Empirically-Grounded Construction of Bug Prediction and Detection Tools

Inauguraldissertation
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Haidar Osman
von Syrien

Leiter der Arbeit:
Prof. Dr. Oscar Nierstrasz
Institut für Informatik

Empirically-Grounded Construction of Bug Prediction and Detection Tools

Inauguraldissertation
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Haidar Osman
von Syrien

Leiter der Arbeit:
Prof. Dr. Oscar Nierstrasz
Institut für Informatik

Von der Philosophisch-naturwissenschaftlichen Fakultät
angenommen.

Bern, 11.12.2017 Der Dekan:

Prof. Dr. Gilberto Colangelo

Acknowledgments

Doing a PhD at SCG has been a beautiful journey full of rewards. The doctorate title is only one of them.

First and foremost, I would like to express my deep gratitude to Oscar. From academic guidance and scientific discussions to personal counselling and after-coffee fun puzzles, Oscar has offered everything that made my PhD journey that enjoyable. And on the personal level, being away from home, I always felt safe knowing that Oscar is there. Oscar ... thank you.

I would like to thank Martin Pinzger for accepting to serve on the PhD committee and for his valuable feedback on this thesis. I also thank Paolo Favaro for chairing the PhD defence.

A big thank you to the SCG family. I am grateful for each and everyone of you. You really made SCG feel like home. I highly value our discussions, fights, papers, darts, swims, barbecues, beers, and coffees. They will always be great memories. I thank Mircea for his lifting spirit and valuable input, Andrei for being a supportive friend, Boris for the great debates and fun, Andrea for the manly gossip, Jan for your delightfully-different character, Nevena for breaking the masculine atmosphere, Claudio for the heated discussions, Manuel for the fruitful collaborations, Yuriy for the context switching, Oli for the deep technical knowledge, Mohammad for bringing a different academic style, and Leo for the fun experiments. You guys are wonderful people and I'm honoured to have served my term with you.

I am grateful to Iris for never saying no when I needed help. I am also indebted to my students: Manuel, Andi, Ensar, Jakob, Sebastien, Simon, Aliya, Cedric, and Mathias. Your hard work has contributed a lot to my PhD.

I would like to thank my friends and brothers back home for their support. Yazan, Montajab, Alaa, Sami, Mothanna, Monther, Safwan, and Rami ... thank you for always being there for me. My in-laws Ahmad and Samira, thank you for your prayers and warm wishes.

I dedicate this work to my family ...

> to my parents Mohsen and Ghada, and to my sister Riham for their unconditional love and support, for believing in me no matter what, and for setting me on the right path throughout my life ...
>
> to my kids Rima and Yosef ... You are the joy of my life ... Your faces bring light to my darkest nights ... I hope you grow up to be proud of your father one day ...
>
> and finally, to Dima ... my friend, my love, my all. A wise man once said, "Give me one constant and I can build the universe". You are that one constant in my life. Without you nothing is possible, nothing makes sense, and nothing matters. Dima ... this is to you.

Abstract

There is an increasing demand on high-quality software as software bugs have an economic impact not only on software projects, but also on national economies in general. Software quality is achieved via the main quality assurance activities of testing and code reviewing. However, these activities are expensive, thus they need to be carried out efficiently.

Auxiliary software quality tools such as bug detection and bug prediction tools help developers focus their testing and reviewing activities on the parts of software that more likely contain bugs. However, these tools are far from adoption as mainstream development tools. Previous research points to their inability to adapt to the peculiarities of projects and their high rate of false positives as the main obstacles of their adoption.

We propose empirically-grounded analysis to improve the adaptability and efficiency of bug detection and prediction tools. For a bug detector to be efficient, it needs to detect bugs that are conspicuous, frequent, and specific to a software project. We empirically show that the null-related bugs fulfill these criteria and are worth building detectors for. We analyze the null dereferencing problem and find that its root cause lies in methods that return null. We propose an empirical solution to this problem that depends on the wisdom of the crowd. For each API method, we extract the nullability measure that expresses how often the return value of this method is checked against null in the ecosystem of the API. We use nullability to annotate API methods with nullness annotation and warn developers about missing and excessive null checks.

