspection activities.

Ad Hoc Models. Jiang et al. [162] proposed the concept of "personalized" defect prediction by proposing a technique able to create a different model for each developer. Their results report that such a technique outperforms existing *just-in-time* defect prediction models. Along with this line, Xia et al. [163] improved the aforementioned technique by Jiang et al. using a multi-objective genetic algorithm that firstly builds a defect prediction model for each developer, and then combine these models assigning different weights with the aim of maximizing F-Measure and cost-effectiveness. With respect to these papers, our approach has not the goal to build a prediction model for each developer, but instead that of providing feedback on defect-prone classes within the scope of a commit: further analysis will evaluate the possible benefits provided by the creation of personalized models in the context of fine-grained just-in-time defect prediction.

[2]http://checkstyle.sourceforge.net

Just-In-Time. The studies by Kamei et al. [59, 151] are great source of inspiration for our work. They proposed a *just-in-time* quality assurance technique that predict defects at commit-level trying to reduce the effort of a reviewer [59]. Later on, they also evaluated how *just-in-time* models perform in the context of cross-project defect prediction [151]. Their main findings report good accuracy for the models in terms of both precision and recall, but also in terms of saved inspection effort. Our work is complementary to these papers. In particular, we start from their basis of detecting defective commits and complement this model with the attributes necessary to filter only those files that are defect-prone and should be more thoroughly reviewed. Rahman et al. [164], Yang et al. [62], and Barnett et al. [152] proposed the usage of alternative techniques for *just-in-time* quality assurance, such as cached history, deep learning, and textual analysis, reporting promising results. We did not investigate these further in the current paper, but studies can be designed and carried out to determine if and how these techniques can be used within the model we present in this paper to further increase its accuracy.

3.2.3 Motivating Example

We discuss an example in which a developer uses long- vs. short-term defect prediction models while inspecting a commit, in order to show some of the limitations of these approaches.

The top part of Figure 3.1 depicts an example history of a software system, with the activities made on the versioning system after a system's release. A set of commits $C = \{c_x, ..., c_{x+4}\}$ are performed by developers to evolve the system. For sake of clarity, suppose that the files A.c, B.c, and C.c are always changed in the considered commits after the system's release. The small circles in the top bar represent changes made to files in each commit and the colors represent whether these changes introduce a defect (purple) or not (dark green) in these files. In addition, a black box surrounds all the files in the same commit. In the following we describe the behavior of the two aforementioned prediction models:

Long-term Defect Prediction. Based on the information gathered before the system's release, a long-term defect prediction model would mark certain files as defect-prone for the entire period leading to the issue of the next release. In our example, the model marks the files B.c and C.c as defect-prone and A.c as non-defective. The model classifies both B.c and C.c as defective starting from the system's release onward, in Figure 3.1 we depict this behavior with a horizontal small arrows, thus showing that B.c and C.c are considered as defective in *every commit*. Indeed, the model does not provide any information about the exact commit that will likely lead to the introduction of a defect. This model would issue warnings about these files on each commit involving them. In our example, this represents an unjustified extra-effort for the developer inspecting the commit. As found in previous research, this unjustified extra-effort derived from using a tool can reduce the developers' confidence in the prediction [165], thus leading to miss important defects in future commits involving actual defect-prone artifacts. Finally, we see that the model does not classify as defective the code in file A.c, also when a defect is introduced in commit c_{x+2} (the missed defective file is depicted with a yellow circle).

Short-term Defect Prediction. As an alternative, a reviewer may adopt a short-term

defect prediction model such as the *just-in-time* one proposed by [59]. In this scenario, a developer is pointed to analyze more in depth only the files referring to a commit marked as potentially defective by the model. However, the number of resources to inspect might be still high depending on the number of files committed and the wasted effort on how many are defect-free. For instance, in the commit c_{x+1} shown in Figure 3.1, only the file `B.c` introduces a defect, while the others are defect-free, yet a warning from the tool would be issued; the developer may need to analyze some non-defective files before finding the actual defect. Thus, while short-term solutions can significantly reduce the reviewers' effort, they might still produce extra-effort in cases a commit is partially defective. Furthermore, in our example, file `B.c` is again defective in commit c_{x+4}, but it is not marked as such, since the model does not recognize the commit as defective (in the figure, the missed defective file is depicted with a yellow circle).

The goal of our work is to make the first steps in supporting software developers during the inspection of a commit (e.g., in a code review), by striving to overcoming the aforementioned limitations of existing defect prediction models in this context. The next section details our research questions and the research method.

3.3 Methodology

This section defines the overall goal of our study, motivates our research questions, and outlines our method.

3.3.1 Research Questions

The *goal* of the study is to investigate how frequently commits are only partially defective and to devise a defect prediction model able to identify the files with the changes that are more likely to introduce a defect. We set up our work around three research questions. The first one is a preliminary analysis aimed at assessing the extent to which a defect prediction model is actually able to estimate the defect-proneness of files within a commit. To this aim, we investigate the ratio of commits that contain both defective and not defective files. Should the frequency of partially commits be low, standard just-in-time models, such as the one devised by Kamei et al. [59] would be sufficient, while in case there should be a notable percentage of commits presenting both defective and non-defective files, then defect prediction models working at a finer granularity than standard just-in-time ones would be desirable.

RQ$_1$. *What is the ratio of partially defective commits?*

Once assessed the actual need for finer grained solutions, we devise a defect prediction model to predict defective files at commit scope.

RQ$_2$. *To what extent can we predict defect-inducing changes at file-level in a commit?*

In addition to assessing the model as a whole, we also evaluate which features provide the highest contribution to the achieved performance.

Table 3.1: Characteristics of the subject software systems.

Project	ID	KLOC	Developers	Commits	Defective Commits
Accumulo	P0	102	65	8,639	1,154
Angular-js	P1	87	923	8,102	2,852
Bugzilla	P2	239	80	9,250	3,873
Gerrit	P3	79	230	24,340	6,269
Gimp	P4	102	306	37,329	8,734
Hadoop	P5	291	165	15,689	2,301
JDeodorant	P6	70	10	1,101	348
Jetty	P7	88	70	13,784	2,698
JRuby	P8	129	308	41,256	9,174
OpenJPA	P9	822	29	4,263	2,921
Overall		**2k**	**2k**	**164k**	**39k**

RQ$_3$. *What are the features of the devised model that the most to its performance?*

Finally, we are also interested in understanding how much effort could be saved when using the proposed model, comparing it to the just-in-time defect prediction model proposed by Kamei et al. [59] as our baseline.

RQ$_4$. *How much effort can be saved using a fine-grained just-in-time defect prediction model with respect to a standard just-in-time model?*

In the following sections, we describe the steps we perform to answer our three research questions.

3.3.2 Subject Systems

To conduct our analysis, we focused on open-source software systems and defined multiple selection criteria: We selected software systems (i) written in the most common programming languages (C, C++, Java, JavaScript, Ruby, and Perl, i.e., the most popular programming languages [117]), (ii) having different size and scope, and (iii) having a change history composed of at least 1,000 commits. We preferred open-source systems where a versioning system is used to track all changes. The access to the source code history enables the computation of metrics with static analysis tools. Moreover, to increase the generalizability of our research, we selected software projects having different domains and programming languages. Note that our selection is not intended to be statistically significant, but rather we just aim at selecting a various set of systems to assess the performance of our prediction model in different contexts (e.g., when considering projects having different change history sizes). In practice, we started from the entire list of open source projects available on GitHub; then we filtered out systems not implemented in the considered programming languages and with less than 1,000 commits in their history.

Successively, among 2,362,287 project candidates, we considered only the most popular projects for a given domain or scope; finally, we randomly selected the ten open-source software systems reported in Table 3.1. For each system, the table reports size (in terms of KLOCs), number of developers, number of commits, and the information on the number of defective commits.

3.3.3 RQ_1 - Investigating Defective Commits

To answer our first research question, we analyze the ratio of the defective files (i.e., source code, configuration, and auxiliary files) contained in defective commits. To this aim, for each commit c_i of the change history of a system S, we identify the set $defectiveFiles(c_i)$ composed of the defective resources contained in c_i. To the best of our knowledge, there is not a publicly available dataset reporting this information: Previous work defined datasets of defective commits [59], without providing details on which of the resources in a certain commit were actually defective. For this reason, we build our own dataset as detailed in the following.

Data Extraction. To automatically identify the set of defective files in each of the commits of the considered systems, we rely on the SZZ algorithm [136, 166]. SZZ exploits the annotation/blame feature of a versioning system to *estimate* the lines of code of a file that induced a certain defect, thus retrieving files that are defect-inducing in each commit. More formally, the algorithm implements the following steps:

1. For each file f_i (where $i = 1...n$) involved in a defect fixing commit dfc, the algorithm $prevVersion(commit, file)$ extracts the last version of the file before the defect fixing commit: $prevVersion(dfc, f_i)$;

2. Starting from the commit $prevVersion(dfc, f_i)$, for each line of code in f_i changed to fix the defect in dfc, the algorithm uses `git blame` to detect the file revision where the last change to that line occurred. We identify comments and empty lines using island parsing [167] and we exclude f_i if no other code is touched. This step outputs the commits in which a defect in file f_i is introduced.

The SZZ algorithm takes as input the list of defects that are *already fixed* by developers, excluding the open ones,[3] but the analysis and the effect of considering open issues will be considered in future work (e.g., exploiting tools such as ReLink [118]).

Data Analysis. Once extracted the defective files involved in defective commits, we answer $\mathbf{RQ_1}$ in two ways. First, we measure how many defective commits are *partially* defective, i.e., they contain a mixture of both defective and non-defective resources. This analysis allow us to understand the magnitude of the problem investigated: If the vast majority of defective commits is composed of only defective artifacts, then standard *just-in-time* defect prediction models would suffice; conversely, if a significant part of defective commits is *partially* defective, then the introduction of fine-grained solutions might be worthwhile. Second, we further analyzed the set of *partially* defective commits, by measuring the ratio between defective and non-defective files they contain. More formally, we computed the $defectiveFiles_{dc}$ ratio as follow:

[3] Open issues might be not verified by developers (i.e., they might be not real defects).

$$defectiveFiles_{dc} = \frac{\#defectiveFiles(dc)}{\#files(dc)} \tag{3.1}$$

where $\#defectiveFiles(dc)$ represents the number of defective files in the defective commit dc, and $\#files(dc)$ the total number of files in dc. This analysis helped us to understand the intrinsic characteristics of *partially* defective commits. Also in this case, if the resulting ratio is high (most files are defective in *partially* defective commits), then the adoption of fine-grained solutions would be not worthwhile.

3.3.4 RQ$_2$ - The Fine-Grained JIT Model

Table 3.2: List of the independent/predicting variables adapted from Rahman et al. [100]* and Kamei et al. [59]**

Acronym	Name	Description	Ref.
COMM	Commit Count	Number of changes to the file up to the considered commit	*
ADEV	Active Dev Count	Number developers who modified to the file up to the considered commit	*
DDEV	Distinct Dev Count	Cumulative number of distinct developers that contributed to the file up to the considered commit	*
ADD	Normalized Lines Added	Normalized number of lines added to the file in the considered commit	*
DEL	Normalized Lines Deleted	Normalized number of lines removed to the file in the considered commit	*
OWN	Owner's Contributed Lines	Boolean value indicating whether the commit is done by the owner of the file	*
MINOR	Owner's Contributions	Number of contributors who contributed less than 5% of the file up to the considered commits	*
SCTR	Changed Code Scattering	Number of packages modified by the committer in the commit	*
NADEV	Neighbor's Active Dev Count	Number of developers who changed the files involved in commits where the file has been modified	*
NDDEV	Neighbor's Distinct Dev Count	Cumulative number of distinct developers who changed the files involved in commits where the file has been modified	*
NCOMM	Neighbor's Commit Count	Number of commits made to files involved in commits where the file has been modified	*
NSCTR	Neighbor's Package Count	Number of different packages touched by the developer in commits where the file has been modified	*
OEXP	Percentage of Lines	Percentage of lines authored in the project	*
EXP	All Committer's Experience	Mean of the experiences of all the developers	*
ND	Number of modified directories	Number of modified directories	**
Entropy	Distribution of modified code across each file	Entropy of changes of the file up to the considered commit	**
LA	Lines of code added	Number of lines added to the file in the considered commit (absolute number of the ADD metric)	**
LD	Lines of code deleted	Number of lines removed to the file in the considered commit (absolute number of the DEL metric)	**
LT	Lines of code in a file before the change	Lines of code in the file before the change	**
AGE	Average interval between the last and the current change	The average time interval between the last and the current change	**
NUC	Number of unique changes to the modified files	Number of times the file has been modified alone up to considered commit	**
CEXP	Experience of the committer	Number of commits made on the file by the committer up to the considered commit	**
REXP	Recent developer experience (last x months)	Number of commits made on the file by the committer in the last month	**
SEXP	Developer experience on a subsystem	Number of commits made by the developer in the package containing the file	**

To answer our second research question, we build a *fine-grained just-in-time* defect prediction model and evaluate its performance. In the following, we describe (i) the independent variables, i.e., the metrics on which the model relies, (ii) the dependent variable, i.e., the characteristic that the model have to predict, (iii) the machine learner performing the predictions, and (iv) the validation methodologies to estimate the accuracy.

i. Independent Variables. This step consists in extracting and quantifying the characteristics of each file involved in a commit. To this purpose, we considered the 24 basic features shown in Table 3.2. These features represent a modified version of those previously proposed by [59] and [100]. We adapted the previous metrics to work at file-level in a commit. The column 'Description' in Table 3.2 details the implementation of the metrics in our context. The choice of the independent variable is driven by two goals: (i) to understand the value of standard *just-in-time* measures in a fine-grained context; (ii) to investigate whether metrics originally proposed in the context of long-term defect prediction to predict defective files may also provide useful contributions when employed in the prediction of defective files contained in a change set.

Furthermore, the chosen metrics help us to characterize commits under different perspectives, thus allowing us to evaluate which metric types are more relevant in our context.

Specifically, we selected metrics to measure (i) the developers' experience (e.g., the experience of the committer [59]), (ii) structural and process factors of the files in the commit (e.g., the lines of code added or the number of previous changes of a committed file [100]), and (iii) factors related to the neighbors' of a committed file, which have been shown to be relevant for predicting the defectiveness of files [100]. Although other metrics have been proposed in the contexts of both code review (e.g., by [168] and [169]) and defect prediction (e.g., [91, 112]), the selected metrics better allow us to verify the role of a larger set of metrics that have been previously adopted for traditional short- and long-term defect prediction. Further studies can be conducted to investigate the addition of other metrics in our context.

From a methodological standpoint, the process metrics adapted from [100] (i.e., COMM, ADEV, DDEV, ADD, DEL, OWN, MINOR, SCTR, NADEV, NDDEV, NCOMM, NSCTR, OXEP, and EXP) were always evaluated considering the commits up to the commit of interest. Similar adjustments were applied for the metrics proposed by [59]. For instance, the NUC metric represents the number of unique changes to the files modified in a commit. In our case, we adjust NUC to represent the number of times a single file involved in a commit is modified alone up to the considered commit. Descriptions of how we adapted the [59] and [100] metrics are reported in Table 3.2.

ii. Dependent Variable. The characteristic to measure is the defectiveness of files contained in a commit. To this aim, we exploited the dataset built in the context of $\mathbf{RQ}_1$ (i.e., we used the output of the SZZ algorithm as a dependent variable to predict).

iii. Machine Learner. In this stage, we needed to select a machine learning classifier able to use the independent variables to infer the defectiveness of files in a change set [126]. To this aim, we tested different classifiers (using the validation methodologies described later in this section), i.e., *Binary Logistic Regression* [170], *J-48* [132], *ADTree* [171], *Multilayer Perceptron* [172], *Naive Bayes* [173], and *Random Forest* [174]. As a result, we found that the *Random Forest* technique [174] is the one having the highest performance, in line with previous findings [175, 176].

Such classifiers builds several decision trees, each of them containing nodes representing a condition on a certain feature that splits the dataset into two. A condition is chosen based on the so-called *Mean Decrease in Impurity* (MDI) [177], a metric able to measure the extent to which the value of a feature can correctly discriminate the dependent variable. It is important to point out that the selected classifier automatically performs a feature selection, thus avoiding the well-known problem of multi-collinearity [122] that occurs when two or more independent variables correlate with each other, possibly affecting the performance of the classifier.

iv. Validation Methodologies. The final step to answer $\mathbf{RQ}_2$ is related to the validation of the model. Commonly used techniques such as *ten-fold cross* [127, 178], or *leave-one-out cross-validation* [179] are not suitable for the validation of just-in-time defect prediction models because the data points (i.e., the commits) follow a certain time order: *Time-insensitive* validation strategies might cause a model to be trained using future data that should not be known at the time of the prediction [127]. For this reason, we adopt a *time-sensitive* analysis where the defectiveness of a commit c_i is evaluated by a model trained

using the data coming from the previous three months of history of the system considered. In other words, while the training set is composed of three-month data, the test set is represented by each commit singularly. Doing so, we exclude the first three months of change history, because of the lack of data needed to perform a proper validation [127]. Our choice of considering three-month periods is based on: (i) choices made in previous work [89, 91, 127] and (ii) the results of an empirical assessment we performed on such a parameter. The empirical assessment showed that the best performance for the devised model is achieved by using three-month periods. In particular, we experimented with time windows of one, two, three, and six months. The complete results are available in our replication package [82].

Afterward, we measure the performance of the model using *precision* and *recall* [119]:

$$precision = \frac{|TP|}{|TP + FP|} \tag{3.2}$$

$$recall = \frac{|TP|}{|TP + FN|} \tag{3.3}$$

where TP, FP, and FN are:

- True Positives (TP): elements that are correctly retrieved by the *fine-grained just-in-time* prediction model (i.e., defective files correctly classified as such);
- False Positives (FP): elements that are wrongly classified by the *fine-grained just-in-time* prediction model (i.e., non-defective files misclassified as defective by the model);
- False Negatives (FN): elements that are not retrieved by the *fine-grained just-in-time* prediction model (i.e., defective files misclassified as non-defective by the model).

In addition, to have a unique value that synthesizes precision and recall we also measure the *F-measure*, i.e., the harmonic mean of precision and recall:

$$F - Measure = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall} \tag{3.4}$$

While the metrics described so far have been widely used in the past to evaluate defect prediction models [32], most of the classifiers output a probability ranging between 0 and 1 representing the likelihood of a code component to be part of a certain class (i.e., in our case, to be *defective* or *non-defective*). The threshold used to discriminate the two classes (in most cases—as well as in this work—such threshold is set to 0.5) influences the computation of both precision and recall, and as a consequence of F-Measure. ROC plots the true positive rates against the false positive rates for all possible thresholds between 0 and 1; the diagonal represents the expected performance of a random classifier. AUC computes the area below the ROC and allows us to have a comprehensive measure for comparing different ROCs: An area of 1 represents a perfect classifier (all the defective

methods are recognized without any error), whereas for a random classifier an area close 0.5 is expected (since the ROC for a random classifier tends to the diagonal). To have a detailed view of the performance of the model in the different cases found in **RQ**$_1$, in Section 3.4.2 we report the evaluation metrics achieved when ran the model over the set of (i) all the defective commits in the dataset, (ii) partially defective commits only, and (iii) fully defective commits only.

3.3.5 RQ$_3$ - Investigating the Importance of the Features

While in **RQ**$_2$ we provide an overview of the accuracy of the devised model in predicting defective files within a commit, **RQ**$_3$ has the goal of investigating which features contribute the most to the prediction capabilities. To address this point, we use an *information gain* algorithm [180] to quantify the gain provided by each independent variable to the prediction of defective files within commits. Formally, let M be the devised *fine-grained just-in-time* prediction model, let $F = \{f_1, \ldots, f_n\}$ be the set of features used by M, the *information gain* algorithm [180] applies the following formula to compute the difference in entropy:

$$InfoGain(M|f_i) = H(M) - H(M|f_i) \tag{3.5}$$

where the function $H(M)$ indicates the entropy of the model that includes the feature f_i, while the function $H(M|f_i)$ measures the entropy of the model that does not include f_i. Entropy is computed as follow:

$$H(M) = -\sum_{i=1}^{n} prob(f_i) \log_2 prob(f_i) \tag{3.6}$$

The algorithm quantifies the degree of uncertainty in M that is reduced by considering the feature f_i. In our work, we employ the *Gain Ratio Feature Evaluation* algorithm [180], which ranks $f_1, \ldots, f_n$ in descending order based on the contribution provided by each feature to the decisions made by M. More specifically, the output of the algorithm is represented by a ranked list in which the features having the highest expected reduction in entropy are placed at the top.

3.3.6 RQ$_4$ - Measuring the Saved Effort

For **RQ**$_4$ we investigate the potential benefits in terms of saved effort that the *fine-grained just-in-time* defect prediction model provides to a developer analyzing the committed files to discover possible defects (e.g., in a code review). Specifically, we perform an effort-aware validation as recommended by Kamei et al. [96]. In this formulation, a technique is assessed on the fraction of defects it can detect while varying the effort required to locate them. As done in previous work [59], we first rank the files to inspect according to their probability of being defective, as it is assigned by the automated classifier (in our case, *Random Forest*); then we measure the percentage of defects that a developer would identify as the effort spent in analyzing the suggested defective files increases. To approximate such an effort, we use the *number of lines of code to inspect*; this metric has been shown to be a surrogate measure of the effort needed for testing or reviewing a module, as code and cognitive complexity are strongly related to size [157]. Thus, size can be considered as a lightweight and efficient solution to estimate the developer's effort in inspecting a code change [55, 154, 181, 182].

We compare our model to the traditional *just-in-time* defect prediction model proposed by Kamei et al. [59]. The selection of this baseline is driven by experimental tests, where we found that this approach works better than the twelve unsupervised techniques proposed by Yang et al. [153]. In particular, the model by Kamei et al. [59] achieves an AUC-ROC 6% higher than the best unsupervised technique, which was the one that predicts a commit as defective in case of a number of committed files higher than eight. We report the results of this additional analysis in our online appendix [82].

To perform a fair comparison, the baseline relies on the same predictors used by Kamei et al. in their experiments and is trained using the best performing classifier (i.e., *Random Forest*, the same used by our approach). We also empirically evaluate the performance of the several classifiers, namely *Binary Logistic Regression* [170], *J-48* [132], *ADTree* [171], *Multilayer Perceptron* [172], *Naive Bayes* [173], and *Random Forest* [174], when applied on the model by Kamei et al. [59]. Also in this case, *Random Forest* classifier outperforms the others.

As the baseline can only assign defect probabilities to commits (because of the commit-level granularity), we assume that the same probability holds for all files within that commit. In other words, if a commit is considered defective by Kamei et al.'s technique, then all the files within that commit are considered as potentially defective and have the same probability to require further inspection by a developer. To determine in which sequence the developer would inspect the files in the same commit, we use the *alphabetical* order, because it is the normal order offered by both IDEs and code review tools [21]. Once we assign the probabilities/order to all the files, we rank them in descending order and compare it with the ranking provided by our technique. It is worth noting that we expect our technique to outperform this baseline, as by definition it aims at lowering the granularity of the information presented to developers. Nevertheless, we still consider this comparison useful because we can verify whether and how much our approach actually meet the expected goal.

Finally, we perform a comparison with the *optimal* approach that ranks all the actual defective files first, starting from the smallest to the largest. In this way, we can investigate how far our technique is with respect to optimal scenario as well as how much it improves upon existing *just-in-time* approaches.

Data Analysis. To quantify the differences between our model and the baselines, we use the P_{opt} and P_k evaluation metrics [181]. P_{opt} is defined as the Δ_{opt} between the effort-based cumulative lift charts of the optimal model and the devised prediction model. Similarly, Δ_k is defined as the Δ_k between our technique and the one by Kamei et al. [59]. Larger values of P_{opt} and P_k indicate smaller differences between the compared techniques. Such values are normalized in the range [0,1] to ease their interpretation [59].

3.3.7 Threats to validity

The results of our study may be affected by a number of threats.

Threats to construct validity. As for factors threatening the relation between theory and observation, in our context, these are mainly concerned with the measurements we performed. Above all, we rely on the results of the SZZ algorithm [136] to answer our

research questions. Although the intrinsic imprecisions of SZZ [137] still represent a threat for the validity of our results, it is the most effective algorithm available in literature.

To compute the CEXP, REXP, SEXP metrics, we mined commits to count the number of modifications applied by a developer in different time windows. However, it might be possible that the actual author of a commit is not the same person as the committer. That may be especially true in large projects where sometimes developers (e.g., newcomers) can modify the source code but do not have rights to perform a push onto the repository. This potential problem might have influenced the way the metrics are computed and used within the devised prediction model. To verify the extent to which this represents an actual issue for our analyses, we quantified in how many cases there was a mismatch between author and committer in the analyzed commits. Specifically, for each commit of the considered projects, we ran the command `git show –format=full`[4] to obtain the full set of information available for the commit. That includes data on both author and committer email addresses. Thus, we could compute the number of times in which the two email addresses differ, i.e., in how many cases the author of a change was not the actual committer. Out of the 160,515 total commits considered in our study, we found 4,173 mismatches, meaning that we are not accurate in only 2.6% of the cases. Based on this result, we can argue that such mismatches represent corner cases rather than systematic problems that threats our analysis. To further verify the impact of this potential threat, we completely re-ran our study excluding those 4,173 commits. However, we did not observe any difference for the results achieved when including the commits. That indicates that mismatches between authors and committers do not influence our findings.

Threats to conclusion validity. Although the metrics used to evaluate the performance of the *fine-grained just-in-time* defect prediction model, (i.e., precision, recall, F-measure, and AUC-ROC), are widely used in the field [112], future studies can be conducted to validate our model from a different angle, e.g., by evaluating its industrial impact.

A possible threat concerning the results achieved in $\mathbf{RQ}_1$ is related to the co-presence of production and test files within a commit, which may lead to a over-estimation of the number of partially defective commits. We conducted an additional analysis to assess the effect of excluding test files on our findings. We could not find differences with respect to the results reported in the original submission (a complete report of this additional analysis is available in our online appendix [82]). These results are in line with recent work: Even test code may be defective [183] and test files have the same proneness of production files to be affected by functional issues [184]. It seems reasonable to keep test files in our analysis/approach to maintain developers' awareness also on bugs in tests.

Another threat regards how we assess the cost-effectiveness of the models experimented. As done in previous research [55, 154, 181, 182, 185, 186], we measure the inspection cost in terms of lines of code to be inspected by a reviewer. LOC has been evaluated as a valid proxy measure [154] since it is correlated with code and cognitive complexity [157]. However, this is an approximation. Function points (FPs) [187] represent an alternative that we do not consider in this study, since it requires setting parameters that only original

[4] https://git-scm.com/docs/git-show

developers/managers or expert effort estimation consultants might properly set and a third-party analysis done by the authors of this work would introduce noise/bias. Future work can be designed and conducted to investigate how much size approximates defect inspection effort.

In the context of $\mathbf{RQ}_2$, we adopt a time-sensitive validation strategy where a single commit c_i represents the test set and the data of the previous three months form the training set. We select this strategy because this is the most similar to a real-case scenario where developers use the devised approach as soon as a new commit is performed, for example in a code review. While other researchers adopted slight variations of this strategy (e.g., Tan et al. [127] used a gap between training and test sets to add in the training set defective commits that were discovered and fixed), we preferred it for its stronger ecological validity.

We statistically compare the differences between our model and the standard *just-in-time* model proposed by Kamei et al. [59]. We do not perform statistical tests with the Bonferroni correction [188]: This is a conscious decision taken on the basis of the findings by Perneger [189], who explained why such a correction is unnecessary and deleterious for sound statistical inference. Finally, we assess the model for the presence of multicollinearity [122], relying on *Random Forest*, which can automatically remove non-relevant features.

As a final note, we compare our model with the one proposed by Kamei et al. [59] in the context of $\mathbf{RQ}_4$ (the cost-effectiveness analysis) but not in $\mathbf{RQ}_2$ (the accuracy analysis). On the one hand, the model by Kamei et al. [59] targets a different problem (i.e., detecting defective commits rather than defective files within commits), thus it cannot be fairly compared with the proposed model in terms of accuracy. This statement is supported by experimental data, which showed that the model by Kamei et al. [59] achieved an overall F-Measure of 31% and AUC-ROC of 53% when employed in our context (by considering all the files within an identified defective commit as defective). On the other hand, the comparison performed regarding cost-effectiveness allows us to understand and quantify the gain provided by our approach against state of the art.

Threats to external validity. The main issue concerns the generalizability of the results. To alleviate this issue, we take into account a variety of projects having different characteristics, scope, and size. Nevertheless, future studies must be devised to replicate and extend our investigation on a larger set of systems, possibly taking into consideration industrial projects as well.

Table 3.3: Results for RQ1 on partially defective commits

	Ratio		
Systems	Partially defective commits	Defective files	Avg. files per commit
Accumulo	39%	63%	4.1
Angular-js	10%	40%	2.2
Bugzilla	35%	36%	5.4
Gerrit	29%	47%	3.3
Gimp	16%	38%	4.3
Hadoop	55%	44%	3.1
JDeodorant	33%	43%	3.4
Jetty	34%	33%	3.8
JRuby	25%	71%	3.5
OpenJPA	21%	24%	4.0
Overall	**30%**	**44%**	**3.7**

3.4 Results and Analysis

In this section, we present the results of the study by research question.

3.4.1 RQ_1. What is the ratio of partially defective commits?

The analysis of the results associated to the first research question aims to understand the prominence of *partially* defective commits, hence the importance of devising a *fine-grained* solution for just-in-time defect prediction. Table 3.3 reports the results for each considered system: The second column reports the percentage of *partially* defective commits contained in the considered systems, the third column shows the percentage of defective files for each projects (computed using Formula 3.1), and the fourth column reports the average number of files per commit in the considered systems. The last row ("Overall") represents the average ratio computed taking into account all the projects as a single dataset.

Among all the defective commits investigated we found that 30% of them are *partially* defective, i.e., they contain a mixture of both defective and non-defective files, while 70% of defective commits only contain one resource. Thus, while standard *just-in-time* models can be adopted in most cases, there still exists a consistent part of defective commits for which they cannot provide developers with detailed information.

Investigating the *partially* defective commits more in depth, we found that on overall only 44% of committed files are defective; this is quite surprising, since it implies that less than the half of the elements in a *partially* defective commit is actually defective. Considering the perspective of a developer who has to inspect the files in a change set, she might spend more than half of the time inspecting non-defective resources before finding an actual defect.

For instance, let us consider the commit `a0641ea475` belonging to the Angular.js project.[5] In this case, the developer committed 9 different files with the aim of making

[5]https://github.com/angular/angular.js/pull/15881

configurable the errors to show in case of wrong usage of the tool. However, there was only one defective file in the whole change set, i.e., the `minErr.js` one. As a consequence, the usage of *coarse-grained just-in-time prediction* model such as the one proposed by Kamei et al. might not provide the adequate support in these cases. The observations made until now still hold when considering the "best" scenario reported in the table, i.e., the one of the JRuby project, where we found that 71% of the resources in a defective commit is affected by a problem, enforcing a developer to inspect many non-defective resources before diagnosing the defect.

With the aim of further understanding the characteristics of defective commits, we also computed the Kendall's τ correlation [190] between the number of files per commit and the number of defective files. This is a non-parametric statistical test used to measure the ordinal association between two measured quantities, with a value ranging between -1 and +1.[6] In our case, the correlation between number of files per commit and number of defective files turned to be equals to 0.42, thus indicating a positive concordance between the two variables. This confirms previous findings reporting that the more resources a developer changes the higher the chances to introduce defects [191].

In conclusion, the results show the need of fine-grained techniques to reduce the number of resources to inspect in a defective commit.

Result 1: 44% of defective commits in our subjects are *partially* defective, i.e., composed of both files that are changed without introducing defects and files that are changed introducing defects. Further, in almost 31% of the changed files a defect is introduced, while the remaining files are defect-free.

3.4.2 RQ_2. To what extent can the model predict defect-inducing changes at file-level?

Table 3.4: Results of the RQ2 considering all commits in the history of the subject software systems.

Systems	Precision	Recall	F-measure	AUC-ROC
Accumulo	93%	94%	92%	92%
Angular-js	85%	86%	85%	91%
Bugzilla	91%	93%	91%	81%
Gerrit	83%	84%	80%	82%
Gimp	90%	91%	88%	88%
Hadoop	92%	92%	90%	86%
JDeodorant	77%	90%	77%	76%
Jetty	88%	88%	86%	87%
JRuby	87%	88%	84%	91%
OpenJPA	92%	90%	92%	90%
Overall	**87%**	**89%**	**85%**	**86%**

[6] (i) -1 represents a perfect negative linear relationship, (ii) +1 a perfect positive linear relationship, and (iii) the values in between indicate the degree of linear dependence between the two measured quantities

Table 3.5: Results of the RQ2 considering only partially defective commits in the history of the subject software systems.

Systems	Precision	Recall	F-measure	AUC-ROC
Accumulo	97%	97%	97%	90%
Angular-js	98%	98%	97%	96%
Bugzilla	84%	97%	97%	86%
Gerrit	97%	89%	86%	88%
Gimp	88%	96%	94%	93%
Hadoop	95%	95%	95%	93%
JDeodorant	65%	64%	63%	70%
Jetty	94%	85%	92%	94%
JRuby	83%	85%	81%	89%
OpenJPA	95%	96%	96%	96%
Overall	**89%**	**91%**	**89%**	**90%**

Table 3.6: Results of the RQ2 considering only fully defective commits in the history of the subject software systems.

Systems	Precision	Recall	F-measure	AUC-ROC
Accumulo	82%	82%	80%	83%
Angular-js	79%	79%	79%	87%
Bugzilla	90%	90%	89%	84%
Gerrit	78%	79%	76%	81%
Gimp	79%	81%	78%	82%
Hadoop	85%	86%	85%	90%
JDeodorant	87%	91%	88%	84%
Jetty	75%	75%	74%	80%
JRuby	79%	80%	79%	82%
OpenJPA	89%	88%	88%	91%
Overall	**82%**	**83%**	**81%**	**84%**

To answer our second research question we evaluate the effectiveness of the prediction model described in Section 3.3.4 based on a machine learning algorithm built using the *Random Forest* classifier. For sake of clarity, we report the results of both RQ_2 and RQ_3 in three separated tables that have a similar structure. The columns "RQ_2" report the evaluation metrics, i.e., *precision*, *recall*, *F-measure*, and *AUC-ROC*, for each system. Table 3.4 is obtained evaluating our model considering indiscriminately all commits in the history of the projects, instead Table 3.5 considers only *partially* defective commits and Table 3.6 represents only *fully-defective* commits.

Looking at the full-inclusive results of Table 3.4, we observe that the precision ranges between 77% and 93% (overall=87%), the recall between 84% and 94% (overall=89%), while the overall F-measure is equal to 85%. Interesting are the results in terms of precision: in a context where the recommendations are given when developers are committing their changes on the repository, having a tool able to pinpoint the files that are likely defective

can avoid the introduction of a consistent number of defects in a system. Assuming that developers can recognize a defect should they get a true positive warning from our model, the adoption of our model has the potential to be useful in practice, since its precision is higher than 85% in most of the cases. The recall values tell us that our model locates more than half of the defects actually present in the subject systems.

Considering also the AUC-ROC we observe that the model obtains levels between 76% and 92% (overall=86%). The worst case observed in our dataset regards the JDeodorant project, where our model achieves the lowest F-measure (77%). Investigating the likely causes behind this result, we found that our model was not able to achieve better performance because of the size of the project that limits also the size of the training set. An additional cause that decreases the performance of our model is the variety of programming languages. Projects such as Angular-js, Gimp, and JRuby that use *JavaScript*, *C/C++*, or *Ruby* tend to perform lower than projects developed in *Java*. A speculative cause of this difference can be found in the way how changes are applied to programming languages such as *C*: when *C* sources are touched to fix a defect also several other files (i.e., headers) are modified, this affects the metrics that consider the number of added or removed lines.

We do not observe large decays between the overall metric values and the highest/lowest ones (i.e., the difference is always within 10%). This means that the *fine-grained just-in-time* model is consistent across the projects.

At the same time, we consider the performance degradation noticed on *fully* defective commits as reasonable. Unfortunately, we are not able to speculate on the specific reasons causing such degradation. Likely, the addition of further independent variables able to characterize the defectiveness of commits as a whole (e.g., the metrics devised by Kamei et al. [59]) can be beneficial to improve the performance of the model further. A future research effort can be devoted to the potential combination between just-in-time and fine-grained just-in-time models. In any case, our results show that in the majority of the cases the model can provide further recommendations also when considering fully-defective commits. Finally, the model including all commits inherits pros and cons observed in the cases of the models built on *partially* and *fully* defective commits only. In other words, it can predict *partially* defective commits better than *fully* defective ones, having higher performance on the former and lower on the latter; that shifts performance in the middle to the individual models.

Result 2: The proposed model achieves an overall *AUC-ROC* of 86% and obtains stable performance across the considered projects.

3.4.3 RQ_3. What are the features of the devised model that the most to its performance?

Table 3.7 reports the results achieved when applying the *Gain Ratio Feature Evaluation* algorithm [180] to understand which are the most relevant features that allow the model to identify defect-inducing changes within the files of a commit. For each variable and project we report the expected entropy reduction the variable gives to the model for the specific project.

Table 3.7: Contribution provided by each feature to the prediction model, as computed by the *Gain Ratio Feature Evaluation* algorithm [180], by software project.

Variable	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	All
COMM	0.022	0.031	0.122	0.015	0.015	0.036	0.016	0.016	0.000	0.081	0.017
ADEV	0.003	0.025	0.007	0.007	0.019	0.005	0.000	0.007	0.013	0.002	0.026
DDEV	0.007	0.049	0.024	0.017	0.014	0.037	0.013	0.011	0.032	0.087	0.015
ADD	0.072	0.277	0.024	0.034	0.050	0.047	0.022	0.076	0.051	0.068	0.045
DEL	0.044	0.152	0.002	0.009	0.050	0.003	0.000	0.058	0.033	0.042	0.014
OWN	0.003	0.003	0.018	0.006	0.000	0.021	0.002	0.007	0.008	0.022	0.009
MINOR	0.000	0.000	0.006	0.001	0.000	0.001	0.001	0.004	0.000	0.001	0.001
SCTR	0.486	0.303	0.120	0.151	0.165	0.291	0.095	0.436	0.196	0.477	0.272
NADEV	0.001	0.105	0.014	0.004	0.000	0.011	0.004	0.017	0.000	0.017	0.018
NDDEV	0.001	0.102	0.012	0.001	0.000	0.010	0.004	0.004	0.000	0.014	0.012
NCOMM	0.001	0.105	0.014	0.004	0.000	0.011	0.004	0.017	0.000	0.017	0.018
NSCTR	0.002	0.131	0.019	0.007	0.000	0.020	0.000	0.011	0.000	0.014	0.034
OEXP	0.128	0.287	0.082	0.061	0.074	0.167	0.004	0.131	0.095	0.132	0.161
EXP	0.126	0.285	0.083	0.067	0.081	0.168	0.000	0.132	0.056	0.135	0.163
ND	0.003	0.027	0.008	0.009	0.019	0.006	0.000	0.007	0.038	0.004	0.033
Entropy	0.008	0.124	0.012	0.003	0.034	0.009	0.000	0.038	0.029	0.021	0.039
LA	0.064	0.131	0.041	0.036	0.056	0.041	0.014	0.162	0.079	0.121	0.046
LD	0.011	0.043	0.006	0.017	0.028	0.002	0.007	0.123	0.038	0.069	0.012
LT	0.029	0.291	0.006	0.003	0.040	0.028	0.008	0.035	0.000	0.038	0.026
AGE	0.304	0.309	0.025	0.015	0.079	0.100	0.027	0.215	0.000	0.211	0.171
NUC	0.000	0.036	0.003	0.007	0.000	0.003	0.007	0.001	0.000	0.003	0.002
CEXP	0.006	0.040	0.007	0.003	0.013	0.002	0.013	0.015	0.055	0.006	0.004
REXP	0.004	0.020	0.000	0.002	0.013	0.002	0.005	0.042	0.011	0.007	0.004
SEXP	0.008	0.156	0.013	0.006	0.025	0.026	0.004	0.013	0.061	0.014	0.040

SCRT (i.e., number of packages modified by the commit) is the factor that provides the highest contribution to the model, consistently across all the considered software projects (this is in line with previous results reeporting that non-focused modifications tend to have an adverse effect on source code quality [91]). EXP and OEXP (i.e., the experience of the author, measured in terms of authored commits and lines) follow as the second and third highest contributors, in all but few projects (e.g., Accumulo and JDeodorant). Finally, worth mentioning is AGE (i.e., the time span between the current commit and the previous one), which is a high contributing factor for several projects.

Overall, this analysis let emerge that defective commits are more likely the result of changes that are scattered across multiple packages, done by authors not as expert of the project, and close to each other in time. Our results are in partial agreement with the findings by Kamei et al. [59]: indeed, only the experience of the committer is a powerful predictor in both traditional and fine-grained just-in-time defect prediction.

Result 3: Factors computing how much a change is scattered across packages, the expertise of the commit author, and the frequency of changes are those that provide the highest contribution to the prediction model.

3.4.4 RQ4. How much effort can be saved using a fine-grained just-in-time defect prediction model with respect to a standard just-in-time model?

Table 3.8: Results for RQ4: The area under the effort curve (higher values indicate that less effort is required to find more defects), by prediction models and their differences in percentage point (p.p.).

Systems	Area under the effort curve			Area difference (in p.p.): Ours vs. ..	
	Optimal	Ours	Kamei et al.	.. Optimal	.. Kamei et al.
Accumulo	90%	64%	46%	-26	18
Angular-js	84%	63%	49%	-21	14
Bugzilla	77%	51%	52%	-26	-1
Gerrit	78%	55%	51%	-23	4
Gimp	78%	61%	52%	-17	9
Hadoop	80%	58%	49%	-22	9
JDeodorant	79%	62%	50%	-17	12
Jetty	77%	54%	51%	-23	3
JRuby	86%	41%	52%	-45	-9
OpenJPA	84%	69%	52%	-15	17
Mean	81%	58%	50%	-24	8

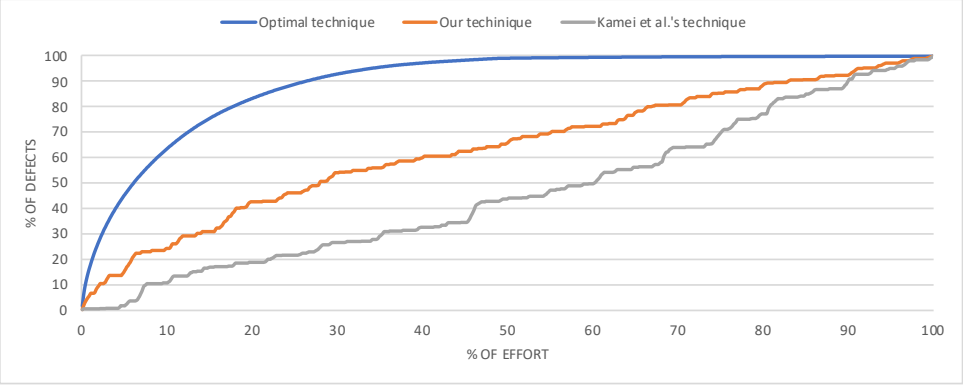

Figure 3.2: Results achieved for Accumulo.

This analysis is intended to provide evidence on the effort developers can save using our model to guide the inspection of commits for defects. We consider the state of the art *just-in-time* model (i.e., the one devised by Kamei et al. [59]) and the optimal results for comparison. Figures 3.2 to 3.11 plot the effort-based cumulative lift charts of the experimented techniques across the different projects and Table 3.8 summarizes these results in terms of the area under the curves and their differences, by project. A higher value measured for the area under the curve indicates that the model is able to pinpoint the defective files earlier, thus the developer spends less effort compared to a model with a lower value. On average, our model achieves better results than the technique of Kamei

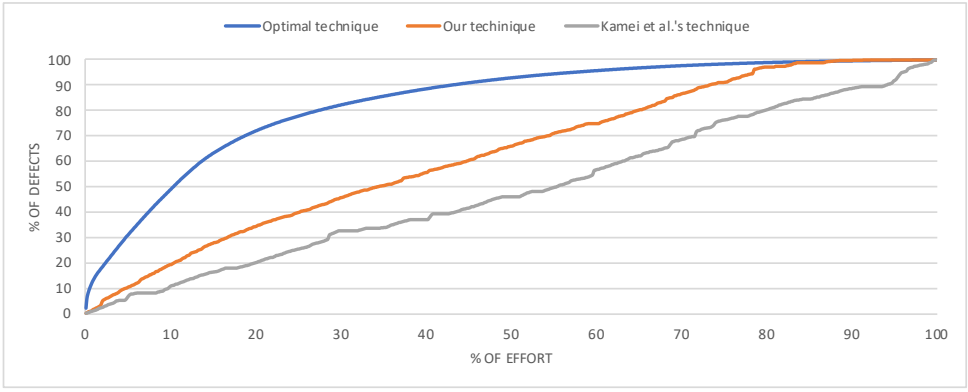

Figure 3.3: Results achieved for Angular-js.

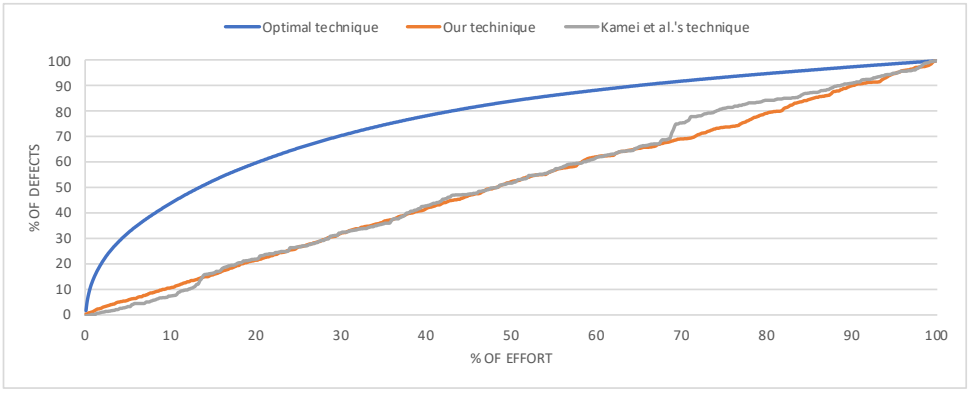

Figure 3.4: Results achieved for Bugzilla.

et al. [59], with an average increase in the area of 8% (Table 3.8). In only two systems the results are lower for our model: Bugzilla and JRuby (i.e.,-1 and -9 percentage points, respectively). Therefore, our technique seems to represents a more viable solution for predicting defects at commit-level.

If we consider the differences between our technique and the optimal model, the difference in the two areas is -24 percentage points on average. This means that, as expected, the optimal model outperforms ours. Nonetheless, we can also see that the difference is closer when considering a reduced effort budget, i.e., in cases where developers have limited time to dedicate to defect fixing activities. For example, let consider a hypothetical limited budget of 10%: in this case, using our technique, it is possible to identify 10% to more than 20% of the defective files. This indicates that, at least in the first phases, our model can be considered as a valid solution to speed up the identification of defects. At the same time, we argue that more research on the topic would be needed, as there is still ample room for improvement.

Finally, it is worth remarking that the results achieved on the entire set of defective

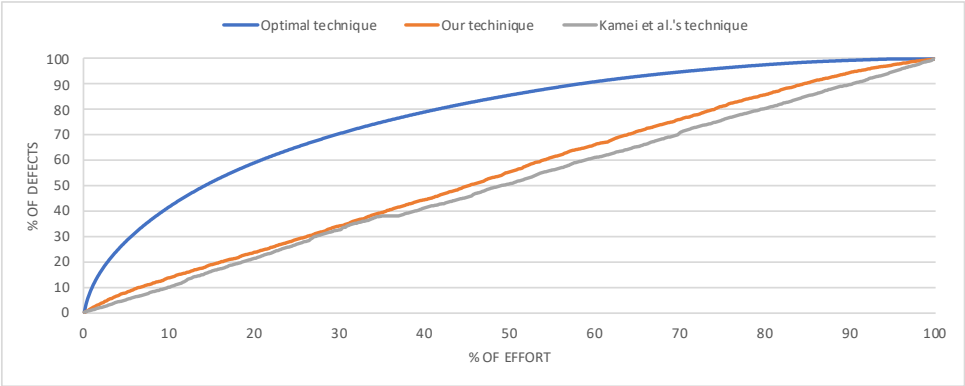

Figure 3.5: Results achieved for Gerrit.

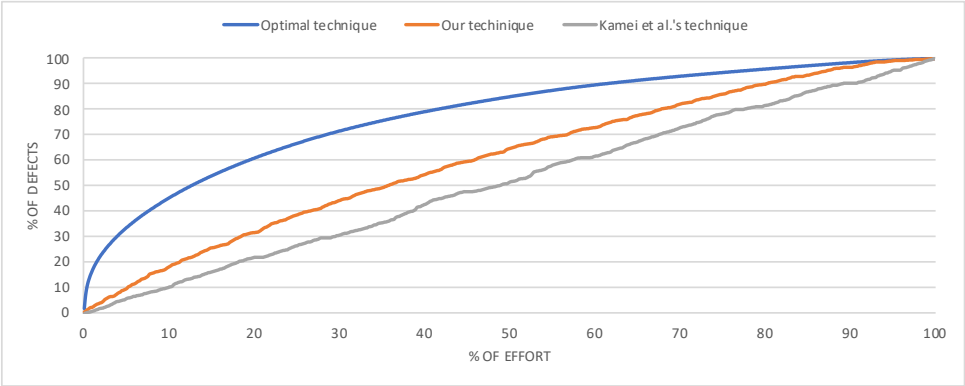

Figure 3.6: Results achieved for Gimp.

commits were also confirmed when considering partially and fully defective commits independently.

Result 4: In comparison with the state of the art, our technique represents a more effective solution to locate defects at commit-time.

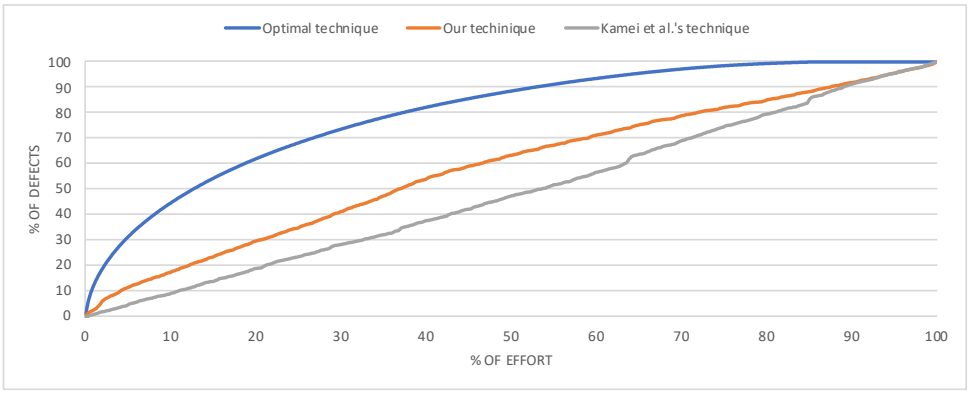

Figure 3.7: Results achieved for Hadoop.

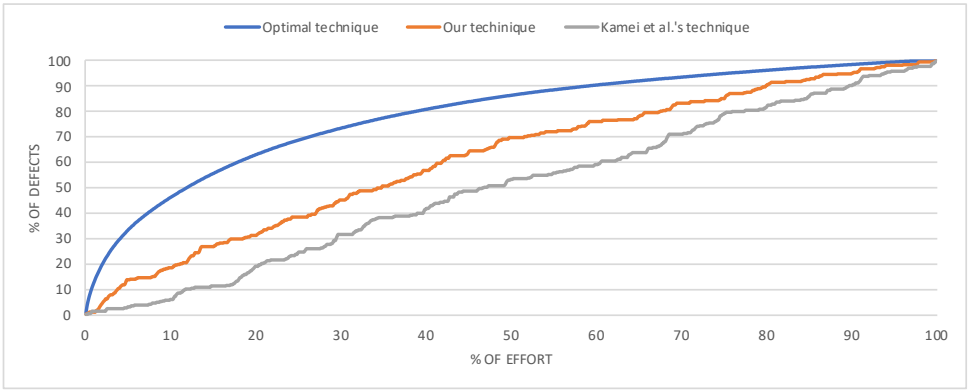

Figure 3.8: Results achieved for JDeodorant.

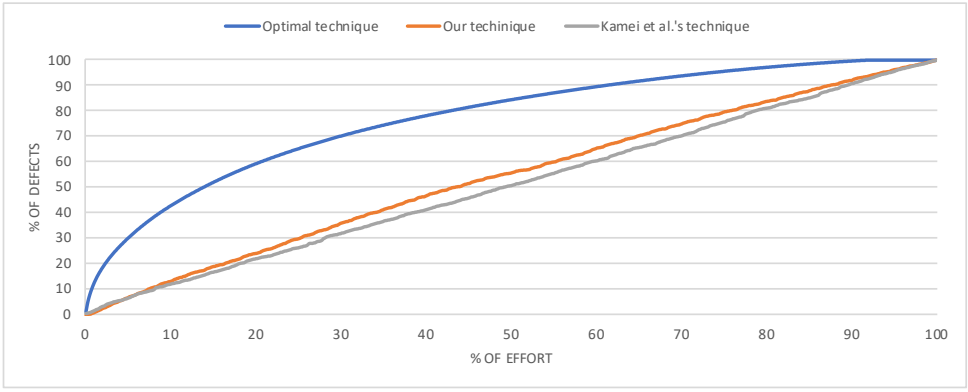

Figure 3.9: Results achieved for Jetty.

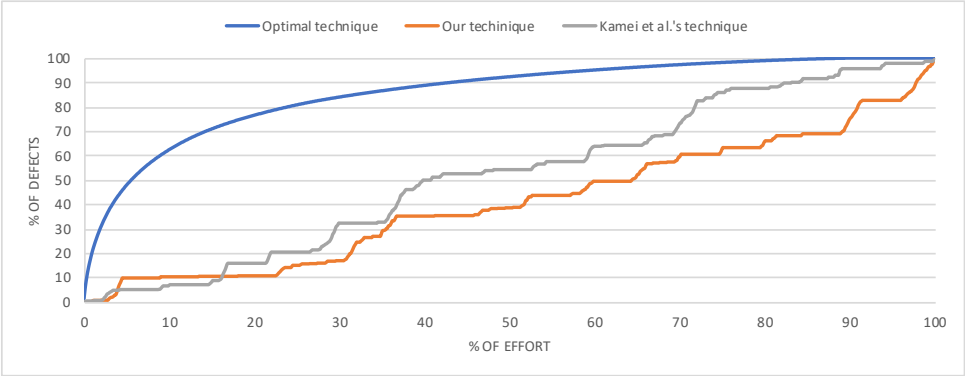

Figure 3.10: Results achieved for JRuby.

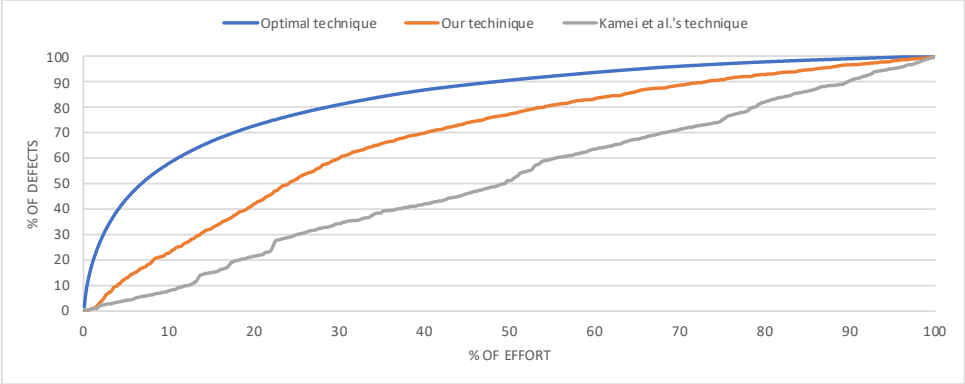

Figure 3.11: Results achieved for OpenJPA.

3.5 Conclusion

Many defect prediction models have been proposed to locate defect-prone files or commits exploiting *long-term* or *short-term* techniques, respectively. Nevertheless, such models suffer from limitations due to the coarse-grained granularity of the predictions performed, which hinder their practical applicability (e.g., in code review). For this reason, we investigated the possibility to devise a fine-grained *just-in-time* defect prediction model to locate defective files contained in a commit. Moreover, for replicability purposes, we re-implemented with a different framework the pipeline of steps needed to achieve our results and release these scripts as a further contribution of this work. The study considered 10 open-source systems written in different programming languages and having different size and scope. In total we analyzed 164k commits of which 39k defective created by 2k developers.

The main contributions made by this Chapter are:

1. An empirical validation aimed at understanding the prominence of partially defective commits, i.e., commits containing both defective and non-defective files on a set of 10 different open source software projects. The results highlight that almost half of defective commits contain both defect-inducing and defect-free changes.

2. A fine-grained *just-in-time* defect prediction model and its empirical evaluation, which showed overall performance up to 86% in terms of AUC-ROC.

3. An assessment of the *cost-effectiveness* of our model and its comparison with the standard *just-in-time* model proposed by Kamei et al. [59], with evidence that our model is more cost-effective.

4. An online appendix [82] with the scripts and data to reproduce the analyses mentioned in the chapter and enable the same analysis on other projects.

Based on the results, our future agenda includes the replication of our study on a larger set of systems, possibly performing an in-depth study in an industrial context. At the same time, future studies can be designed and conducted to investigate (i) the role of other independent variables, e.g., those reported by McIntosh et al. [168], on the performance of fine-grained defect prediction, (ii) the model in the context of cross-project defect prediction, and (iii) the benefits provided by the usage of personalized defect prediction [162, 163] as well as more sophisticated ensemble techniques [192]. Moreover, we plan to evaluate the extent to which standard just-in-time approaches working at commit-level can be combined with the fine-grained solution we proposed, e.g., through a multi-stage classification process where the defective commits are identified first and then the specific defective files are detected. Furthermore, the effectiveness of our model should be evaluated in-field, through a controlled study with practitioners to incorporate in our model some of the guidelines suggested by Lewis et al. [24] to make defect prediction more actionable in practice and support human activities (e.g., by introducing a graphical user interface supporting code reviewers when diagnosing defect-prone code components).

4

Classifying code comments in Java software systems

Code comments are a key software component containing information about the underlying implementation. Several studies have shown that code comments enhance the readability of the code. Nevertheless, not all the comments have the same goal and target audience. In this paper, we investigate how 14 *diverse Java open and closed source software projects use code comments, with the aim of understanding their purpose. Through our analysis, we produce a taxonomy of source code comments; subsequently, we investigate how often each category occur by manually classifying more than 40,000 lines of code comments from the aforementioned projects. In addition, we investigate how to automatically classify code comments at line level into our taxonomy using machine learning; initial results are promising and suggest that an accurate classification is within reach, even when training the machine learner on projects different than the target one. Data and Materials [https://doi.org/10.5281/zenodo.2628361].*

This chapter is partly based on

- *L. Pascarella, A. Bacchelli. Classifying code comments in Java open-source software systems, MSR'17* [193],
- *L. Pascarella. Classifying code comments in Java Mobile Applications, MOBILESoft'18 (Student Research Competition)* [194], and
- *L. Pascarella, M. Bruntink, A. Bacchelli. Classifying code comments in Java software systems , EMSE'19* [195],

4.1 Introduction

While writing and reading source code, software engineers routinely introduce code comments [196]. Several researchers investigated the usefulness of these comments, showing that thoroughly commented code is more readable and maintainable. For example, Woodfield et al. conducted one of the first experiments demonstrating that code comments improve program readability [197], then Tenny et al. confirmed these results with more experiments [198, 199]. Hartzman et al. investigated the economical maintenance of large software products showing that comments are crucial for maintenance [200]. Jiang et al. found that comments that are not aligned with the annotated functions confuse authors of future code changes [201]. Overall, given these results, having abundant comments in the source code is a recognized good practice [202]. Accordingly, researchers proposed to evaluate code quality with a metric based on code/comment ratio [203, 204].

Nevertheless, not all the comments are the same. This is evident, for example, by glancing through the comments in a source code file[1] from the Java Apache Hadoop Framework [205]. In fact, we see that some comments target end-user programmers (e.g., Javadoc), while others target internal developers (e.g., *inline* comments); moreover, each comment is used for a different purpose, such as providing the implementation rationale, separating logical blocks, and adding reminders; finally, the interpretation of a comment also depends on its position with respect to the source code. Defining a taxonomy of the source code comments is still an open research problem.

Haouari et al. [206] and Steidl et al. [207] presented the earliest and most significant results in comments' classification. Haouari et al. investigated developers' commenting habits, focusing on the position of comments with respect to source code and proposing an initial taxonomy that includes four high-level categories [206]; Steidl et al. proposed a semi-automated approach for the quantitative and qualitative evaluation of comment quality, based on classifying comments in seven high-level categories [207]. In spite of the innovative techniques they proposed to understand developers' commenting habits and to assess comments' quality, the classification of comments was not in their primary focus.

In the work presented in this article, we focus on increasing our empirical understanding of the types of comments that developers write in source code files. This is a key step to guide future research on the topic. Moreover, this increased understanding has the potential to (1) improve current quality analysis approaches that are restricted to the comment ratio metric only [203, 204] and to (2) strengthen the reliability of mining approaches that use comments as input (e.g., [208, 209]).

To this aim, we conducted an in-depth analysis of the comments in the Java source code files of six major OSS systems and eight industrial projects. We set up our study as an exploratory investigation. We started without hypotheses regarding the content of source code comments, with the aim of discovering the comments' purposes and roles, their format, and their frequency. To this end, we (1) conducted three iterative content analysis sessions (involving four researchers) over 50 source files including about 250 comment blocks to define an initial taxonomy of code comments, (2) validated the taxonomy externally with 3 developers, (3) inspected 2,000 open source and 4,000 closed source code files and manually classified (using a new application we devised for this purpose) over 24,000 comment blocks

[1] https://tinyurl.com/zqeqgpq

comprising more than 40,000 lines, (4) used the resulting dataset to evaluate how effectively comments can be automatically classified, and (5) investigated how many comments from an unseen project should be manually classified to improve the performance of an automatic classification approach trained on other projects.

Our results show that developers write comments with a large variety of different meanings and that this should be taken into account by analyses and techniques that rely on code comments. The most prominent category of comments summarizes the purpose of the code, confirming the importance of research related to automatically creating these type of comments. Finally, our automated classification approach, based on supervised algorithms, reaches promising initial results, even when training on software projects that are different than the target project.

4.2 Motivating Example

Listing 4.1 shows an example Java source code file that contains both code and comments. In a well-documented file, comments help the reader with a number of tasks, such as understanding the code, knowing the choices and rationale of authors, and finding additional references. When developers perform software maintenance, the aforementioned tasks become mandatory steps that practitioners should attend. The fluency in performing maintenance tasks depends on the quality of both code and comments. When comments are omitted, much depends on the ability of developers and the complexity of the code; when well-written comments are present, the maintenance could be simplified.

4.2.1 Code/comment ratio to measure software maintainability

When developers want to estimate the maintainability of code, one of the simplest solutions is to compute the code/comment ratio, as proposed by Garcia et al. [204]. By evaluating the aforementioned metric in the snippet in Listing 4.1, we find an overall indicator of quality, which—however—is inaccurate. The inaccuracy arises from the fact that this metric considers only one kind of comment. More precisely, Garcia et al. focus only on the presence or absence of comments, omitting the possibility of use comments with different benefits for different end-users. The previous sample of code represents a case where the author used comments for different purposes. The comment on line 31 represents a note that developers use to remember an activity, an improvement, or a fix. On line 20 the author marks his contribution on the file. Both these two comments represent real cases where the presence of comments increases the code/comment ratio without any real effect on code readability or maintanability. This situation hinders the validity of this kind of metric and indicates the need for a more accurate approach to tackle the problem.

4.2.2 An existing taxonomy of source code comments

A great source of inspiration for our work comes from Steidl et al. who presented a first detailed approach for evaluating comment quality [207]. One of the key steps of their approach is to first automatically categorize the comments to differentiate between different comment types. They define a preliminary taxonomy of comments that comprises 7 high-level categories: Copyright, Header, Member, Inline, Section, Code, and Task. They

Figure 4.1: Example of Java file.

```
1  public class STSubscriptExpression extends STExpression {
2
3      private static CSpellingService fInstance;
4
5      /**
6       * Returns the created expression, or null in case of error.
7       * @deprecated Replaced by {@link #getExpression()}
8       */
9      @Deprecated
10     public STExpression getSubscriptExpression(){
11         if (fInstance == null) {
12             fInstance = new Expression(ConsoleEditors.getPreferenceStore());
13         }
14         return fInstance;
15     }
16
17     /**
18      * Handle terminated sub-launch
19      * @param launch a terminable launch object.
20      * @author Jesse MC Wilson
21      */
22     private void STLaunchTerminated(ILaunch launch) {
23         // See com.vaadin.data.query.QueryDelegate#getPrimaryKeyColumns
24         if (this == launch)
25             return;
26         // Remove sub launch, keeping the processes of the terminated launch to
27         // show the association and to keep the console content accessible
28         if (subLaunches.remove(launch) != null) {
29             // terminate ourselves if this is the last sub launch
30             if (subLaunches.size() == 0) {
31                 // TODO: Check the possibility to exclude it
32                 //monitor.exlude();
33                 monitor.subTask("Terminated"); //$NON-NLS-1$
34                 fTerminated = true;
35                 fireTerminate();
36                 // %%%
37             }
38         }
39     }
40 }
```

provide evidence that their quality model, based on this taxonomy, provides important insights on documentation quality and can reveal quality defects in practice.

The study of Steidl et al. demonstrates the importance of treating comments in a way

that suits their different categories. However, the creation of the taxonomy was not the focus of their work, as also witnessed by the few details given about the process that led to its creation. In fact, we found a number of cases in which the categories did not provide adequate information or did not differentiate the type of comments enough to obtain a clear understanding. To detail this, we consider three examples from Listing 4.1:

Member category. Lines 5, 6, 7 and 8 correspond to the Member category in the taxonomy by Steidl et al. In fact, Member comments describe the features of a method or field being located near to definition [207]. Nevertheless, we see that the function of line 6 differs from that of line 7; the former summarizes the purpose of the method, the latter gives notice about replacing the usage of the method with an alternative. By classifying these two lines together, one would lose this important difference.

IDE directives. Lines 33 does not belong to any explicit category in the taxonomy by Steidl et al. In this case, the target is not a developer, but the Integrated Development Environment (IDE). Similarly, line 23 does not have a category in the taxonomy by Steidl et al., but it is a possibly important external reference to read for more details.

Unknown. Line 36 represents a case of a comment that should be disregarded from any further analysis. Since it does not separate parts, the Section would not apply and an automated classification approach would try to wrongly assign it to one of the other categories. The taxonomy by Steidl et al. does not consider *unknown* as a category.

With our work, we specifically focus on devising an empirically grounded, fine-grained classification of comments that expands on the previous initial efforts by Steidl et al. Our aim is to get a comprehensive view of the comments, by focusing on the purpose of the comments written by developers. Besides improving our scientific understanding of this type of artifacts, we expect this work to be also beneficial, for example, to the effectiveness of the quality model proposed by Steidl et al. and other approaches relying on mining and analyzing code comments (e.g., [203, 208, 209]).

4.3 Methodology

This section defines the overall goal of our study, motivates our research questions, and outlines our research method.

4.3.1 Research Questions

The ultimate goal of this study is to understand and classify the primary purpose of code comments written by software developers. In fact, past research showed evidence that comments provide practitioners with a great assistance during maintenance and future development, but not all the comments are the same or bring the same value.

We started by analyzing past literature searching for similar efforts on analysis of code comments. We observed that a few studies completed a taxonomy of comments, in a preliminary fashion. Indeed, most of past work focuses on the impact of comments on

software development processes such as code understanding, maintenance, or code review and the classification of comments is only treated as a side outcome (e.g., [198, 199]).

RQ$_1$. How can code comments be categorized?

Given the importance of comments in software development, the natural next step is to apply the resulting taxonomy and investigate on the primary use of comments. Therefore, we investigate whether some classes of comments are predominant and whether patterns across different projects or domains (e.g., open source and industrial systems) exist. This investigation is reflected in our second research question:

RQ$_2$. How often does each category occur in OSS and industrial projects?

Subsequently, we investigate to what extent an automated approach can classify unseen code comments according to the taxonomy defined in RQ1. An accurate automated classification mechanism is the first essential step in using the taxonomy to mine information from large-scale projects and to improve existing approaches that rely on code comments. This leads to our third research question:

RQ$_3$. How effective is an automated approach, based on machine learning, in classifying code comments in OSS and industrial projects?

Finally, we expect that practitioners or researchers could benefit by applying our machine learning algorithm to an unseen real project. For this reason, we investigate how much the performance are improved by manually classifying an increasing number of comments in a new project and providing this information to our machine learning algorithm. This evaluation leads to our last research question:

RQ$_4$. How does the performance of an automated approach improve by adding classified comments from the system under classification?

4.3.2 Selection of subject systems

To conduct our analysis, we focused on a single programming language (i.e., Java, one of the most popular programming languages [117]) and on projects that are either developed in an open source setting or in an industrial one.

OSS context: Subject systems. We selected six heterogeneous software systems: Apache Spark [210], Eclipse CDT, Google Guava, Apache Hadoop, Google Guice, and Vaadin. They are all open source projects and the history of the changes are controlled with GIT version control system. Table 4.1 details the selected systems. We select unrelated projects emerging from the context of different four software ecosystems (i.e., Apache,

Table 4.1: Details of the selected open source systems.

Project	Java source lines			Commits	Contributors	Sample sets		
	Code	Comment	Ratio			Files	Blocks of comments	Ratio
Apache Spark	753k	287k	38%	38k	1,351	61	465	7.7%
Eclipse CDT	1,239k	466k	38%	26k	211	799	6,009	7.5%
Google Guava	252k	88k	35%	4k	185	158	1,100	6.9%
Apache Hadoop	1,258k	396k	31%	15k	171	672	4,228	6.3%
Google Guice	9k	5k	56%	2k	32	59	718	12.1%
Vaadin	2,643k	1,101k	42%	91k	726	401	3,340	8.3%
Overall	6M	2.3M	38%	176k	2.7k	2k	16k	8%

Google, Eclipse, and Vaadin); the development environment, the number of contributors, and the project size are different: Our aim is to increase the diversity of comments that we find in our dataset.

Industrial context: Subject systems. We also include heterogeneous industrial software projects, which are clients of the company in which the second author works. Table 4.2 reports the anonymized characteristics of such projects, respecting their non-disclosure agreements.

Table 4.2: Details of the anonymize industrial systems.

Project	Java source lines			Sample sets		
	Code	Comments	Ratio	Files	Blocks of comments	Ratio
P1	478k	12k	2.5%	159	1,386	8.7%
P2	69k	0.1k	0.1%	236	102	0.4%
P3	2,044k	19k	1.0%	503	1,673	3.3%
P4	1,065k	5k	0.5%	761	1,223	1.6%
P5	1,026k	7k	0.6%	506	1,073	2.1%
P6	3,088k	5k	0.2%	503	273	0.5%
P7	1,173k	13k	1.1%	290	1,058	3.6%
P8	1,208k	4k	0.3%	1,042	1,179	1.1%
Overall	10M	65k	0.7%	4k	8k	2%

4.3.3 RQ_1. Categorizing code comments

To answer our first research question, we (1) defined the comment granularity we consider, we (2) conducted three iterative *content analysis sessions* [211] involving four software engineering researchers with at least three years of programming experience, and we (3) validated our categories involving other three professional developers.

Comment granularity. Java offers three alternative ways to comment source code: *inline* comments, *multi-line* comments, and *JavaDoc* comments (which is a special case of multi-line comments). A comment (especially multi-line ones) may contain difference pieces of information with different purposes, hence belonging to different categories. Moreover,

a comment may be a natural language word or an arbitrary sequence of characters that, for example, represent a delimiter or a directive for the preprocessor. For this reason, we conducted our manual classification at *character level*. The user specifies the starting and ending character of each comment block and its classification. For example, the user could categorize two parts of a single inline comment into two different classes. By choosing a fine-grained granularity at *character level* users are responsible for identifying comment's delimiters (i.e., the text is not automatically split into tokens). Even if this choice may complicate the user's work, this flexibility, chosen during the manual classification, allowed us both to define the taxonomy precisely and to have a basis to decide the appropriate comment granularity for the automatic classification, i.e., line granularity (see Section 4.3.5 – 'Classification granularity').

Definition phase. This phase involved four researchers in software engineering (three Ph.D. candidates and one faculty member). Two of these researchers are authors of this paper. In the first iteration, we started choosing six appropriate OSS projects (reported in Table 4.1) and sampling 35 files with a large variety of code comments. Subsequently, together we analyzed all source code and comments. During this analysis we could define some obvious categories and left undecided some comments; this resulted in the first draft taxonomy defining temporary category names. In the course of the second phase, we first conducted an individual work analyzing 10 new files, in order to check or suggest improvements to the previous taxonomy, then we gathered together to discuss the findings. The second phase resulted in a validation of some clusters in our draft and the redefinitions of others. The third phase was conducted in team and we analyzed five files that were previously unseen. During this session, we completed the final draft of our taxonomy verifying that each kind of comments we encountered was covered by our definitions and those overlapping categories were absent.
Through this iterative process, we defined a hierarchical taxonomy with two layers. The top layer consists of six categories and the inner layer consists of 16 subcategories.

Validation phase. We validated the resulting taxonomy externally with three professional developers who had three to five years of Java programming experience and were not involved in the writing of this work. We conducted one session with each developer. At the beginning of the session, the developer received a printed copy of the description of the comment categories in our taxonomy (similar to the explanation we provide in Section 4.4.1) and was allowed to read through it and ask questions to the researcher guiding the session. Afterwards, each developer was required to login into ComMean (a web application, described in Section 4.3.4) and classify—according to the provided taxonomy—each piece of comment (i.e., by arbitrary specifying the sequence of adjacent characters that identify words, lines, or blocks belonging to the same category) in three Java source code files (the same files have been used for all the developers) that contained a total of 138 different lines of comments. During the classification, the researcher was not in the experiment room, but the printed taxonomy could be consulted. At the end of the session, the guiding researcher came back to the experiment room and asked the participant to comment on the taxonomy and the classification task. At the end of all the three sessions, we compared the differences (if any) among the classifications that the developers produced.

All the participants found the categories to be clear and the task to be feasible; however,

they also reported the need for consulting the printed taxonomy several times during the session to make sure that their choice was in line with the description of the category. Although they observed that the categories were clear, the analysis of the three sets of answers showed differences. We computed the inter-rater reliability by using Fleiss' kappa value [212] and found a corresponding k value above 0.9 (i.e., very good) for the three raters and the 138 lines they classified. We individually analyzed each case of disagreement by asking the participants to re-evaluate their choices after better extrapolating the context. Following this approach, the annotators converged on a common decision.

Exhaustiveness by cross-license validation. As a side effect of investigating the second industrial dataset (which covers different commercial licenses and market strategies) for RQ_2, we cross-validated the exhaustiveness of the categories and subcategories present in our taxonomy in a different context.

In fact, we directly applied our definitions to the comments of 8 industrial projects producing a manually classified set of 8,000 block of comments. During the labeling phase, we adopted all definitions present in the proposed codebook and even though we found a significant different distribution of each category, we found that such definitions are sufficiently exhaustive to cover all types of comment present also in the new set of projects.

This second validation provided additional corroborating evidence that the proposed taxonomy (initially generated involving 4 software engineering researchers, which iteratively observed the content of a sample of 50 Java open source files) fits all the comments that we encountered in both open and closed source projects, adequately.

4.3.4 A dataset of categorized code comments

To answer the second research question about the frequencies of each category, we needed a statistically significant set of code comments classified accordingly to the taxonomy produced as an answer to RQ_1. Since the classification had to be done manually, we relied on random sampling to produce a statistically significant set of code comments. Combining the sets of OSS and industrial project, we classified comments in a total of 6,000 Java files from six open source projects and eight industrial projects. Our aim is to give a representative overview of how developers use comments and how these comments are distributed.

To reach this number of files for which we manually annotate the comments, we adopted two slightly different sampling strategies for OSS and industrial projects; we detail these strategies in the following.

OSS Projects: Sampling files. To establish the size of statistically significant sample sets for our manual classification, we used as a unit the number of files, rather than the number of comments: This results in the creation of a sample set that gives an additional overview of how comments are distributed in a system. We established the size (n) of such set with the following formula ([213], pp. 328-331):

$$n = \frac{N \cdot \hat{p}\hat{q}(z_{\alpha/2})^2}{(N-1)E^2 + \hat{p}\hat{q}(z_{\alpha/2})^2}$$

The size has been chosen to allow the simple random sampling without replacement. In the formula, $\hat{p}$ is a value between 0 and 1 that represents the proportion of files containing

a given category of code comment, while $\hat{q}$ is the proportion of files not containing such kind of comment (i.e., $\hat{q} = 1 - \hat{p}$). Since the *a-priori* proportion of $\hat{p}$ is not known, we consider the worst case scenario where $\hat{p} \cdot \hat{q} = 0.25$. In addition, considering we are dealing with a small population (i.e., 557 Java files for Google Guice project) we use the finite population correction factor to take into account their size (N). We sample to reach a confidence level of 95% and error (E) of 5% (i.e., if a specific comment is present in f% of the files in the sample set, we are 95% confident it will be in $f\% \pm 5\%$ files of our population). The suggested value for the sample set is 1,925 files. In addition, since we split the sample sets in two parts with an overlapped chunk for validation, we finally sampled 2,000 files. This value does not change significantly the error level that remains close to 5%. This choice only validates the quality of our dataset as a representation of the overall population: It is not related to the *precision* and *recall* values presented later, which are actual values based on manually analyzed elements.

Industrial projects: Sampling files. As done in the OSS case we selected a statistically significant sample of files belonging to industrial projects. We relied on simple random sampling without replacement to select a sufficient amount of files representative of the eight industrial projects that we considered in this study. According to the formula [213] used for the sampling in the OSS context, we defined a sample of 2,000 Java files with a confidence level of 95% and error of 5% (i.e., if a specific comment is present in f% of the files in the sample set, we are 95% confident it will be in $f\% \pm 5\%$ files of our population). Since we expected a similar workload for both domains we started with the same number of files. However, during the inspection, we found out that we still had resources to conduct a deep investigation, because we found a less number of comments per file. Therefore, we decided to double the number of files to inspect manually. This condition led to the creation of a sample set of 4,000 Java files for the industrial study.

Manual classification. Once the sample of files with comments was selected, each of them had to be manually classified according to our taxonomy. For the manual classification, we rely on the human ability to understand and categorize written text expressed in natural language, specifically, code comments. To support the users during this tedious works that may be error-prone due to the repetitiveness of the task (especially for large datasets), we developed a web application named ComMean to conduct the classification. ComMean (i) shows one file at a time, (ii) allows the user to save the current progress for further inspections, and (iii) highlights the classified instances with different colors and opacity. During the inspection, the user can arbitrarily choose the selection granularity (e.g., s/he can select a part of a line, an entire line, or a block composed of multiple lines) by selecting the starting and ending characters. For the given selection, the user can assign a label corresponding to one of the categories in our taxonomy.

The first and last authors of this paper manually inspected the sample set composed of 2,000 open source files and 4,000 industrial files. One author analyzed 100% of these files, while another analyzed a random, overlapping subset comprising 10% of the files. These overlapped files were used to verify their agreement, which, similarly to the external validation of the taxonomy with professional developers (Section 3.3), highlighted only negligible differences. More precisely, every participant independently read and labeled his own set of comments. If labels matched we accepted those cases as resolved, otherwise, we

discussed each unmatched case. During the discussion, we evaluated the reasons behind a certain decision. Then, we came to a conclusion of choosing a single label. In most cases, the opinions differed due to the ambiguous nature of the comments. In these cases, we analyzed the context and tried a second run. Finally, we resolved these comments by carefully analyzing comments and the code context.

This large-scale categorization helped giving an indication of the exhaustiveness of the taxonomy created in RQ_1 with respect to the comments present in our sample: None of the annotators felt that comments, or parts of the comments, should have been classified by creating a new category. Although promising, this finding is applicable only to our dataset and its generalizability to other contexts should be assessed in future studies. The annotations referring to open source projects as well as ComMean are publicly available [214]; the dataset constructed with industrial data cannot be made public due to non-disclosure agreements.

4.3.5 Automated classification of source code comments

In the third research question, we set to investigate to what extent and with which accuracy source code comments can be automatically categorized according to the taxonomy resulting from the answer to RQ_1 (Section 4.4.1). Employing sophisticated classification techniques (e.g., based on deep learning approaches [215]) to accomplish this task goes beyond the scope of the current work. Our aim is twofold: (1) Verifying whether it is feasible to create an automatic classification approach that provides fair accuracy and (2) defining a reasonable baseline against which future methods can be tested.

Classification granularity. We set the automated classification to work *at line level*. In fact, from our manual classification, we found several blocks of comments that had to be split and classified into different categories (similarly to the block defined in the lines 5–8 in Listing 4.1) and in the vast majority of the cases (96%), the split was at line level. In only less than 4% of the cases, one line had to be classified into more than one category. In these cases, we replicated the line in our dataset for each of the assigned categories, to get a lower bound on the effectiveness in these cases.