For a bug predictor to be efficient, it needs to be optimized as both a machine learning model *and* a software quality tool. We empirically show how feature selection and hyperparameter optimizations improve prediction accuracy. Then we optimize bug prediction to locate the maximum number of bugs in the minimum amount of code by finding the most cost-effective combination of bug prediction configurations, *i.e.*, dependent variables, machine learning model, and response variable. We show that using both source code and change metrics as dependent variables, applying feature selection on them, then using an optimized Random Forest to predict the number of bugs results in the most cost-effective bug predictor.

Throughout this thesis, we show how empirically-grounded analysis helps us achieve efficient bug prediction and detection tools and adapt them to the characteristics of each software project.

Contents

1 Introduction

Delivering high-quality software requires extensive software quality assurance (SQA) activities. These activities ensure that the product conforms to requirements and satisfies user needs. The problem is that SQA activities such as testing and code reviewing are expensive. With finite budgets of software projects, a thorough and complete verification and validation is impossible, especially with large, complex, and rapidly evolving modern software systems. To linearly increase the confidence that all bugs have been found, SQA costs grow exponentially [118, 133]. This exponential relationship between cost and quality rapidly exhausts project budgets.

The manual nature of reviewing code and writing tests is the main source of SQA costs. This fact drives researchers to devise techniques and develop tools that help developers focus their reviewing and testing activities to catch more bugs with fewer resources. Bug prediction and bug detection are the main tools to achieve this task. Bug prediction hints approximate locations of bugs in software entities (*e.g.*, packages, classes, methods) using a machine learning approach. Bug detection on the other hand hints exact location of bugs (*i.e.*, line of code) using source code analysis. When *designed carefully*, bug prediction and detection tools can largely reduce the costs of software testing and code reviewing and help developers catch bugs before they slip into production.

1.1 Auxiliary Software Quality Assurance Tools

Software bugs vary in their costs and severity. However, it is an established fact in software engineering that bugs that are detected early are cheaper to fix. When bugs slip into advanced phases in development, they become more expensive to fix. Even worse, bugs that make it to production can have disastrous consequences on the reputation, cost, and overall success of software projects. Therefore, developers perform software testing and code reviewing to detect and fix bugs as early as possible and prevent them from reaching production code. Software testing is an important and widely adopted technique to detect bugs. Modern development methodologies (*e.g.*, Scrum [173, 182], XP [14], TDD [7]) acknowledge software testing as an essential activity for improving software quality. Code reviewing is also widely adopted and has several benefits such as bug detection, code improvement, and knowledge transfer. It has been shown that code reviewing, in its traditional rigorous form (*i.e.*, code inspection [51]) and in its modern lightweight tool-oriented form [10], is very effective in spotting defects in the code [2, 127].

However software testing and code reviewing are costly activities. Software testing has a non-negligible overhead on the development time (approx. 30%) [210] and code reviewing is by design a resource-intensive activity because it highly involves people, making it the longest phase of code integration. On average, a developer spends six hours a week on reviewing other developers' code [24]. These immense costs, along with the limited resources available for software projects, call for tools that help developers optimize testing and reviewing so that more bugs are caught with fewer resources. We call these tools: *auxiliary software quality assurance (ASQA) tools* to emphasize the fact that they only complement the main methods of testing and reviewing and do not replace them. There are two types of tools available for this task: bug prediction and bug detection tools.

A bug predictor is an intelligent system (model) trained on data derived from software (metrics) to make a prediction about bugs (*e.g.*, number of bugs, bug proneness) in software entities (*e.g.*, packages, classes, files, methods). Bug prediction helps developers focus their quality assurance efforts on the parts of the system that are more likely to contain bugs. Bug prediction takes advantage of the fact that bugs are not evenly distributed across the system but they rather tend to cluster [185]. The distribution of bugs follows the Pareto principle, *i.e.*, 80% of the bugs are located in 20% of the files [160].

The other family of ASQA tools is Bug detection tools (*e.g.*, FindBugs [85], PMD [166], and Jlint [91]). These tools employ static analysis to detect generic problems in the source code and provide developers with explanations of these problems and how to fix them. These tools harvest prior knowledge of generic bugs related to the target language built-in libraries (*e.g.*, objects of the class `java.io.InputStream` should be closed some time after being initiated and used). Other static analysis approaches focus on one family of problems, such as the null dereferencing bug [117, 120, 147].