Classification technique. Having created a reasonably large dataset to answer RQ_2 (it comprises more than 15,000 comment blocks totaling over 30,000 lines in OSS and up to 8,000 comment blocks that correspond to 10,000 lines in industrial systems), we employ *supervised* machine learning [126] to build the automated classification approach. This kind of machine learning uses a pre-classified set of samples to infer the classification function. In particular, we tested two different classes of supervised classifiers: (1) *probabilistic classifiers*, such as naive Bayes or naive Bayes Multinominal, and (2) *decision tree algorithms*, such as J48 and Random Forest. These classes make different assumptions on the underlying data, as well as have different advantages and drawbacks in terms of execution speed and overfitting.

Data balancing. Chawla et al. study the effect of an approach to limit the problem of data imbalance named Synthetic Minority Over-sampling Technique (SMOTE) [121]. Specifically, their method tries to over-sample the minority occurrences and under-sample the majority classes to achieve better performance in a classification task. Data imbalance, in fact, is a frequent issue in classification problems occurring when the number of instances that refer to frequent classes is higher than uncommon instances (in our case Discarded,

Under development, and Style & IDE classes). To ensure that our results would not have been biased by confounding factors, such as data imbalance [121], we adopt the SMOTE package available in Weka toolkit[2] with the aim of balancing our training sets. In addition, we relied on the work of O'brien [122] to mitigate the issues that can derive from the multi-collinearity of independent variables. To this purpose, we compared the results of different classification techniques. Specifically, in our study, we address this problem by applying the Random Over-Sampling algorithm [123] implemented as a supervised filter in the Weka toolkit. The filter re-weights the instances in the dataset to give them the same total weight for each class maintaining unchanged the total sum of weights across all instances.

Classification evaluation. To evaluate the effectiveness of our automated technique to classification code comments into our taxonomy, we use two well known Information Retrieval (IR) metrics for the quality of results [216], namely *precision* and *recall*:

$$Precision = \frac{|TP|}{|TP + FP|}$$

$$Recall = \frac{|TP|}{|TP + FN|}$$

TP, FP, and FN are based on the following definitions:

- True Positives (TP): elements that are correctly retrieved by the approach under analysis (i.e., comments categorized in accord to annotators)
- False Positives (FP): elements that are wrongly classified by the approach under analysis (i.e., comments categorized in a different way by the oracle)
- False Negatives (FN): elements that are not retrieved by the approach under analysis (i.e., comments present only in the oracle)

The union of TP and FN constitutes the set of correct classifications for a given category (or overall) present in the benchmark, while the union of TP and FP constitutes the set of comments as classified by the used approach. In other words, *precision* represents the fraction of the comments that are correctly classified into a given category, while *recall* represents the fraction of *relevant comments* in that category, where the *relevant comments* definition includes both true positive and false negative.

Effort/performance estimation. With the fourth research question, we aim at estimating how many new code comments a researcher or developers should manually classify from an unseen project, in order to obtain higher performance when using our classification algorithm on such a project. Since the presence of classified comments from a project in the training set positively influences the performance of the classification algorithm on other comments from the same project, we want to measure how many instances are required in the training set to reach the knee of a hypothetical effort/performance curve.

To this aim, we quantify the exact extent to which each new manually classified block of comments contributes to produce better results, if any. We consider the project for which we obtained the worst results in cross-project validation when answering RQ_3; then,

[2]https://www.cs.waikato.ac.nz/ml/weka/

we progressively add to the training set a fixed number of randomly selected comments belonging to the subject project and for each iteration we measure the performance of the model.

This evaluation starts from a cross-project setting (i.e., we train only on different projects and test on an unseen one) and slowly gets to a within-project validation setting (i.e., we train not only from different projects, but also from comments in the project that we are going to test with on unseen comments). We investigate what is among the lowest reasonable amount of comments that one should classify to get as close as possible to a full within-project setting.

4.3.6 Threats to validity

Taxonomy validity. To ensure that the comments categories emerged from our content analysis sessions were clear and accurate, and to evaluate whether our taxonomy provides an exhaustive and effective way to organize source code comments, we conducted a validation session that involved three experienced developers (see Section 4.3.3) external to content analysis sessions. These software engineers held an individual session on three unrelated Java source files. They found the categories to be clear and the task feasible, and the analysis of the three sets of answers showed a few minor differences. We counted the number of lines of comments classified with the same label by all participants and the number of lines of comments that at least two experts were in conflict. Finally, considering these two values we could calculate the percentage of comments that were classified with the same label by all participants. We measured that only 8% of the considered comments in the first run led to mismatches. Moreover, we individually analyzed each case by asking the participants to re-evaluate their choices after better extrapolate the context. Following that approach, the annotators converged on a common decision. In addition, we reduce the impact of human errors during the creation of the dataset by developing ComMean, a web application to assist the annotation process.

External validity. One potential criticism of a scientific study conducted on a small sample of projects is that it could deliver little knowledge. In addition, the study highlights the characteristics and distributions of 6 open source frameworks and 8 industrial projects mainly focusing on developers practices rather than end-users patterns. Historical evidence shows otherwise: Flyvbjerg gave many examples of individual cases contributing to discoveries in physics, economics, and social science [217]. To answer to our research questions, we read and inspected more than 28,000 lines of comments belonging to 2,000 open source Java files and 12,000 lines of comments belonging to 4,000 closed source Java files (see Section 4.3.4) written by more than 3,000 contributors in a total of 14 different projects (in accord to Table 1 and Table 2). We also chose projects belonging to different ecosystems and with different ment environments, number of contributors, and size of the project. To have an initial assessment of the generalizability of the approach we tested our results simulating this circumstance using the cross-project validation and cross-license validation (i.e., training on OSS systems and testing on industrial systems, and viceversa) involving both open and closed source projects. Similarly, another threat concerning the generalizability is that our taxonomy refers only to a single object-oriented programming language i.e., Java. However, since many object-oriented languages descend to common ancestor languages, many functionalities across object-oriented programming are similar and

it is reasonable to expect the same to happen for their corresponding comments. Further research can be designed to investigate whether our results hold in other programming paradigms.

After having conducted the entire manual classification and the experiment, we realized that the exact location and the surrounding context of a code comment may be a valuable source of information to extract the semantic of the comment. Unfortunately, our tool ComMean did not record this information, thus we could not investigate how the performance of a machine learner would benefit from it. Future work should take this feature into account when designing similar experiments.

Moreover, RandomForest can be prone to overfitting, thus provide results that are too optimistic. To mitigate this threat, we use different training and testing mechanisms that create conditions that should decrease this problem (e.g., within-project and cross-project).

Finally, a line of comment may have more than one meaning. We empirically found that this was the case for 4% of the inspected lines. We discarded these lines, as we considered this effect marginal, but this is a limitation of both our taxonomy and automatic classification mechanism.

4.4 Results and Analysis

In this section, we present and analyze the results of our research questions aimed at understanding what developers write in comments and with which frequency, as well as at evaluating the results of an automated classification approach and how manually classified comments from a project help improving the performance of a classifier trained on different projects.

4.4.1 RQ_1. How can code comments be categorized?

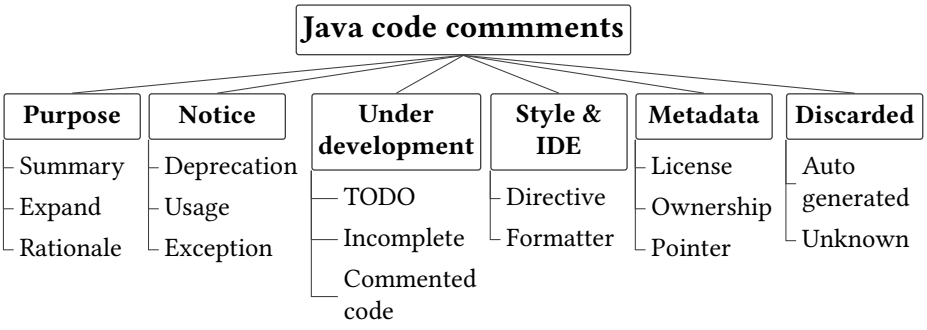

Figure 4.2: Taxonomy of code comments.

Our manual analysis led to the creation of a taxonomy of comments having a hierarchy with two layers (Section 4.3.3). The top level categories gather comments with similar overall purpose, the internal levels provide a fine-grained definition using explanatory names. Figure 4.2 outlines all categories. The top level categories are composed of 6 distinct groups and the second level categories are composed of 16 definitions. We now describe each category with the corresponding subcategories.

A. **Purpose**

The Purpose category contains the code comments used to describe the functionality of linked source code either in a shorter way than the code itself or in a more exhaustive manner. Moreover, these comments are often written in a natural language and are used to describe the purpose or the behavior of the referenced source code. The keywords *'what'*, *'how'* and *'why'* describe the actions that take place in the source code in summary, expand, and rationale groups, respectively, which are the subcategories of Purpose:

A.1 Summary: This type of comment contains a brief description of the behavior of the referenced source code. More generically, this class of comments represents answers to the question word *'what'*. Intuitively, this category incorporates comments that represent a sharp description of what the code does. Often, this kind of comments is used by developers to provide a summary that helps to understand the behavior of the code without reading it.

A.2 Expand: As with the previous category, the main purpose of reading this type of comment is to obtain a description of the associated code. In this case, the goal is to provide *more details* on the code itself. The question word *'how'* can be used to easily recognize the comments belonging to this category. Usually, these comments explain in detail the purpose of short parts of the code, such as details about a variable declaration.

A.3 Rationale: This type of comment is used to explain the rationale behind some choices, patterns, or options. The comments that answer the question *'why'* belong to that category (e.g., "Why does the code use that implementation?" or "Why did the developer use this specific option?").

B. **Notice**

The Notice category contains the comments related to the description of warning, alerts, messages, or in general, functionalities that should be used with care. It covers the description of deprecated artifacts, as well as, the adopted strategies to move to new implementations. Further, it includes the use case examples giving to developer additional advice over parameters or options. Finally, it covers examples of use cases or warnings about exceptions.

B.1 Deprecation: This type of comment contains explicit warnings used to inform the users about deprecated interface artifacts. This subcategory contains comments related to alternative methods or classes that should be used (e.g., "do not use [this]", "is it safe to use?" or "refer to: [ref]"). It also includes the description of the future or scheduled deprecation to inform the users of candidate changes. Sometimes, a tag comment such as *@version*, *@deprecated*, or *@since* is used.

B.2 Usage: This type of comment regards explicit suggestions to users that are planning to use a functionality. It combines pure natural language text with examples, use cases, snippets of code, etc. Often, the advice is preceded by a metadata mark e.g., *@usage*, *@param* or *@return*

B.3 Exception: This category describes the reasons for the occurred exception. Sometimes it contains potential suggestions to prevent the unwanted behavior or actions to do when that event arise. Some tags are used also in this case, such as *@throws* and *@exception*.

C. **Under development**

The Under development category covers the topics related to ongoing and future development. In addition, it envelopes temporary tips, notes, or suggestions that developers use during development. Sometimes informal requests of improvement or bug correction may also appear.

C.1 TODO: This type of comment regards explicit actions to be done or remarks both for the owners of the file and for other developers. It contains explicit fix notes about bugs to analyze and resolve, or already treated and fixed. Furthermore, it references to implicit TODO actions that may be potential enhancements or fixes.

C.2 Incomplete: This type comprises partial, pending or empty comment bodies. It may be introduced intentionally or accidentally by developers and left in the incomplete state for some reason. This type may be added automatically by the IDE and not filled in by the developer e.g., empty "@param" or "@return" directives.

C.3 Commented code: This category is composed of comments that contain source code commented out by developers. It envelopes functional code in a comment to try hidden features or some work in progress. Usually, this type of comment represents features under test or temporarily removed. The effect of this kind of comments is directly transposed to the program flow.

D. **Style & IDE**

The Style & IDE category contains comments that are used to logically separate the code or provide special services. These comments may be added automatically by the IDE or used to communicate with it.

D.1 Directive: This is an additional text used to communicate with the IDE. It is in form of comments to be easily skipped by the compiler and it contains text of limited meaning to human readers. These comments are often added automatically by the IDE or used by developers to change the default behavior of the IDE or compiler.

D.2 Formatter: This type of comment represents a simple solution adopted by the developers to separate the source code in logical sections. The occurrence of patterns or the repetition of symbols is a good hint at the presence of a comment in the formatter category.

E. **Metadata**

The Metadata category aims to classify comments that define meta information about the code, such as authors, license, and external references. Usually, some specific tags (e.g.,

"@author") are used to mark the developer name and its ownership. The license section provides the legal information about the source code rights or the intellectual property.

E.1 License: Generally placed on top of the file, this types of comments describes the end-user license agreement, the terms of use, the possibility to study, share and modify the related resource. Commonly, it contains only a preliminary description and some external references to the complete policy agreement.

E.2 Ownership: These comments describe the authors and the ownership with different granularity. They may address methods, classes or files. In addition, this type of comment includes credentials or external references about the developers. A special tag is often used e.g., *"@author"*.

E.3 Pointer: This types of comments contains references to linked resources. The common tags are: *"@see"*, *"@link"* and *"@url"*. Other times developers use custom references such as *"FIX #2611"* or *"BUG #82100"* that are examples of traditional external resources.

F. **Discarded**

This category groups the comments that do not fit into the categories previously defined; they have two flavors:

F.1 Automatically generated: This category defines auto-generated notes (e.g., "Auto-generated method stub"). In most case, the comment represents the skeleton with a comment's placeholder provided by the IDE and left untouched by the developers.

F.2 Unknown: This category contains all remaining comments that are not covered by the previous categories. In addition, it contains the comments whose meaning is hard to understand due to their poor content (e.g., meaningless because out of context).

Finding 1: The manual inspection of a sample of representative Java files from diverse open source software systems has led to the creation of a taxonomy for code comments composed by two layers with 6 top coarse-grained categories and 16 inner fine-grained categories.

4.4.2 RQ_2. How often does each category occur in OSS and industrial projects?

The second research question investigates the occurrence of each category of comments in the 6,000 source files that we manually classified from our six OSS systems and eight industrial projects. We first describe the results separately, then we contrast how the comments are distributed in the two settings.

OSS projects: Distribution of comments. Figure 4.3 shows the distribution of the comments across the categories in the considered OSS systems. The figure reports the cumulative value for the top level categories (e.g., Notice) and the absolute value for the

inner categories (e.g., Exception). For each category, the top red bar indicates the number of *blocks of comments* in the category, while the bottom blue bar indicates the number of non-blank *lines of comments* in the category.

Comparing blocks and lines, we see that the longest type of comments is License, with more than 11 lines on average per block. The Expand category follows with a similar average length. The Summary category has only an average length of 1.4 lines, which is surprising, since it is used to describe the purpose of possibly very long methods, variables, or blocks of code. The other categories show negligible differences between number of blocks and lines.

We consider the quality metric code/comment ratio, which was proposed at line granularity [203, 204], in the light of our results. We see that 59% of lines of comments should not be considered (i.e., categories from C to F), as they do not reflect any aspect of the readability and maintainability of the code they pertain to; this would significantly change the results. On the other hand, if one considers blocks of comments, the result would be closer to the original code/comment metric purpose. In this case, a simple solution would be to only filter out the Metadata category, because the other categories seem to have a more negligible impact.

Considering the distribution of the comments, we see that the Summary subcategory is the most prominent one. TThis is in line with the value of research efforts that attempt to generate summaries for functions and methods automatically, by analyzing the source code [218]. In fact, these methods would alleviate developers from the burden of writing a significant amount of the comments we found in source code files. On the other hand, the Summary accounts for only 24% of the overall lines of comments, thus suggesting that they only give a partial picture on the variety and role of this type of documentation. The second most prominent category is Usage. Together with the prominence of Summary, this suggests that the comments in the systems we analyzed are targeting end-user developers more frequently than internal developers. This is also confirmed by the low occurrence of the Under development category. Concerning Under development, the low number of comments in this category may also indicate that developers favor other channels to keep track of tasks to be done in the code.

Finally, the variety of categories of comments and their distribution underlines once more the importance of a classification effort before applying any analysis technique on the content and value of code comments. The low number of discarded cases corroborates the completeness of our taxonomy.

Industrial projects: Distribution of comments. Figure 4.4 shows the distribution of the comments across the categories in the considered industrial systems. To differentiate from the case of OSS systems, we use other colors: The top *green* bar indicates the number of blocks of comments, while the bottom *yellow* bar indicates the number of non-blank lines of comments.

Comparing number of blocks and number of lines, we see that most categories show a negligible differences between the two granularities. The largest, yet unremarkable difference is in the Purpose category (this is expected since this category includes both the Summary and the Expand subcategories), in which we found 4,167 lines distributed over 3,436 blocks, with an average of 1.21 lines per block.

Considering the quality metric code/comment ratio in the industrial context, we see

that 31% of the lines of comments should not be considered (i.e., categories from C to F). This percentage is significantly lower than the case of OSS systems, whose distribution of comment lines is skewed by the License category. Past research has shown that these types of comments, which are especially structured, can be detected with high precision and recall even in free form documents [219].

The Summary subcategory is the most prominent one, thus corroborating the importance of research investigating ways to automatically generate this kind of comments (e.g., [218]), also in the industrial setting. Matching the case of OSS systems, the second most prominent subcategory is Usage, immediately followed by Incomplete. This indicates that most comments target internal developers in the system, which is to be expected in a close source setting.

Finally, also in the industrial setting, the taxonomy was extensive enough to allow us to categorize all the source code comments without dropping any instance, even though we created this taxonomy from comments in OSS projects.

OSS vs. Industrial: Comparison of the distributions. Figure 4.5 shows a comparison of the distribution of comments for the considered OSS systems and industrial projects, as a proportion of the total number of lines/blocks of comments in each context. The large difference in the frequency of License lines is evident, while we see that the categories Purpose, Notice, Style & IDE, and Discarded have substantially similar distributions. Another large difference regards the Under development category: The industrial projects we analyzed use source code comments for commenting code and leave incomplete comments far more frequently than OSS systems. This could be an indication that, if we exclude the License category, using code comments as an indicator for quality could be more appropriate for OSS systems. In fact, Incomplete and Commented code subcategories could be an indication of bad practices and low readability and maintainability of code, thus hindering the value of a comment/code metric. Investigating this hypothesis is beyond the scope of our current work, but studies can be devised and conducted to verify to which extent some types of comments indicate *problems* in the code, rather than a higher quality.

Finding 2: By comparing the distribution of comments in open and closed source software projects, we found that on average the former class of projects uses 4 times the number of comments compared to second set. The Metadata category is the most popular category in OSS and Purpose is the most popular category in closed source.

4.4.3 RQ_3. How effective is an automated approach, based on machine learning, in classifying code comments in OSS and industrial projects?

To evaluate the effectiveness of machine learning in classifying code comments we employed a supervised learning method. Supervised machine learning bases the decision evaluating on a pre-defined set of features, training on a set of labeled instances. Since we set to classify *lines* of code comments, we computed the features at line granularity.

Text preprocessing. We preprocessed the comments by doing the following in this order: (1) tokenizing the words on spaces and punctuation (except for words such as '@usage' that would remain compounded), (2) splitting identifiers based on camel-casing

(e.g., 'ModelTree' became 'Model Tree'), (3) lowercasing the resulting terms, (4) removing numbers and rare symbols, and (5) creating one instance per line.

Feature creation. Table 4.3 shows all the features that appear in the final model (these features are a subset of all that we initially devised). Since the optimal set of features is not known *a priori*, we started with some simple, traditional features and iteratively experimented with others more sophisticated features, in order to improve precision and recall for all the projects we analyzed.

A set of features commonly used in text recognition [220] consists in measuring the occurrence of the words; in fact, words are the fundamental tokens of all languages we want to classify. To avoid overfitting to words too specific to a project, such as code identifiers, we considered only words above a certain threshold t. We found this value experimentally starting with a minimum of 3 and increasing up to 10, in one-unit steps. Since the values above 7 do not change the precision and recall quality, we chose that threshold.

In addition, other features consider the information about the *context* of the line, such as the text length, the comment position in the whole file, the number of rows, the nature of the adjacent rows, etc.

The last set of features is category specific. We defined regular expressions to recognize specific patterns. We report three detailed examples:

- This regular expression is used to match comments in single line or multiple lines with empty body.

```
^\s*\/(\*|\s)*(\/|\*\s*\*\/)\n*$
```

- This regular expression matches the special keywords used in the *Usage* category.

```
(?i)@param|@usage|@since|@value|@return
```

- The following regular expression is used to find patterns of symbols that may be used in *Formatter* category.

```
([^*\s])(\1\1)|^\s*\/\/\/\s*\S*|\$\S*\s*\S*\$
```

Machine learning validation with 10-fold. We tested both probabilistic classifiers and decision tree algorithms. When using probabilistic classifiers, the average values of precision and recall were usually lower those obtained using decision tree algorithms. While, using decision tree algorithms, the percentage value associated with the correctly classified instances is improved. Particularly, with Random Forest we obtain up to 98.4% correctly classified instances. Nevertheless, in the latter case, many comments belonging to classes with a low occurrence were wrongly classified. Since the purpose of our tool is to best fit the aforementioned taxonomy we found that the best classifier is based on a probabilistic approach.

In Table 4.4 we report only the results (precision, recall, and weighted average TP rate) for the naive Bayes Multinominal classifier that on average, considering whole categories, achieves a better result accordingly to the aforementioned considerations. In Table 4.4 we intentionally leave empty cells that correspond to categories of comments that are not present in related projects. For the evaluation, we started with a standard 10-fold cross validation. Table 4.4 and Table 4.5 show these results in the columns '10-fold' for open and closed source, respectively. In both cases, we obtain promising performance for the six high-level categories. Generically, the performance in the open source case is slightly higher than the closed source one. For OSS systems, precision and recall are always above 93%; for closed source projects, we have a drop of the performance up to 70% of precision for the Discarded category. This difference is most likely attributable to the smaller number of instances available for the training set of closed source projects. Indeed, the same trend is also visible in fine-grained categories. The precision for inner categories is in average better for OSS projects (with a minimum of 50% in the case of the Rationale category). In the closed source projects, both precision and recall for inner categories reach high value up to 100% for several categories, however, there are categories (Rationale Deprecation, and Unknown) where the performance is below 70%.

Cross-project validation. Different systems have comments describing different code artifacts and are likely to use different words and jargons. Thus, term-features working for the comments in one system may not work for others. To better test the generalizability of the results achieved by the classifier, we conduct a *cross-project validation*, as also previously proposed and tested by Bacchelli et al. [221]. In practice, cross-project validation for OSS case consists in a 6-fold cross validation, in which folds are neither stratified nor randomly taken, but correspond exactly to the different systems: In the open source case, we train the classifiers on five systems and we try to predict the classification of the comments in the remaining system. We do this six times rotating the test system. Similarly, in the industrial context we divided the dataset in eight folds corresponding to the eight industrial projects, then we used one fold as test dataset and the remaining folds to train the model. We repeated this process eight times to evaluate the performance for each project. The right-most columns (i.e., 'cross-project') in Table 4.4, 4.5, and 4.6 show the results by tested systems.

Cross-license validation. The different development setting, i.e., OSS or industrial, may have an impact on software development [222]. In line with the hypothesis of Paulson et al. [222], we indeed found a difference in the comments usage between these two different categories of development processes. We found that open source projects have on average up to 8 blocks of comments per file, while the industrial projects decrease have an average of 2 blocks of comments per file. Therefore, these differences during the training of a machine learning classifier can become crucial, as they may impact on the performance of the model.

To evaluate the impact of the different development setting on an automated approach to classify code comments, we define and conduct a *cross-license validation.* We define cross-license validation when the training set differs from the testing set for the license/setting of the file to which the comment pertain, i.e., OSS or industrial. In our study, we conduct a *2-fold* cross-license validation, in which we train on projects from one setting (e.g., OSS) and we test on projects from the other setting (e.g., industrial). In this validation, we

alternate OSS and industry as test and training sets. Table 4.7 contains the results, in terms of precision and recall, obtained by evaluating our model on the top categories. The first row represents the results obtained training the model on the OSS projects and testing it on the industrial ones, instead the second row refers to the opposite situation where we trained the model with the industrial comments and tested it with OSS ones. Even though the differences are not major (e.g., 0.73 of precision for both Discarded categories), training the model with the open source data achieves better results on average (e.g., the precision is up to 10% higher for the category Under Development using open source training set); this result may be due to the higher number of comments in the OSS dataset or more diverse distributions of the features across the data. Overall, the within-project performance is marginally better than the cross-project one, when the training is accomplished with open-source data. Indeed, cross-project validation achieves performance above 0.73 in terms of weighted average TP rate, while within-project validation conducted only on open-source projects is up to 0.88 in terms of weighted average TP rate. Based on our experience gained through the manual classification, we argue that many comments in OSS systems are written with a different purpose than comments in closed-source projects. For example, OSS programmers rely on code comments to communicate their development strategies, vice-versa, industrial driven developers seem to rely on alternative channels to communicate with their team. This observation is also reflected the different number of comments present in the two domains as well as the different distribution across found categories. Moreover, this difference would have an impact on the creation of new tools aimed at helping developers to increase their productivity and to improve software reliability.

Summary. The values for 10-fold cross validation reported in Table 4.4 show accurate results (mostly above 95%) achieved for top-level categories. This means that the classifier could be used as an input for tools that analyze source code comments of the considered systems. For inner-categories, the results are lower; nevertheless, the weighted average TP rate remains 0.85. Furthermore, we do not see large effects due to the prominent class imbalance. This suggests that the amount of training data is enough for each class.

As expected, testing with cross-project validation, the classifier performance drops. However, this is a more reliable test for what to expect with Java comments from unseen projects. The weighted average TP rate that goes as low as 0.74. This indicates that project-specific terms are key for the classification and either an approach should start with some supervised data or more sophisticated features must be devised.

The last analysis (i.e., the cross-license validation), where we divided the dataset in two parts gathering in the same dataset all projects with the same commercial license (i.e., OSS and industrial projects), shows that results are higher when using open source dataset to train the model (up to 15% of Precision for Style & IDE category). Even though cross-license shows a negative performance when using industrial comments to train the model, the differences are in average below 7% for Purpose and Metadata categories in terms of Precision. In the end, Discarded achieves the same performance for both categories (0.73 in Precision). These results suggest that the proposed open source dataset may be used by both open source organization and industrial companies to categorize Java code comments. The higher performance of training on OSS may be due to the higher number of manually classified instances from the OSS projects; a further study could

investigate whether a higher number of training instances from the industrial context would lead to similar results.

Finding 3: The within-project validation achieves the best performance (up to 0.95 for weighted average TP rate) compared to cross-project or cross-license validation where the performance is typically 15% lower. Although this, it remains above 0.74 in terms of weighted average TP for unseen files that may be due to the use a project-specific set of terms. In addition to differences in the terminology used in the comments, this difference may be increased by the different communication strategies used in the inspected developmental domains.

4.4.4 RQ_4. How does the performance of an automated approach improve by adding classified comments from the system under classification?

The answer to our previous research question shows that it is possible to create an automatic classifier for code comments. However, when such a classifier is tested on an unseen project, it achieves lower results, compared to testing it on a project for which some of the comments are part of the training set. This is expected, since words used in the text are parts of the training features. In this research question, we investigate how many instances should one classify from an unseen system to make the classification algorithm reach higher results.

To this aim, we selected the industrial project that achieved the worst performance in cross-project validation, then, we progressively added to the training set a fixed number of manual classified comments (i.e., in steps of 5 comments). For each iteration, we evaluated the performance of a Random Forest classifier and computed precision and recall.

Figure 4.6 shows the classifier's results by progressively including new manually classified comments. The *blue* line indicates the evolution of the precision curve by progressively adding 5 random selected manual classified comments of the subject system; the *red* line indicates the trend of the recall values. The lines show that the classifier starts from a minimum of 0.65 and 0.74 for precision and recall, respectively. This is the scenario in which no comments belonging to the unseen project are included in the training set. The maximum performance corresponds to 0.89 of precision and 0.94 of recall, and it is reached when at least 100 manually classified comments of the subject system are added to the training set. The trend shows that the performance reaches a plateau after 100 manually classified instances.
Finally, we observe that in the starting phase (left side of the chart) the performance of the model remains stably below 0.80 of precision and recall until the comments contribution is below 30 threshold; instead it improves rapidly in the interval included between 30 and 70 blocks of comments. This observation seems to indicate the presence of an optimal interval of comments that a human classifier should manually classify to boost the performance of the proposed solution for a novel project.

The investigation highlights that the performance of the proposed model can be easily and significantly increased by manually classifying a small sample of new comments (e.g., in our case the manual classification of just 60 blocks of comments boosted the *prediction*

of 30%). We empirically found this sample size to be between 40 and 80 block of comments, which corresponds to about 10 Java open source files (or about 3 hours of labeling efforts).

Finding 4: The improvement achieved by manual classifying a sample between 40 and 80 code comment from an unseen project is up to a 37% and 27% gain in precision and recall, respectively. This indicates that it is possible to significantly improve the accuracy of the automatic classification for an unseen project with a minor effort.

Figure 4.3: Frequencies of comments per category in open source projects. Top, red bars show the occurrences by blocks of comments and bottom, blue bars by lines.

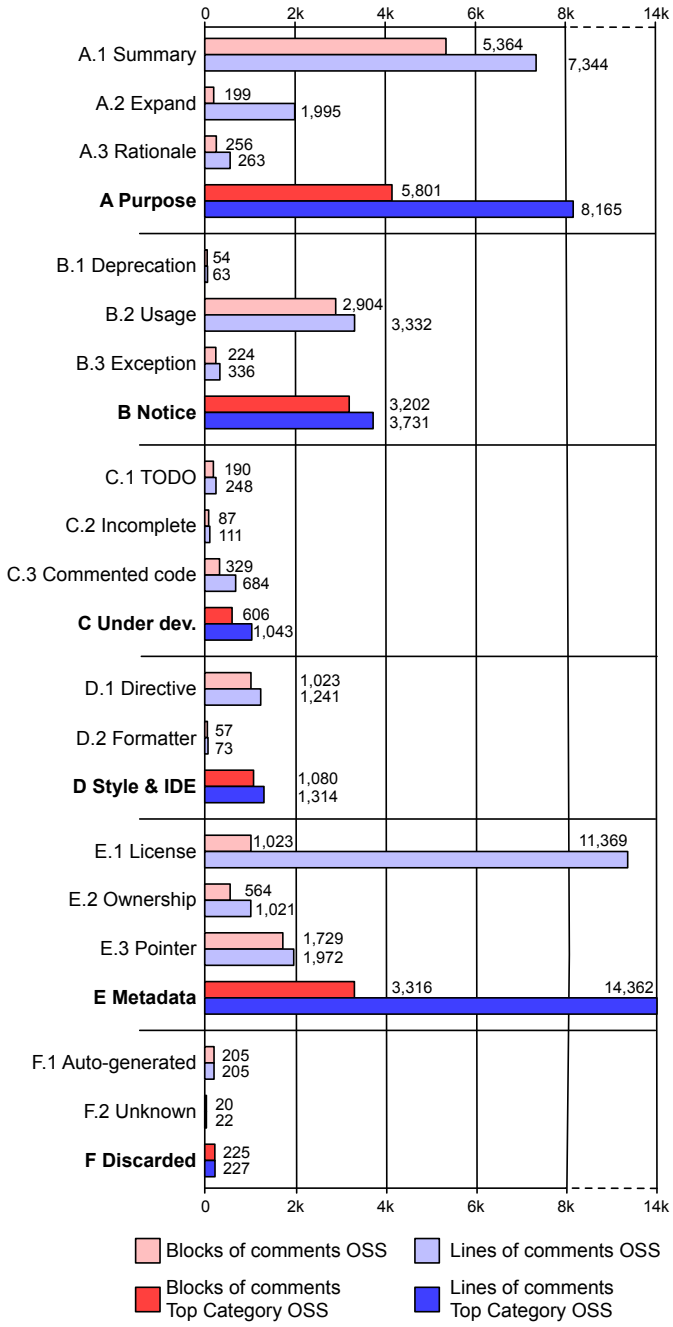

Figure 4.4: Frequencies of comments per category industrial projects. Top, green bars show the occurrences by blocks of comments and bottom, yellow bars by lines.

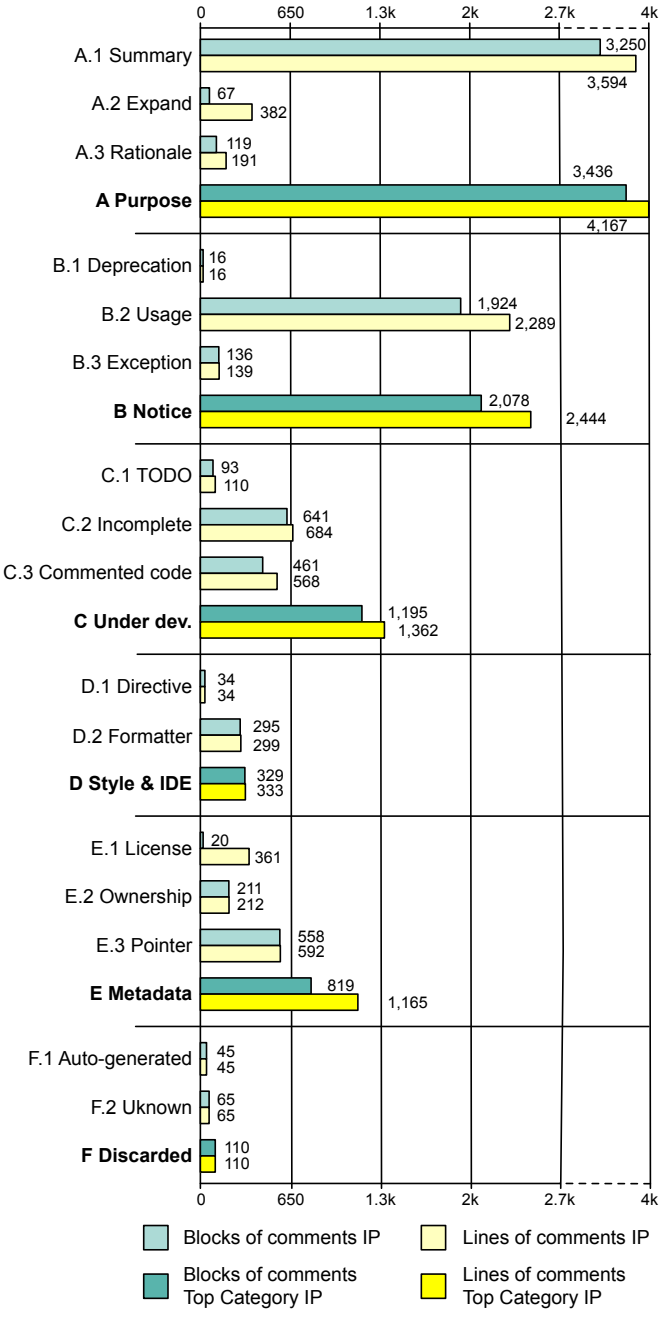

Figure 4.5: Frequencies of comments per category. Top, red bars show the occurrences by blocks of comments and bottom, blue bars by lines.

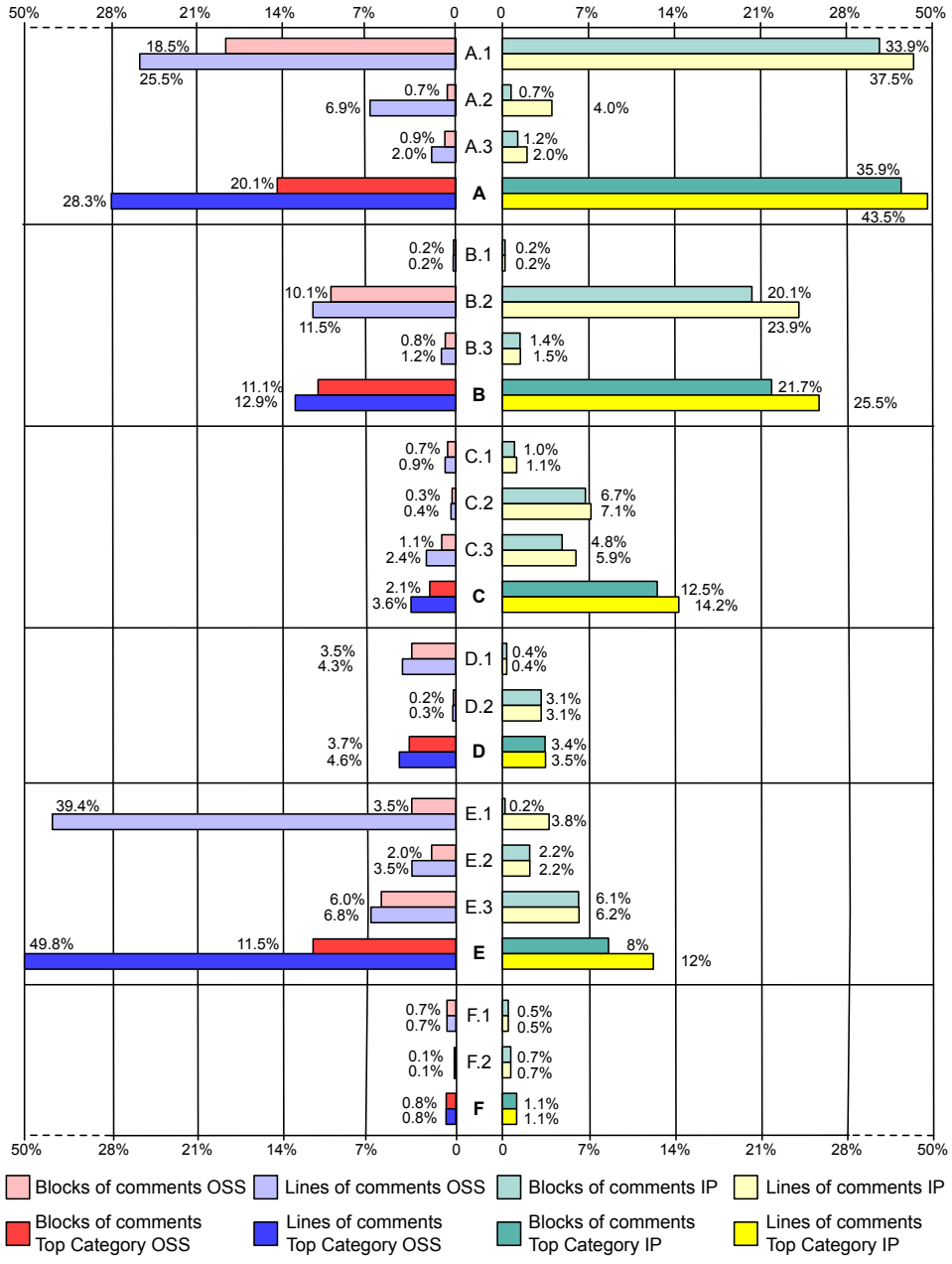

Table 4.3: Machine learning features for comments classification.

Feature	Type	Description
words	numeric	counts the occurrence of each word in the bag of unique words
punctuation	boolean	used in combination of a regular expression to distinguish source code from natural language e.g., object.method(par1, par2);
words count	numeric	measures the length of the comment, using the words as unit size
unique words count	numeric	measures the length of the comment, only unique words are counted
row position	numeric	detects the absolute position of the comment
adjacent rows	numeric	recognizes the nature of the adjacent rows e.g., comments or code
deprecation	boolean	true if a comment contains special tags like @deprecation
usage	boolean	true if a comment contains special tags such as @usage, @return or @value
exception	boolean	true if a comment contains special tags such as @exception or @throws
TODO	boolean	true if a comment contains keywords such as todo or fix or a link to a bug is detected
incomplete	boolean	true if a comment contains an empty body
commented code	boolean	true if a comment contains code snippets
directive	boolean	true if a comment contains special sequence of symbols used by IDE
formatter	boolean	true if a comment is composed of patterns of symbols or characters
license	boolean	true if a comment contains words such as license, copyright, legal or law
ownership	boolean	true if a comment contains tags such as @author or @owner
pointer	boolean	true if a comment contains a reference to an external linkable resource
automatic generated	boolean	true if a comment contains text automatically inserted by IDE e.g., Auto-generated method stub

Table 4.4: Results of the classification with naive bayes multinomial classifier in OSS.