Both bug prediction and bug detection tools have the potential to cut down testing and code reviewing costs, but they are still not widely adopted in industry. The main reasons lie in their low efficiency and adaptability. Efficiency means that these tools should help software engineers focus their SQA efforts on those software parts that most likely contain bugs. Adaptability means that these tools should be custom-tailored to the peculiarities of individual projects because generic tools produce sub-optimal results. Researchers seem to ignore these two concepts in their approaches.

For instance, current bug prediction research focuses on optimizing bug prediction from the machine learning perspective and seems to overlook optimizing it as a software quality tool, that is, it must be optimized to locate the maximum number of bugs in the minimum amount of code [129, 130, 6]. Also it has been shown that although developers acknowledge the potential benefits of static analysis tools, they are still hesitant to adopt them because of the high number of false positives and poor adaptability [92]. In summary, bug prediction and detection tools need to be tuned for the projects they operate on and optimized as quality tools that help developers in their software quality assurance activities.

Problem Statement
The low efficiency and adaptability in current auxiliary software quality assurance tools prevent them from meeting their goal of enhancing the main SQA activities of testing and code reviewing.

1.2 Our Approach: Empirically-Grounded Analysis

We propose empirically-grounded analysis (EGA) as an approach to solve the efficiency and adaptability issues in ASQA tools. In general, EGA is the method of establishing relevant research questions using empirical case studies, action research methods, or interviews. In this thesis, we use EGA to build efficient and adaptable bug prediction and detection tools. We formally state our thesis as follows:

Thesis Statement
In order for bug prediction and detection tools to be useful in practice, they need to be adapted to the different characteristics of different projects, and optimized to locate the maximum number of bugs in the minimum amount of analyzed code. Empirically-Grounded Analysis (EGA) provides a unified framework for achieving these goals.

1.2.1 Adaptability

There are many factors that vary among software projects: domain, development method, architecture, libraries and frameworks, *etc.*. These factors play an important role in how, where, and what bugs are introduced in the software. Bug prediction and detection tools should be adapted to the particularity of each software system.

In current bug detection tools, a bug rule in one project may not apply in another. The generic design of these rules limits their adaptability to the project-specific bugs. Although current bug detection tools are indeed useful, more customized and empirically-grounded methods are needed to detect bugs that are specific to the software at hand.

Bug predictors adapt to software systems using software-derived metrics and then predict any type of bugs within these systems. However, a bug predictor has more design choices than just the used metrics. More particularly designing a bug predictor means making decisions regarding the used metrics, the machine learning model, and the response variable. These *bug prediction configuration* options also need to be adapted. A bug prediction configuration that works with one system may not work with another because the aforementioned differentiating factors affect the correlation between different metrics and software defects. In fact, it has been shown previously that a model trained on data from a specific project does not perform well on another project and so called cross-project defect prediction rarely works [216].

1.2.2 Efficiency

Auxiliary software quality assurance tools exist to help developers carry out the main SQA activities more efficiently. This necessitates that these tools be efficient, that is, they be optimized to help developers locate as many bugs as possible with as few resources as possible.

The generic bug rules and the nature of static analysis of bug detection tools lead to the false positives problem. Indeed, the high number of false positives in these tools hinders their adoption in software projects [21, 92]. Besides, it has been shown that domain-specific rules are better than generic rules at bug prevention [84]. Although bug detection rules are efficient in the sense that they pinpoint exact locations of potential bugs, false positives decrease this efficiency as developers need to examine many locations that do not actually contain bugs. Another problem is that static analysis itself is expensive. Rules that rarely apply or do not represent serious bugs slow down bug detection tools and decrease their appeal to developers. Bug rules should be carefully designed to catch bugs with conspicuous severity and adequate recurrence.

The main promise of bug prediction is to help software engineers focus their testing and reviewing efforts on those software parts that most likely contain bugs. Under this promise, for a bug predictor to be useful in practice, it must be *efficient*, that is, it must be optimized to locate the maximum number of bugs in the minimum amount of code [129, 130, 6]. A bug predictor should be tuned and evaluated to fulfill this promise.

1.2.3 EGA to the Rescue

In this thesis, we demonstrate how to improve the adaptability and efficiency of ASQA tools by using EGA in requirement collection, problem analysis, and tool crafting.

Requirement collection in this context means the discovery of bugs that are worth building tools for. There are various kinds of software bugs but not all of them can be detected or are recurrent enough to automate their detection. We propose a scalable analysis of bug-fix code changes to discover bug patterns and their prevalence. We empirically show that missing null checks and missing method calls are the most frequently present categories of bugs in Java projects.