			Validation						
	P = Precision R = Recall			Cross project					
Top categories	Inner categories		10-fold	CDT	Guava	Guice	Hadoop	Vaadin	Spark
Purpose	Summary	P	0.88	0.96	0.68	0.61	0.72	0.62	0.65
		R	0.82	0.99	0.61	0.69	0.56	0.69	0.84
	Expand	P	1.00	0.84	0.00	0.00	0.09	0.00	0.00
		R	0.98	0.64	0.00	0.00	0.05	0.00	0.00
	Rationale	P	0.50	0.56	0.15	0.00	0.10	0.03	0.67
		R	0.69	0.84	0.23	0.00	0.41	0.17	0.17
	Purpose	P	0.99	0.77	0.77	0.81	0.80	0.83	0.68
		R	0.99	0.98	0.98	0.81	1.00	1.00	1.00
Notice	Deprecation	P	0.74	0.75	0.22			0.14	
		R	0.78	0.81	1.00			1.00	
	Usage	P	0.86	0.85	0.50	0.43	0.67	0.90	0.56
		R	0.90	0.87	0.45	0.64	0.61	0.65	0.15
	Exception	P	0.76	0.75	0.43	0.00	0.58	0.69	0.13
		R	0.98	0.95	0.87	0.00	0.88	0.97	0.29
	Notice	P	1.00	0.50	0.50	0.36	0.60	1.00	1.00
		R	0.98	0.50	0.50	1.00	0.41	0.33	0.17
Under dev.	TODO	P	0.61	0.97	0.57	0.29	0.03	0.19	
		R	0.52	0.96	0.83	1.00	0.16	0.11	
	Incomplete	P	0.91	0.92			0.11	0.95	
		R	0.96	1.00			0.88	0.91	
	Commented code	P	0.91	0.91			0.05	0.92	
		R	0.91	0.95			0.06	0.50	
	Under development	P	0.98	1.00	0.00	0.00	0.00	0.00	
		R	0.93	0.67	0.00	0.00	0.00	0.00	
Style & IDE	Directive	P	0.96	0.96				0.00	
		R	1.00	1.00				0.00	
	Formatter	P	0.81	0.93	0.00			0.00	
		R	0.77	0.28	0.00			0.00	
	Style & IDE	P	0.97	1.00	1.00			0.00	
		R	0.99	1.00	1.00			0.00	
Metadata	License	P	0.99	1.00	0.98	1.00	0.99	0.99	1.00
		R	0.98	0.99	1.00	0.95	0.99	1.00	1.00
	Ownership	P	0.80	1.00	1.00	0.57	0.00	1.00	
		R	0.96	1.00	0.08	0.27	0.00	0.98	
	Pointer	P	0.84	0.80	0.82	0.81	0.79	0.97	1.00
		R	0.94	0.74	0.52	0.54	0.70	0.85	0.60
	Metadata	P	1.00	1.00	1.00	1.00	1.00	1.00	0.89
		R	1.00	0.68	0.68	0.57	0.57	0.95	1.00
Discarded	Auto generated	P	0.90	0.91			0.13	0.84	
		R	1.00	1.00			1.00	1.00	
	Unknown	P	0.65	1.00		0.00	0.00	0.00	
		R	0.77	0.39		0.00	0.00	0.00	
	Discarded	P	0.96	0.00		0.00	0.00	0.00	
		R	0.98	0.00		0.00	0.00	0.00	
Weighted average TP rate			0.85	0.88	0.77	0.79	0.74	0.80	0.83

Table 4.5: Results of the classification with random forest classifier in industrial projects.

			Validation								
	P = Precision R = Recall			Cross project							
Top categories	Inner categories		10-fold	P1	P2	P3	P4	P5	P6	P7	P8
Purpose	Summary	P	0.97	0.52	0.64	0.84	0.73	0.63	0.78	0.83	0.68
		R	1.00	0.86	0.87	0.89	0.86	0.97	0.69	0.89	0.77
	Expand	P	1.00	0.75	1.00	0.86	0.57	0.96	0.13	0.60	
		R	1.00	0.14	0.29	0.90	0.57	0.53	0.10	0.53	
	Rationale	P	1.00	0.18	0.50	0.13	0.71	0.00	0.54	0.53	0.00
		R	0.70	0.30	0.10	0.83	0.71	0.00	0.43	0.16	0.00
	Purpose	P	0.98	0.56	0.92	0.92	0.95	0.72	0.88	1.00	0.70
		R	0.99	0.99	0.97	0.91	0.84	0.94	0.79	0.72	0.83
Notice	Deprecation	P	0.83	0.00		1.00	0.67	0.85			
		R	0.45	0.00		0.86	0.54	0.72			
	Usage	P	1.00	0.40	0.00	0.77	0.90	0.81	0.97	0.84	0.67
		R	1.00	0.91	0.00	0.86	0.69	0.98	0.85	0.84	0.81
	Exception	P	1.00	1.00		1.00	1.00	1.00	0.72	1.00	1.00
		R	0.96	0.94		0.25	1.00	1.00	0.70	0.31	0.96
	Notice	P	1.00	1.00	0.00	0.73	0.65	0.80	0.87	1.00	0.69
		R	1.00	0.58	0.00	0.88	0.99	0.97	0.85	0.58	0.89
Under dev.	TODO	P	0.99	0.56			0.00			1.00	0.15
		R	0.86	0.55			0.00			0.53	0.08
	Incomplete	P	1.00				1.00			0.50	1.00
		R	1.00				0.24			0.75	0.79
	Commented code	P	0.98	0.68		1.00	1.00	0.27	1.00	0.70	0.37
		R	0.98	0.62		1.00	1.00	0.25	1.00	0.58	0.41
	Under development	P	0.99	0.96	0.95	0.11	0.67	0.27	0.75	0.95	0.37
		R	0.97	0.78	0.96	0.34	0.69	0.33	0.88	1.00	0.45
Style & IDE	Directive	P	0.94	0.00		0.78		0.00		0.71	
		R	0.92	0.00		0.53		0.00		0.42	
	Formatter	P	1.00		0.90						0.88
		R	0.84		0.55						0.68
	Style & IDE	P	1.00	0.00	0.99	0.54	0.00	0.00	0.00	0.81	0.87
		R	0.85	0.00	0.08	0.34	0.00	0.00	0.00	0.52	0.55
Metadata	License	P	1.00	0.00	0.63	0.90		0.00		1.00	1.00
		R	1.00	0.00	0.69	0.87		0.00		1.00	1.00
	Ownership	P	1.00	0.54	1.00	1.00		1.00	1.00	1.00	
		R	1.00	0.51	0.97	0.78		1.00	1.00	0.81	
	Pointer	P	1.00	0.51	1.00	1.00	1.00	1.00	0.89	0.57	0.57
		R	0.98	0.70	0.97	0.32	0.71	0.45	0.73	0.63	0.31
	Metadata	P	1.00	1.00	0.84	0.91	0.98	1.00	0.96	0.96	0.97
		R	0.99	0.32	0.61	0.87	0.26	0.47	0.79	0.81	0.98
Discarded	Auto generated	P	0.94				0.10	0.00			
		R	1.00				0.56	0.00			
	Unknown	P	0.60		0.25						
		R	0.99		0.20						
	Discarded	P	0.70	0.78	0.28		0.00	0.00			
		R	0.99	1.00	0.15		0.00	0.00			
Weighted average TP rate			0.95	0.75	0.68	0.73	0.77	0.70	0.78	0.76	0.71

Table 4.6: Results of the classification with random forest classifier in cross-project validation.

	P = Precision R = Recall		Cross project validation													
			Open source						Closed source							
Top categories	Inner categories		CDT	Guava	Guice	Hadoop	Vaadin	Spark	P1	P2	P3	P4	P5	P6	P7	P8
Purpose	Summary	P	0.95	0.85	0.93	0.65	0.79	0.74	0.84	0.69	0.89	0.88	0.84	0.85	0.75	0.93
		R	0.98	0.96	1.00	1.00	0.96	1.00	1.00	0.97	1.00	1.00	0.96	0.99	0.90	0.94
	Expand	P	0.85	0.70	1.00	0.15	0.15	1.00	1.00	0.42	1.00	1.00	1.00	0.24	0.57	
		R	0.64	0.85	1.00	0.13	0.75	0.25	1.00	1.00	1.00	1.00	1.00	0.19	0.57	
	Rational	P	0.55	0.00	1.00	0.15	0.00	0.00	1.00	0.40	1.00	1.00	0.32	0.56	0.72	0.36
		R	0.68	0.00	1.00	0.51	0.00	0.00	0.86	0.29	0.86	0.87	0.62	0.36	0.71	0.56
	Purpose	P	0.99	0.85	0.79	.98	0.70	0.70	0.97	0.86	0.90	0.89	0.90	0.99	0.89	0.74
		R	1.00	1.00	1.00	1.00	1.00	0.99	1.00	0.99	0.98	0.99	0.99	0.97	0.84	0.78
Notice	Deprecation	P	0.75	0.00			0.00		1.00		1.00	1.00	1.00			
		R	0.81	0.00			0.00		1.00		1.00	1.00	1.00			
	Usage	P	0.88	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.84
		R	0.90	0.85	0.90	1.00	0.35	0.33	0.92	0.38	0.92	0.93	0.84	0.92	1.00	0.81
	Exception	P	0.72	1.00		0.63	1.00	1.00	1.00		1.00		1.00	0.74	0.68	1.00
		R	0.98	1.00		0.88	1.00	1.00	0.55		0.65		0.50	0.75	0.43	0.87
	Notice	P	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.74	0.96
		R	0.99	0.76	0.73	0.97	0.24	0.65	0.95	0.81	0.87	0.91	0.81	0.73	0.86	0.85
Under dev.	TODO	P	0.59	1.00		0.00	0.21		0.96			0.00			0.77	0.15
		R	0.50	0.65		0.00	0.32		0.67			0.00			0.67	0.08
	Incomplete	P	0.90				0.87					0.78			0.89	1.00
		R	0.88				0.73					0.63			0.10	0.25
	Commented code	P	0.89	1.00			1.00		1.00		1.00	1.00	1.00	1.00		0.17
		R	0.97	1.00			1.00		1.00		1.00	1.00	1.00	0.99		0.41
	Under development	P	0.99	1.00	0.76	0.98	1.00		1.00	1.00	1.00	1.00	1.00	1.00	0.87	1.00
		R	0.95	0.55	0.85	0.86	0.67		0.99	0.86	0.85	0.86	0.87	0.86	0.87	0.99
Style & IDE	Directive	P	1.00	0.00					1.00		1.00		1.00		1.00	
		R	1.00	0.00					1.00		1.00		0.75		1.00	
	Formatter	P	0.90							0.87						0.98
		R	1.00							0.42						0.57
	Style & IDE	P	1.00	0.00	1.00				1.00	1.00	1.00		1.00		1.00	0.99
		R	1.00	0.00	0.79				1.00	0.89	0.89		0.82		0.90	0.84
Metadata	License	P	1.00	1.00	1.00	0.99	1.00	1.00	1.00	0.93	1.00		1.00		1.00	1.00
		R	1.00	1.00	1.00	0.99	1.00	1.00	1.00	1.00	1.00		1.00		1.00	1.00
	Ownership	P	1.00	1.00			1.00		1.00	1.00	1.00		0.99	1.00	1.00	
		R	0.98	1.00			0.97		1.00	1.00	1.00		0.97	1.00	0.91	
	Pointer	P	0.97	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.97	1.00	0.95	0.57	0.67
		R	0.99	0.82	1.00	0.68	1.00	0.45	0.42	0.99	0.71	0.85	0.86	0.99	0.57	0.40
	Metadata	P	1.00	1.00	0.99	0.99	0.94	0.90	1.00	0.99	0.91	0.99	0.99	0.99	1.00	0.99
		R	0.95	0.94	0.89	0.99	1.00	1.00	1.00	1.00	0.97	0.95	0.98	0.98	0.91	0.95
Discarded	Auto generated	P	0.00			0.23	0.78					0.10	0.00			
		R	0.00			1.00	0.89					0.78	0.00			
	Unknown	P	1.00							0.21						
		R	0.40							0.17						
	Discarded	P	0.38		0.55	0.00	0.10			0.00		0.00	0.00			
		R	0.15		0.56	0.00	0.15			0.00		0.00	0.00			
Weighted average TP rate			0.90	0.92	0.90	0.98	0.76	0.86	0.94	0.94	0.95	0.91	0.83	0.90	0.86	0.88

Table 4.7: Results of the cross-license validation in terms of precision (P) and recall (R), using a random forest classifier.

	Top categories											
	Purpose		Notice		Under development		Style & IDE		Metadata		Discarded	
Testing on:	P	R	P	R	P	R	P	R	P	R	P	R
Industrial systems	0.75	0.98	0.96	0.49	0.87	0.69	0.78	0.13	0.99	0.69	0.73	0.91
OSS systems	0.68	0.99	0.88	0.67	0.77	0.50	0.63	0.66	0.98	0.55	0.73	0.30

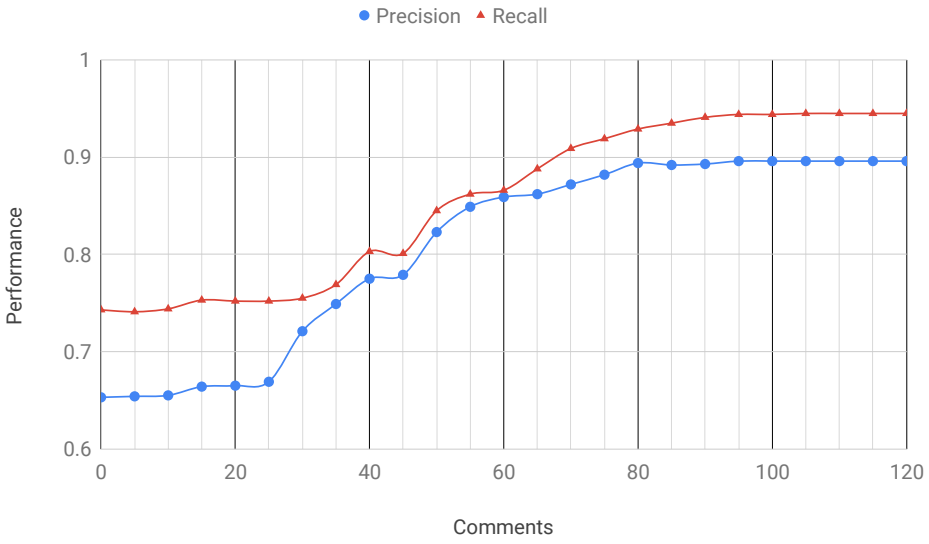

Figure 4.6: Number of comments required to increase the performance for an unseen project.

4.5 Related Work

4.5.1 Information Retrieval Technique

Lawrie et al. [223] use information retrieval techniques based on cosine similarity in vector space models to assess program quality under the hypothesis that *"if the code is high quality, then the comments give a good description of the code"*. Marcus and Maletic propose a novel information retrieval techniques to automatically identify traceability links between code and documentation [224]. Similarly, de Lucia et al. focus on the problem of recovering traceability links between the source code and connected free text documentation. They propose a comparison between a probabilistic information retrieval model and a vector space information retrieval [225]. Even though comments are part of software documentation, previous studies on information retrieval focus generally on the relation between code and free text documentation.

4.5.2 Comments Classification

Several studies regarding code comments in the 80's and 90's concern the benefit of using comments for program comprehension [197–199]. Stamelos et al. suggest a simple ratio metric between code and comments, with the weak hypothesis that software quality grows if the code is more commented [226]. Similarly, Oman and Hagemeister propose a tree structure of maintainability metrics that also consider code comments [203] and Garcia et al. also use lines of comments to measure the maintainability of a module [204].

New recent studies add more emphasis to the code comments in a software project. Fluri et al. present a heuristic approach to associate comments with code investigating whether developers comment their code. Marcus and Maletic propose an approach based on information retrieval technique [227]. Maalej and Robillard investigate API reference documentation (such as javadoc) in Java SDK 6 and .NET 4.0 proposing a taxonomy of knowledge types. They use a combination of grounded and analytical approaches to create such taxonomy [228]. Instead Witte et al. used Semantic Web Technologies to connect software code and documentation artifacts [229]. However, both approaches focus on external documentation and do not investigate evolutionary aspects or quality relationship between code and comments, i.e., they do not track how documentation and source code changes together over time or the combined quality factor. More in focus is the work of Steidl et al. where they investigate the quality of the source code comments [207]. They proposed a model for comment quality based on different comment categories and use a classification based on machine learning technique tested on Java and C/C++ programs. They define 7 high-level categories that are generically available in both Java and C/C++ programming languages. Moreover, they evaluated the quality of their taxonomy with a survey by involving 16 experienced software developers. Despite the quality of the work, they found only 7 high-level categories of comments based mostly on comment syntax, i.e., inline comments, section separator comments, task comments, etc. This limitation may be a consequence of the aggregation of different programming languages such as Java and C/C++. In our study, we refine such categories by including a fine-grained taxonomy composed of 16 categories. Our taxonomy is tailored specifically for Java programming language, indeed, the study involved only Java sources.

Padioleau et al. [209] also conducted an extensive evaluation and classification of source

code comments. They had a different aim than ours: They focused on understanding to what extent developers' needs can be derived from code comments (e.g., how comments are used for code annotations or to communicate intentions behind the software development paradigm); moreover they considered a different context (i.e., code comments from three Unix-like operating systems (i.e., Linux, FreeBSD, and OpenSolaris) written in C). They conducted a classification along four dimensions: *content*, *people involved*, *code location*, and *time and evolution*. The 'content' dimension is the most aligned with our work. Although their focus (developers' needs and software reliability) and data sources (C systems) were different, some of their categories along the 'content' dimension share strong similarities with our taxonomy (e.g., 'PastFuture' includes TODOs, as our 'C. Under Development' does). Especially if we account for the subjective differences that are common in manual classification studies, these similarities seem to indicate that it would be possible to derive a taxonomy of code comments that goes beyond the boundaries of a single programming language (indeed they also performed a preliminary investigation trying to classify comments of a Java system according to their taxonomy that further suggests this possibility [209]). Interestingly, Padioleau et al. also showed how more than 50% of the comments can be *exploited* by existing or to be proposed tools. We did not consider this aspect in our work, but future studies can be devised to investigate it using our publicly available dataset. An additional difference between our work and that of Padioleau et al. is that we studied how our manually analyzed code comments can be automatically classified according to our taxonomy using machine learning.

4.6 Conclusion

Code comments contain valuable information to support software development, especially during code reading and code maintenance. Nevertheless, not all the comments are the same: For accurate investigations, analyses, usages, and mining of code comments, this has to be taken into account. By recognizing different kinds of code comments, as well as different meaning brought by code comments we want to simplify developers' activities and provide better data for metrics. Following this direction, a developer may exclude comments that do not involve a specific task and focus only on a subset (e.g., a developer that plans a maintainability task may be not interested in comments related to metadata information such as license and ownership). Relying on our findings, a developer may dynamically choose which categories highlight during a specific development phase. Moreover, our classification system can be used to improve techniques that use comments as a way to measure the maintainability of a software system. In fact, in this work, we investigated how comments can be categorized, also proposing an approach for their automatic classification.

The contributions of our work are:

- A novel, empirically validated, hierarchical taxonomy of code comments for Java projects, comprising 16 inner categories and 6 top categories.
- An assessment of the relative frequency of comment categories in 6 OSS Java software systems.
- An estimation of the relative frequency of comment categories in 8 industrial Java software systems.

- An exhaustiveness by cross-license validation of proposed taxonomy in a different context.
- A publicly available dataset of more than 2,000 source code files with manually classified comments, also linked to the source code entities they refer to.
- An empirical evaluation of an enhanced machine learning approach to automatically classify code comments according to the aforementioned taxonomy.
- An empirical evaluation aimed at understanding how many instances should be manually classified from an unseen system to make the classification algorithm perform similarly to an already seen project.

5

On the Performance of Method-Level Bug Prediction: A Negative Result

Bug prediction is aimed at identifying software artifacts that are more likely to be defective in the future. Most approaches defined so far target the prediction of bugs at class/file level. Nevertheless, past research has provided evidence that this granularity is too coarse-grained for its use in practice. As a consequence, researchers have started proposing defect prediction models targeting a finer granularity (particularly method-level granularity), providing promising evidence that it is possible to operate at this level. Particularly, models mixing product and process metrics provided the best results.

We present a study in which we first replicate previous research on method-level bug-prediction, by using different systems and timespans. Afterwards, based on the limitations of existing research, we (1) re-evaluate method-level bug prediction models more realistically and (2) analyze whether alternative features based on textual aspects, code smells, and developer-related factors can be exploited to improve method-level bug prediction abilities. Key results of our study include that (1) the performance of the previously proposed models, tested using the same strategy but on different systems/timespans, is confirmed; but, (2) when evaluated with a more practical strategy, all the models show a dramatic drop in performance, with results close to that of a random classifier. Finally, we find that (3) the contribution of alternative features within such models is limited and unable to improve the prediction capabilities significantly. As a consequence, our replication and negative results indicate that method-level bug prediction is still an open challenge.

This chapter extends Chapter 2 and to avoid repetitions it focuses on new parts only. It is based on

L. Pascarella, F. Palomba, A. Bacchelli. On the Performance of Method-Level Bug Prediction: A Negative

5.1 Introduction

The necessary evolution of software systems often leads to the introduction of defects, which possibly preclude the correct functioning of a piece of software and reduce its overall reliability [88]. To tackle this problem, researchers have been developing several techniques to support developers (e.g., verification and testing [231]): one of the most investigated areas is *bug-prediction* [32], which consists in detecting the areas of a software more likely to contain bugs in the future. Researchers have proposed and evaluated a variety of bug prediction models based on product [90, 97, 150], process [92, 100, 147], socio-technical [49, 232], and developer-related [43, 91] metrics. These models have been evaluated both in within-project scenarios and in cross-project ones [36–38], with several approaches achieving remarkable prediction performance [233]. Nevertheless, the practical relevance of bug prediction research has been put into question by studies that suggest that bug prediction does not address any real need of developers [24, 93, 94]. One of the main criticisms regards the *granularity* at which bugs are found [93]. In fact, most of the presented models predict bugs in modules or files – a granularity that is deemed not informative enough for practitioners, because files and modules can be arbitrarily large and inspecting them can require too much work [35]. In addition, considering that larger classes tend to be more bug-prone [44, 46], the effort required to identify the defective part in these classes is even more pronounced [90, 95, 96, 144].

To tackle this limitation, Menzies et al. [97] and Tosun et al. [98] conducted the first investigations on a finer granularity, i.e., function-level. Successively, Hata et al. [99] applied this idea to the context of object-oriented systems, proposing a method-level prediction model built using a set of historical metrics and that reported promising performance. Giger et al. [35] went even further and investigated the value of both product and process metrics for method-level prediction model. Specifically, Giger et al. devised three prediction models based on the combination of the two sets of features and evaluated how well they could classify which methods would have at least a bug (binary classification) within a specified time frame. They considered single snapshots of 21 open source software (OSS) projects in Java and reported promising results: 84% precision and 88% recall.

In this paper, we present a work that continues on this line of research.[1] First, we replicate the investigation conducted by Giger et al. [35] on bug prediction at method-level. We use the same features and classifiers as the reference work, but on a different dataset to test the generalizability of their findings. While our results show similar performance as the reference work, we observed two key limitations that may possibly bias the interpretation of the achieved findings. On the one hand, Giger et al. [35] took the change history and predicted bugs from the same time frame (which could lead to incorrect results) and used cross-validation (which have been reported as at the risk of producing biased estimates in certain circumstances [116]). On the other hand, they did not consider a number of alternative features that have been proved to impact the performance of class-level bug prediction models, namely textual, code smell-related, and developer-based metrics

Result, JSS'19 [230]

[1] The work presented here is an extension of the conference paper 'Re-evaluating Method-Level Bug Prediction' [87], appeared in the proceedings of the 25th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, 2018. pp. 592-601.

[113, 147, 234]. We tackle these limitations by (1) estimating the models' performance using data from subsequent releases (as done by more recent studies, which did it at a coarser granularity [100]), and by (2) adding a set of new alternative features to the considered method-level bug prediction model.

Our results show that—when evaluated on a release-by-release strategy—all the existing method-level bug prediction models present lower performance, close to that of a random classifier. As a consequence, even though we could replicate the reference work, a more realistic evaluation lead to negative results. Furthermore, all the alternative features we experiment with only marginally improve the performance, suggesting that method-level bug-prediction is still not a solved problem.

5.2 Background and Related Work

Research in the field of bug prediction is highly active [32, 235] and can be roughly divided in two sets: On the one hand, researchers focused on the characteristics relating to source code being more defect prone [39, 40, 42–47, 49, 148]; on the other hand, researchers defined bug prediction techniques based on unsupervised [50–52] and supervised [53–57] approaches. More recently, the concept of *just-in-time* bug-prediction has been introduced—techniques with the purpose of recommending defective files as developers commit them [59, 60, 62, 87, 236–238].

The current paper presents a work that focuses on investigating how well supervised approaches can identify bug-prone methods. In this section we discuss the literature related to class-/method-level bug prediction and describe the role of textual information for software quality.

5.2.1 Class-level Bug-Prediction

The approaches in this category differ from each other mainly for the underlying prediction algorithm, e.g., *Logistic Regression* vs *Random Forest*, and for the considered features, e.g., *product* (e.g., lines of code) or *process metrics* (e.g., number of changes performed to a class).

Product metrics. Basili et al. [90] found that five CK metrics [101] can help one to determine defective classes and that Coupling Between Objects (CBO) is the most related to bugs. These results were re-confirmed in further studies [95, 102, 103]. Ohisson et al. [104] focused on design metrics (e.g., 'number of nodes') to identify bug-prone modules, revealing the applicability of such metrics for the identification of buggy modules. Nagappan and Ball exploited two static analysis tools to predict the pre-release bug density for Windows Server [105]. Nagappan et al. [106] experimented with code metrics for predicting buggy components across five Microsoft projects, finding that no single metric is the best across all projects. Zimmerman et al. [57] investigated complexity metrics for bug-prediction and reported a positive correlation between code complexity and bugs. Nikora et al. [107] showed that measurements of a system's structural evolution (e.g., 'number of executable statements') can serve as bug-predictors. More recently, Dam et al. [239] reported an experience report of using product metrics and abstract representation of source code in practice, showing that it is possible to achieve good prediction accuracy when employing them within deep learning models.

Process metrics. Graves et al. [108] experimented with both product and process metrics for bug-prediction, finding that product metrics are poorer predictors for bugs. Moser et al. [110, 111] performed two comparative studies, which provided additional corroborating evidence on the superiority of process metrics in predicting buggy code components. Later on, D'Ambros et al. [112] performed an extensive comparison of bug-prediction approaches relying on both product and process metrics, finding that no technique works better in all contexts.

Despite the aforementioned promising results, a study by Shihab et al. [93] reported that developers consider class or module level bug-prediction too coarse-grained for being useful in practice. Also, a study by Lewis et al. [24] reported similar issues when trying defect prediction in practice at Google. This situation calls for the creation of methods able to provide a more fine-grained prediction (e.g., at *method-level*), re-evaluating and adapting what has been learned in the preceding work.

Alternative metrics. Despite product and process features have been the most widely used for bug prediction purposes, researchers have been also investigating the value of alternative metrics. In this category, several researchers exploited developer-related factors. For example, Hassan investigated a technique based on the entropy of developers' code changes [89], finding that it has better performance than models based on changes to code components. Ostrand et al. [113, 114] proposed the use of the number of developers who modified a code component as a bug-proneness predictor: however, the performance of the resulting model was poorly improved with respect to existing models. Later on, Di Nucci et al. [91] defined a bug-prediction model based on a mixture of code, process, and developer-based metrics outperforming the performance of existing models. As part of their experimentation, Di Nucci et al. also re-assessed the contribution given by the metric proposed by Ostrand et al. [113, 114], showing that in certain circumstances the number of developers may represent a relevant factor to characterize the bug-proneness of classes. Bird et al. [240] found that the addition of an information related to the code ownership of classes can make bug prediction models more accurate; these findings were later confirmed by other researchers [241, 242]. On a similar line of research, socio-technical factors have been also exploited for bug prediction: for instance, Posnett et al. [49] proposed two novel metrics based on the developer's social-network that may be used as predictors of faults in production, while Bird et al. [232] studied how developers contribute to source code and found that lack of collaboration and coordination are associated with an increase of the number of bugs.

Other researchers focused on improving bug prediction capabilities using the information coming from code smells [243], i.e., sub-optimal design implementations applied by developers. Khomh et al. [44] and Palomba et al. [46] indeed reported that such smells have a strong, negative impact on the bug proneness of source code. Following these findings, Taba et al. [244] studied how the addition of variables characterizing the presence of 13 different types of code smells can improve the performance of bug prediction models built using standard product metrics: they reported an improvement close to 13% in terms of F-Measure. Later on, Palomba et al. [147] showed that the intensity of code smells, namely a measure of their severity, can further improve bug prediction capabilities with respect to the work of Taba et al. [244].

Finally, a less explored yet worth to discuss bug prediction angle concerns the usage of

textual metrics. In the first place, the use of textual information has represented a promising and orthogonal dimension to improve software quality assessment [197, 200, 245]. For instance, textual information has already been used in several software engineering tasks or activities such as information retrieval [224, 245, 246], code smell detection [47, 247, 248], refactoring [249–251], and meaning extraction [193]. Perhaps more importantly, the addition of textual-related information has been proved to enhance the performance of bug prediction models. Marcus et al. [245] defined the Conceptual Cohesion of Classes (C3) and added it within a bug prediction model based on product metrics, finding that textual information can provide a boost of ≈23% in terms of F-Measure. Walid et al. [234] provided initial compelling evidence that a lack of coherence between code comments and corresponding source code (due to the missing update of code documentation) impacts the bug-proneness of code elements. Aman et al. [252] further explored the problem and found that certain types of code comments (e.g., those explaining functionalities implemented in a class) are associated with a higher bug-proneness of the code. Later on, Buse and Weimer [253] found that poor readability of source code contributes to the identification of buggy classes. These findings were also confirmed by Binkley et al. [254], who showed that a lower source code readability is often associated to an increase of the production code fault-proneness; When employed within predictive models, readability metrics provide an additional contribution that allow these models to perform ≈10% better than models built without using them.

Inspired by the results of these previous studies, in this work we aim at evaluating the effect of characterizing bug-prone methods considering alternative features, thus providing a wider overview of the performance achievable with method-level bug prediction.[2]

5.2.2 Method-level Bug-Prediction

While the seminal idea of lowering the granularity of bug prediction is to attribute to Menzies et al. [97] and Tosun et al. [98], the work by Giger et al. [35] was the first explicitly aimed at predicting bugs at method-level in object-oriented software systems. Giger et al. defined a set of product and process metrics to characterize a method and evaluated these metrics in three method-level bug prediction models, respectively based on: (i) product metrics, (ii) process metrics, and (iii) their combination. Giger et al. [35] found that both product and process metrics contribute to the identification of buggy methods and their combination achieves the best performance (i.e., F-Measure=86%). To produce the dataset used in their evaluation, Giger et al. took the following steps [35]: they (1) considered a large time frame in the history of 21 Java OSS systems, (2) focused on the methods present at the end of the time frame, (3) computed product metrics for each method at the end of the time frame, (4) computed process metrics (e.g., number of changes) for each method throughout the time frame, and (5) counted the number of bugs for each method throughout the time frame, relying on bug fixing commits. Finally, they used 10-fold cross-validation [115] to evaluate the three aforementioned models, considering the presence/absence of bug(s) in a method as the dependent (binary) variable. Similarly to the paper discussed above, Hata et al. [99] proposed a fine-grained prediction model in which they computed a number of historical metrics to predict the bug-proneness of Java methods. Their results reported that method-level predictions are more effective than file-

[2]This part is a novel contribution of this paper.

and package-level ones when considering the effort required by developers to locate and debug a potential defect in source code.

In this work, we re-evaluate the paper by Giger et al. [35] using data from subsequent releases (i.e., a release-by-release validation), which better models a real-case scenario where a prediction model is updated as soon as new information is available.[3] The choice of focusing on the work of Giger et al. [35] rather than the one of Hata et al. [99] is motivated by the fact that Giger et al. experimented with both code and process metrics (as opposed to Hata et al. who only considered process metrics), thus giving us the opportunity of providing a wider overview of the performance of method-level defect prediction.

5.3 Research Goals and Context

In this section, we define both the research questions guiding our study and the context of our investigation.

5.3.1 Research Questions

The *goal* of the empirical study is to re-evaluate how bug prediction can be applied at method-level, with the *purpose* of understanding the performance of models built using different sets of features. We start our investigation by replicating the study conducted by Giger et al. [35] on a partially overlapping set of software systems (but considering different moments in time) to evaluate the generalizability of their findings. Thus, we ask:

RQ$_1$. *How can code comments be categorized?*

While replicating the methodology proposed by Giger et al. [35], we detected some limitations concerning the validation approach: (1) it uses 10-fold cross-validation, which is at the risk of producing biased estimates in certain circumstances [116], (2) product metrics are considered only at the end of the time frame (while bugs are found *within* the time frame), and (3) the number of changes and the number of bugs were both considered in the same time frame (this *time-insensitive* validation strategy may have led to biased results). Thus, in the second part of our study we try to overcome the aforementioned limitations by re-evaluating the performance using data from subsequent releases. A *release-by-release* validation better models a real-case scenario where a prediction model is updated as soon as new information is available. Our expectation is that the performance is going to be weaker in this setting. This leads to our second research question:

RQ$_2$. *How often does each category occur in OSS and industrial projects?*

A second limitation we identify is related to the independent variables exploited by Giger et al. [35]. While they performed an extensive analysis of product and process metrics, the role of other types of information—that have been shown to boost the performance of bug prediction models in the past [234, 245, 254]—was not assessed. In the context of

[3] This part was previously presented at an academic software engineering conference [87].

our study, we assess the impact of three families of metrics such as: (1) textual features, whose aim is to capture the readability of the considered code as well as its alignment with code comments and their types; (2) code smells [243], which describe potential design flaws in source code; and (3) developer-related factors, that analyze properties related to the developers working on a system. Hence, we ask:

RQ$_3$. *How effective is an automated approach, based on machine learning, in classifying code comments in OSS and industrial projects?*

Table 5.1: Overview of the projects used in this study.

Projects	LOC	Developers	Releases	Methods	Buggy Methods
Ant	213k	15	4	42k	2.3k
Checkstyle	235k	76	6	31k	4.1k
Cloudstack	1.16M	90	2	85k	13.4k
Eclipse JDT	1.55M	22	33	810k	3.3k
Eclipse Platform	229k	19	3	7k	2.7k
Emf Compare	3.71M	14	2	9k	0.7k
Gradle	803k	106	4	73k	4.6k
Guava	489k	104	17	262k	1.2k
Guice	19k	32	4	9k	0.5k
Hadoop	2.46M	93	5	179k	5.8k
Lucene-solr	586k	59	7	213k	8.7k
Vaadin	7.06M	133	2	43k	11.3k
Wicket	328k	19	2	30k	4.9k
Overall	**19M**	**782**	**91**	**1.8M**	**63.5k**

5.3.2 Subject systems

The *context* of our work consists of 13 software systems whose characteristics are reported in Table 5.1. For each system, the table reports its size (in terms of LOCs) and how many developers contributed over the entire history, as well as the number of releases, methods, and buggy methods. In particular, we focus on systems implemented in Java (i.e., one of the most popular programming languages [117]), since both the metrics previously defined by Giger et al. [35] and the alternative features proposed in this study mainly target this programming language. In addition, we select projects whose source code and change history are publicly available (i.e., open-source software projects using a version control system) to enable the extraction of product, process, and alternative metrics.

Starting from the 81,327,803 open-source systems written in Java available at the time of the analysis on Github,[4] we first filter out those having less that 1,000 commits and more than 5,000 methods: this filter gives us a total of 6,753,654 systems. Finally, we randomly select 13 of them. Compared to Giger et al. [35], we consider fewer, but larger

[4]https://github.com

systems, which are composed of a much larger number of methods (1.8M vs 112,058) and bugs (63,400 vs 23,762). This choice allows us to test the effectiveness of method-level bug prediction on software systems of a different scale.

5.4 RQ_3. Evaluating Alternative Metrics

Our $\mathbf{RQ_3}$ seeks to evaluate the potential of alternative metrics for method-level bug prediction.

5.4.1 RQ_3 - Methodology

We define a set of metrics based on textual aspects of source code as well as on code smells, and developer-related factors, which we include in the method-level bug prediction models experimented in our previous research questions. In the following, we first describe the metrics and the rationale for their choice, and then we detail the model definition and evaluation process.

Textual Metrics. As summarized in Section 5.2, previous work [234, 252–254] highlighted that textual aspects of source code could impact its bug proneness and be useful when considered within bug prediction models. Thus, we first challenge these findings and explore the role of textual features when applied to method-level bug prediction. Table 5.2 lists the considered metrics. We include:

Code Readability. Based on the findings by Buse and Weimer [253] and Binkley et al. [254], we compute a measure of readability of source code. We directly employ the tool proposed in [253]: This outputs an index, which is a decimal score ranging between 0 and 1, where 0 represents unreadable code and 1 refers to easily readable code. This tool relies on a readability model composed of 19 metrics (including line length, number and length of identifiers, number of a predefined list of characters, branches, loops). To compute it, we rely on the publicly available version of the tool provided by the authors.[5] Code readability is not code complexity, which we already defined in other metrics previously.

Textual Coherence. Based on the findings reported by Walid et al. [234], we measure the textual coherence, i.e., the extent to which comment and source code of a method are aligned. To compute it, we first normalize comments and source code using a standard Information Retrieval (IR) process [119].[6] Then, we apply the Vector Space Model (VSM) [119] and measure the textual similarity between comments and source code (i.e., the vectors of VSM) using the cosine distance.

Comment Classification. Based on the findings by Aman et al. [252], who reported that different comments types are associated to different bug proneness, we classify source code comments exploiting the model proposed by Pascarella and Bacchelli [193]: It analyzes the text contained in a comment and classifies its semantic. Specifically,

[5] URL: `http://www.arrestedcomputing.com/readability`.

[6] In detail, we (i) separate composed identifiers, (ii) lower case the extracted words, (iii) remove special characters, programming keywords, and common English stop words, and (iv) stem words to their original roots via Porter's stemmer [255].

Table 5.2: List of the considered method-level textual metrics.

Metric name	Description (applies to method level)
Readability	Source code readability index [253]
Textual Coherence	Measure of the textual coherence between source and code comments [234]
Purpose	# of code comments used to describe the functionality of linked source code
Notice	# of code comments related to the description of warning, alerts, or messages
Under Development	# of code comments covers the topics related to the ongoing and future development
Style&IDE	# of code comments used to logically separate the code or provide special services
Metadata	# of code comments used to classify comments that define meta-information about the code
Other	# of code comments that do not fit into the previously defined categories

Pascarella and Bacchelli defined a hierarchical taxonomy with two levels: the first coarse-grained contains 6 categories, while the second fine-grained contains 16 sub-categories. In our study, we define 6 new method-level textual metrics based on the first level, as described by the last six rows in Table 5.2.

Code smells. These are poor design or implementation choices introduced by developers when maintaining and/or evolving software systems [243, 256]. The addition of the information on the design quality of classes into existing bug prediction models has been proved to improve bug identification capabilities [147, 244]. The contribution of measures of code smell severity appeared to be particularly useful in class-level bug prediction [147]. Following these findings, in the context of our work, we first identify code smell types that affect methods and then compute their intensity. We focus on:

Long Method. This smell refers to methods implementing more than one functionalities and is generally detected by considering its size [243]. Methods affected by this smell are poorly cohesive and possibly impact their understandability, change- and defect-proneness [46].

Long Parameter List. This smell refers to methods having a long list of parameters. Instances of this smell can lower the maintainability of methods and possibly indicate that the method is poorly cohesive [243].

Message Chains. This smell occurs when a client requests an object; this requires yet another one, and so on, thus creating a long concatenation of method calls [243]. This smell has been associated to a higher defect-proneness of the affected methods [41, 46].

The rationale behind the selection of these code smell types is twofold. In the first place, these have been reported to occur in software projects [46]. Perhaps more importantly, they influence the bug-proneness of the affected methods [41, 46], thus perfectly fitting the goal of our paper.

To detect them we rely on Decor [257], a method to define code smell detection rules using a Domain-Specific Language. The approach uses a set of rules, called "rule cards",[7] which describe the intrinsic characteristics that a method should have to be affected by a certain code smell type. In the case of *Long Method*, Decor identifies it by considering the number of lines of code of a method: If this is higher than 80, then a code smell is detected. As for *Long Parameter List*, it considers a method affected by this smell if it has more than three parameters. Finally, *Message Chains* instances are detected if a method contains a statement in which more than three method calls are performed.

According to several empirical studies [47, 247, 258], the accuracy of Decor is relatively high both in terms of precision and recall, with typical values of F-Measure around 75%. This makes the detector more accurate than other available tools [259] and, therefore, suitable for our study.

Once detected code smell instances, we compute their intensity. We follow a similar approach as previous work [47, 260]: as Decor classifies a method as smelly if a specific condition is satisfied, for instance, if its lines of code > 80, we can say that the higher the distance between the actual code metric value and the fixed threshold, the higher the intensity of the smell. We use this approach to compute the intensity of all the three smells considered.

Developer-related factors. Aspects capturing how developers work on source code and what is the change process they apply when performing software maintenance and evolution activities have been often successfully applied in bug prediction as they showed a great potential for improving predictive models [43, 89, 91, 114, 240].

These previous findings lead us to consider how developer-related factors work when employed in the context of method-level bug prediction. In particular, we focus on three orthogonal aspects such as:

Number of developers. In the first place, for each method of the considered dataset, we compute how many distinct developers worked on it over the history of the project. The contribution of this metric to bug prediction capabilities was firstly assessed by Ostrand et al. [113, 114], who reported that individual developer's data provides a limited boost to bug prediction models; Nevertheless, Di Nucci et al. [91] performed a larger empirical evaluation of the value of this metric, showing that it can improve the performance of these models by up to 10%. This is the reason why we seek to understand its value at a finer-level. We compute the metric by (i) mining all commits in the change history that changed a method *m* and (ii) counting the number of developers who made changes to it. To distinguish different developers, we consider the e-mail address they left on Github—we are aware that this computation may be imprecise in cases where a developer uses multiple e-mails when working on the project, however there is no practical way to solve this problem.

[7] http://www.ptidej.net/research/designsmells/

Code ownership. According to Bird et al. [240], developers having a higher experience on the source code they touch are less prone to introduce bugs. The authors assessed this relation by computing the code ownership of classes and measuring its impact on the performance of bug prediction models, finding that models including this metric have an accuracy that is 24% higher than those not including it as a feature. In our work, we compute code ownership at method-level by following the same approach of Bird et al. [240]: given a method m, we compute the ratio of number of commits that a contributor c has made on m with respect to the total number of commits made by c. Once computed the metric for all developers who contributed to m, we assign to the method the maximum ownership computed.

Entropy of code changes. Finally, we take into account the way developers make changes in a system and compute the entropy of changes originally defined by Hassan [89] for class-level bug prediction. This metric has been shown to strongly impact on the performance of predictive maintenance models [89, 261]. Following the algorithm of Hassan [89], we first identify all commits which modified a method m and then run a pattern-based technique that can detect the so-called *feature-introduction* modifications, namely changes applied to introduce new or enhancing existing features. Afterwards, the entropy of changes on m is computed exploiting the concept of Shannon entropy [262] as in the following equation:

$$entropy(m, \alpha) = -(p_{m,\alpha} \cdot \log_2 p_{m,\alpha}) \tag{5.1}$$

where $p_{m,\alpha}$ indicates the probability that m was changed received *feature introduction* modifications over its change history. This probability is computed considering the fraction between the number of *feature introduction* modifications applied on m in the change history over the total number of *feature introduction* modifications applied by developers.

It is important to remark that other alternative metrics have been proposed by researchers in the bug prediction field, like for instance socio-technical [49, 91] or code coverage features [263]. Being aware of those alternative metrics, we decided to exclude them from our study as they cannot be easily computed at method-level. For instance, let consider the case of socio-technical congruence [264]: this measures how much the organizational structure of a development community reflects the actual technical organization among developers. While the metric has been defined in terms of how much the developers' social network matches the modularization of packages, no definition is available with respect to the socio-technical congruence between developers' social network and division of methods among classes. Thus, we preferred to be conservative and not define any novel metric that would have deserved a separate validation before being used in our context. Rather, we relied on metrics that can be directly computed at method-level, e.g., the considered code smells are all directly computable at method-level.

Model Definition and Data Analysis. To test the contribution given by the considered families of features, we build five classes of method-level bug prediction models on the basis of those defined for **RQ**$_1$. Our methodology is inspired by the one of Bird et al. [240]: starting from the baseline one, namely the *product + process* one defined by Giger et al. [35], we progressively fed up the model with additional features, so that we can measure the

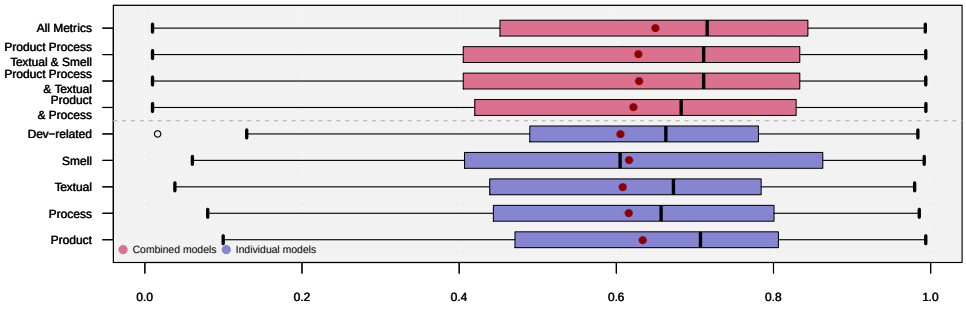

Figure 5.1: Comparison of the distribution of F-measure values considering the combination of product, process and textual metrics.

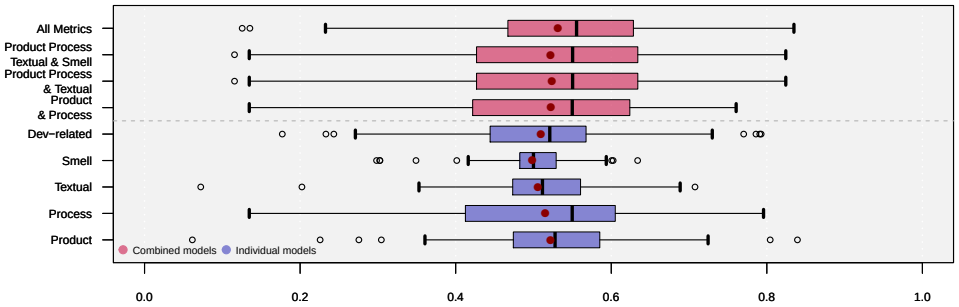

Figure 5.2: Comparison of the distribution of AUC-ROC values considering the combination of product, process and textual metrics.

contribution given by each family of features to the capabilities of method-level bug prediction (if any). Hence, we build models relying on (i) *product + process + textual* features, (ii) *product + process + textual + code smell-related* features, and (iii) *product + process + textual + code smell-related + developer-related* features. Furthermore, we also build models relying on the various families of features independently so that we can assess the independent value of each of them for method-level bug prediction.

As done in the context of the previous research questions, we apply the Random Over-Sampling and *Correlation-based Feature Selection* [125] algorithms to deal with data balancing and multicollinearity. The performance of all experimented method-level bug prediction models are evaluated using the same validation strategy (i.e., release-by-release) and evaluation metrics (i.e., precision, recall, F-Measure, and AUC-ROC) used in the context of the previous research question. We also conduct a statistical comparison of the performance of the prediction models considered. The Mann-Whitney test [265] is not recommended in the case of comparisons of multiple models over multiple datasets, since the performance of a specific model might vary between two datasets [266]. Thus, we compared the AUC-ROC values of the experimented models over the different systems using the Scott-Knott Effect Size Difference (ESD) test [133].

5.4.2 RQ_3 - Results

Figures 5.1 and 5.2 show the distribution of F-Measure and AUC-ROC, respectively, achieved by the prediction models built progressively using the various families of features considered in our study. We also report on the performance achieved by the models *only* relying on individual sets of features. Consistently with the methodology adopted for the first research question, we analyze how their performance varies when considering different classifiers (i.e., *Simple Logistic*, *Logistic*, *Multilayer Perceptron*, *Random Forest*, *J48*, *Decision Table*, and *Naive Bayes*). However, we limit the discussion of the results to *Random Forest* because it was the classifier providing slightly better performance (see Table **??**).[8]

When looking at the figures, we can see that the models built using individual sets of features have poor performance: for instance, the AUC-ROC of the *only textual* model is close to 53%, which indicates that it is just slightly better than a random classifier. A similar discussion can be drawn for the other individual models, thus confirming that taking those features alone does not give advantages when it comes to the prediction of defective methods. These results confirm what has been previously reported in the literature on the importance of combining multiple features to boost the performance of bug prediction models [90, 91, 100].

Turning the attention to the combined models, we notice that the progressive addition of metrics only gives a marginal contribution to the overall classification accuracy. Specifically, the inclusion of textual metrics into a model containing product and process metrics allows the model to have 4% and 5% more F-Measure and AUC-ROC. In the first place, this indicates that these metrics provide a limited amount of additional information to predict future bugs. At the same time, our findings represent a negative result with respect to the findings reported by all prior studies that we exploited to derive the textual features [234, 252–254]; indeed, we could not find any textual measure able to significantly increase the performance of the experimented prediction models.

The discussion is similar when considering the addition of code smell-related information. According to previous findings in the field [147, 244], including a measure of code smell severity within models relying on a combination of process and product metrics provides an increase of ≈15% in terms of F-Measure. Unfortunately, this is not the case when lowering the granularity of the predictions. In our case, indeed, the F-Measure of the *product + process + textual + code smell-related* model is even lower than the one not including any smell-related information (-2%). This may indicate that code smells computed at method-level have limited predictive power when compared to class-level code smells. Thus, our findings are again negative with respect to previous work [147, 244]. Perhaps more importantly, we could not confirm the results of D'Ambros et al. [41], in which the authors reported that one of the considered smells, i.e., *Message Chains*, is the one that mostly affects the bug-proneness of source code methods.

As for the developer-related factors (named *All metrics* in Figures 5.1 and 5.2), their inclusion provides an increase of 4% with respect to the second best performing model (the *product + process + textual* one). So, also in this case, we claim that the contribution is marginal and that we could not confirm the role of developer-related factors for bug prediction when the granularity is that of methods.

[8] A complete overview of the results achieved when using other classifiers is available in our appendix [129].

The results discussed so far are all statistically significant. According to the ranking of the performance provided by the Scott-Knott ESD test [133], there is no model performing statistically better than the others, thus indicating that the addition of alternative metrics does not boost the performance of bug prediction.[9]

To conclude the discussion, based on the findings discussed so far we argue that the research on method-level bug prediction still needs notable steps to do for better supporting developers. Our paper provides initial compelling evidence of the need of novel, specific metrics able to capture method-level information that actually relate to the bug-proneness of methods.

Result 3: The addition of alternative features based on textual, code smells, and developer-related factors improves the performance of the existing models only marginally, if at all.

5.5 Threats to Validity

We describe the factors that can affect the validity of our empirical results.

Threats to Construct Validity. A first factor influencing the relationship between theory and observation is related to the dataset exploited. In our study, we relied on the same methodology previously adopted by Giger et al. [35] to build our own repository of buggy methods, i.e., we first retrieved bug-fixing commits using the textual-based technique proposed by Fisher et al. [69] and then considered as buggy the methods changed in that commits. While we cannot exclude possible imprecision and/or incompleteness of the data used in this study, we have re-evaluated the performance of the tool by Fisher et al. in our context finding that it could detect buggy commits with a precision of 84% correctly. Still in this category, we re-implemented the product and process metrics used to build the experimented models. This was due to the lack of a publicly available tool. When re-implementing such metrics we faithfully followed the descriptions provided by Giger et al. [35]. As for the alternative metrics, instead, we used available tools whenever possible (e.g., in the case of the comment classification [193] or when detecting method-level code smells [257]). To enable and stimulate the replicability of our study, we made all tools and scripts exploited publicly available in our online appendix [129]. As for the selection of the classifier to use when building the bug prediction model, we tested the performance of different classifiers, finding Random Forest to be the one providing the best performance. All the tested classifiers use the default parameters, since finding the best configuration for all of them would have been too expensive [143]. Future work can be devised to investigate the impact of parameters' configuration on our findings.

Threats to Conclusion Validity. A first point of discussion regards the data pre-processing techniques adopted before the construction of the experimented bug prediction models. To ensure that the results would not have been biased by confounding effects such as data unbalance [121] or multi-collinearity [124], we adopted formal procedures aimed at (i) over-sampling the training sets [121] and (ii) removing non-relevant independent variables through feature selection [125]. As for the evaluation of the models, we

[9]Detailed statistical results are reported in our appendix [129].

complemented the results concerning the F-measure by relying on a *threshold-independent* metric such as the AUC-ROC. Furthermore, we supported our findings with an appropriate statistical test like the Scott-Knott ESD one [133].

Threats to External Validity. This category refers to the generalizability of our findings. While in the context of this work we analyzed software projects having different size and scope, we limited our focus to Java systems because some of the tools exploited to compute the considered metrics only work for this programming language (e.g., certain code smells have been only defined for Java [243]). Thus, we cannot claim generalizability concerning systems written in different languages as well as to projects belonging to industrial environments. Similarly, we considered a subset of the available metrics in each of the five families of features considered: in particular, we limited ourselves to the analysis of the metrics which previous works have analyzed, while we cannot exclude that different results could be achieved when considering different metrics (e.g., other method-level code smell types).

5.6 Conclusion

We investigated (i) the performance of different types of method-level bug prediction models when applied in a real-case scenario and (ii) the contribution given by textual features to existing bug prediction models. The main contributions made by our study are:

1. A re-evaluation on different systems/timespans of previously defined method-level bug prediction models. The results confirm previous findings in the field [35].
2. An empirical analysis of how the performance of existing method-level bug prediction models changes when applied to a more realistic, release-by-release scenario. Our results provide evidence that current method-level bug prediction models do not dramatically outperform a random classifier; hence we reveal the need for further research in this area.
3. An empirical analysis of whether the performance of existing method-level bug prediction models can be improved by considering alternative features. Our results reveal that the overall prediction capabilities lead to negligible improvements.
4. An online appendix [129] that reports the dataset and all the additional analyses performed in the work described in this paper.

Based on the results achieved so far, our future agenda includes (i) the replication of our study on a broader set of systems also considering ensemble methods [54, 132], (ii) the investigation of novel metrics to properly work for method-level bug prediction, and (iii) an *in-vivo* analysis of the capabilities of method-level bug prediction models, involving practitioners during their daily activities [94].

6

Information Needs in Contemporary Code Review

Contemporary code review is a widespread practice used by software engineers to maintain high software quality and share project knowledge. However, conducting proper code review takes time and developers often have limited time for review. In this paper, we aim at investigating the information that reviewers need to conduct a proper code review, to better understand this process and how research and tool support can make developers become more effective and efficient reviewers.

Previous work has provided evidence that a successful code review process is one in which reviewers and authors actively participate and collaborate. In these cases, the threads of discussions that are saved by code review tools are a precious source of information that can be later exploited for research and practice. In this paper, we focus on this source of information as a way to gather reliable data on the aforementioned reviewers' needs. We manually analyze 900 code review comments from three large open-source projects and organize them in categories by means of a card sort. Our results highlight the presence of seven high-level information needs, such as knowing the uses of methods and variables declared/modified in the code under review. Based on these results we suggest ways in which future code review tools can better support collaboration and the reviewing task.

This chapter is based on

L. Pascarella, D. Spadini, F. Palomba, A. Bacchelli. Information Needs in Contemporary Code Review, CSCW'18 [267]

6.1 Introduction

Peer code review is a well-established software engineering practice aimed at maintaining and promoting source code quality, as well as sustaining development community by means of knowledge transfer of design and implementation solutions applied by others [7]. Contemporary code review, also known as *Modern Code Review* (MCR) [7, 268], represents a lightweight process that is (1) informal, (2) tool-based, (3) asynchronous, and (4) focused on inspecting new proposed code changes rather than the whole codebase [14]. In a typical code review process, developers (the *reviewers*) other than the code change author manually inspect new committed changes to find as many issues as possible and provide feedback that needs to be addressed by the author of the change before the code is accepted and put into production [269].

Modern code review is a collaborative process in which reviewers and authors conduct an asynchronous online discussion to ensure that the proposed code changes are of sufficiently high quality [7] and fit the project's direction [15] before they are accepted. In code reviews, discussions range from low-level concerns (*e.g.*, variable naming and code style) up to high-level considerations (*e.g.*, fit within the scope of the project and future planning) and encompass both functional defects and evolutionary aspects [270]. For example a reviewer may ask questions regarding the structure of the changed code [271] or clarifications about the rationale behind some design decisions [272], another reviewer may respond or continue the thread of questions, and the author can answer the questions (*e.g.*, explaining the motivation that led to a change) and implement changes to the code to address the reviewers' remark.

Even though studies have shown that modern code review has the *potential* to support software quality and dependability [16, 268, 273], researchers have also provided strong empirical evidence that the outcome of this process is rather erratic and often unsatisfying or misaligned with the expectations of participants [7, 270, 274]. This erratic outcome is caused by the cognitive-demanding nature of reviewing [21], whose outcome mostly depends on the time and zeal of the involved reviewers [268].

Based on this, a large portion of the research efforts on tools and processes to help code reviewing is explicitly or implicitly based on the assumption that reducing the *cognitive load* of reviewers improves their code review performance [21]. In the current study, we continue on this line of better supporting the code review process through the reduction of reviewers' cognitive load. Specifically, *our goal is to investigate the information that reviewers need to conduct a proper code review.* We argue that—if this information would be available at hand—reviewers could focus their efforts and time on correctly evaluating and improving the code under review, rather than spending cognitive effort and time on collecting the missing information. By investigating reviewers' information needs, we can better understand the code review process, guide future research efforts, and envision how tool support can make developers become more effective and efficient reviewers.

To gather data about reviewers' information needs we turn to one of the collaborative aspects of code review, namely the discussions among participants that happen during this process. In fact, past research has shown that code review is more successful when there is a functioning collaboration among all the participants. For example, Rigby *et al.* reported that the efficiency and effectiveness of code reviews are most affected by the amount of review participation [275]; Kononenko *et al.* [169] showed that review participation metrics

are associated with the quality of the code review process; McIntosh *et al.* found that a lack of review participation can have a negative impact on long-term software quality [16, 276]; and Spadini *et al.* studied review participation in production and test files, presenting a set of identified obstacles limiting the review of code [184]. For this reason, from code review communication, we expect to gather evidence of reviewers' information needs that are solved through the collaborative discussion among the participants.

To that end, we consider three large open-source software projects and manually analyze 900 code review discussion threads that started from a reviewer's question. We focus on what kind of questions are asked in these comments and their answers. As shown in previous research [277–280], such questions can implicitly represent the information needs of code reviewers. In addition, we conduct four semi-structured interviews with developers from the considered systems and one focus group with developers from a software quality consultancy firm, both to challenge our outcome and to discuss developers' perceptions. Better understanding what reviewers' information needs are can lead to reduced cognitive load for the reviewers, thus leading, in turn, to better and shorter reviews. Furthermore, knowing these needs helps driving the research community toward the definition of methodologies and tools able to properly support code reviewers when verifying newly submitted code changes.

Our analysis led to seven high-level information needs, such as knowing the uses of methods and variables declared/modified in the code under review, and their analysis in the code review lifecycle. Among our results, we found that the needs to know (1) whether a proposed alternative solution is valid and (2) whether the understanding of the reviewer about the code under review is correct are the most prominent ones. Moreover, all the reviewers' information needs are replied to within a median time of seven hours, thus pointing to the large time savings that can be achieved by addressing these needs through automated tools. Based on these results, we discuss how future code review tools can better support collaboration and the reviewing task.

6.2 Background and Related Work

This section describes the basic components that form a modern code review as well as the literature related to information needs and code review participation.

6.2.1 Background: The code review process

Figure 6.1 depicts a code review (pertaining to the OpenStack project) done with a typical code review tool. Although this is one of the many available review tools, their functionalities are largely the same [18]. In the following we briefly describe each of the components of a review as provided by code review tools.

Code review tools provide an ID and a status (part ① in Figure 6.1) for each code review, which are used to track the code change and know whether it has been *merged* (*i.e.*, put into production) or *abandoned* (*i.e.*, it has been evaluated as not suitable for the project). Code review tools also allow the change author to include a textual description of the code change, with the aim to provide reviewers with more information on the rationale and behavior of the change. However, past research has provided evidence that the quality and level of detail of the descriptions that accompany code changes are often suboptimal

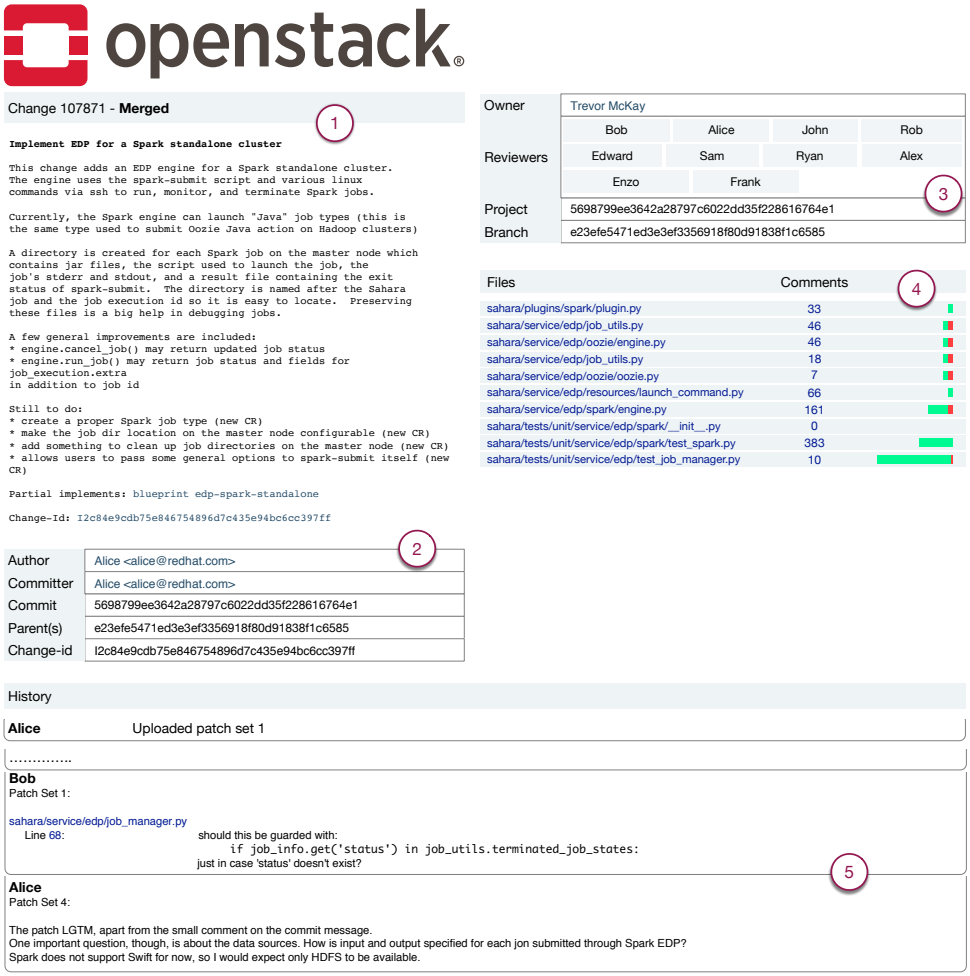

Figure 6.1: Example of code review mined from Gerrit.

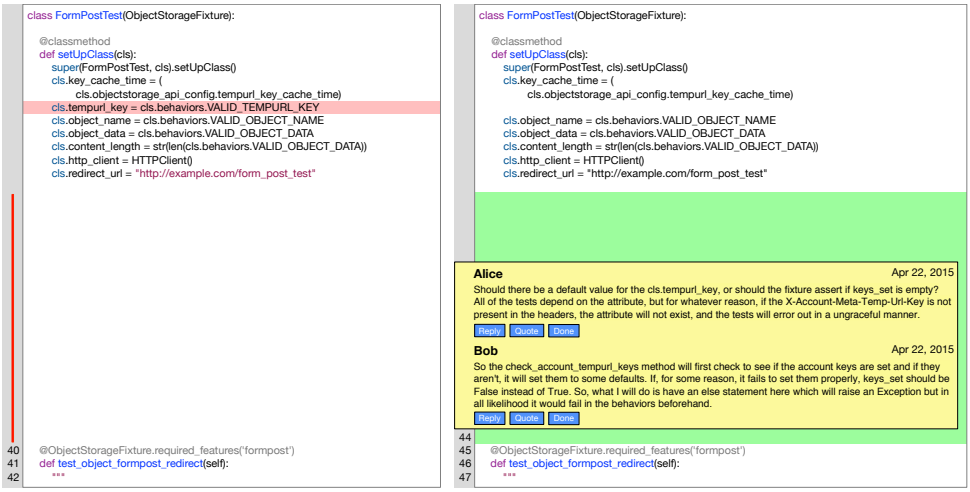

Figure 6.2: Example of code review comments mined from Gerrit.

[271], thus making it harder for reviewers to properly understand the code change through this support. The fact that the change description is often not optimal strengthens the importance of the goal of our study: An improved analysis of developers' needs in code review can provide benefits in terms of review quality [169].

The second component of a typical code review tool is a view on the technical meta-information on the change under review (part ② in Figure 6.1). This meta-information include author and committer of the code change, commit ID, parent commit ID, and change ID, which can be used to track the submitted change over the history of the project.

Part ③ of the tool in Figure 6.1 reports, instead, more information on who are the reviewers assigned for the inspection of the submitted code change, while part ④ lists the source code files modified in the commit (*i.e.*, the files on which the review will be focused).

Finally, part ⑤ is the core component of a code review tool and the one that involves most collaborative aspects. It reports the discussion that author and reviewers are having on the submitted code change. In particular, reviewers can ask clarifications or recommend improvements to the author, who can instead reply to the comments and propose alternative solutions. This mechanism is often accompanied by the upload of new versions of the code change (*i.e.*, revised patches or *iterations*), which lead to an iterative process until all the reviewers are satisfied with the change or decide to not include it into production. Figure 6.2 shows a different view that contains both reviews and authors comments. In this case, the involved developers discuss about a specific line of code, as opposed to Alice from the previous example who commented on the entire code change (Figure 6.1, end of part ⑤).

6.2.2 Related Work

Over the last decade the research community spent a considerable effort in studying code reviews (*e.g.*, [184, 268, 270, 281–284]). In this section, we compare and contrast our work to previous research in two areas: first, we consider studies that investigate the information needs of developers in various contexts, then we analyze previous research that focused

on code review discussion, participation, and time.

Information needs

Breu *et al.* [279] conducted a study—which has been a great inspiration to the current study we present here—on developers' information needs based on the analysis of collaboration among users of a software egineering tool (*i.e.*, issue tracking system). In their study, the authors have quantitatively and qualitatively analyzed the questions asked in a sample of 600 bug reports from two open-source projects, deriving a set of information needs in bug reports. The authors showed that active and ongoing participation were important factors needed for making progress on the bugs reported by users and they suggested a number of actions to be performed by the researchers and tool vendors in order to improve bug tracking systems.

Ko *et al.* [277] studied information needs of developers in collocated development teams. The authors observed the daily work of developers and noted the types of information desired. They identified 21 different information types in the collected data and discussed the implications of their findings for software designers and engineers. Buse and Zimmermann [280] analyzed developers' needs for software development analytics: to that end, they surveyed 110 developers and project managers. With the collected responses, the authors proposed several guidelines for analytics tools in software development.

Sillito *et al.* [285] conducted a qualitative study on the questions that programmers ask when performing change tasks. Their aim was to understand what information a programmer needs to know about a code base while performing a change task and also how they go about discovering that information. The authors categorized and described 44 different kinds of questions asked by the participants. Finally, Herbsleb *et al.* [286] analyzed the types of questions that get asked during design meetings in three organizations. They found that most questions concerned the project requirements, particularly what the software was supposed to do and, somewhat less frequently, scenarios of use. Moreover, they also discussed the implications of the study for design tools and methods.

The work we present in this paper is complementary with respect to the ones discussed so far: indeed, we aim at making a further step ahead investigating the information needs of developers that review code changes with the aim of deepening our understanding of the code review process and of leading to future research and tools to better support reviewers in conducting their tasks.

Code Review Participation and Time

Extensive work has been done by the software engineering research community in the context of code review participation. Abelein *et al.* [287] investigated the effects of user participation and involvement on system success and explored which methods are available in literature, showing that it can have a significant correlation with system quality. Thongtanunam *et al.* [288] showed that reviewing expertise (which is approximated based on review participation) can reverse the association between authoring expertise and defect-proneness. Even more importantly, Rigby *et al.* [275] reported that the level of review participation is the most influential factor in the code review efficiency. Furthermore, several studies have suggested that patches should be reviewed by at least two developers to maximize the number of defects found during the review, while minimizing the reviewing workload on the development team [14, 289–291].

Thongtanunam *et al.* [276] showed that the number of participants that are involved with a review has a large relationship with the subsequent defect proneness of files in the Qt system: A file that is examined by more reviewers is less likely to have post-release defects. Bavota *et al.* [292] also found that the patches with low number of reviewers tend to have a higher chance of inducing new bug fixes. Moreover, McIntosh *et al.* [16, 168] measured review investment (i.e., the proportion of patches that are reviewed and the amount of participation) in a module and examined the impact that review coverage has on software quality. They found that patches with low review investment are undesirable and have a negative impact on code quality. In a study of code review practices at Google, Sadowski *et al.* [293] found that Google has refined its code review process over several years into an exceptionally lightweight one, which–in part–seems to contradict the aforementioned findings. Although the majority of changes at Google are small (a practice supported by most related work [294]), these changes mostly have one reviewer and have no comments other than the authorization to commit. Ebert *et al.* [295] made the first step in identifying the factors that may confuse reviewers since confusion is likely impacts the efficiency and effectiveness of code review. In particular, they manually analyzed 800 comments of code review of Android projects to identify those where the reviewers expressed confusion. Ebert *et al.* found that humans can reasonably identify confusion in code review comments and proposed the first binary classifier able to perform the same task automatically; they also observed that identifying confusion factors in inline comments is more challenging than general comments. Finally, Spadini *et al.* [184] analyzed more than 300,000 code reviews and interviewed 12 developers about their best practices when reviewing test files. As a result, they presented an overview of current code review practices, a set of identified obstacles limiting the review of test code, and a set of issues that developers would like to see improved in code review tools. Based on their findings, the authors proposed a series of recommendations and suggestions for the design of tools and future research.

Furthermore, previous research investigated how to make a code review shorter, hence making patches be accepted at a faster rate. For example, Jiang *et al.* [296] showed that patches developed by more experienced developers are more easily accepted, reviewed faster, and integrated more quickly. Additionally, authors stated that reviewing time is mainly impacted by submission time, the number of affected subsystems by the patch and the number of requested reviewers. Baysal *et al.* [297] showed that size of the patch or the part of the code base being modified are important factors that influenced the time required to review a patch, and are likely related to the technical complexity of a given change.

Recently, Chatley and Jones have proposed an approach aimed at enhancing the performance of code review [298]. The authors built Diggit to automatically generate code review comments about potentially missing changes and worrisome trends in the growth of size and complexity of the files under review. By deploying Diggit at a company, the authors found that the developers considered Diggit's comments as actionable and fixed them with an overall rate of 51%, thus indicating the potential of this approach in supporting code review performance.

Despite many studies showing that code review participation has a positive impact on the overall software development process (*i.e.*, number of post-release defects and time spent in reviewing), none of these studies focused on what are the developers needs

when performing code review. To fill this gap, our study aims at increasing our empirical knowledge on this field by mean of quantitative and qualitative research, with the potential of reducing the cognitive load of reviewers and the time needed for the review.

6.3 Methodology

The *goal* of our study is to increase our empirical knowledge on the reviewers' needs when performing code review tasks, with the *purpose* of identifying promising paths for future research on code review and the next generation of software engineering tools required to improve collaboration and coordination between source code authors and reviewers. The perspective is of researchers, who are interested in understanding what are the developers' needs in code review, therefore, they can more effectively devise new methodologies and techniques helping practitioners in promoting a collaborative environment in code review and reduce discussion overheads, thus improving the overall code review process.

Starting from a set of discussion threads between authors and reviewers, we start our investigation by eliciting the actual needs that reviewers have when performing code review:

- **RQ$_1$:** *What reviewers' needs can be captured from code review discussions?*

Specifically, we analyze the types of information that reviewers may need when reviewing, we compute the frequency of each need, and we challenge our outcome with developers from the analyzed systems and from an external company. Thus, we have three sub-questions:

- **RQ$_{1.1}$:** *What are the kinds of information code reviewers require?*
- **RQ$_{1.2}$:** *How often does each category of reviewers' needs occur?*
- **RQ$_{1.3}$:** *How do developers' perceive the identified needs?*

Once investigated reviewers' needs from the reviewer perspective, we further explore the collaborative aspects of code review by asking:

- **RQ$_2$:** *What is the role of reviewers' needs in the lifecycle of a code review?*

Specifically, we first analyze how much each reviewers' need is accompanied by a reply from the author of the code change: in other words, we aim at measuring how much authors of the code under review interact with reviewers to make the applied code change more comprehensible and ease the reviewing process. To complement this analysis, we evaluate the time required by authors to address a reviewer's need; also in this case, the goal is to measure the degree of collaboration between authors and reviewers. Finally, we aim at understanding whether and how the reviewers' information needs vary at different iterations of the code review process. For instance, we want to assess whether some specific needs arise at the beginning of the process (*e.g.*, because the reviewer does not have enough initial context to understand the code change) or, similarly, if clarification questions only

appear at a later stage (*e.g.*, when only the last details are missing and the context is clear). Accordingly, we structure our second research question into three sub-questions:

- **RQ$_{2.1}$:** *What are the reviewers' information needs that attract more discussion?*
- **RQ$_{2.2}$:** *How long does it take to get a response to each reviewers' information need?*
- **RQ$_{2.3}$:** *How do the reviewers' information needs change over the code review process?*

The following subsections describe the method we use to answer our research questions.

6.3.1 Subject Systems

The first step leading to address our research goals is the selection of a set of code reviews that might be representative for understanding the reviewers' needs when reviewing source code changes. We rely on the well-known Gerrit platform,[1] which is a code review tool used by several major software projects. Specifically, Gerrit provides a simplified web based code review interface and a repository manager for Git.[2] From the open-source software systems using Gerrit, we select three: OpenStack,[3] Android,[4] and QT.[5] The selection was driven by two criteria: (i) These systems have been extensively studied in the context of code review research and have been shown to be highly representative of the types of code review done over open-source projects *et al.* [16, 168, 292]; (ii) these systems have a large number of active authors and reviewers over a long development history.

6.3.2 Gathering Code Review Threads

We automatically mine Gerrit data by relying on the publicly available APIs it provides. For the considered projects, the number of code reviews is over one million: this makes the manual analysis of all of them practically impossible. Thus, as done by Breu *et al.* [279], we select a random subset composed of 300 code reviews per project, for which we identify up to 1,800 messages (*i.e.*, we extract a total of 900 code review threads). Since we are interested in discussions, we take into account only closed code reviews by considering both merged and abandoned patches, while we do not consider recently opened or pending requests.

We detect reviewers' questions (considering the presence of a '?' sign) that start a discussion thread and we extract all the subsequent comments (made by the author, the reviewer, or other developers) in the whole thread.

The considered threads refer to both patch sets and inline discussions. To better illustrate the mining process of general discussions, Figure 6.1 reports a code review extracted from OpenStack. As shown in the bottom of the figure (part ⑤), author and reviewers opened a discussion on the performed change. Figure 6.2 shows a thread of discussion started at line level. In both cases, all the comments among the participants

[1] https://www.gerritcodereview.com/
[2] https://git-scm.com/
[3] https://review.openstack.org/
[4] https://android-review.googlesource.com/
[5] https://codereview.qt-project.org

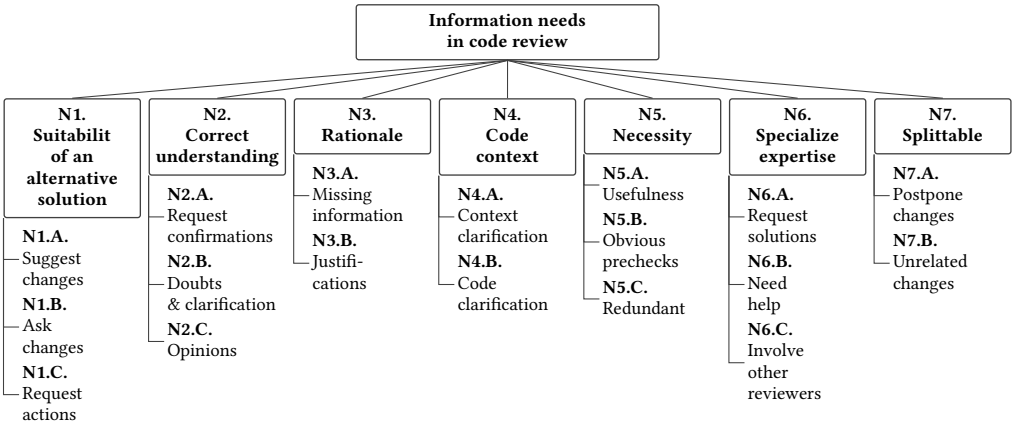

Figure 6.3: The taxonomy of reviewers' information needs that emerged from our analysis

represent the types of discussion threads that we use to detect the information needs of reviewers.

For each identified *thread*, we store the following information:

- the *Gerrit id* of the code review;
- the *revision id* that identifies the patch set of a code review;
- the opening *question*, the *answers*, and the *source code* URL identifier of the change;
- the practitioner *name e.g.*, author or reviewer;
- the code review *status*, *i.e.*, whether it is merged or abandoned;
- the *size* of the thread counting the number of comments present into discussion;
- the creation and the update *time*.

We use the aforementioned pieces of information to answer our research questions as detailed in the following.

6.3.3 RQ_1 - Identifying the Reviewers' Needs from Code Review Discussions

To answer $\mathbf{RQ}_{1.1}$, we manually identify the reviewers' needs in code review by following a similar strategy as done in previous work on information needs [277, 279, 280, 285, 286]. Specifically, we perform a card sorting method [76] that involves all the authors of this paper (2 graduate students, 1 research associate, and 1 faculty member - who have at least seven years of programming experience). From now on, we refer to them as the *inspectors*. This method represents a well-established sorting technique that is used in information architecture with the aim of creating mental models and allowing the definition of taxonomies from input data [76]. In our case, it is used to organize code review

threads into hierarchies and identify common themes. We rely on code review threads (*i.e.*, questions and answers) to better understand the meaning behind reviewers' questions that may implicitly define the reviewers' need. Finally, we apply an *open card sorting*: We have no predefined groups of reviewers' information needs, rather the needs emerge and evolve during the procedure. In our case, the process consists of the three iterative sessions described as follow.

Iteration 1: Initially, two inspectors (the first two authors of this paper) independently analyze an initial set of 100 OpenStack code review threads each. Then, they open a discussion on the reviewers' needs identified so far and try to reach a consensus on the names and types of the assigned categories. During the discussion, also the other two inspectors participate with the aim of validating the operations done in this iteration and suggesting possible improvements. As an output, this step provides a draft categorization of reviewers' needs.

Iteration 2: The first two inspectors re-categorize the 100 initial reviewers' needs according to the decisions taken during the discussion; then, they use the draft categorization as a basis for categorizing the remaining set of 200 code review threads belonging to OpenStack. This phase is used for both assessing the validity of the categories emerging from the first iteration (by confirming some of them and redefining others) and for discovering new categories. Once this iteration is completed, all the four inspectors open a new discussion aimed at refining the draft taxonomy, merging overlapping categories or better characterizing the existing ones. A second version of the taxonomy is produced.

Iteration 3: The first two inspectors re-categorize the 300 code review threads previously analyzed. Afterwards, the first inspector classifies the reviewers' needs concerning the two remaining considered systems. In doing so, the inspector tries to apply the defined categories on the set of code review threads of Android and QT. However, in cases where the inspector cannot directly apply the categories defined so far, the inspector reports such cases to the other inspectors so that a new discussion is opened. Unexpectedly this event did not eventually happen in practice; in fact, the inspector could fit all the needs in the previously defined taxonomy, even when considering new systems. This result suggests that the categorization emerging from the first iterations reached a saturation [299], valid at least within the considered sample of threads.

Additional validation. To further check and confirm the operations performed by the first inspector, the third author of this paper—who was only involved in the discussion of the categories, but not in the assignment of the threads into categories—independently analyzed all the code review threads belonging to the three considered projects. The inspector classified all the 900 threads according to the second version of the taxonomy, as defined through iteration 2. The inspector did not need to define any further categories (thus suggesting that the taxonomy was exhaustive for the considered sample), however in six cases there were a disagreement between the category he assigned and the one assigned by the first author: as a consequence, the two authors opened a discussion in order to reach an agreement on the actual category to assign to those code review threads. Overall, the inter-rater agreement between this inspector and the first one, computed using the Krippendorff's k [300], was 98%.

Following this iterative process, we defined a hierarchical categorization composed of two layers. The top layer consists of *seven* categories, while the inner layer consists of 18 subcategories. Figure 6.3 depicts the identified top- and sub-categories. During the iterative sessions, ≈4% of the analyzed code review threads are discarded from our analysis since they do not contain useful information to understand the reviewers' needs. We assign these comments to four temporary sub-categories that indicate the reasons why they are discarded (e.g., they are noise or sarcastic comments), successively, we gathered these comments, in an additional top-category *Discarded*.

To answer $\mathbf{RQ}_{1.1}$, we report the reviewers' needs belonging to the categories identified in the top layer.

Subsequently, to answer $\mathbf{RQ}_{1.2}$ and understand how frequently each category of our needs appears, we verify how many information needs are assigned to each category. In this way, we can overview the most popular reviewers' needs when performing code review tasks. We answer this research question by presenting and discussing bar plots showing the frequency of each identified category.

To answer $\mathbf{RQ}_{1.3}$, we discuss the outcome of the previous sub-RQs with developers of the three considered systems and an external company. This gives us the opportunity to challenge our findings, triangulate our results, and complement our vision on the problem.

Table 6.1: Interviewees' experience (in years) and their working context.