Problem analysis is a vital phase in the development of efficient ASQA tools. It leads to a better understanding of why the problem exists and how it can be possibly solved. We analyze the missing null check problem by carrying out an empirically-grounded analysis of null checks themselves. We track null checks in a plethora of Java systems to understand where null values come from and how they are used. We discover that methods that return null are the main source of null-related bugs. This means that a tool that helps developers decide whether to check the returned value of a method against null or not, would cover a wide spectrum of the missing null check problem.

Tool crafting is concerned with the approaches and design decisions behind ASQA tools. Bug prediction and detection tools should be adapted to the particularity of each software system and should be optimized as quality tools. We demonstrate how EGA helps us build such tools in two use cases: an empirical approach for detecting missing null checks and an empirically-grounded optimization of bug prediction.

1.3 Contributions in Detail

1.3.1 Building an Efficient Null Dereference Detection Approach

We empirically discover and analyze the null dereferencing problem, and then design an empirical approach to detect it in source code.

First, we set out to discover recurrent bug patterns in Java systems. Analyzing bug-fix code changes in 810 open-source Java projects reveals many interesting bug patterns, of which missing null check is the most common [158]. This means that designing a tool for detecting missing null checks would benefit Java developers in most projects.

Second, to be able to propose a solution, we analyze the null checks in 810 open-source Java systems to understand when and why developers use null [157]. We find that 35% of all conditional statements contain null checks. A deeper investigation reveals many questionable practices with respect to using null. Uninitialized member variables and passing null as a method parameter are recurrent reasons for introducing null checks, but returning null in methods is by far the most dominant as 71% of the values checked for null are returned from method calls.

Third, we employ the acquired knowledge to devise an empirical approach that detects missing null checks on API method calls. We build a prototype Eclipse plugin that detects, with certain confidence, missing null checks in clients of Apache Lucene. We collect and analyze a large number of Lucene clients and track how they handle the returned values from Lucene API calls, *i.e.*, how often they are checked to be non-null. The results are then aggregated per library method in a *nullability* measure defined as follows:

$$Nullability(Method) = \frac{CheckedDereferences(Method)}{Dereferences(Method)}$$

Method nullness denotes whether a method might return null (nullable) or never returns null (non-null). The nullability measure serves as a proxy for method nullness and expresses the confidence that a particular method returns null. A nullability of zero indicates that a method never returns null, *i.e.*, the returned value is always dereferenced in clients without a null check. The method's nullness is therefore non-null. Conversely, a non-zero nullability indicates that a method is nullable. The Eclipse IDE plugin then adds nullability information, along with confidence and support, to the method's documentation. It also adds a nullness annotation to the return type of the method to assist null analysis tools. Developers can customize the nullness warnings by setting confidence and support thresholds in the plugin settings. The nullability documentation enables developers to make more informed decisions about adding or removing a null check. The null analysis points developers at locations where a null check is unnecessary or potentially missing.

This empirical bug detection approach is designed with efficiency in mind from requirement collection (*i.e.*, bug pattern discovery) and problem analysis, till output design (*i.e.*, weighted warnings) and control (*i.e.*, warning thresholds). Also this approach fulfils the adaptability requirement by reporting bugs that are directly related to the external APIs used in a software project and allowing developers to configure the plugin so that it only reports warnings with certain confidence and support.

This research is published in two full technical research papers [157, 112], a tool demo paper [113], and an early research achievement paper [158].

1.3.2 Efficient Bug Prediction

Bug prediction employs machine learning to predict bugs in software entities. Thus building an efficient bug predictor requires optimizations from two point of views: machine learning and software quality.