ID	Years as developer	Years as reviewer	Working context
P_1	15	10	OpenStack
P_2	20	10	OpenStack
P_3	25	20	Qt
P_4	10	10	Android
FG_1	8	7	Company A
FG_2	10	10	Company A
FG_3	7	5	Company A

Interviews with reviewers from the subject systems. To organize the discussion with the developers of Android, OpenStack, and Qt, we use semi-structured interviews–a format that is often used in exploratory investigations to understand phenomena and seek new insights [301]. A crucial step in this analysis is represented by the recruitment strategy, *i.e.*, the way we select and recruit participants for the semi-structured interviews. With the aim of gathering feedback and opinions from developers having a solid experience with the code review practices of the considered projects, we select only developers who had conducted at least 100 reviews[6] in their respective systems. Then, we randomly select 10 per system and invite them via email to participate in an online, video interview. Four experienced code reviewers accepted to be interviewed: two from OpenStack, one from Qt, and one from Android. The response rate achieved (17%) is in line with the one achieved by many previous works involving developers [47, 302, 303]. Table 6.1 summarizes the interviewees' demographic.

[6]This minimum number of reviews to ensure an appropriate experience of the interviewees is aligned with the numbers used in previous studies on code review (*e.g.*, [7]).

The interviews are conducted by the first two authors of this work via Skype. With the participants' consent all the interviews are recorded and transcribed for analysis. Each interview starts with general questions about programming and code reviews experience. In addition, we discuss whether the interviewees consider code reviews important, which tool they prefer, and generally how they conduct reviews. Overall, we organize the interview structure around five sections:

1. General information regarding the developer;
2. General perceptions on and experience with code review;
3. Specific information needs during code review;
4. Ranking of information needs during code review;
5. Summary.

The main focus regarding the information needs is centered around points 3 and 4: We iteratively discuss each of the categories emerged from our analysis (also showing small examples where needed). Afterwards, we discuss the following main questions with each interviewee:

1. What is your experience with <category>?
2. Do you think <category> is important to successfully perform a code review? Why?
3. Do you think current code review tools support this need?
4. How would you improve current tools?

Our goal with these questions is to allow us to better understand the relevance of each developer's need and whether developers feel it is somehow incorporated in current code review tools or, if not, how they would envision this need incorporated. Successively, we ask developers to rank the categories according to their perceived importance. Our goal is to understand what the interviewees perceive as the most important needs and why. To conclude the interview, the first two authors of this paper summarize the interview, and before finalizing the meeting, these summaries are presented to the interviewee to validate our interpretation of their opinions.

Focus group with an external commercial company. While the original developers provide an overview of the information needs identified in the context of the systems analyzed in this study, our findings may not provide enough diversity. To improve this aspect, we complement the aforementioned semi-structured interviews with an additional analysis targeting experts in assessing the source code quality of systems. In particular, we recruited three employees from a firm in Europe specialized in software quality assessments for their customers. The mission of the firm is the definition of techniques and tools able to diagnose design problems in the clients' source code, with the purpose of providing consultancy on how to improve the productivity of their clients' industrial developers. Our decision to involve these quality experts is driven by the willingness to receive authoritative opinions from professionals who are used to perform code reviews for their customers.

The three participants have more than 15 years in assess code quality and more than 10 years of experience in code review.

In this case we proceed with a *focus group* [70, 71] because it better fits our methodology. Indeed, this technique is particularly useful when a small number of people is available for discussing about a certain problem [70, 71] and consists of the organization of a meeting that involves the participants and a moderator. The moderator starts the discussion by asking general questions on the topic of interest and then leaves the participants to openly discuss about it with the aim of gathering additional qualitative data useful for the analysis of the results. In the context of this paper, the first two authors of the paper are the moderators in a meeting directly organized in the consultancy firm. The focus group is one hour long and the participants reflected on and discuss the information needs we identified and what are the factors influencing their importance. From this analysis, our aim is also to better understand the external validity of our taxonomy.

6.3.4 RQ_2 - On the role of reviewers' needs in the lifecycle of a code review

In the context of the second research question we perform a fine-grained investigation of the role of reviewers' needs in code review. We analyze which of them capture more replies, what is the time required for getting an answer, and whether reviewers' needs change throughout the iterations.

Specifically, we consider code review threads related to the same reviewer's need independently. Then, to answer $\mathbf{RQ}_{2.1}$ we computed the number of replies that each group received: this is a metric that represents how much in deep reviewers and authors should interact to be able to exchange the information necessary to address the code review. We do not assess the quality of the responses, since we aim at reporting quantitative observations on the number of answers provided by authors to a reviewer's need.

As for $\mathbf{RQ}_{2.2}$, this represents a follow-up of the previously considered aspect. Indeed, besides assessing the number of replies for each reviewers' need, we also measure the time (in terms of minutes) needed to get a response. This complementary analysis can possibly provide insights on whether certain needs require authors to spend more time to make their change understandable, thus providing information on the relative importance of each need which might be further exploited to prioritize software engineering research effort when devising and developing new techniques to assist code reviewers.

Finally, to answer $\mathbf{RQ}_{2.3}$ and understand how the reviewers' needs change over the code review iterations, we measure the number of times a certain need appears in each iteration of a code review. This analysis may possibly lead to observations needed by the research community to promptly provide developers with appropriate feedback during the different phases of the code review process.

As a final step of our methodology, we compute pairwise statistical tests aimed at verifying whether the observations of each sub-research question are statistically significant. We apply the Mann-Whitney test [304]. This is a non-parametric test used to evaluate the null hypothesis stating that it is equally likely that a randomly selected value from one sample will be less than or greater than a randomly selected value from a second sample. The results are intended as statistically significant at α=0.05. We also estimate the magnitude of the measured differences by using the Cliff's Delta (or d), a non-parametric

effect size measure for ordinal data [305]. We follow well-established guidelines to interpret the effect size values: negligible for $|d|$<0.10, small for 0.10≤ $|d|$ < 0.33, medium for 0.33 ≤ $|d|$ < 0.474, and large for $|d|$ ≥ 0.474 [305].

6.4 Results

In this section, we present and analyze the results of our study by research question.

6.4.1 RQ_1 - A Catalog of Reviewers' Information Needs

We report the results of our first research questions, which aimed at cataloging reviewers' information needs in code review and assessing their diffusion. For the sake of comprehensibility, we answer each sub-research question independently.

$RQ_{1.1}$: *What are the kinds of information code reviewers require?*

Following the methodology previously described (Section 6.3.3), we obtained 22 groups of reviewers' information needs. They were then clustered according to their intention into *seven* high-level categories that represent the classes of information needs associated with the discussion threads considered in our study. We describe each high-level category also including representative examples.

N1. Suitability of An Alternative Solution
This category emerged by grouping threads in which the reviewer poses a question to discuss options and alternative solutions to the implementation proposed by the author in the first place. The purpose is not only to evaluate alternatives but also to trigger a discussion on potential improvements. The example reported in the following reports a case where the reviewer starts reasoning on how much an alternative solution is suitable for the proposed code change.

> R: "… Since the change owner is always admin, this code might be able to move out of the loop? The following should be enough for this? [lines of code]"

N2. Correct understanding
In this category, we group questions in which the reviewers try to ensure to have captured the real meaning of the changes under review; in other words, this category refers to questions asked to get a consensus of reviewers' interpretation and to clarify doubts. This is more frequent when code comments or related documentation is missing. as reported in the example shown in the following.

> R: "This is now an empty heading …Or do you feel it is important to point out that these are C++ classes?"
> A: "The entire page is split up into [more artifacts]. The following sections only refer to [one artifact]. I added a sentence introducing the section."

N3. Rationale
This category refers to questions asked to get missing information that may be relevant to justify why the project needs the submitted change set or why a specific change part was implemented/designed in a certain way. For example, a reviewer may request more details about the issue that the patch is trying to address. These details help the reviewer in better understanding whether the change fits with the project scope and style. For instance, in the example reported below the reviewer (R) asks why the author replaced a piece of code.

> R: "Can you explain why you replaced [that] with [this] and where exactly was failing?"

N4. Code Context
In this category, we grouped questions asked to retrieve information aimed at clarifying the context of a given implementation. During a code review, a reviewer has access to the entire codebase and, in this way, may reconstruct the invocation path of a given function to understand the impact of the proposed change. However, we observed that the reviewer needs contextual information to clarify a particular choice made by authors. These questions range from very specific (*i.e.*, aimed at understanding the code behavior) to more generic (*i.e.*, aimed at clarifying the context in which such code is executed). The author replies to such questions by providing additional explanations on the code change or contextual project details. For instance, let consider the thread reported below, where the Author (A) replies to the Reviewer (R) by pointing R to the file (and the line) containing the asked clarification.

> R: "In what situations would [this condition] be false, but not undefined?"
> A: "See [file], exactly in line [number], in this case the evaluation of the expression returns false."
> R: "It may be helpful to add a comment documenting these situations to avoid future regressions."

N5. Necessity
In this category, the reviewer needs to know whether a (part of) the change is really necessary or can be simplified/removed. For example, a reviewer may spot something that seems like a duplicated code, yet is unsure if whether the existing version is a viable solution or it should be implemented as proposed by the author. In the example below, the reviewer asks whether a certain piece of code could be removed.

> R: "Is this needed?"
> A: "I believe its only required if you have methods after the last enum value, but I generally add it regardless. We have a pretty arbitrary mix."

N6. Specialized Expertise
Threads belonging to this category regard situations in which a reviewer finds or feels there is a code issue, however, the reviewer's knowledge is not appropriate to propose a solution. In these cases, typically a reviewer asks other reviewers to step in and contribute with their specialized expertise. Sometimes, reviewers may ask the author to propose informal alternatives that may better address the found issue. The examples reported below show two cases where the reviewer encourages other developers to reason on how to fix an issue.

> R: "… Lars, Simon, any ideas? We really need to fix this for [the next release] and the time draws nigh"

> R: "I need a better way to handle this …not a good idea to hard code digits in there. example also needs to be removed, its there just to make the tests pass."

N7. Splittable
For several reasons (including reducing the cognitive load of reviewers [306]), authors want to propose changes that are *atomic* and *self-contained* (*e.g.*, address a single issue or add a single feature). However, sometimes, what authors propose may be perceived by reviewers as something that can be addressed by different code changes, thus reviewed separately. For this reason, a reviewer needs to understand whether the split she has in mind can be done; based on this the reviewer asks questions aimed at finding practical evidence behind this idea. In other words, this category gathers questions proposed by reviewers who need to understand whether the proposed changes can be split into multiple, separated patches. For example, the thread below reports a question where the reviewer (R) asks the author about the possibility of splitting unrelated changes, but the feasibility of this split is not confirmed.

> R: "This looks like an unrelated change. Should it be in a separate commit?"
> A: "Actually its related. The input object is needed to log the delete options."
> R: "OK, I wasn't sure because in the previous version we don't pass 'force' into the method, but now we do pass it in via the 'input'."

In addition to the aforementioned categories, we found several cases in which the presence of a question or question mark did not correspond to a real information need, similarly to the aforementioned categorized information needs, we provide an example in the following.

> R: "I hate `name` as a name. What kind of name is this?"

> R: 'If you thought it was necessary to check `exe()` for errors, then why'd you leave out [another part] here? :)"

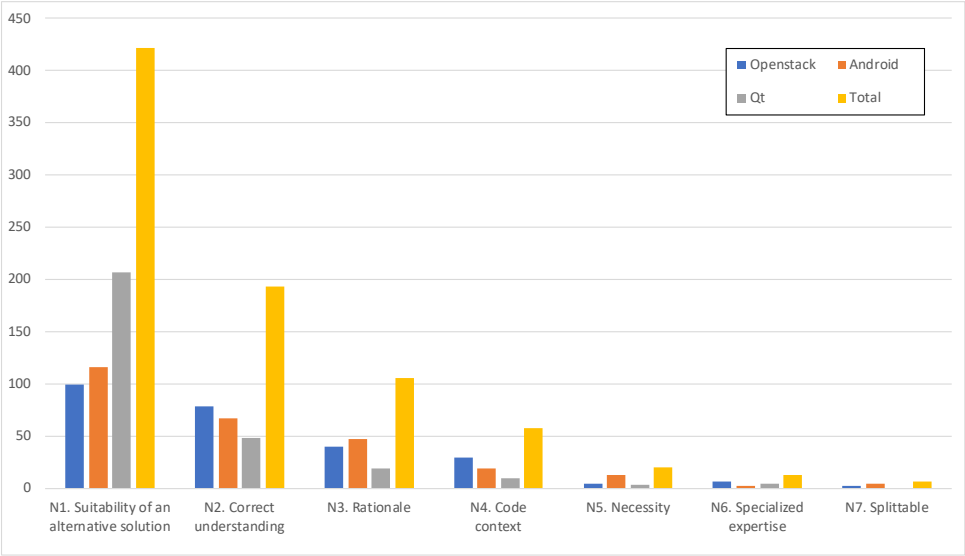

Figure 6.4: Distribution of reviewers' information needs across the considered systems.

RQ$_{1.2}$: *How often does each category of reviewers' needs occur?*

Figure 6.4 depicts bar plots that show the distribution of each reviewers' need over the considered set of code review threads. The results clearly reveal that not all the information needs are equally distributed and highlight the presence of a particular type (*i.e.*, N1. 'Suitability of an alternative solution'), which has a way larger number of occurrences with respect to all the others (the result is consistent over the three considered systems). Thus, we can argue that one of the most useful tools for reviewers would definitively be one that allows them to have just-in-time feedback on the actual practicability of possible alternative solutions with respect to the one originally implemented by the authors of the committed code change.

The second most popular category is represented by 'Correct understanding' (N2), *i.e.*, questions aimed at assessing the reviewers' interpretation of the code change and to clarify doubts. This finding basically confirms one the main outputs of the work by Bacchelli and Bird [7], who found that *code review is understandability*. The popularity of this category is similar in all the considered projects, confirming that this need is independent from the type of system or the developers working on it.

Still, a pretty popular need is 'Rationale' (N3). This has also to do with the understandability of the code change, however in this case it seems that a common reviewers' need is having detailed information on the motivations leading the author to perform certain implementation choices.

Other categories are less diffused, possibly indicating that reviewers do not always need such types of information. For instance, 'Splittable' (N7) is the category having the lowest number of occurrences. This might be either because of the preventive operations that the

development community adopts to limit the number of *tangled changes* [307] or because of the attention that developers put when performing code changes. In any case, this category seems to be less diffused and, as a consequence, one can claim that future research should spend more effort on different (most popular) reviewers' information needs.

RQ$_{1.3}$: *How do developers' perceive the identified needs?*

In this section, we present the results of our interviews and focus group with developers. First, we report on the participants' opinions on the taxonomy derived from the previous two sub-research questions, then we describe the most relevant themes that emerged from the analysis of the transcripts. We refer to individual interviewees using their identifiers ($P_{\#}$ and $FG_{\#}$).

Participants' opinion on the taxonomy. In general, all interviewees agreed on the information needs emerged from the code review threads: For all the categories, the developers agreed that they were asking those types of question themselves, several times and repeatedly. Furthermore, the order of importance of the categories was also generally agreed upon: According to the interviewees, the most important and discussed topic is 'suitability of an alternative solution' (N1), followed by 'understanding' (N2), 'rationale' (N3), and 'code context' (N4). Interestingly, the 'splittable' (N7) category is perceived as very important for the interviewed developers, but they confirmed that it happens rarely to receive big and long patches to review.

Although also the participants in the focus group agreed with the taxonomy of needs and their ranking, they stated that questions regarding 'correct understanding' (N2) are not common (in our taxonomy is ranked second). When discussing this difference with the focus group participants, they argued that this discrepancy was probably due to the type of projects we analyzed: Indeed, we analyzed open-source systems, while the focus group was conducted with participants working in an industrial, closed-source setting. One developer said: *"if I don't understand something of the change, I just go to my colleague that created it and ask to him. This is possible because we are all in the same office in the same working hours, while this is not the case in the projects you analyzed."*

Understanding a code change to review. An important step for all the interviewees when it comes to reviewing a patch is *to understand the rationale behind the changes* (N3). P_1 explained that to understand why the author wrote the patch, he first reads the commit message, since *"[it] should be enough to understand what's going on."* Interviewees said that it is very useful to have attached a ticket to the commit message, for example a JIRA issue, to really understand why it was necessary submitting the patch [P_{1-4}, FG_{1-3}]. However, sometime the patch is difficult to understand, and this leads to reviewers asking for more context or rationale of the change, as P_1 put it: *"Sometime the commit message just says "Yes, fix these things." And you say "Why? Was it broken? Is there a bug report information?" So in this case there is not enough description, and I would have to ask for it"*. Interestingly, P_1 reported that this issues generally happens with new contributors or with novice developers. During the focus group, FG_3 said he also uses tests to obtain more context about the change: *"In general, to get more context I read the Java docs or the tests."* Finally, all the interviewees explained that to obtain more context, or the rationale behind the change, they use external IRC channels (outside Gerrit) to get in touch with reviewers/authors, *e.g.*, by emails, or Slack.

Authors' information needs. Considering the point of view of the *author* of a change, the interviewees explained that code review is sometimes used as a way to *get information* from specialized experts, thus underling the dual nature of the knowledge exchange happening in code review [15, 15]. P_2 explained that it is sometimes difficult for an author to have all the information they need to make the change, for example if the change is in a part of the system where they are not expert. In this case, P_2 explained: *"when you make a change, you usually add the experts of the system to your review, and then you ping them on IRC, asking for a review, if they have some time."* Interestingly, this point also came up during the focus group, where a developer said *"if it's a new system [...] my knowledge lacks at one front, it may be technology, it may be knowledge of the system."* In this case, the developer would ask the help of colleagues. This is also in line with another need we discovered in the previous research question, that is the 'Specialized Expertise' (N6). Indeed, interviewees said that when they are not familiar with the change, or they do not have the full context of the change, they ask an expert to contribute: *"[in the project where I work] we have sub-system maintainers: they are persons with knowledge in that area and have more pleasure or willingness to work on those specialized areas. If the reviewers do not reach consensus during the review, we always ask to those experts."* [P_4].

Small and concise patches. When discussing with developers the 'Splittable' (N7) need, all agreed that patches should be self-contained as much as possible [P_{1-4}, FG_{1-3}]. P_3 said: *"I always ask to split it, because in the end it will be faster to get it in [the system]."* P_4 added that it is something that they do all the time, because usually people do not see this issue. P_2 said: *"It's always better to have 10 small reviews than one big review with all the changes, because no one will review your code. It's like that. So if you want to merge something big, it's always better to do it in small changes."*

Another point raised during the interviews is that large patch sets are difficult to review and require a lot of time to read [P_{1-4}, FG_{1-3}], thus this may delay the acceptance of the patches. P_4 explained: *"You can have a large patch set that is 90% okay and 10% that not okay: the 10% will generate a lot of discussion and will block the merge of 90% of the code. So yes, it's something that I do all the time. I ask people, you need to organize better the patch."*

When talking about the issue, P_1 also added that having small patches is very important for making it easier to revert them: *"yes this is something I find it to be really, really important. Bugs are everywhere – there is always another bug to fix. So the patch should be small enough that [in case of bugs] you can revert it without breaking any particular code."*

Interestingly, all the interviewees agreed that tools could help reviewers and authors in solving this need: for example, when submitting a large patch, the tool could suggest the author to split it into more parts to ease the reviewing process.

Offering a solution. All interviewees agreed that to do a proper code review, reviewers should always pinpoint the weak parts in the code and offer a solution [P_{1-4}, FG_{1-3}]. P_3 said: *"When I request the change, I usually put a link or example because I know that maybe the other guy doesn't know about the other approach. This is usually the main reason why somebody didn't do something: because he didn't know it was possible."* P_1 added: *"[...] whenever you propose a change, you should always explain why you need to change it and what. Just putting [the score to] minus two, or even minus one without explanation, is bad because then people don't know what to do. We have to try being more friendly as a community."* This constructive behavior was also agreed upon during the focus group: one developer said that the worst

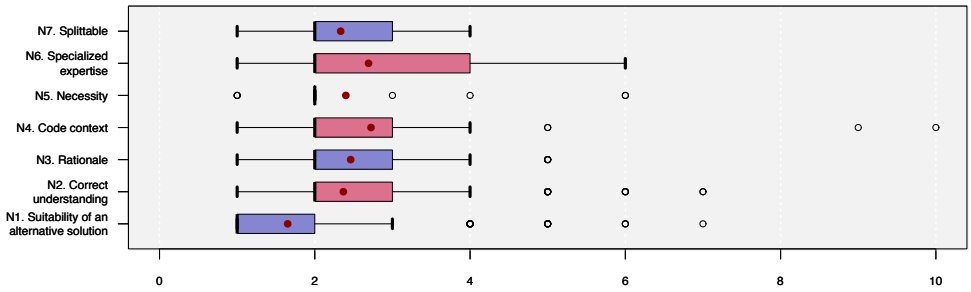

Figure 6.5: Distribution of the number of replies for each reviewers' need.

thing that can happen in a code review is a non-constructive comment. Interestingly, this reported behavior confirms what we discovered in our previous research question: indeed, 'Suitability of an alternative solution' (N1) is the most frequent type of question when doing code review.

In addition, concerning constructive feedback, interviewees said that when they do not fully understand a change, they first ask the author explanations: *"For example, if you don't understand correctly the change this person is trying to add, you just ask him, and they are forced to answer you. And if you don't have the context information, they should be able to provide it to you."*[P_2] Interviewees said that it is better to ask explanation to the author first, and only after decide to/not merge the patch. P_4 also explained that sometime it is better to accept a patch than start a big discussion on small detail: *"Even though I understand that a better solution will be doable, I'll probably won't propose it because a lot of times people won't have time to actually rework on a new proposal, and you need to balance how you want the project to move forward: Sometimes it's better to have a code that is not the best solution, but at least does not regress and it fixes a bug."*

6.4.2 RQ_2 - The Role of Reviewers' Information Needs In A Code Review's Lifecycle

We present the results achieved when answering our second research question, which was focused on the understanding of the role of reviewers' information needs in the lifecycle of the code review process. We report the results by considering each sub-research question independently.

$RQ_{2.1}$: *What are the reviewers' information needs that attract more discussion?*

To answer **$RQ_{2.1}$** and understand to what extent the reviewers' needs attract developers' discussions, we compute, for each discussion thread that we manually categorized, the number of iterations that involve the developers of a certain code review.

Figure 6.5 depicts box plots reporting the distribution of the number of answers for each reviewers' need previously identified (red dots indicate the mean). Approximately 18% of code review threads (considering both merged and abandoned patches of every projects) do not have an answer. The first observation regards the median value of each distribution: as shown, all of them are within one and three, meaning that most of the threads are concluded with a small amount of discussion. From a practical perspective,

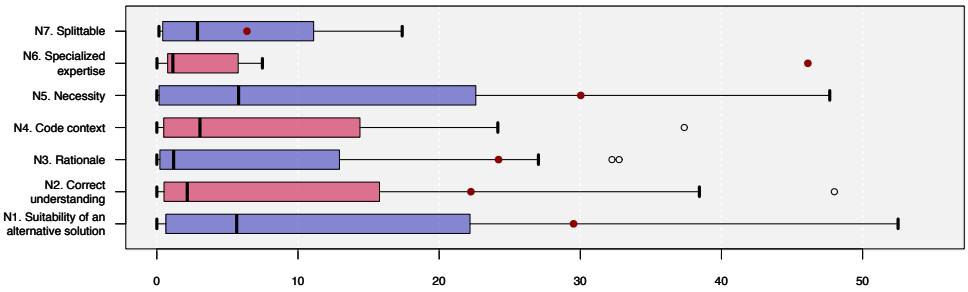

Figure 6.6: Distribution of the number of hours needed to answer each reviewers' need category.

this result highlights that authors can address almost immediately the need pointed out by a reviewer; at the same time, it might highlight that tools able to address the reviewers' needs identified can be particularly useful to even avoid the discussion and lead to an important gain in terms of time spent to review source code. Among the reviewers' needs, the 'Specialized expertise' (N6) is the one with the largest scattering of discussion rate. This result seems to indicate that the more collaboration is required due the largest number of replies a discussion receives, which possibly preclude the integration in the codebase of important changes that require the expertise of several people.

The statistical tests confirmed that there are no statistically significant differences among the investigated distributions, with the only exception of 'Suitability of an alternative solution' (N1), for which the ρ-value is lower than 0.01 and the Cliff's d is 'medium'. This category is the one having the lowest mean (1.7) and we observed that often authors of the code change tend to directly implement the alternative solution proposed by the reviewer without even answering to the original comment. This tendency possibly explain the motivation behind this statistical difference.

Overall, according to our results, most of the reviewers' information needs are satisfied with few replies–most discussions are closed shortly. The only category having more scattered results is the one where reviewers ask for the involvement of more people in the code review process.

RQ$_{2.2}$: *How long does it take to get a response to each reviewers' information need?*

Figure 6.6 reports the distribution of the number of hours needed to get an answer for each group of reviewers' information need. In this analysis we could only consider the questions having at least one answer; similarly, if a reviewer's comment got more than one reply, we considered only the first one to compute the number of hours needed to answer the comment.

Looking at the results, we can observe that the median is under 7 hours for almost all the categories. A possible reason for that consists of the nature of the development communities behind the subject systems. Indeed, all the projects have development teams that span across different countries and timezones: thus, we might consider as expected the fact to not have an immediate reaction to most of the comments made by reviewers. Some differences can be observed in the distributions of two reviewers' information needs such as 'Necessity' (N5) and 'Suitability of an alternative solution' (N1). In this case, the median number of hours is higher with respect to the other categories (7 vs 5), while the

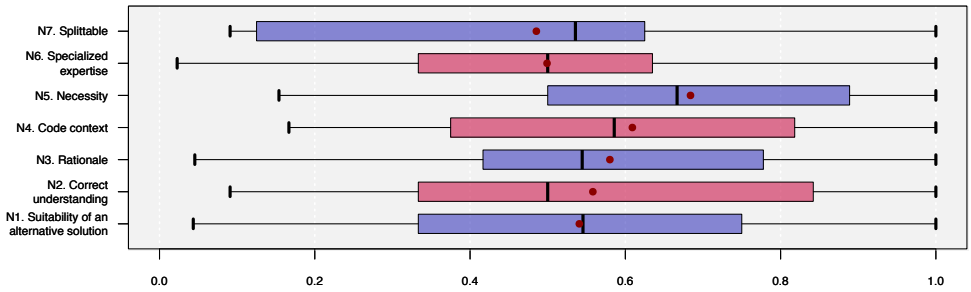

Figure 6.7: Distribution of reviewers' needs over different iterations of the code review process.

3rd quartiles are around one day (meaning that 25% of the questions in this category took more than one day to have a response). Conversely, the discussion of other categories generally took less time to start. For instance, the 'Specialized expertise' (N6) need has a median of one hour and a 3rd quantile equal to four. Such differences, however, are not statistically significant.

To conclude the analysis of our findings for this research question, we can argue that developers generally tend to respond slower to questions regarding the proposal of alternatives and the evaluation of the actual necessity of a certain code change; on the other hand, questions where more reviewers are called to discuss seem to get a faster response time.

RQ$_{2.3}$: *How do the reviewers' information needs change over the code review process?*

The last research question targets the understanding of whether reviewers' needs vary over the different iterations of the code review process. Figure 6.7 presents the result of our analysis, with box plots depicting the distribution of each reviewers' information need in the various iterations: for the sake of better comprehensibility of the results, we considered the normalized number of iterations available in each of the 900 code review threads analyzed.[7]

Almost all the categories have their median around 0.5, meaning that the majority of reviewers' information needs are raised in the first half of the review process. Moreover, we are not able to map reviewers' information needs with any specific iteration. This result might indicate that there is not a time-sensitive relationship between those needs and that they arise independently from how much discussion has already been going on in the review.

Besides this general conclusion, we also notice some differences between the category 'Necessity' (N5) and the others. In the case of the 'Necessity' (N5) category the median and mean reach both 0.67, thus indicating that most of such questions come later in the process. It is interesting to note that modifications aimed at performing perfective changes that improve the overall design/style of the source code rather than solving issues are mainly requested by reviewers in a later stage of the code review process, *i.e.*, likely after that most important fixes solving problems impacting the functioning of the system are already submitted by the author and answers about understanding the context of the change are

[7]We also conducted an analysis using the absolute number of iterations, yet results were equivalent.

given. Such an observation may need further investigations and validation, however it may possibly reveal the possibility to devise strategies to guide the next generation of code review tools toward a *selection* of the information that a reviewer might need in a earlier/later stage of the code review process.

6.5 Threats to Validity

Our study might have been subject to a number of threats to validity that may affect our results. This section summarizes the limitations of our study and how we tried to mitigate them.

Validity of the defined reviewers' needs. Since the meaning of a question may be dependent by the context, we may lack of a full understanding of its nature and background. This type of threat may first apply to our study when we identify code review threads composed of both questions and answers: to this aim, we automatically mined the Gerrit repository that is a reliable source for the extraction of code review data [270, 308]. To extract code review threads we employed the publicly available APIs of such repository: For this reason, we are confident on the completeness of the extracted data.

The adopted open card sorting process is also inherently subjective because different themes are likely to emerge from independent card sorts conducted by the same or different people. To ensure the correctness and completeness of the categories associated to the reviewers' needs identified with the card sorting, we iteratively conducted the process by merging and splitting reviewers' need categories if needed. As an additional step, we also took into account authors responses and discussion threads when classifying questions made by reviewers, with the aim of properly understand the context in which a certain question has been made. Moreover, all the authors of this paper, who have more than seven years of experience in software development, assessed the validity of the emerged categories, thus increasing its overall completeness. Of course, we cannot exclude the missing analysis of specific code review threads that point to categories that were not identified in our study.

We consider questions asked by reviewers through the Gerrit platform as indicators of the actual reviewers' needs. This assumption may not hold for all projects, as many active projects do not use the Gerrit platform. For example, Tsay *et al.* [309] highlighted how several developers contribute to the software development by using different platforms (*e.g.*, GitHub). However, we partly mitigated this threat to validity by carefully selecting software systems broadly studied in code review research [16, 168, 292] and having a large number of code review data (which indicates they actively use Gerrit). The study of different platforms such as GitHub, GitLab, or Collaborator is left for future work.

External validity. As for the generalizability of the results, we conducted this study on a statistically significant sample of 900 code reviews that include more than 1,800 messages belonging to three well-known projects that use the Gerrit platform since 2011. A threat to validity in this category may arise when we consider closed-source projects. In that case, the experience of closed-source reviewers may affect the need to asks clarification questions, therefore, the findings that we found in open-source project may be not generalizable to a closed-source context. As part of our future research agenda, we plan to extend this study by including closed-source projects.

6.6 Discussion and Implications

Our quantitative and qualitative results showed that reviewers have a diversity of information needs at different conceptual levels and pertaining to different aspects of the code under review. In this section we discuss how our results lead to recommendations for practitioners and designers, as well as implications for future research.

1. **Selection of assistant experts.** The results achieved by mining code review repositories and interviewing practitioners indicate that 'Suitability of An Alternative Solution' (N1) and 'Correct Understanding' (N2) are not only the most recurring needs, but also those perceived as the most important. When discussing these topics with developers from both open-source and industrial systems, we uncovered possible areas where current code review tools can offer better features. For example, a key need for the reviewers is being able to communicate with the experts of the sub-system under review; this underlines the importance of tools able to recognize developers' expertise and create recommendations.
   Researchers have conducted the first steps into this direction. For instance, among others, Patanamon *et al.* [22] proposed RevFinder, an approach to search and recommend reviewers based on similarity of previous reviewed files, while Thongtanunam *et al.* [310] validated the performance of a reviewer recommendation model based on file paths similarity. An interesting novelty that emerged from our analysis, with respect to existing previous work on reviewers recommendation, is the *target* of the recommendation. In fact, existing reviewer recommendation mechanisms target the author of the change who has to select the reviewer and propose reviewers for full changes or files. Instead, we found that also the reviewers have the need to consult an external expert, maybe for a more specific part of the entire change under review. For instance P_3 explained that *"reviewers sometimes ask for other reviewers that may be more expert"*, thus having an assistant that can help the selection of an expert reviewer may increase her productivity. Targeting reviewers instead of change authors and having a finer grained focus for the recommendation mechanism can lead to interesting changes in both the model (which may use different features to compute expertise and the *difference* of expertise among reviewers) and the evaluation approach (which may no longer be based on just matching actually selected reviewers). Further studies can be designed and conducted to better understand this novel angle.

2. **Early detection of splittable changes.** Even if the 'Splittable' (N7) category is the less frequently occurring, interviewees argued that it is really useful to automatically detect splittable code changes before a submission. For example, in the focus group all the participants (FG_{1-3}) suggested: *"if it's an unrelated change [...], pull it out of this ticket and put it on another issue."* In fact, this would (1) decrease the time spent in detecting this issue and asking the author to re-work the change, as well as (2) reduce the risks of introducing defects in the source code [307].
   Researchers have already underlined the risks of tangled code changes (*i.e.*, non-cohesive code changes that may be divided in atomic commits) for mining software repositories approaches [307] and have proposed mechanisms for automatically splitting them. For instance, Herzig and Zeller [307] proposed an automated approach

relying on static and dynamic analysis to identify which code changes should be separated; Yamauchi suggested a clustering algorithm tuned to identify unrelated changes in a commit message [311]; and Dias *et al.* [312] proposed a methodology to untangle code changes at a finer-granularity, *i.e.*, by selecting the single statement of a code review that should be placed in other commits. More recently researchers also proposed untangling techniques tailored explicitly to code review [306, 313] and conducted the first experiments to measure the effects of tangled code changes on code review [313, 314] substantiating the value of separating unrelated changes. Despite these advances in splitting algorithms and their immediate practical value, no commercial code review tool offers this feature. Our analysis underlines even more the relevance of having such a feature integrated as early as possible in the development process, possibly in the development environment, so that authors send already self-contained patches for review. Moreover, despite the notable research advances in the field, we believe that there is still room for improvement, *e.g.*, by complementing state-of-the-art methods with conceptual-related information aimed at capturing the semantic relationships between different code changes.
Also, early improvements of code changes before review are in line with the work by Balachandran [315]. He reported that the time to market can be reduced also creating automatic bots able to conduct preliminary reviews [315]. In this regard, there are still plenty of opportunities and challenges on how and when bots can automatically help reviewers during their activities and whether they may be employed to assist some of the developers' needs in code review.

3. **Automatically detecting alternative solutions.** In connection with the most frequent need (*i.e.*, 'Suitability of an alternative solution' (N1)), an interviewee from one of the open-source projects explained to prefer to propose an alternative solution before rejecting a patch: *"[...] usually I put a link or an example."* In this light, a promising avenue for an impactful improvement in code review is to integrate a toll that automatically mines alternative solutions. Accordingly, a first interesting step would be to investigate how to integrate at code review time an approach such as the one proposed by Ponzanelli *et al.*, which systematically mines community based resources such as forums or Stack Overflow to propose related changes [316]. Another promising starting point in this direction is the concept of programming with "Big Code," as proposed by Vechev and Yahav [317], to automatically learn alternative solutions from the large amount of code available in public code repositories such as GitHub.

4. **Synchronous communication support.** The absence of a proper real-time communication channel within code review tools was a common issue that emerged from both the interviews with the open-source developers and the focus group. In fact, two interviewees ([FG_1] and [FG_3]) explained: *"you can just go to the author and ask to him in person, and maybe it would be a long discussion [...]"*. This is in line with the experience reported by developers at Microsoft in a previous study by Bacchelli and Bird [7]. Nevertheless, in-person discussions can happen only if both author and reviewer are co-located, otherwise logistic barriers could impose serious constraints [7]. Yet, open-source developers are able to fulfill this real-time commu-

nication need using alternative channels; P_2 stated: *"we usually have an IRC channel [...]"*. The two observations suggest that, when it is possible, developers prefer to rely on direct communication to discuss feedback; this may be to avoid discussing difficult criticism online in a public forum and to have a higher communication bandwidth than small online thread comments. In both scenarios, our results show that current code review tools are clearly not able to fully satisfy the communication need of the involved people. Future work should be conducted to understand how communication can be facilitated within the code review tool itself (thus improving traceability of discussions, which is relevant for future developers' information needs [272]); in principle, this future analysis should take into account not only technical aspects to increase the communication bandwidth, but also the social aspects that could currently hinder developers from discussing certain arguments with the current tools.

5. **Automatic change summarization.** 'Correct understanding' (N2) and 'Rationale' (N3) are also key information needs for reviewers. Normally this is achieved by perusing the code change description or additional comments. Nevertheless, our interviewees reported cases in which these sources of information were insufficient to fulfill this need; on this P_3 reported: *"I even had cases where the description didn't have anything in common with the code"*. Indeed this shows that another significant source of delay in a code review process is when patches contain unaligned or missing information (*i.e.*, the commit message is not clear enough or it does not match with the actual patch). Code summarization techniques appear to be a good fit for this task: Indeed, past literature presented different summarization techniques that can be used to both produce or check the current documentation. For example, Buse and Weimer proposed a technique to synthesize human-readable documentation starting from code changes [318], but also several other researchers have been contributing with more approachess: Canfora*et al.* experimented Ldiff [319], Parnin and Görg developed CILDiff [320], and Cortés-Coy *et al.* designed ChangeScribe [321]. Our analysis suggests that supporting code review is a ripe opportunity for research on code summarization techniques to have another angle of impact on a real-world application.

6.7 Conclusions

Modern code review is an important technique used to improve software quality and promote collaboration and knowledge sharing within a development community. In a typical code review process, authors and reviewers interact with each other to exchange ideas, find bugs, and discuss alternative solutions to better design the structure of a submitted code change. Often reviewers are required to inspect author patches without knowing the rationale or without being aware of the context in which a code change is supposed to be plugged-in. Therefore, they must ask questions aimed at addressing their doubts, possibly waiting for a long time before getting the expected clarifications. This might potentially result in causing delays in the integration of important changes into production.

In this work we investigate the reviewers' information needs by analyzing 900 code review threads of three popular open-source software systems (OpenStack, Android,

and QT). Moreover, we conduct four semi-structured interviews with developers from the considered projects and one focus group with developers from a software quality consultancy company, with the aim of challenging and discussing our outcome.

We discovered the existence of seven high-level reviewers' information needs, which are differently distributed and have, therefore, different relevance for reviewers. Furthermore, we analyzed the role played by each category of reviewers' information needs across the lifecycle of a code review, and in particular what are the reviewers' information needs that attract more discussion, for how long a reviewer should wait to get a response, and how the information needs change over the code review lifecycle.

Based on our findings, we provide recommendations for practitioners and researchers, as well as viable directions for impactful tools and future research. We hope that the insights we have discovered will lead to improved tools and validated practices which in turn may lead to higher code quality overall.

7

Conclusion

This closing chapter revisits our original research questions, presents study implications, and contains concluding remarks on the effectiveness of using fine-grained defect prediction in code review with an outlook into future work on improving code review further.

7.1 Research Questions Revisited

This section revisits the research questions defined in Chapter 1.

> **RQ$_1$.** *Are current defect prediction algorithms a feasible solution for supporting code review?*

Despite the initial expectations, we found that current prediction algorithms are loosely suitable for actual usage. We obtained this result by conducting a study where we replicated previous research on method-level defect prediction. We evaluated the performance of the state of the art of defect prediction models considering different systems and timespans—we focused on 13 Java open-source software systems whose repositories are publicly available on GitHub. Successively, we revisited the evaluation strategy by proposing a more realistic scenario based on a release-by-release approach.

In our initial replication, we observed that although our results are ten percentage points lower than those obtained in the literature, they are far from being random. In particular, method-level defect prediction models trained with process metrics perform better than those based on product metrics. This result confirms the replicability of the state of the art on defect prediction and provides evidence that the combination of predictors of different nature does not dramatically improve the prediction capabilities. However, when those results are re-evaluated with a release-by-release evaluation strategy, all the experimented method-level defect prediction models show a significant drop in performance (up to 20 points percentage less in terms of AUC-ROC) that are close to the results of a random classifier.

We learn from this research question that existing defect prediction algorithms cannot support code review because the performance achieved in a realistic scenario is similar to a random classifier and would not reduce reviewer effort.

RQ$_2$. *To what extent do fine-grain defect prediction models improve prediction performance?*

With the second research question, we seek to understand how much effort a fine-grained just-in-time defect prediction algorithm can save if hypothetically employed in a context of code review. We achieved this by examining the performance of advanced fine-grained just-in-time defect prediction models. In particular, we first investigated the prominence of partially defective commits (*e.g.*, commits that contain both defective and non-defective files) because not all files in a defective commit are defective. Then, we proposed and evaluated a novel fine-grained just-in-time defect prediction model to predict files that are likely to be defective. Finally, we estimated how much effort, in terms of lines of code to inspect, a fine-grained prediction model can save during a code review. We considered ten different open-source repositories to evaluate the performance of our model.

We observed that almost half of the defective commits contain both defect-inducing and defect-free changes (*i.e.*, composed of both files that are changed without introducing defects and files that are changed introducing defects). In those partially defective commits, more than half of the files are defect-free. Consequently, these defect-free files unnecessarily attract reviewers' effort while inspecting sources. Then, by building a fine-grained prediction model that achieves an overall stable performance across the considered projects, we confirmed how developer-related factors are those that generally provide the highest contribution to the prediction of defective files within commits.

By answering this research question, we understand how a fine-grained prediction model could save reviewers' effort by localizing half of the defects in only a quarter of the lines of code. We conjecture that such a model would save effort during a code review.

RQ$_3$. *How can alternative features improve defect prediction performance?*

With the third research question, we aim at understanding whether the performance of a method-level defect prediction model—evaluated with a more realistic release-by-release strategy—can be improved further by using alternative features based on textual aspects, code smells, and developer-related factors. We answered this question by investigating how developers use code comments in open and closed software, and how alternative metrics derived by the above factors impact the prediction performance.

Specifically, Chapter 4 focuses on how diverse software projects use code comments to understand their role in source code. In this case, we merged the results of three studies that analyze six traditional open-source software systems, five mobile open-source applications, and eight industrial projects. Finally, the Chapter 5 extends the work presented in Chapter 2 and evaluates to what extent the performance of a release-by-release defect prediction model can be improved by using the aforementioned additional features.

With the Chapter 4, our manual inspection of 40,000 lines of code belonging to different project domains has led to the creation of a taxonomy for code comments composed of two layers with six top coarse-grained categories and 16 fine-grained inner categories. Then, by comparing the distribution of comments in open and closed source software, we found that

on average, the former class of projects uses four times the number of comments compared to the second with a difference also in the distribution of comments for each category.

Finally, although within validation achieves 95% of true positive, the cross-project or cross-license validation is typically 15% lower. However, by manually adding a small sample of 80 lines of code comments of an unseen project, the performance of the proposed machine learning can achieve a 37% boost in precision. Consequently, defect prediction aimed at supporting code review can benefit from features derived by those metrics.

Regarding the Chapter 5, we observed that the performance of release-by-release defect prediction algorithms does not improve in a realistic way when trained with alternative features based on textual aspects, code smells, and developer-related factors. We registered a gain of only two percentage points when using all proposed metrics as machine learning features.

By answering this research question, we assessed that method-level release-by-release defect prediction models gain only a marginal advantage when trained with alternative features.

RQ$_4$. *What are reviewers' information needs and how does defect prediction fulfill them?*

This research question investigates the role of the information that reviewers need while conducting code review and how-to guide future research effort to design tools that better support code review to make reviews more effective and efficient. We answered this research question by conducting a quantitative and a qualitative study on the role of information needs in code review. We analyzed the threads of discussions that are recorded by code review tools. Then, we built a double-layer taxonomy that summarizes the purpose of needs argued in the threads' discussions. Finally, we validated our findings with expert developers from both open-source and industrial domains. We started with a manual analysis of 900 code review comments from three large open-source projects. We later continued with four semi-structured interviews with at least a senior reviewer from each open-source project and a focus group with developers of a software quality consultancy company.

In this study, we observed the presence of seven high-level information needs such as insights on the questions and answers discussed by reviewers for knowing the uses of methods and variables declared/modified in the code under review which are differently distributed and have, therefore, different relevance for reviewers. Furthermore, we clarified the role played by each category of reviewers' information needs across the lifecycle of a code review, and in particular the reviewers' information needs that attract more discussion, for how long a reviewer should wait to get a response, and how the information needs change over the code review lifecycle.

By conducting this study, we explore alternative ways to reduce further the mental load of reviewers during code review without compromise their effectiveness—other than supporting code review with defect prediction. We provided recommendations for practitioners and researchers, as well as viable directions for impactful tools and future research. In addition, we assess that code review needs more support when it comes to dealing with large systems to reduce the review effort in catching defects and improving the overall quality.

7.2 Implications

We discuss the implications that we derive from this thesis.

7.2.1 Combining the performance of defect prediction models

During software maintenance and evolution, developers modify the source code to add new features or fix defects encountered by users. Such continuous changes naturally lead to the risk of introducing new defects: for this reason, developers must carefully verify that the performed modifications do not introduce new defects in the code. This task is usually executed directly during the development (e.g., by running test cases) or when the changes are reviewed (e.g., through code review). Defect prediction represents a potential way to allocate inspection and testing resources to the portion of source code more likely to be defective.

In our research, we first investigated the state of the art of defect prediction models in a realistic scenario, confirming a dramatic drop in performance with results close to that of a random model. Then, to decrease the effort required by developers during code inspection, we proposed a file-level just-in-time defect prediction model that anticipates feedback at commit time. Later, to allow real applicability of defect prediction models, we propose to tackle the problem from a different perspective by exploring what sets of alternative metrics can capture non-functional properties (*e.g.*, by retrieving non-functional properties through code comments taxonomy in Chapter 4) with the goal of improving the performance of defect prediction further.

However, to help developers, research should focus on boosting prediction performance further by combining fine-grained (*e.g.*, method-level) with just-in-time defect prediction models. Those models would generate localized recommendations to guide developers during a code review with the aim of delivering better software while saving developers' effort.

7.2.2 Better support for code review tools

With our research, we aim at establishing to what extent code review can benefit from defect prediction models. However, when we investigate the information that reviewers need to conduct a proper code review, we discovered how several areas of code review need further support or new development. To this aim, we suggest the following research directions for supporting code review further in different review phases:

- **Selection of experts**. A key need for the reviewers is being able to communicate with the experts of the sub-system under review; this underlines the importance of tools able to recognize developers' expertise and create recommendations. Researchers have conducted the first steps into this direction. For instance, among others, Patanamon *et al.* [22] proposed RevFinder, an approach to search and recommend reviewers based on similarity of previous reviewed files, while Thongtanunam *et al.* [310] validated the performance of a reviewer recommendation model based on file path similarity.

  An interesting novelty that emerged from our analysis, with respect to previous work on reviewer recommendation, is the target of the recommendation. In fact,

existing reviewer recommendation mechanisms target the author of the change who has to select the reviewer and propose reviewers for full changes or files. Instead, we found that also the reviewers have the need to consult an external expert, maybe for a more specific part of the entire change under review. Targeting reviewers instead of change authors and having a finer grained focus for the recommendation mechanism can lead to interesting changes in both the model (which may use different features to compute expertise and the difference of expertise among reviewers) and the evaluation approach (which may no longer be based on just matching actually selected reviewers).

- **Detection of splittable changes.** Automatically splitting changes into small parts would (1) decrease the time spent in detecting this issue and asking the author to re-work the change, as well as (2) reduce the risks of introducing defects in the source code [307]. Researchers have already underlined the risks of tangled code changes (*i.e.*, non-cohesive code changes that may be divided in atomic commits) for mining software repositories approaches [307] and have proposed mechanisms for automatically splitting them. For instance, Herzig and Zeller [307] proposed an automated approach relying on static and dynamic analysis to identify which code changes should be separated; Yamauchi suggested a clustering algorithm tuned to identify unrelated changes in a commit message [311]; and Dias *et al.* [312] proposed a methodology to untangle code changes at a finer-granularity, *i.e.*, by selecting the single statement of a code review that should be placed in other commits. More recently researchers also proposed untangling techniques tailored explicitly to code review [306, 313] and conducted the first experiments to measure the effects of tangled code changes on code review [313, 314] substantiating the value of separating unrelated changes. Despite these advances in splitting algorithms and their immediate practical value, no commercial code review tool offers this feature. Our analysis underlines even more the relevance of having such a feature integrated as early as possible in the development process, possibly in the development environment, so that authors can easily send already self-contained patches for review. Despite the notable research advances in the field, there is still room for improvement, *e.g.*, by complementing state of the art methods with conceptual-related information aimed at capturing the semantic relationships between different code changes.

- **Explore alternative solutions.** A promising avenue for improving code review is to integrate a tool that automatically mines alternative solutions. Accordingly, a first interesting step would be to investigate how to integrate at code review time an approach such as the one proposed by Ponzanelli *et al.*, which systematically mines community based resources such as forums or Stack Overflow to propose related changes [316]. Another promising starting point in this direction is the concept of programming with "Big Code," as proposed by Vechev and Yahav [317], to automatically learn alternative solutions from the large amount of code available in public code repositories such as GitHub.

- **Synchronous communication channels.** The absence of a proper real-time communication channel within code review tools is a common issue. This is in line with the experience reported by developers at Microsoft in a previous study by Bacchelli

and Bird [7]. Nevertheless, in-person discussions can happen only if both author and reviewer are co-located, otherwise logistic barriers could impose serious constraints [7]. When it is possible, developers prefer to rely on direct communication to discuss feedback; this may be to avoid discussing difficult criticism online in a public forum or to have a larger communication bandwidth than small online thread comments. In both scenarios, our results show that current code review tools are not able to fully satisfy the communication need of the involved people. Future work should be conducted to understand how communication can be facilitated within the code review tool itself (thus improving traceability of discussions, which is relevant for future developers' information needs [272]); in principle, this future analysis should take into account not only technical aspects to increase the communication bandwidth, but also the social aspects that could currently hinder developers from discussing certain arguments with the current tools.

- **Change summarization.** Developers typically write code change summarizations at commit time. Nevertheless, our research reported cases in which these sources of information were insufficient to fulfill reviewers' needs. These cases, together with patches that contain unaligned or missing data (i.e., the commit message is not clear enough or it does not match with the actual patch), are a dramatic source of delay. Code summarization techniques appear to be a good fit for this task: Indeed, past literature presented different summarization techniques that can be used to both produce or check the current documentation. For example, Buse and Weimer proposed a technique to synthesize human-readable documentation starting from code changes [318], but also several other researchers have been contributing with more approaches: Canfora*et al.* experimented with Ldiff [319], Parnin and Görg developed CILDiff [320], and Cortés-Coy *et al.* designed ChangeScribe [321]. Our findings suggest that supporting code review tools with summarization techniques is a promising research direction to have another angle of impact on a real-world application.

7.3 Concluding Remarks

In this thesis, we focus primarily on improving code review. We study this assuming that the human nature of reviewers is the point that can be more successfully improved. We seek to understand how to improve defect prediction to support code review. We analyze how the meaning of code comments combined with other alternative sets of metrics improve defect prediction. We study how anticipating feedback at commit time reduces the code review inspection effort. And finally, we consider how reviewers' needs can guide the improvement of code review further. With these studies, we show how to support code review with boosted prediction models while reporting a list of guidelines to design efficient code review tools that reduce the mental load of reviewers.

We see that code review needs to be provided with a lot more support when it comes to dealing with software quality. Aware that our study has scratched only the surface setting the ground of this topic, we hope that future research can bring further investigation in this direction.

Bibliography

URLs in this thesis have been archived on Archive.org. Their link target in digital editions refers to this timestamped version.

References

[1] Marc Andreessen. Why software is eating the world. *The Wall Street Journal*, 20(2011):C2, 2011.

[2] Zhengrui Jiang and Sumit Sarkar. Free software offer and software diffusion: The monopolist case. *ICIS 2003 proceedings*, page 81, 2003.

[3] Nancy L Martin, J Michael Pearson, and Kimberly A Furumo. Is project management: size, complexity, practices and the project management office. In *Proceedings of the 38th Annual Hawaii International Conference on System Sciences*, pages 234b–234b. IEEE, 2005.

[4] Thomas W Mastaglio and Robert Callahan. A large-scale complex virtual environment for team training. *Computer*, 28(7):49–56, 1995.

[5] David Lorge Parnas. Software aging. In *Proceedings of the 16th international conference on Software engineering*, pages 279–287. IEEE Computer Society Press, 1994.

[6] Rajiv D Banker, Gordon B Davis, and Sandra A Slaughter. Software development practices, software complexity, and software maintenance performance: A field study. *Management science*, 44(4):433–450, 1998.

[7] Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. *Proceedings - International Conference on Software Engineering*, pages 712–721, 2013.

[8] A Frank Ackerman, Priscilla J Fowler, and Robert G Ebenau. Software inspections and the industrial production of software. In *Proc. of a symposium on Software validation: inspection-testing-verification-alternatives*, pages 13–40. Elsevier North-Holland, Inc., 1984.

[9] A Frank Ackerman, Lynne S Buchwald, and Frank H Lewski. Software inspections: an effective verification process. *IEEE software*, 6(3):31, 1989.

[10] Michael E Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, 1976.

[11] Ieee guide to classification for software anomalies. *IEEE Std 1044.1-1995*, pages 1–60, 1996.

[12] Ieee standard classification for software anomalies. *IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993)*, pages 1–23, 2010.

[13] Jason Cohen. Modern code review. *Making Software: What Really Works, and Why We Believe It*, pages 329–336, 2010.

[14] Peter C Rigby and Christian Bird. Convergent contemporary software peer review practices. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering*, FSE 2013, pages 202–212. ACM, 2013.

[15] Georgios Gousios, Andy Zaidman, Margaret-Anne Storey, and Arie Van Deursen. Work practices and challenges in pull-based development: the integrator's perspective. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 358–368. IEEE Press, 2015.

[16] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E Hassan. An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering*, pages 1–44, 2015.

[17] Murtuza Mukadam, Christian Bird, and Peter C Rigby. Gerrit software code review data from Android. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 45–48. IEEE, 2013.

[18] Yuriy Tymchuk, Andrea Mocci, and Michele Lanza. Code review: Veni, vidi, vici. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 151–160. IEEE, 2015.

[19] Gary McGraw. Automated code review tools for security. *Computer*, 41(12):108–111, 2008.

[20] The most intelligent Java IDE, author=IntelliJ, IDEA, journal=JetBrains [online].[cit. 2016-02-23]. URL: https://www. jetbrains. com/idea/# chooseYourEdition, year=2011.

[21] Tobias Baum, Kurt Schneider, and Alberto Bacchelli. On the optimal order of reading source code changes for review. In *33rd International Conference on Software Maintenance and Evolution*, ICSME 2017, pages 329–340, 2017.

[22] P. Thongtanunam, C. Tantithamthavorn, R. G. Kula, N. Yoshida, H. Iida, and K. Matsumoto. Who should review my code? a file location-based code-reviewer recommendation approach for modern code review. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 141–150, March 2015.

[23] Tianyi Zhang, Myoungkyu Song, and Miryung Kim. Critics: An interactive code review tool for searching and inspecting systematic changes. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 755–758. ACM, 2014.

[24] Chris Lewis, Zhongpeng Lin, Caitlin Sadowski, Xiaoyan Zhu, Rong Ou, and E James Whitehead Jr. Does bug prediction support human developers? findings from a Google case study. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 372–381. IEEE Press, 2013.

[25] Fred GWC Paas and Jeroen JG Van Merriënboer. Instructional control of cognitive load in the training of complex cognitive tasks. *Educational psychology review*, 6(4):351–371, 1994.

[26] Davide Spadini, Fabio Palomba, Tobias Baum, Stefan Hanenberg, Magiel Bruntink, and Alberto Bacchelli. Test-driven code review: An empirical study. In *Proceedings of the 41st International Conference on Software Engineering*. ACM, 2019.

[27] Marco di Biase, Magiel Bruntink, Arie van Deursen, and Alberto Bacchelli. The effects of change decomposition on code review—a controlled experiment. *PeerJ Computer Science*, 5:e193, 2019.

[28] Ying He. The study on expert selection in peer-review based on knowledge management-the new application of scientometrics. In *2009 First International Conference on Information Science and Engineering*, pages 4605–4608. IEEE, 2009.

[29] Ritu Kapur and Balwinder Sodhi. Estimating defectiveness of source code: A predictive model using github content. *arXiv preprint arXiv:1803.07764*, 2018.

[30] David Bowes, Steve Counsell, Tracy Hall, Jean Petric, and Thomas Shippey. Getting defect prediction into industrial practice: the ELFF tool. In *2017 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 44–47. IEEE, 2017.

[31] Sunghun Kim, Thomas Zimmermann, E James Whitehead Jr, and Andreas Zeller. Predicting faults from cached history. In *Proceedings of the 29th international conference on Software Engineering*, pages 489–498. IEEE Computer Society, 2007.

[32] Tracy Hall, Sarah Beecham, David Bowes, David Gray, and Steve Counsell. A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering*, 38(6):1276–1304, 2012.

[33] F. Akiyama. An example of software system debugging. In *Proceedings of the International Federation of Information Processing Societies Congress*, page 353–359. IFIP'71, 1971.

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

[35] Emanuel Giger, Marco D'Ambros, Martin Pinzger, and Harald C Gall. Method-level bug prediction. In *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 171–180. ACM, 2012.

[36] Foyzur Rahman, Daryl Posnett, and Premkumar Devanbu. Recalling the imprecision of cross-project defect prediction. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 61. ACM, 2012.

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

[38] Thomas Zimmermann, Nachiappan Nagappan, Harald Gall, Emanuel Giger, and Brendan Murphy. Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 91–100. ACM, 2009.

[39] 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 Centre for Advanced Studies on Collaborative Research, October 27-30, 2008, Richmond Hill, Ontario, Canada*, page 23. IBM, 2008.

[40] Alberto Bacchelli, Marco D'Ambros, and Michele Lanza. Are popular classes more defect prone? In *International Conference on Fundamental Approaches to Software Engineering*, pages 59–73. Springer, 2010.

[41] Marco D'Ambros, Alberto Bacchelli, and Michele Lanza. On the impact of design flaws on software defects. In *Quality Software (QSIC), 2010 10th International Conference on*, pages 23–31. IEEE, 2010.

[42] Gabriele Bavota, Bernardino De Carluccio, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Orazio Strollo. When does a refactoring induce bugs? an empirical study. In *Source Code Analysis and Manipulation (SCAM), 2012 IEEE 12th International Working Conference on*, pages 104–113. IEEE, 2012.

[43] Gemma Catolino, Fabio Palomba, Andrea De Lucia, Filomena Ferrucci, and Andy Zaidman. Enhancing change prediction models using developer-related factors. *Journal of Systems and Software*, 143:14–28, 2018.

[44] Foutse Khomh, Massimiliano Di Penta, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. An exploratory study of the impact of antipatterns on class change- and fault-proneness. *Empirical Softw. Engg.*, 17(3):243–275, June 2012.

[45] Ganesh J Pai and Joanne Bechta Dugan. Empirical analysis of software fault content and fault proneness using bayesian methods. *IEEE Transactions on software Engineering*, 33(10), 2007.

[46] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Fausto Fasano, Rocco Oliveto, and Andrea De Lucia. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. *Empirical Software Engineering*, pages 1–34, 2017.

[47] Fabio Palomba, Annibale Panichella, Andy Zaidman, Rocco Oliveto, and Andrea De Lucia. The scent of a smell: An extensive comparison between textual and structural smells. *IEEE Transactions on Software Engineering*, 2017.

[48] Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. On the naturalness of buggy code. In *Proceedings of the 38th International Conference on Software Engineering*, pages 428–439. ACM, 2016.

[49] Daryl Posnett, Raissa D'Souza, Premkumar Devanbu, and Vladimir Filkov. Dual ecological measures of focus in software development. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 452–461, Piscataway, NJ, USA, 2013. IEEE Press.

[50] Daniel Joseph Dean, Hiep Nguyen, and Xiaohui Gu. Ubl: Unsupervised behavior learning for predicting performance anomalies in virtualized cloud systems. In *Proceedings of the 9th international conference on Autonomic computing*, pages 191–200. ACM, 2012.

[51] Tung Thanh Nguyen, Tien N Nguyen, and Tu Minh Phuong. Topic-based defect prediction (NIER track). In *Proceedings of the 33rd international conference on software engineering*, pages 932–935. ACM, 2011.

[52] Feng Zhang, Quan Zheng, Ying Zou, and Ahmed E Hassan. Cross-project defect prediction using a connectivity-based unsupervised classifier. In *Proceedings of the 38th International Conference on Software Engineering*, pages 309–320. ACM, 2016.

[53] Gerardo Canfora, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, Annibale Panichella, and Sebastiano Panichella. Multi-objective cross-project defect prediction. In *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*, pages 252–261. IEEE, 2013.

[54] Dario Di Nucci, Fabio Palomba, Rocco Oliveto, and Andrea De Lucia. Dynamic selection of classifiers in bug prediction: An adaptive method. *IEEE Transactions on Emerging Topics in Computational Intelligence*, 1(3):202–212, 2017.

[55] Yasutaka Kamei, Shinsuke Matsumoto, Akito Monden, Ken-ichi Matsumoto, Bram Adams, and Ahmed E Hassan. Revisiting common bug prediction findings using effort-aware models. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–10. IEEE, 2010.

[56] Fabio Palomba, Marco Zanoni, Francesca Arcelli Fontana, Andrea De Lucia, and Rocco Oliveto. Smells like teen spirit: Improving bug prediction performance using the intensity of code smells. In *Proceedings of the 32nd International Conference on Software Maintenance and Evolution (ICSME 2016). Raleigh, USA: IEEE*, 2016.

[57] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. Predicting defects for eclipse. In *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, PROMISE '07, pages 9–, 2007.

[58] Sunghun Kim, E James Whitehead Jr, and Yi Zhang. Classifying software changes: Clean or buggy? *IEEE Transactions on Software Engineering*, 34(2):181–196, 2008.

[59] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering*, 39(6):757–773, 2013.

[60] Shivkumar Shivaji, E James Whitehead, Ram Akella, and Sunghun Kim. Reducing features to improve code change-based bug prediction. *IEEE Transactions on Software Engineering*, 39(4):552–569, 2013.

[61] Takafumi Fukushima, Yasutaka Kamei, Shane McIntosh, Kazuhiro Yamashita, and Naoyasu Ubayashi. An empirical study of just-in-time defect prediction using cross-project models. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 172–181. ACM, 2014.

[62] Xinli Yang, David Lo, Xin Xia, Yun Zhang, and Jianling Sun. Deep learning for just-in-time defect prediction. In *Software Quality, Reliability and Security (QRS), 2015 IEEE International Conference on*, pages 17–26. IEEE, 2015.

[63] Shane McIntosh and Yasutaka Kamei. Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction. *IEEE Transactions on Software Engineering*, 2017.

[64] Qiao Huang, Xin Xia, and David Lo. Supervised vs unsupervised models: A holistic look at effort-aware just-in-time defect prediction. In *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*, pages 159–170. IEEE, 2017.

[65] Victor R Basili. The role of experimentation in software engineering: past, current, and future. In *Proceedings of IEEE 18th International Conference on Software Engineering*, pages 442–449. IEEE, 1996.

[66] Barry Boehm, Hans Dieter Rombach, and Marvin V Zelkowitz. *Foundations of empirical software engineering: the legacy of Victor R. Basili.* Springer Science & Business Media, 2005.

[67] R Burke Johnson and Anthony J Onwuegbuzie. Mixed methods research: A research paradigm whose time has come. *Educational researcher*, 33(7):14–26, 2004.

[68] Barbara A Kitchenham, Shari Lawrence Pfleeger, Lesley M Pickard, Peter W Jones, David C. Hoaglin, Khaled El Emam, and Jarrett Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on software engineering*, 28(8):721–734, 2002.

[69] Michael Fischer, Martin Pinzger, and Harald Gall. Populating a release history database from version control and bug tracking systems. In *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, pages 23–32. IEEE, 2003.

[70] Mike Kuniavsky. *Observing the user experience: a practitioner's guide to user research.* Elsevier, 2003.

[71] Robert K Merton and Patricia L Kendall. The focused interview. *American journal of Sociology*, 51(6):541–557, 1946.

[72] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. The promises and perils of mining github. In *Proceedings of the 11th working conference on mining software repositories*, pages 92–101. ACM, 2014.

[73] Mark Mason. Sample size and saturation in phd studies using qualitative interviews. In *Forum qualitative Sozialforschung/Forum: qualitative social research*, volume 11, 2010.

[74] George Ritzer et al. *The Blackwell encyclopedia of sociology*, volume 1479. Blackwell Publishing New York, NY, USA, 2007.

[75] Klaas-Jan Stol, Paul Ralph, and Brian Fitzgerald. Grounded theory in software engineering research: a critical review and guidelines. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 120–131. IEEE, 2016.

[76] Hazel E Nelson. A modified card sorting test sensitive to frontal lobe defects. *Cortex*, 12(4):313–324, 1976.

[77] Luca Pascarella, Achyudh Ram, Azqa Nadeem, Dinesh Bisesser, Norman Knyazev, and Alberto Bacchelli. Investigating type declaration mismatches in Python. In *2018 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)*, pages 43–48. IEEE, 2018.

[78] Franz-Xaver Geiger, Ivano Malavolta, Luca Pascarella, Fabio Palomba, Dario Di Nucci, and Alberto Bacchelli. A graph-based dataset of commit history of real-world Android apps. In *Proceedings of the 15th International Conference on Mining Software Repositories*, pages 30–33. ACM, 2018.

[79] Luca Pascarella, Fabio Palomba, Massimiliano Di Penta, and Alberto Bacchelli. How is video game development different from software development in open source? In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pages 392–402. IEEE, 2018.

[80] Mariaclaudia Nicolai, Luca Pascarella, Fabio Palomba, and Alberto Bacchelli. Healthcare Android apps: a tale of the customers' perspective. In *Proceedings of the 3rd ACM SIGSOFT International Workshop on App Market Analytics*, pages 33–39. ACM, 2019.

[81] Study replicability dataset 1. Zenodo hosted dataset. `https://doi.org/10.5281/zenodo.3606276`, 2020.

[82] Study replicability dataset 2. Zenodo hosted dataset. `https://doi.org/10.5281/zenodo.3606288`, 2020.

[83] Study replicability dataset 3. Zenodo hosted dataset. `https://doi.org/10.5281/zenodo.3606320`, 2020.

[84] Study replicability dataset 4. Zenodo hosted dataset. https://doi.org/10.5281/zenodo.3698929, 2020.

[85] Wikipedia. Open science. https://en.wikipedia.org/wiki/Open_science, 2020.

[86] NWO. Open science. https://www.nwo.nl/en/policies/open+science, 2020.

[87] Luca Pascarella, Fabio Palomba, and Alberto Bacchelli. Re-evaluating method-level bug prediction. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 592–601. IEEE, 2018.

[88] Manny M Lehman and Laszlo A Belady. *Program evolution: processes of software change*. Academic Press Professional, Inc., 1985.

[89] Ahmed E. Hassan. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 78–88, Washington, DC, USA, 2009. IEEE Computer Society.

[90] Victor R Basili, Lionel C. Briand, and Walcélio L Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on software engineering*, 22(10):751–761, 1996.

[91] D. Di Nucci, F. Palomba, G. De Rosa, G. Bavota, R. Oliveto, and A. De Lucia. A developer centered bug prediction model. *IEEE Transactions on Software Engineering*, PP(99):1–1, 2017.

[92] Nachiappan Nagappan, Andreas Zeller, Thomas Zimmermann, Kim Herzig, and Brendan Murphy. Change bursts as defect predictors. In *Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on*, pages 309–318. IEEE, 2010.

[93] Emad Shihab, Ahmed E Hassan, Bram Adams, and Zhen Ming Jiang. An industrial study on the risk of software changes. In *Proceedings of the 20th International Symposium on the Foundations of Software Engineering*, FSE 2012, page 62. ACM, 2012.

[94] M. Lanza, A. Mocci, and L. Ponzanelli. The tragedy of defect prediction, prince of empirical software engineering research. *IEEE Software*, 33(6):102–105, Nov 2016.

[95] Tibor Gyimóthy, Rudolf Ferenc, and István Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering (TSE)*, 31(10):897–910, 2005.

[96] Thomas J Ostrand, Elaine J Weyuker, and Robert M Bell. Predicting the location and number of faults in large software systems. *IEEE Transactions on Software Engineering*, 31(4):340–355, 2005.

[97] Tim Menzies, Jeremy Greenwald, and Art Frank. Data mining static code attributes to learn defect predictors. *IEEE transactions on software engineering*, 33(1):2–13, 2007.

[98] Ayşe Tosun, Ayşe Bener, Burak Turhan, and Tim Menzies. Practical considerations in deploying statistical methods for defect prediction: A case study within the turkish telecommunications industry. *Information and Software Technology*, 52(11):1242–1257, 2010.

[99] Hideaki Hata, Osamu Mizuno, and Tohru Kikuno. Bug prediction based on fine-grained module histories. In *Proceedings of the 34th International Conference on Software Engineering*, pages 200–210. IEEE Press, 2012.

[100] Foyzur Rahman and Premkumar Devanbu. How, and why, process metrics are better. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 432–441. IEEE Press, 2013.

[101] Shyam R Chidamber and Chris F Kemerer. A metrics suite for object oriented design. *IEEE Transactions on software engineering*, 20(6):476–493, 1994.

[102] Khaled El Emam, Walcelio Melo, and Javam C Machado. The prediction of faulty classes using object-oriented design metrics. *Journal of Systems and Software*, 56(1):63–75, 2001.

[103] Ramanath Subramanyam and M. S. Krishnan. Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects. *Software Engineering, IEEE Transactions on*, 29(4):297–310, 2003.

[104] Niclas Ohlsson and Hans Alberg. Predicting fault-prone software modules in telephone switches. *IEEE Transactions on Software Engineering*, 22(12):886–894, 1996.

[105] 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.

[106] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. Mining metrics to predict component failures. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 452–461, New York, NY, USA, 2006. ACM.

[107] Allen P. Nikora and John C. Munson. Developing fault predictors for evolving software systems. In *Proceedings of the 9th IEEE International Symposium on Software Metrics*, pages 338–349. IEEE CS Press, 2003.

[108] Todd L Graves, Alan F Karr, James S Marron, and Harvey Siy. Predicting fault incidence using software change history. *IEEE Transactions on software engineering*, 26(7):653–661, 2000.

[109] Amit Nandi Taghi M. Khoshgoftaar, Nishith Goel and John McMullan. Detection of software modules with high debug code churn in a very large legacy system. In *Software Reliability Engineering*, pages 364–371. IEEE, 1996.

[110] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. Analysis of the reliability of a subset of change metrics for defect prediction. In *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '08, pages 309–311, New York, NY, USA, 2008. ACM.

[111] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 30th international conference on Software engineering*, pages 181–190. ACM, 2008.

[112] Marco D'Ambros, Michele Lanza, and Romain Robbes. Evaluating defect prediction approaches: a benchmark and an extensive comparison. *Empirical Software Engineering*, 17(4-5):531–577, 2012.

[113] RobertM. Bell, ThomasJ. Ostrand, and ElaineJ. Weyuker. The limited impact of individual developer data on software defect prediction. *Empirical Software Engineering*, 18(3):478–505, 2013.

[114] Thomas J. Ostrand, Elaine J. Weyuker, and Robert M. Bell. Programmer-based fault prediction. In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, PROMISE '10, pages 19:1–19:10, New York, NY, USA, 2010. ACM.

[115] J Kittler et al. Pattern recognition. a statistical approach. 1982.

[116] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, and Kenichi Matsumoto. An empirical comparison of model validation techniques for defect prediction models. *IEEE Transactions on Software Engineering*, 43(1):1–18, 2017.

[117] Stephen Cass and Parthasaradhi Bulusu. Interactive: The top programming languages 2018. *IEEE Spectrum*, `https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2018`, Jul 2018.

[118] Rongxin Wu, Hongyu Zhang, Sunghun Kim, and Shing-Chi Cheung. Relink: Recovering links between bugs and changes. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 15–25, New York, NY, USA, 2011. ACM.

[119] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley, 1999.

[120] Thomas J McCabe. A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320, 1976.

[121] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16:321–357, 2002.

[122] R.M. O'Brien. A caution regarding rules of thumb for variance inflation factors. *Quality & Quantity*, 41(5):673, 2007.

[123] Nitesh V Chawla. Data mining for imbalanced datasets: An overview. In *Data mining and knowledge discovery handbook*, pages 875–886. Springer, 2009.

[124] Tom Dietterich. Overfitting and undercomputing in machine learning. *ACM computing surveys (CSUR)*, 27(3):326–327, 1995.

[125] Mark Andrew Hall. Correlation-based feature selection for machine learning. 1999.

[126] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. *The elements of statistical learning*, volume 1. Springer series in statistics Springer, Berlin, 2001.

[127] Ming Tan, Lin Tan, Sashank Dara, and Caleb Mayeux. Online defect prediction for imbalanced data. In *Proceedings of the 37th International Conference on Software Engineering-Volume 2*, pages 99–108. IEEE Press, 2015.

[128] James A Hanley and Barbara J McNeil. The meaning and use of the area under a receiver operating characteristic (roc) curve. *Radiology*, 143(1):29–36, 1982.

[129] Luca Pascarella, Fabio Palomba, and Alberto Bacchelli. Re-evaluating method-level bug prediction - online appendix. `https://doi.org/10.5281/zenodo.3518990`, 2018.

[130] Anna Corazza, Sergio Di Martino, Filomena Ferrucci, Carmine Gravino, Federica Sarro, and Emilia Mendes. Using tabu search to configure support vector regression for effort estimation. *Empirical Software Engineering*, 18(3):506–546, 2013.

[131] C Huang, LS Davis, and JRG Townshend. An assessment of support vector machines for land cover classification. *International Journal of remote sensing*, 23(4):725–749, 2002.

[132] Ruchika Malhotra. A systematic review of machine learning techniques for software fault prediction. *Appl. Soft Comput.*, 27(C):504–518, February 2015.

[133] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, and Kenichi Matsumoto. Automated parameter optimization of classification techniques for defect prediction models. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*, pages 321–332. IEEE, 2016.

[134] A. J. Scott and M. Knott. A cluster analysis method for grouping means in the analysis of variance. *Biometrics*, 30:507–512, 1974.

[135] Segla Kpodjedo, Filippo Ricca, Philippe Galinier, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. Design evolution metrics for defect prediction in object oriented systems. *Empirical Software Engineering*, 16(1):141–175, 2011.

[136] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? *SIGSOFT Softw. Eng. Notes*, 30(4):1–5, May 2005.

[137] Daniel Alencar da Costa, Shane McIntosh, Weiyi Shang, Uirá Kulesza, Roberta Coelho, and Ahmed E Hassan. A framework for evaluating the results of the szz approach for identifying bug-introducing changes. *IEEE Transactions on Software Engineering*, 43(7):641–657, 2017.

[138] Luca Pascarella, Fabio Palomba, and Alberto Bacchelli. Re-evaluating method-level bug prediction - online appendix. `https://figshare.com/articles/RENE_zip/5909194`, 2018.

[139] B. Ghotra, S. McIntosh, and A. E. Hassan. Revisiting the impact of classification techniques on the performance of defect prediction models. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 789–800, May 2015.

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

[141] Klaus Krippendorff. *Content analysis: An introduction to its methodology*. Sage, 2004.

[142] Jean-Yves Antoine, Jeanne Villaneau, and Anaïs Lefeuvre. Weighted krippendorff's alpha is a more reliable metrics for multi-coders ordinal annotations: experimental studies on emotion, opinion and coreference annotation. In *EACL 2014*, pages 10–p, 2014.

[143] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–305, 2012.

[144] Luca Pascarella, Fabio Palomba, and Alberto Bacchelli. Fine-grained just-in-time defect prediction. *Journal of Systems and Software*, 150:22–36, 2019.

[145] M. M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, Sept 1980.

[146] Ian Sommerville. *Software Engineering: (Update) (8th Edition) (International Computer Science)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[147] Fabio Palomba, Marco Zanoni, Francesca Arcelli Fontana, Andrea De Lucia, and Rocco Oliveto. Toward a smell-aware bug prediction model. *IEEE Transactions on Software Engineering*, 45(2):194–218, 2017.

[148] Davide Spadini, Fabio Palomba, Andy Zaidman, Magiel Bruntink, and Alberto Bacchelli. On the relation of test smells to software code quality. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME). IEEE*, 2018.

[149] Michele Tufano, Gabriele Bavota, Denys Poshyvanyk, Massimiliano Di Penta, Rocco Oliveto, and Andrea De Lucia. An empirical study on developer-related factors characterizing fix-inducing commits. *Journal of Software: Evolution and Process*, 29(1):e1797–n/a, 2017. e1797 JSME-15-0185.R2.

[150] T. Menzies, A. Butcher, D. Cok, A. Marcus, L. Layman, F. Shull, B. Turhan, and T. Zimmermann. Local versus global lessons for defect prediction and effort estimation. *IEEE Transactions on Software Engineering*, 39(6):822–834, June 2013.

[151] Yasutaka Kamei, Takafumi Fukushima, Shane McIntosh, Kazuhiro Yamashita, Naoyasu Ubayashi, and Ahmed E Hassan. Studying just-in-time defect prediction using cross-project models. *Empirical Software Engineering*, 21(5):2072–2106, 2016.

[152] Jacob G. Barnett, Charles K. Gathuru, Luke S. Soldano, and Shane McIntosh. The relationship between commit message detail and defect proneness in Java projects on github. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 496–499, New York, NY, USA, 2016. ACM.

[153] Yibiao Yang, Yuming Zhou, Jinping Liu, Yangyang Zhao, Hongmin Lu, Lei Xu, Baowen Xu, and Hareton 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*, FSE 2016, pages 157–168, New York, NY, USA, 2016. ACM.

[154] Erik Arisholm, Lionel C Briand, and Eivind B Johannessen. A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. *Journal of Systems and Software*, 83(1):2–17, 2010.

[155] Israel Herraiz, Jesus M Gonzalez-Barahona, and Gregorio Robles. Towards a theoretical model for software growth. In *Mining Software Repositories, 2007. ICSE Workshops MSR'07. Fourth International Workshop on*, pages 21–21. IEEE, 2007.

[156] Marek Leszak, Dewayne E Perry, and Dieter Stoll. Classification and evaluation of defects in a project retrospective. *Journal of Systems and Software*, 61(3):173–187, 2002.

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

[158] John C Munson and Sebastian G Elbaum. Code churn: A measure for estimating the impact of code change. In *Software Maintenance, 1998. Proceedings., International Conference on*, pages 24–31. IEEE, 1998.

[159] Audris Mockus and David M Weiss. Predicting risk of software changes. *Bell Labs Technical Journal*, 5(2):169–180, 2000.

[160] Lech Madeyski and Marcin Kawalerowicz. Continuous defect prediction: the idea and a related dataset. In *Proceedings of the 14th International Conference on Mining Software Repositories*, pages 515–518. IEEE Press, 2017.

[161] Jon Eyolfson, Lin Tan, and Patrick Lam. Do time of day and developer experience affect commit bugginess? In *Proceedings of the 8th Working Conference on Mining Software Repositories*, pages 153–162. ACM, 2011.

[162] Tian Jiang, Lin Tan, and Sunghun Kim. Personalized defect prediction. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, pages 279–289. IEEE Press, 2013.

[163] Xin Xia, David Lo, Xinyu Wang, and Xiaohu Yang. Collective personalized change classification with multiobjective search. *IEEE Transactions on Reliability*, 65(4):1810–1829, 2016.

[164] Foyzur Rahman, Daryl Posnett, Abram Hindle, Earl Barr, and Premkumar Devanbu. Bugcache for inspections: Hit or miss? In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 322–331, New York, NY, USA, 2011. ACM.

[165] D. L. Parnas and M. Lawford. The role of inspection in software quality assurance. *IEEE Transactions on Software Engineering*, 29(8):674–676, Aug 2003.

[166] Chadd Williams and Jaime Spacco. Szz revisited: Verifying when changes induce fixes. In *Proceedings of the 2008 Workshop on Defects in Large Software Systems*, DEFECTS '08, pages 32–36, New York, NY, USA, 2008. ACM.

[167] Leon Moonen. Generating robust parsers using island grammars. In *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, WCRE '01, pages 13–, Washington, DC, USA, 2001. IEEE Computer Society.

[168] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E Hassan. The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 192–201. ACM, 2014.

[169] Oleksii Kononenko, Olga Baysal, Latifa Guerrouj, Yaxin Cao, and Michael W Godfrey. Investigating code review quality: Do people and participation matter? In *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, pages 111–120. IEEE, 2015.

[170] S. le Cessie and J.C. van Houwelingen. Ridge estimators in logistic regression. *Applied Statistics*, 41(1):191–201, 1992.

[171] Yoav Freund and Llew Mason. The alternating decision tree learning algorithm. In *Proceedings of the Sixteenth International Conference on Machine Learning*, ICML '99, pages 124–133. Morgan Kaufmann Publishers Inc., 1999.

[172] C. Van Der Malsburg. *Frank Rosenblatt: Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*, pages 245–248. Springer Berlin Heidelberg, 1986.

[173] George H. John and Pat Langley. Estimating continuous distributions in bayesian classifiers. In *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence*, UAI'95, pages 338–345. Morgan Kaufmann Publishers Inc., 1995.

[174] Andy Liaw and Matthew Wiener. Classification and regression by randomforest. *R News*, 2(3):18–22, 2002.

[175] Yue Jiang, Bojan Cukic, and Tim Menzies. Can data transformation help in the detection of fault-prone modules? In *Proceedings of the 2008 Workshop on Defects in Large Software Systems*, DEFECTS '08, pages 16–20. ACM, 2008.

[176] Marko Robnik-Sikonja. Improving random forests. In *ECML*, volume 3201, pages 359–370, 2004.

[177] Johannes L. Grabmeier and Larry A. Lambe. Decision trees for binary classification variables grow equally with the gini impurity measure and pearson's chi-square test. *Int. J. Bus. Intell. Data Min.*, 2(2):213–226, June 2007.