Optimizing Bug Prediction as a Machine Learning Model

The quality of the trained model is directly dependent on the quality of the features. Irrelevant and correlated features degrade the performance of prediction models. The more features are fed into a model, the more complex the model is, and the less accurate the model becomes. Feature selection is the process of selecting relevant features for model training. Feature selection reduces model complexity by eliminating noise and correlated features to reduce the generalization error. There are two main types of feature selection: filters and wrappers. Filters apply statistical measures to give scores to features independently of the machine learning model. The features are then ranked based on the score and a subset of the most relevant features is selected based on a certain score threshold. Example filters are Correlation-based Feature Selection (CFS), Information Gain (InfoGain), and Principal Component Analysis (PCA). Wrappers choose a feature subset that gives the best performance of a certain machine learning model. They are called wrappers because the learning algorithm is wrapped into the selection procedure. They try different subsets and choose the one that gives the best accuracy of the machine learning model at hand. We empirically investigate the impact of these techniques on building bug predictors. We show that for some machine learning models, it is essential to perform feature selection while other models are robust against noise and redundancy in the feature set. In general, wrapper methods consistently reduce prediction error while filters are unstable and can actually hinder the accuracy of a bug predictor. We also observe that the same feature selection method chooses different features in different projects. This means that applying feature selection not only improves the accuracy of a bug predictor, but also adapt it to the software project at hand. These findings are submitted as a journal article and have passed the first reviewing phase [155].

Machine learning models often have what is known as Hyperparameters. These are the parameters that need to be set before training the model and affect its learning, construction, and evaluation. Example hyperparameters are the complexity parameter in support vector machines and the number of neurons in the hidden layer in a feed-forward neural network. Different machine learning problems have different characteristics and the hyperparameters need to be tuned accordingly. Using a standard hill-climbing grid search algorithm, we investigate the effect of hyperparameter optimization of a model on its prediction accuracy. Our results reveal that tuning model hyperparameters has a statistically significant positive effect on the prediction accuracy of the models, suggesting that adapting hyperparameters to each software project is a necessary step in building an accurate bug predictor for that project. These results are published in a workshop paper [154].

Optimizing Bug Prediction as a Software Quality Tool

Hyperparameter optimization and feature selection are optimization techniques from the machine learning field. However a bug predictor is more than just a machine learning model. It should help developers steer maintenance activities towards the buggy parts of a software, by locating the highest number of bugs in the least amount of code. Hence, it should be optimized for that purpose.

There are many design aspects to a bug predictor, each of which has several options, *i.e.*, software metrics, machine learning model, and response variable. These design decisions should be judiciously made because an improper choice in any of them might lead to wrong, misleading, or even useless results. We argue that bug prediction *configurations* are intertwined and thus need to be evaluated in their entirety, in contrast to the common practice in the field where each aspect is investigated in isolation. We evaluate 60 different bug prediction configuration combinations on five open source Java projects. Interestingly, source code metrics alone are not cost-effective whereas change metrics extracted from the evolution of the system can be used alone to build efficient bug predictors. We find out that the best choices for building a cost-effective bug predictor are change metrics mixed with source code metrics as independent variables, Random Forest as the machine learning model, and the number of bugs as the response variable. Combining these configuration options results in the most efficient bug predictor across all subject systems. This study is published in full technical research paper [156].

1.4 Outline

This dissertation is organized as follows:

In **chapter 2**, we discuss the state of the art in the domains directly related to this thesis. We focus on bug prediction approaches, frequent bug patterns studies, and solutions to the most frequent bugs.

In **chapter 3**, we present our approach to mining bug-fix code changes and extracting patterns from them. In this chapter, we show how missing null checks are the most frequent and are worth building detection tools for.

In **chapter 4**, we analyze null usage in Java systems to learn about the root causes of null-related problems. Our findings in this chapter helps designing an efficient detection tool for the missing null bug pattern.

In **chapter 5**, we design and implement an empirical solution to the missing null check problem.

In **chapter 6**, we investigate the effect of applying filter and wrapper feature selection on bug prediction.

In **chapter 7**, we investigate the effect of hyperparameter optimization on predicting the number of bugs.

In **chapter 8**, we optimize bug prediction as a software quality tool by empirically choosing the most efficient bug prediction configurations.

In **chapter 9**, we conclude this thesis and outline future research directions.

In **Appendix A**, we present KOWALSKI, a tool that can collect clients of a specific Java API to facilitate API usage mining research.

In **Appendix B**, we present BICO, a tool that can be used to analyze the evolution of software projects. It links source code repositories with issue trackers, detects multi-purpose and miscategorized commits, and calculates source code and change metrics of projects.

2 State of the Art

There is a large body of research in the domain of auxiliary software quality assurance tools such as bug prediction and detection tools. In this chapter, we survey the state of the art only in the fields that are directly related to our work. First, we review existing approaches in exploring bug-fix code changes to discover frequent bug categories. Second, we survey existing solutions for detecting the two most prevalent bug categories: method invocation bugs and missing null check bugs. This is directly related to our missing null check detector as it combines API mining and static analysis approaches behind the scenes. Finally, we review the related work in bug prediction and how researchers approached its optimization, highlighting the research gaps in the field.