[178] Pierre A. Devijver and Josef Kittler. *Pattern Recognition: A Statistical Approach*. 1982.

[179] Claude Sammut and Geoffrey I. Webb, editors. *Leave-One-Out Cross-Validation*, pages 600–601. Springer US, Boston, MA, 2010.

[180] J. R. Quinlan. Induction of decision trees. *Mach. Learn.*, 1(1):81–106, March 1986.

[181] Thilo Mende and Rainer Koschke. Revisiting the evaluation of defect prediction models. In *Proceedings of the 5th International Conference on Predictor Models in Software Engineering*, page 7. ACM, 2009.

[182] Thilo Mende and Rainer Koschke. Effort-aware defect prediction models. In *Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on*, pages 107–116. IEEE, 2010.

[183] Arash Vahabzadeh, Amin Milani Fard, and Ali Mesbah. An empirical study of bugs in test code. In *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, pages 101–110. IEEE, 2015.

[184] Davide Spadini, Maurício Aniche, Margaret-Anne Storey, Magiel Bruntink, and Alberto Bacchelli. When testing meets code review: Why and how developers review tests. page to appear, 2018.

[185] Gerardo Canfora, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, Annibale Panichella, and Sebastiano Panichella. Defect prediction as a multiobjective optimization problem. *Softw. Test. Verif. Reliab.*, 25(4):426–459, June 2015.

[186] Annibale Panichella, Carol V. Alexandru, Sebastiano Panichella, Alberto Bacchelli, and Harald C. Gall. A search-based training algorithm for cost-aware defect prediction. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, GECCO '16, pages 1077–1084, New York, NY, USA, 2016. ACM.

[187] Jack E Matson, Bruce E Barrett, and Joseph M Mellichamp. Software development cost estimation using function points. *IEEE Transactions on Software Engineering*, 20(4):275–287, 1994.

[188] Hervé Abdi. Bonferroni and šidák corrections for multiple comparisons. *Encyclopedia of measurement and statistics*, 3:103–107, 2007.

[189] Thomas V Perneger. What's wrong with bonferroni adjustments. *BMJ: British Medical Journal*, 316(7139):1236, 1998.

[190] Maurice G Kendall. A new measure of rank correlation. *Biometrika*, 30(1/2):81–93, 1938.

[191] Abram Hindle, Daniel M German, and Ric Holt. What do large commits tell us?: a taxonomical study of large commits. In *Proceedings of the 2008 international working conference on Mining software repositories*, pages 99–108. ACM, 2008.

[192] Xinli Yang, David Lo, Xin Xia, and Jianling Sun. Tlel: A two-layer ensemble learning approach for just-in-time defect prediction. *Information and Software Technology*, 87:206–220, 2017.

[193] Luca Pascarella and Alberto Bacchelli. Classifying code comments in Java open-source software systems. In *Proceedings of the 14th International Conference on Mining Software Repositories*, pages 227–237. IEEE Press, 2017.

[194] Luca Pascarella. Classifying code comments in Java mobile applications. In *2018 IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 39–40. IEEE, 2018.

[195] Luca Pascarella, Magiel Bruntink, and Alberto Bacchelli. Classifying code comments in Java software systems. *Empirical Software Engineering*, pages 1–39, 2019.

[196] Beat Fluri, Michael Wursch, and Harald C Gall. Do code and comments co-evolve? on the relation between source code and comment changes. In *Reverse Engineering, 2007. WCRE 2007. 14th Working Conference on*, pages 70–79. IEEE, 2007.

[197] Scott N Woodfield, Hubert E Dunsmore, and Vincent Yun Shen. The effect of modularization and comments on program comprehension. pages 215–223, 1981.

[198] Ted Tenny. Procedures and comments vs. the banker's algorithm. *SIGCSE Bull.*, 17(3):44–53, September 1985.

[199] Ted Tenny. Program readability: Procedures versus comments. *IEEE Transactions on Software Engineering*, 14(9):1271–1279, 1988.

[200] Carl S. Hartzman and Charles F. Austin. Maintenance productivity: Observations based on an experience in a large system environment. pages 138–170, 1993.

[201] Zhen Ming Jiang and Ahmed E. Hassan. Examining the evolution of code comments in postgresql. In *Proceedings of the 2006 International Workshop on Mining Software Repositories*, MSR '06, pages 179–180, New York, NY, USA, 2006. ACM.

[202] Sergio Cozzetti B de Souza, Nicolas Anquetil, and Kathia M de Oliveira. A study of the documentation essential to software maintenance. In *Proceedings of the 23rd annual international conference on Design of communication: documenting & designing for pervasive information*, pages 68–75. ACM, 2005.

[203] Paul Oman and Jack Hagemeister. Metrics for assessing a software system's maintainability. In *Software Maintenance, 1992. Proceerdings., Conference on*, pages 337–344. IEEE, 1992.

[204] Manuel J Barranco Garcia and Juan Carlos Granja-Alvarez. Maintainability as a key factor in maintenance productivity: a case study. In *icsm*, page 87, 1996.

[205] Apache Software Foundation (ASF) - Apache Hadoop software library. `http://hadoop.apache.org/`. [Online; accessed 10-02-2017].

[206] Dorsaf Haouari, Houari Sahraoui, and Philippe Langlais. How good is your comment? a study of comments in Java programs. In *Empirical Software Engineering and Measurement (ESEM), 2011 International Symposium on*, pages 137–146. IEEE, 2011.

[207] Daniela Steidl, Benjamin Hummel, and Elmar Juergens. Quality analysis of source code comments. In *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*, pages 83–92. IEEE, 2013.

[208] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. /* icomment: Bugs or bad comments?*. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 145–158. ACM, 2007.

[209] Yoann Padioleau, Lin Tan, and Yuanyuan Zhou. Listening to programmers taxonomies and characteristics of comments in operating system code. In *Proceedings of the 31st International Conference on Software Engineering*, pages 331–341. IEEE Computer Society, 2009.

[210] Apache Spark. `http://spark.apache.org`. [Online; accessed 03-Feb-2016].

[211] William Lidwell, Kritina Holden, and Jill Butler. *Universal Principles of Design, Revised and Updated: 125 Ways to Enhance Usability, Influence Perception, Increase Appeal, Make Better Design Decisions, and Teach through Design.* Rockport Publishers, 2nd edition, January 2010.

[212] Joseph L Fleiss. Measuring nominal scale agreement among many raters. *Psychological bulletin*, 76(5):378, 1971.

[213] Mario F Triola. *Elementary statistics.* Pearson/Addison-Wesley Reading, MA, 2006.

[214] Luca Pascarella and Alberto Bacchelli. Manually classified dataset of source code comments. `http://doi.org/10.4121/uuid:232d15bf-ce75-48f5-8a2c-e8e809b8333e`.

[215] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. Adaptive Computation and Machine Learning series. The MIT Press, 2016.

[216] Christopher D Manning, Prabhakar Raghavan, Hinrich Schütze, et al. *Introduction to information retrieval*, volume 1. Cambridge university press Cambridge, 2008.

[217] Bent Flyvbjerg. Five misunderstandings about case-study research. *Qualitative inquiry*, 12(2):219–245, 2006.

[218] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K Vijay-Shanker. Towards automatically generating summary comments for Java methods. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 43–52. ACM, 2010.

[219] Alberto Bacchelli, Michele Lanza, and Romain Robbes. Linking e-mails and source code artifacts. In *Proceedings of ICSE 2010 (32nd International Conference on Software Engineering)*, pages 375–384. ACM Press, 2010.

[220] Fabrizio Sebastiani. Machine learning in automated text categorization. *ACM computing surveys (CSUR)*, 34(1):1–47, 2002.

[221] Alberto Bacchelli, Tommaso dal Sasso, Marco D'Ambros, and Michele Lanza. Content classification of development emails. In *In Proceedings of ICSE 2012 (34th ACM/IEEE International Conference on Software Engineering)*, pages 375–385, 2012.

[222] James W Paulson, Giancarlo Succi, and Armin Eberlein. An empirical study of open-source and closed-source software products. *IEEE Transactions on Software Engineering*, 30(4):246–256, 2004.

[223] Dawn J Lawrie, Henry Feild, and David Binkley. Leveraged quality assessment using information retrieval techniques. In *Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on*, pages 149–158. IEEE, 2006.

[224] Andrian Marcus and Jonathan I Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Software Engineering, 2003. Proceedings. 25th International Conference on*, pages 125–135. IEEE, 2003.

[225] De Lucia et al. Information retrieval models for recovering traceability links between code and documentation. In *Software Maintenance, 2000. Proceedings. International Conference on*, pages 40–49. IEEE, 2000.

[226] Ioannis Stamelos, Lefteris Angelis, Apostolos Oikonomou, and Georgios L Bleris. Code quality analysis in open source software development. *Information Systems Journal*, 12(1):43–60, 2002.

[227] Andrian Marcus, Jonathan I. Maletic, and Andrey Sergeyev. Recovery of traceability links between software documentation and source code. *International Journal of Software Engineering and Knowledge Engineering*, 15(5):811–836, 2005.

[228] Walid Maalej and Martin P. Robillard. Patterns of knowledge in api reference documentation. *IEEE Trans. Softw. Eng.*, 39(9):1264–1282, September 2013.

[229] René Witte, Yonggang Zhang, and Juergen Rilling. Empowering software maintainers with semantic web technologies. In *The Semantic Web: Research and Applications, 4th European Semantic Web Conference, ESWC 2007, Innsbruck, Austria, June 3-7, 2007, Proceedings*, pages 37–52, 2007.

[230] Luca Pascarella, Fabio Palomba, and Alberto Bacchelli. On the performance of method-level bug prediction: A negative result. *Journal of Systems and Software*, 2019.

[231] Ugo Buy, Alessandro Orso, and Mauro Pezze. Automated testing of classes. In *ACM SIGSOFT Software Engineering Notes*, volume 25, pages 39–48. ACM, 2000.

[232] Christian Bird, Nachiappan Nagappan, Harald Gall, Brendan Murphy, and Premkumar Devanbu. Putting it all together: Using socio-technical networks to predict failures. In *Software Reliability Engineering, 2009. ISSRE'09. 20th International Symposium on*, pages 109–119. IEEE, 2009.

[233] Marco D'Ambros, Michele Lanza, and Romain Robbes. An extensive comparison of bug prediction approaches. In *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, pages 31–41. IEEE, 2010.

[234] Walid M. Ibrahim, Nicolas Bettenburg, Bram Adams, and Ahmed E. Hassan. On the relationship between comment update practices and software bugs. *Journal of Systems and Software*, 85(10):2293 – 2304, 2012. Automated Software Evolution.

[235] R Hosseini, Burak Turhan, and Dimuthu Gunarathna. A systematic literature review and meta-analysis on cross project defect prediction. 2017.

[236] Gemma Catolino. Just-in-time bug prediction in mobile applications: the domain matters! In *Mobile Software Engineering and Systems (MOBILESoft), 2017 IEEE/ACM 4th International Conference on*, pages 201–202. IEEE, 2017.

[237] Gemma Catolino, Dario Di Nucci, and Filomena Ferrucci. Cross-project just-in-time bug prediction for mobile apps: An empirical assessment. In *2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 99–110. IEEE, 2019.

[238] Thong Hoang, Hoa Khanh Dam, Yasutaka Kamei, David Lo, and Naoyasu Ubayashi. Deepjit: an end-to-end deep learning framework for just-in-time defect prediction. In *Proceedings of the 16th International Conference on Mining Software Repositories*, pages 34–45. IEEE Press, 2019.

[239] Hoa Khanh Dam, Trang Pham, Shien Wee Ng, Truyen Tran, John Grundy, Aditya Ghose, Taeksu Kim, and Chul-Joo Kim. Lessons learned from using a deep tree-based model for software defect prediction in practice. In *Proceedings of the 16th International Conference on Mining Software Repositories*, pages 46–57. IEEE Press, 2019.

[240] Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu. 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*, pages 4–14. ACM, 2011.

[241] Michaela Greiler, Kim Herzig, and Jacek Czerwonka. Code ownership and software quality: a replication study. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 2–12. IEEE, 2015.

[242] Foyzur Rahman and Premkumar Devanbu. Ownership, experience and defects: a fine-grained study of authorship. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 491–500. ACM, 2011.

[243] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[244] Seyyed Ehsan Salamati Taba, Foutse Khomh, Ying Zou, Ahmed E Hassan, and Meiyappan Nagappan. Predicting bugs using antipatterns. In *2013 IEEE International Conference on Software Maintenance*, pages 270–279. IEEE, 2013.

[245] Andrian Marcus, Denys Poshyvanyk, and Rudolf Ferenc. Using the conceptual cohesion of classes for fault prediction in object-oriented systems. *IEEE Transactions on Software Engineering*, 34(2):287–300, 2008.

[246] Annibale Panichella, Bogdan Dit, Rocco Oliveto, Massimiliano Di Penta, Denys Poshyvanyk, and Andrea De Lucia. How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 522–531. IEEE Press, 2013.

[247] Fabio Palomba, Annibale Panichella, Andrea De Lucia, Rocco Oliveto, and Andy Zaidman. A textual-based technique for smell detection. In *Program Comprehension (ICPC), 2016 IEEE 24th International Conference on*, pages 1–10. IEEE, 2016.

[248] Fabio Palomba, Andy Zaidman, and AD Lucia. Automatic test smell detection using information retrieval techniques. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME). IEEE*, 2018.

[249] Gabriele Bavota, Andrea De Lucia, Andrian Marcus, and Rocco Oliveto. Automating extract class refactoring: an improved method and its evaluation. *Empirical Software Engineering*, 19(6):1617–1664, 2014.

[250] Gabriele Bavota, Malcom Gethers, Rocco Oliveto, Denys Poshyvanyk, and Andrea de Lucia. Improving software modularization via automated analysis of latent topics and dependencies. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(1):4, 2014.

[251] Gabriele Bavota, Rocco Oliveto, Malcom Gethers, Denys Poshyvanyk, and Andrea De Lucia. Methodbook: Recommending move method refactorings via relational topic models. *IEEE Transactions on Software Engineering*, 40(7):671–694, 2014.

[252] Hirohisa Aman, Sousuke Amasaki, Takashi Sasaki, and Minoru Kawahara. Empirical analysis of fault-proneness in methods by focusing on their comment lines. In *2014 21st Asia-Pacific Software Engineering Conference*, volume 2, pages 51–56. IEEE, 2014.

[253] Raymond PL Buse and Westley R Weimer. Learning a metric for code readability. *IEEE Transactions on Software Engineering*, 36(4):546–558, 2010.

[254] David Binkley, Henry Feild, Dawn Lawrie, and Maurizio Pighin. Increasing diversity: Natural language measures for software fault prediction. *Journal of Systems and Software*, 82(11):1793–1803, 2009.

[255] M. F. Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.

[256] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. When and why your code starts to smell bad (and whether the smells go away). *IEEE Transactions on Software Engineering*, 43(11):1063–1088, 2017.

[257] Naouel Moha, Yann-Gael Gueheneuc, Laurence Duchien, and Anne-Francoise Le Meur. Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, 36(1):20–36, 2009.

[258] Fabiano Pecorelli, Fabio Palomba, Dario Di Nucci, and Andrea De Lucia. Comparing heuristic and machine learning approaches for metric-based code smell detection. In *Proceedings of the 27th International Conference on Program Comprehension*, pages 93–104. IEEE Press, 2019.

[259] Eduardo Fernandes, Johnatan Oliveira, Gustavo Vale, Thanis Paiva, and Eduardo Figueiredo. A review-based comparative study of bad smell detection tools. In *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*, EASE '16, pages 18:1–18:12, New York, NY, USA, 2016. ACM.

[260] Fabio Palomba, Damian Andrew Andrew Tamburri, Francesca Arcelli Fontana, Rocco Oliveto, Andy Zaidman, and Alexander Serebrenik. Beyond technical aspects: How do community smells influence the intensity of code smells? *IEEE transactions on software engineering*, 2018.

[261] Gemma Catolino, Fabio Palomba, Francesca Arcelli Fontana, Andrea De Lucia, Andy Zaidman, and Filomena Ferrucci. Improving change prediction models with code smell-related information. *Empirical Software Engineering*, page to appear, 2019.

[262] Claude E Shannon. Prediction and entropy of printed english. *Bell system technical journal*, 30(1):50–64, 1951.

[263] David Bowes, Tracy Hall, Mark Harman, Yue Jia, Federica Sarro, and Fan Wu. Mutation-aware fault prediction. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 330–341. ACM, 2016.

[264] Marcelo Cataldo, James D Herbsleb, and Kathleen M Carley. Socio-technical congruence: a framework for assessing the impact of technical and work dependencies on software development productivity. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 2–11. ACM, 2008.

[265] J. Cohen. *Statistical power analysis for the behavioral sciences*. Lawrence Earlbaum Associates, 2nd edition edition, 1988.

[266] Janez Demšar. Statistical comparisons of classifiers over multiple data sets. *Journal of Machine learning research*, 7(Jan):1–30, 2006.

[267] Luca Pascarella, Davide Spadini, Fabio Palomba, Magiel Bruntink, and Alberto Bacchelli. Information needs in contemporary code review. *Proceedings of the ACM on Human-Computer Interaction*, 2(CSCW):135, 2018.

[268] Jason Cohen. Modern code review. In Andy Oram and Greg Wilson, editors, *Making Software*, chapter 18, pages 329–338. O'Reilly, 2010.

[269] Tobias Baum, Olga Liskin, Kai Niklas, and Kurt Schneider. A faceted classification scheme for change-based industrial code review processes. In *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 74–85. IEEE, 2016.

[270] Moritz Beller, Alberto Bacchelli, Andy Zaidman, and Elmar Juergens. Modern code reviews in open-source projects: which problems do they fix? In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 202–211. ACM, 2014.

[271] Yida Tao, Yingnong Dang, Tao Xie, Dongmei Zhang, and Sunghun Kim. How do software engineers understand code changes?: an exploratory study in industry. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 2012.

[272] Andrew Sutherland and Gina Venolia. Can peer code reviews be exploited for later information needs? In *Software Engineering-Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on*, pages 259–262. IEEE, 2009.

[273] Rodrigo Morales, Shane McIntosh, and Foutse Khomh. Do code review practices impact design quality? a case study of the qt, vtk, and itk projects. In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, pages 171–180. IEEE, 2015.

[274] Mika V Mantyla and Casper Lassenius. What types of defects are really discovered in code reviews? *Software Engineering, IEEE Transactions on*, 35(3):430–448, 2009.

[275] Peter C Rigby, Daniel M German, Laura Cowen, and Margaret-Anne Storey. Peer review on open source software projects: Parameters, statistical models, and theory. *ACM Transactions on Software Engineering and Methodology*, page 34, 2014.

[276] Patanamon Thongtanunam, Shane McIntosh, Ahmed E Hassan, and Hajimu Iida. Investigating code review practices in defective files: An empirical study of the qt system. In *MSR '15 Proceedings of the 12th Working Conference on Mining Software Repositories*, 2015.

[277] Andrew J. Ko, Robert DeLine, and Gina Venolia. Information Needs in Collocated Software Development Teams. In *29th International Conference on Software Engineering (ICSE'07)*, pages 344–353, 2007.

[278] Andrew Sutherland and Gina Venolia. Can peer code reviews be exploited for later information needs? In *Software Engineering-Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on*, pages 259–262. IEEE, 2009.

[279] Silvia Breu, Rahul Premraj, Jonathan Sillito, and Thomas Zimmermann. Information Needs in Bug Reports : Improving Cooperation Between Developers and Users. *Proceedings of the 2010 Computer Supported Cooperative Work Conference*, pages 301–310, 2010.

[280] Raymond P.L. Buse and Thomas Zimmermann. Information needs for software development analytics. In *Proceedings - International Conference on Software Engineering*, pages 987–996, 2012.

[281] Richard A Baker Jr. Code reviews enhance software quality. In *Proceedings of the 19th International Conference on Software Engineering*, pages 570–571. ACM, 1997.

[282] Norihito Kitagawa, Hideaki Hata, Akinori Ihara, Kiminao Kogiso, and Kenichi Matsumoto. Code review participation: game theoretical modeling of reviewers in gerrit datasets. In *Proceedings of the 9th International Workshop on Cooperative and Human Aspects of Software Engineering*, pages 64–67. ACM, 2016.

[283] Jacek Czerwonka, Michaela Greiler, and Jack Tilford. Code reviews do not find bugs. how the current code review best practice slows us down. In *Proceedings of the 2015 International Conference on Software Engineering*. IEEE – Institute of Electrical and Electronics Engineers, May 2015.

[284] Fevzi Belli and Radu Crisan. Towards automation of checklist-based code-reviews. In *Software Reliability Engineering, 1996. Proceedings., Seventh International Symposium on*, pages 24–33. IEEE, 1996.

[285] Jonathan Sillito, Gail C Murphy, and Kris De Volder. Questions programmers ask during software evolution tasks. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 23–34. ACM, 2006.

[286] JD Herbsleb and E Kuwana. Preserving knowledge in design projects: What designers need to know. *Chi '93 & Interact '93*, pages 7–14, 1993.

[287] Ulrike Abelein and Barbara Paech. Understanding the Influence of User Participation and Involvement on System Success: a Systematic Mapping Study. *Empirical Software Engineering*, 20(1):28–81, 2015.

[288] Patanamon Thongtanunam, Shane McIntosh, Ahmed E Hassan, and Hajimu Iida. Revisiting code ownership and its relationship with software quality in the scope of modern code review. In *Proceedings of the 38th international conference on software engineering*, pages 1039–1050. ACM, 2016.

[289] Chris Sauer, D Ross Jeffery, Lesley Land, and Philip Yetton. The effectiveness of software development technical reviews: A behaviorally motivated program of research. *Software Engineering, IEEE Transactions on*, 26(1):1–14, 2000.

[290] Adam Porter, Harvey Siy, Audris Mockus, and Lawrence Votta. Understanding the sources of variation in software inspections. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 7(1):41–79, 1998.

[291] Patanamon Thongtanunam, Shane Mcintosh, Ahmed E. Hassan, and Hajimu Iida. Review Participation in Modern Code Review. *Empirical Software Engineering (EMSE)*, page to appear, 2016.

[292] Gabriele Bavota and Barbara Russo. Four eyes are better than two: On the impact of code reviews on software quality. In *2015 IEEE 31st International Conference on Software Maintenance and Evolution, ICSME 2015 - Proceedings*, pages 81–90, 2015.

[293] Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli. Modern code review: a case study at Google. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, pages 181–190. ACM, 2018.

[294] Achyudh Ram, Anand Ashok Sawant, Marco Castelluccio, and Alberto Bacchelli. What makes a code change easier to review? an empirical investigation on code change reviewability. In *26th Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, page forthcoming, 2018.

[295] Felipe Ebert, Fernando Castor, Nicole Novielli, and Alexander Serebrenik. Confusion detection in code reviews. In *33rd International Conference on Software Maintenance and Evolution (ICSME), Proceedings*. ICSME, 2017.

[296] Yujuan Jiang, Bram Adams, and Daniel M. German. Will my patch make it? And how fast?: Case study on the linux kernel. In *IEEE International Working Conference on Mining Software Repositories*, pages 101–110, 2013.

[297] Olga Baysal, Oleksii Kononenko, Reid Holmes, and Michael W Godfrey. The influence of non-technical factors on code review. In *Reverse Engineering (WCRE), 2013 20th Working Conference on*, pages 122–131. IEEE, 2013.

[298] Robert Chatley and Lawrence Jones. Diggit: Automated code review via software repository mining. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 567–571. IEEE, 2018.

[299] Deborah Finfgeld-Connett. Use of content analysis to conduct knowledge-building and theory-generating qualitative systematic reviews. *Qualitative Research*, 14(3):341–352, 2014.

[300] Klaus Krippendorff. Agreement and information in the reliability of coding. *Communication Methods and Measures*, 5(2):93–112, 2011.

[301] Robert S Weiss. *Learning from strangers: The art and method of qualitative interview studies*. Simon and Schuster, 1995.

[302] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Denys Poshyvanyk, and Andrea De Lucia. Mining version histories for detecting code smells. *IEEE Transactions on Software Engineering*, 41(5):462–489, 2015.

[303] Bogdan Vasilescu, Daryl Posnett, Baishakhi Ray, Mark GJ van den Brand, Alexander Serebrenik, Premkumar Devanbu, and Vladimir Filkov. Gender and tenure diversity in github teams. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, pages 3789–3798. ACM, 2015.

[304] W. J. Conover. *Practical Nonparametric Statistics*. Wiley, 3rd edition edition, 1998.

[305] Robert J. Grissom and John J. Kim. *Effect sizes for research: A broad practical approach*. Lawrence Earlbaum Associates, 2nd edition edition, 2005.

[306] Mike Barnett, Christian Bird, Joao Brunet, and Shuvendu K Lahiri. Helping developers help themselves: Automatic decomposition of code review changesets. In *Proceedings of the 37th International Conference on Software Engineering*, volume 1 of *ICSE 2015*, pages 134–144. IEEE, 2015.

[307] Kim Herzig and Andreas Zeller. The impact of tangled code changes. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 121–130. IEEE Press, 2013.

[308] Kazuki Hamasaki, Raula Gaikovina Kula, Norihiro Yoshida, AE Cruz, Kenji Fujiwara, and Hajimu Iida. Who does what during a code review? datasets of oss peer review repositories. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 49–52. IEEE Press, 2013.

[309] Jason Tsay, Laura Dabbish, and James Herbsleb. Let's talk about it: evaluating contributions through discussion in github. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 144–154. ACM, 2014.

[310] Patanamon Thongtanunam, Raula Gaikovina Kula, Ana Erika Camargo Cruz, Norihiro Yoshida, and Hajimu Iida. Improving code review effectiveness through reviewer recommendations. In *Proceedings of the 7th International Workshop on Cooperative and Human Aspects of Software Engineering*, pages 119–122. ACM, 2014.

[311] Kenji Yamauchi, Jiachen Yang, Keisuke Hotta, Yoshiki Higo, and Shinji Kusumoto. Clustering commits for understanding the intents of implementation. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, pages 406–410. IEEE, 2014.

[312] Martín Dias, Alberto Bacchelli, Georgios Gousios, Damien Cassou, and Stéphane Ducasse. Untangling fine-grained code changes. In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, pages 341–350. IEEE, 2015.

[313] Yida Tao and Sunghun Kim. Partitioning composite code changes to facilitate code review. In *Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on*, pages 180–190. IEEE, 2015.

[314] Marco di Biase, Magiel Bruntink, Arie van Deursen, and Alberto Bacchelli. The effects of change-decomposition on code review-a controlled experiment. *PeerJ Preprints*, 2018.

[315] Vipin Balachandran. Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 931–940. IEEE Press, 2013.

[316] Luca Ponzanelli, Simone Scalabrino, Gabriele Bavota, Andrea Mocci, Massimiliano Di Penta, Rocco Oliveto, and Michele Lanza. Supporting software developers with a holistic recommender system. In *Proceedings of ICSE 2017 (39th ACM/IEEE International Conference on Software Engineering)*. to be published, 2017.

[317] Martin Vechev, Eran Yahav, et al. Programming with "big code". *Foundations and Trends® in Programming Languages*, 3(4):231–284, 2016.

[318] Raymond PL Buse and Westley R Weimer. Automatically documenting program changes. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 33–42. ACM, 2010.

[319] Gerardo Canfora, Luigi Cerulo, and Massimiliano Di Penta. Ldiff: An enhanced line differencing tool. In *Proceedings of the 31st International Conference on Software Engineering*, pages 595–598. IEEE Computer Society, 2009.

[320] Chris Parnin and Carsten G$^{'}$org. Improving change descriptions with change contexts. In *Proceedings of the 2008 international working conference on Mining software repositories*, pages 51–60. ACM, 2008.

[321] Luis Fernando Cortés-Coy, Mario Linares-Vásquez, Jairo Aponte, and Denys Poshyvanyk. On automatically generating commit messages via summarization of source code changes. In *Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on*, pages 275–284. IEEE, 2014.

Curriculum Vitæ

Ing. Luca Pascarella

1987 June 18	Date of birth in Benevento, Italy

Education

1/2016 - 1/2020	Ph.D. candidate, Software Engineering Research Group, Delft University of Technology, The Netherlands
10/2013 - 5/2015	M.Sc. Software Engineering, University of Sannio, Italy
10/2008 - 5/2012	B.Sc. Software Engineering, University of Sannio, Italy

Experience

10/2017 - 1/2019	Research intern, Software Improvement Group (SIG), The Netherlands
5/2016 - 9/2016	Research visitor, IMDEA Software Institute, Spain
1/2010 - 31/2012	Entrepreneur, LP Systems, Italy
1/2008 - 01/2017	Magazine Writer, Elettronica IN, Italy
1/2007 - 1/2009	Teacher, High School L. Palmieri, Italy

Meeting, Talk, & Lecture

Guest	SHONAN “Release Engineering for Mobile Apps”

Guest Lecture	Software Dependability, University of Salerno
Guest Lecture	Software Analytics, Delft University of Technology, Delft University of Technology
Guest Lecture	Seminary, University of Sannio

Awards

2018	Best Paper Award Honorable Mention
2018	ACM International Travel, funds for traveling
2017	ACM SIGSOFT Distinguished Paper Award
2013	ERASMUS-2013 placement, funds for traveling

List of Publications

12. *Luca Pascarella*, Fabio Palomba, Alberto Bacchelli: On the Performance of Method-Level Bug Prediction: A Negative Result (JSS).

11. Mariaclaudia Nicolai, *Luca Pascarella*, Fabio Palomba, and Alberto Bacchelli: Healthcare Android Apps: A Tale of the Customers' Perspective. In Proceedings of the 3rd International Workshop on App Market Analytics. (ESEC/FSE-2019). Aug 26-30 2019. Tallinn, Estonia.

10. *Luca Pascarella*, Magiel Bruntink, Alberto Bacchelli: Classifying code comments in Java software systems. In Journal of Empirical Software Engineering (EMSE).

9. *Luca Pascarella*, Fabio Palomba, Alberto Bacchelli: Fine-Grained Just-In-Time Defect Prediction. In Journal of Systems and Software (JSS).

8. *Luca Pascarella*, Davide Spadini, Fabio Palomba, Magiel Bruntink, Alberto Bacchelli: Information Needs in Contemporary Code Review. In Proceedings of the 21st ACM Conference on Computer-Supported Cooperative Work and Social Computing (CSCW-2018). November 3-7 2018. Jersey City, USA.

7. *Luca Pascarella*: Classifying code comments in Java Mobile Applications. In Proceedings of the 5th IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft-2018) — Student Research Competition — May 27-28 2018. Gothenburg, Sweden.

6. *Luca Pascarella*, Fabio Palomba, Massimiliano Di Penta, Alberto Bacchelli: How Is Video Game Development Different from Software Development in Open Source?. In Proceedings of the 15th International Conference on Mining Software Repositories (MSR-2018), May 27-28 2018. Gothenburg, Sweden.

5. Franz-Xaver Geiger, Ivano Malavolta, *Luca Pascarella*, Fabio Palomba, Dario Di Nucci, Ivano Malavolta, Alberto Bacchelli: A Graph-based Dataset of Commit History of Real-World Android apps. In Proceedings of the 15th International Conference on Mining Software Repositories (MSR-2018), May 27-28 2018. Gothenburg, Sweden.

4. *Luca Pascarella*, Franz-Xaver Geiger, Fabio Palomba, Dario Di Nucci, Ivano Malavolta, Alberto Bacchelli: Self-Reported Activities of Android Developers. In Proceedings of the 5th IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft-2018), May 27-28 2018. Gothenburg, Sweden.

3. *Luca Pascarella*, Fabio Palomba, Alberto Bacchelli: Re-evaluating Method-Level Bug Prediction. In Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering (SANER-2018), March 20-23, 2018. Campobasso, Italy.

2. *Luca Pascarella*, Achyudh Ram, Azqa Nadeem, Dinesh Bisesser, Norman Knyazev, Alberto Bacchelli: Investigating Type Declaration Mismatches in Python. In Proceedings of the Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE-2018), Mar 20-23, 2018. Campobasso, Italy.

1. *Luca Pascarella*, Alberto Bacchelli: Classifying code comments in Java open-source software systems. In Proceedings of the 14th International Conference on Mining Software Repositories (MSR-2017), May 20-21, 2017. Buenos Aires, Argentina.

Included in this thesis.
Won a best paper, tool demonstration, or proposal award.

Titles in the IPA Dissertation Series since 2015

G. Alpár. *Attribute-Based Identity Management: Bridging the Cryptographic Design of ABCs with the Real World.* Faculty of Science, Mathematics and Computer Science, RU. 2015-01

A.J. van der Ploeg. *Efficient Abstractions for Visualization and Interaction.* Faculty of Science, UvA. 2015-02

R.J.M. Theunissen. *Supervisory Control in Health Care Systems.* Faculty of Mechanical Engineering, TU/e. 2015-03

T.V. Bui. *A Software Architecture for Body Area Sensor Networks: Flexibility and Trustworthiness.* Faculty of Mathematics and Computer Science, TU/e. 2015-04

A. Guzzi. *Supporting Developers' Teamwork from within the IDE.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2015-05

T. Espinha. *Web Service Growing Pains: Understanding Services and Their Clients.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2015-06

S. Dietzel. *Resilient In-network Aggregation for Vehicular Networks.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2015-07

E. Costante. *Privacy throughout the Data Cycle.* Faculty of Mathematics and Computer Science, TU/e. 2015-08

S. Cranen. *Getting the point — Obtaining and understanding fixpoints in model checking.* Faculty of Mathematics and Computer Science, TU/e. 2015-09

R. Verdult. *The (in)security of proprietary cryptography.* Faculty of Science, Mathematics and Computer Science, RU. 2015-10

J.E.J. de Ruiter. *Lessons learned in the analysis of the EMV and TLS security protocols.* Faculty of Science, Mathematics and Computer Science, RU. 2015-11

Y. Dajsuren. *On the Design of an Architecture Framework and Quality Evaluation for Automotive Software Systems.* Faculty of Mathematics and Computer Science, TU/e. 2015-12

J. Bransen. *On the Incremental Evaluation of Higher-Order Attribute Grammars.* Faculty of Science, UU. 2015-13

S. Picek. *Applications of Evolutionary Computation to Cryptology.* Faculty of Science, Mathematics and Computer Science, RU. 2015-14

C. Chen. *Automated Fault Localization for Service-Oriented Software Systems.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2015-15

S. te Brinke. *Developing Energy-Aware Software.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2015-16

R.W.J. Kersten. *Software Analysis Methods for Resource-Sensitive Systems.* Faculty of Science, Mathematics and Computer Science, RU. 2015-17

J.C. Rot. *Enhanced coinduction.* Faculty of Mathematics and Natural Sciences, UL. 2015-18

M. Stolikj. *Building Blocks for the Internet of Things.* Faculty of Mathematics and Computer Science, TU/e. 2015-19

D. Gebler. *Robust SOS Specifications of Probabilistic Processes.* Faculty of Sciences, Department of Computer Science, VUA. 2015-20

M. Zaharieva-Stojanovski. *Closer to Reliable Software: Verifying functional behaviour of concurrent programs.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2015-21

R.J. Krebbers. *The C standard formalized in Coq.* Faculty of Science, Mathematics and Computer Science, RU. 2015-22

R. van Vliet. *DNA Expressions – A Formal Notation for DNA.* Faculty of Mathematics and Natural Sciences, UL. 2015-23

S.-S.T.Q. Jongmans. *Automata-Theoretic Protocol Programming.* Faculty of Mathematics and Natural Sciences, UL. 2016-01

S.J.C. Joosten. *Verification of Interconnects.* Faculty of Mathematics and Computer Science, TU/e. 2016-02

M.W. Gazda. *Fixpoint Logic, Games, and Relations of Consequence.* Faculty of Mathematics and Computer Science, TU/e. 2016-03

S. Keshishzadeh. *Formal Analysis and Verification of Embedded Systems for Healthcare.* Faculty of Mathematics and Computer Science, TU/e. 2016-04

P.M. Heck. *Quality of Just-in-Time Requirements: Just-Enough and Just-in-Time.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2016-05

Y. Luo. *From Conceptual Models to Safety Assurance – Applying Model-Based Techniques to Support Safety Assurance.* Faculty of Mathematics and Computer Science, TU/e. 2016-06

B. Ege. *Physical Security Analysis of Embedded Devices.* Faculty of Science, Mathematics and Computer Science, RU. 2016-07

A.I. van Goethem. *Algorithms for Curved Schematization.* Faculty of Mathematics and Computer Science, TU/e. 2016-08

T. van Dijk. *Sylvan: Multi-core Decision Diagrams.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2016-09

I. David. *Run-time resource management for component-based systems.* Faculty of Mathematics and Computer Science, TU/e. 2016-10

A.C. van Hulst. *Control Synthesis using Modal Logic and Partial Bisimilarity – A Treatise Supported by Computer Verified Proofs.* Faculty of Mechanical Engineering, TU/e. 2016-11

A. Zawedde. *Modeling the Dynamics of Requirements Process Improvement.* Faculty of Mathematics and Computer Science, TU/e. 2016-12

F.M.J. van den Broek. *Mobile Communication Security.* Faculty of Science, Mathematics and Computer Science, RU. 2016-13

J.N. van Rijn. *Massively Collaborative Machine Learning.* Faculty of Mathematics and Natural Sciences, UL. 2016-14

M.J. Steindorfer. *Efficient Immutable Collections.* Faculty of Science, UvA. 2017-01

W. Ahmad. *Green Computing: Efficient Energy Management of Multiprocessor Streaming Applications via Model Checking.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2017-02

D. Guck. *Reliable Systems – Fault tree analysis via Markov reward automata.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2017-03

H.L. Salunkhe. *Modeling and Buffer Analysis of Real-time Streaming Radio Applications Scheduled on Heterogeneous Multiprocessors.* Faculty of Mathematics and Computer Science, TU/e. 2017-04

A. Krasnova. *Smart invaders of private matters: Privacy of communication on the Internet and in the Internet of Things (IoT).*

Faculty of Science, Mathematics and Computer Science, RU. 2017-05

A.D. Mehrabi. *Data Structures for Analyzing Geometric Data.* Faculty of Mathematics and Computer Science, TU/e. 2017-06

D. Landman. *Reverse Engineering Source Code: Empirical Studies of Limitations and Opportunities.* Faculty of Science, UvA. 2017-07

W. Lueks. *Security and Privacy via Cryptography – Having your cake and eating it too.* Faculty of Science, Mathematics and Computer Science, RU. 2017-08

A.M. Şutîi. *Modularity and Reuse of Domain-Specific Languages: an exploration with MetaMod.* Faculty of Mathematics and Computer Science, TU/e. 2017-09

U. Tikhonova. *Engineering the Dynamic Semantics of Domain Specific Languages.* Faculty of Mathematics and Computer Science, TU/e. 2017-10

Q.W. Bouts. *Geographic Graph Construction and Visualization.* Faculty of Mathematics and Computer Science, TU/e. 2017-11

A. Amighi. *Specification and Verification of Synchronisation Classes in Java: A Practical Approach.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-01

S. Darabi. *Verification of Program Parallelization.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-02

J.R. Salamanca Tellez. *Coequations and Eilenberg-type Correspondences.* Faculty of Science, Mathematics and Computer Science, RU. 2018-03

P. Fiterău-Broştean. *Active Model Learning for the Analysis of Network Protocols.* Faculty of Science, Mathematics and Computer Science, RU. 2018-04

D. Zhang. *From Concurrent State Machines to Reliable Multi-threaded Java Code.* Faculty of Mathematics and Computer Science, TU/e. 2018-05

H. Basold. *Mixed Inductive-Coinductive Reasoning Types, Programs and Logic.* Faculty of Science, Mathematics and Computer Science, RU. 2018-06

A. Lele. *Response Modeling: Model Refinements for Timing Analysis of Runtime Scheduling in Real-time Streaming Systems.* Faculty of Mathematics and Computer Science, TU/e. 2018-07

N. Bezirgiannis. *Abstract Behavioral Specification: unifying modeling and programming.* Faculty of Mathematics and Natural Sciences, UL. 2018-08

M.P. Konzack. *Trajectory Analysis: Bridging Algorithms and Visualization.* Faculty of Mathematics and Computer Science, TU/e. 2018-09

E.J.J. Ruijters. *Zen and the art of railway maintenance: Analysis and optimization of maintenance via fault trees and statistical model checking.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-10

F. Yang. *A Theory of Executability: with a Focus on the Expressivity of Process Calculi.* Faculty of Mathematics and Computer Science, TU/e. 2018-11

L. Swartjes. *Model-based design of baggage handling systems.* Faculty of Mechanical Engineering, TU/e. 2018-12

T.A.E. Ophelders. *Continuous Similarity Measures for Curves and Surfaces.* Faculty of Mathematics and Computer Science, TU/e. 2018-13

M. Talebi. *Scalable Performance Analysis of Wireless Sensor Network*. Faculty of Mathematics and Computer Science, TU/e. 2018-14

R. Kumar. *Truth or Dare: Quantitative security analysis using attack trees*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-15

M.M. Beller. *An Empirical Evaluation of Feedback-Driven Software Development*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2018-16