2.1 Detection Tools for Frequent Bug Categories

In this section, we focus on the empirically-grounded analysis of bug patterns and their detection approaches. We demonstrate how prevalent categories of software bugs are found, what they are, and how each category is approached by researchers.

2.1.1 Discovering Prevalent Bugs

Different bug pattern extraction mechanisms lead to different patterns and statistics, shedding light on different aspects of software maintenance activities. We share the same vision as the other studies on bug and fix patterns: automating bug detection and fixing as much as possible. This is an interesting topic and has caught the attention of researchers especially in the past decade.

Thung *et al.* [195] manually inspected 500 randomly-selected bugs in three machine learning and NLP processing systems (Apache Mahout, Apache Lucene, and Apache OpenNLP) and grouped them into high-level categories. They found out that bugs related to algorithms have the highest proportion (22.6%). The results are expected due to the algorithm-intensive nature of the domain, and thus, cannot be generalized. Pan *et al.* [161] manually detected some change patterns that correspond to fixes and automatically extracted instances of these patterns from seven open source projects. Martinez *et al.* [125] came up with a new approach for specifying and finding change patterns based on AST differencing. They evaluated their approach by counting 18 change patterns from 14 project repositories. Both studies report that changes to method invocations and `if`-statements are by far the most prevalent bug-fix patterns. However, due to their use of different pattern extraction mechanisms from ours, neither study reported the missing null-check bug-fix pattern. In our study on bug-fix patterns, presented in chapter 3, we not only confirm previous findings that bugs related to method invocations (*i.e.*, missing and undue invocations) are highly recurrent, but also reveal that missing null checks are the most common bugs in Java systems.

2.1.2 Method Invocation Bug Detection

Studies that detect bugs by mining API usage are based on the assumption that the majority is right. In our missing null check detector, we follow the same intuition that bugs are anomalies in the source code [48, 114, 207]. However, none of the previous studies considers the nullness of API methods as a usage pattern.

Li and Zhou [114] propose a tool called PR-Miner that uses frequent itemset mining to find program element correlations. These correlations are called programming rules and their violations are treated as possible bugs. PR-Miner considers method bodies as transactions, and program elements (*e.g.*, method calls) that appear in it as items. Livshits and Zimmermann developed a tool called DynaMine [116] that also employs frequent itemset mining to find recurrent patterns of application-specific method invocations. Based on the bias from these patterns, DynaMine can suggest bug locations for bugs related to method invocation. Dynamine is different from PR-Miner in that it considers code revisions as transactions and inserted method calls as items. Monperrus *et al.* adapt the same approach to work specifically on object-oriented programs [142]. They propose a missing method call detection system (DMMC) that operates based on the idea of code deviants in type usage. Type usage is “a list of method calls on a variable of a given type occurring in the body of a particular calling method” [142]. DMMC collects type usage of all types in a system then detects deviant type usages as missing call bugs. Nguyen *et al.* use graph-based mining to detect object usage patterns in the form of code skeletons [148]. Then they use graph-based anomaly detection to find locations in the source code that deviate from the patterns, and possibly represent bugs. On the other hand, Wang *et al.* apply n-gram language modeling to detect correlations between program elements [205]. Wang *et al.* implement their approach in a tool called Bugram and demonstrate that it detects bugs that cannot be detected by other state-of-the-art tools that use frequent itemset mining.

These previous approaches are applicable only on large systems in order for them to extract rules and detect violations. This is because rules are extracted from the same systems they are applied on. Other approaches take advantage of the ecosystems around libraries and frameworks to extract rules. Zhang *et al.* developed a tool called *Precise* that suggests method parameters of a certain API by mining its usage in other client projects [214]. Zhong *et al.* [215] mine open source repositories to extract sequential rules of API method calls. Salman [178] and Saied [177] use similar techniques to extract multi-level frequent usage API patterns from API clients. These patterns can be used for API element recommendation, automatic documentation, and bug detection. In our null check detector, we also mine the ecosystems of library clients to be able to deduce method nullness rules that can be applied to other clients of any size.

2.1.3 Null Dereferencing Bug Detection

Null-related bugs attract the attention of many researchers and practitioners who propose various approaches to detect them as early as possible.

The first family of solutions incorporates data flow analysis techniques to detect possible null values. Some techniques are simple, fast, and intra-procedural [29, 43, 48, 56, 86] and some are more complex, thorough, and inter-procedural [117, 196, 147, 46]. For instance, Hovemeyer and Pugh [86] perform intra-procedural forward data flow analysis to approximate the static single assignment (SSA) for the values of variables. Then they analyze the dereferences as a backward data flow over the SSA approximation. This algorithm replaces the previous basic forward data flow analysis approach [87] and is now part of FindBugs [85], a static analysis tool for finding bugs in Java. However, their solution cannot find possible null dereferences of values returned from method calls, as this is beyond the scope of an intra-procedural analysis, unless the analysis is aware that the called method is annotated as returning null. These situations arise often, as values returned from methods are the most often checked for null category [157]. The nullness annotations our plugin generates are exactly those missing links between a project and its dependencies that allow an intra-procedural null analysis to reason about values received by calling a method.

Nanda and Sinha present XYLEM, an unsound inter-procedural, path- and context-sensitive data-flow analysis [147]. Contrary to Tomb et. al. [196] they report a 16 times higher number of possible null dereferences in inter-procedural than in intra-procedural analysis, yet they claim a low false positive rate. However, Ayewah and Pugh [9] compare several null dereference analysis tools, including XYLEM, and observe that, besides the high number of reported false positives, many of the reported null dereferences (true positives) do not manifest themselves as bugs at run time. The authors claim that when the null dereference passes the initial software testing, it rarely causes bugs and "reviewing all potential null dereferences is often not as important as many other undone software quality tasks"[9]. Ayewah and Pugh argue that no warning should be issued for dereferences that are unchanged over many versions of stable software. We take this argument into consideration in the design of our missing null check detector. If our analysis finds the same unconditionally dereferenced method return value in many versions, it decreases the originating method's nullability, which reflects that the method is rarely causing problems.

The second family of solutions proposes to annotate the "nullness" in code. Researchers propose to avoid null dereferences by extending the type system with `@Nullable` and `@NonNull` annotations [52, 164, 163, 42]. Loginov *et al.* [117], beside their inter-procedural null dereference analysis, propose null-related annotations to ensure the soundness and safety of the analysis.

The idea of annotations made it to widely-used Java libraries like *Checker Framework*[1] and *Guava*[2]. Also some programming languages introduce the idea of reference declarations that are not null. For instance the Spec# programming system extends the C# programming language with the support of contracts (like non-null types), allowing the Spec# compiler to statically enforce these contracts [12].

With annotation, method, field, variable and collection item declarations can be annotated, so that a static checker can verify that no nullable value is ever dereferenced and the nullness matches for parameters passed to methods. However, if annotations for library methods are missing, either they need to be added manually or tools need to make naïve assumptions about them. For instance, in their analysis, Flanagan *et al.* assumes that all library methods never return null [55]. Our approach gets rid of the required manual work to add the nullness annotations to existing projects, as the annotations are automatically generated from usage.

The third family of solutions tries to solve the problem by introducing language constructs. Haskell [76] and Scala [149] have the "Maybe" and the "Option" types, which are object containers. In a similar fashion, Oracle introduced the "Optional" type in Java 8 recently [201]. Groovy and C# have the safe navigation "?." to safely invoke a method on a possibly-null object.

In our approach of missing null check detection, we combine API usage mining with static analysis. First, we collect clients of a certain API. Second, we statically analyze the source code to detect method calls to that API and whether their returned values are checked against null or not. This step builds up the nullability measure for each method in the API. Finally, in the system we want to apply the detection on, we insert nullness annotations to the API methods, based on the nullability measure and rely on Eclipse static analysis to produce the necessary null-related warnings. We also insert documentation about the confidence and support of the nullness so that developers take informed decisions when they deal with these warnings.

2.2 Bug Prediction Tools

In the following subsections, we review the different aspects of building a bug predictor and highlight our contributions.

[1]`http://types.cs.washington.edu/checker-framework/`
[2]`https://github.com/google/guava`

2.2.1 Software Metrics

There are three types of features used to predict bugs: software metrics extracted from the source code, the versioning system, or the organizational structure.

Many researchers argue that source code metrics are good predictors for future defects [80, 162, 13, 27, 192, 26]. Source code metrics try to capture the quality (*e.g.*, lack of cohesion in methods, coupling between objects) and complexity (*e.g.*, weighted method count, depth of inheritance tree) of the source code itself. The rationale behind using the source code metrics as bug predictors is that there should be a strong relation between source code features (quality and complexity) and software defects [191]. In other words, the more complex a software entity is, the more likely it contains bugs. Also the poorer the software design is, the more bug-prone it is.

A second class of studies considers metrics extracted from the software versioning systems like CVS, Subversion, and Git. These studies establish a relationship between how and when software entities (binaries, modules, classes, methods) change and evolve over time on one hand, and the fault-proneness of these entities on the other hand. The version history metrics (*i.e.*, change metrics) describe software entities with respect to their age [176, 20], past faults [104, 216, 159], past modifications and fixes [69, 159, 79, 77, 102, 143], and developer information [208, 209, 165, 132]. Using software history metrics as bug predictors is motivated by the following heuristics:

1. Entities that change more frequently tend to have more bugs.
2. Entities with a larger numbers of bugs in the past tend to have more bugs in the future.
3. Entities that have been changed by new developers tend to have more bugs.
4. Entities that have been changed by many developers tend to have more bugs.
5. Bug-fixing activities tend to introduce new bugs.
6. The older an entity, the more stable it is.

Other researchers work on metrics extracted from actual changes in source code from one version to another [144, 38, 66]. These changes are called *"Code Churns"*. As explained by Nagappan and Ball [144], *"Code churn is a measure of the amount of code change taking place within a software unit over time."*

Finally some researchers argue that the organizational structure and volatility of the software teams have an impact on the software quality [146, 140]. These studies show that organizational metrics have positive, strong, and statistically significant correlation with bug-proneness.

Many studies show that change metrics are better than source code metrics at predicting bugs [143, 94] [69, 102, 209, 159, 65]. Moreover, Arisholm *et al.* states that models based on object-oriented metrics are no better than a model based on random classification [6]. In chapter 8, we confirm the superiority of change metrics over source code metrics, but also show that mixing both produce the best results, contradicting previous findings [6].

2.2.2 Feature Selection in Bug Prediction

In a recent systematic literature review of the machine learning techniques in the bug prediction field [121], surprisingly, most studies (60%) do not apply feature selection at all. Our investigation shows that researchers in bug prediction often undermine the importance of feature selection.

Among the studies that apply feature selection, correlation based feature selection (CFS) [73] is the most commonly used [121]. Many studies employ the CFS filter technique before training machine learning models [47, 97, 6] [39, 123, 216, 200, 151]. We also apply CFS as the baseline filter technique and compare it to wrapper techniques to show that wrappers outperform this filter in most cases.

Most studies that apply feature selection actually apply filter techniques like principal component analysis (PCA) [101, 38, 145, 96, 189], consistency based selection (CBS) [216], and InfoGain [134, 198, 188, 139]. Very few studies apply wrapper feature selection techniques [30, 68] when classifying software entities as buggy or clean (classification). However, to the best of our knowledge, there is no study that applies wrapper techniques when predicting the number of bugs in software entities (regression). In chapter 6 we compare the CFS filter with different wrappers while treating bug prediction as a regression problem.

Shivaji *et al.* [187] study the impact of multiple feature selection techniques (filters and wrappers) on the performance of Naïve Bayes and Support Vector Machines when classifying code changes as buggy or clean. They report a significant enhancement in the accuracy measures of both classifiers when feature selection is applied. However, Shivaji *et al.* consider *"everything in the source code separated by whitespace or a semicolon"* as a feature [187]. This includes variable names, method names, keywords, comment words, *etc.*. They end up with a staggering number of features ranging from 6,127 to 41,942 features. The vast majority of features in the initial feature set are bound to be irrelevant, hence the results of their study are exaggerated and cannot be generalized.

Challagulla *et al.* [35] report that performing principal component analysis before training the models does not result in a significant improvement on the performance of the bug predictors, while correlation-based feature selection (CFS) and consistency-based subset evaluation (CBS) actually decrease the prediction error of the machine learning models. We actually report different results when applying CFS to regressors.

Khoshgoftaar *et al.* [100] combine filter feature selection and data sampling to achieve better classification accuracy on the package level. They report an improvement of around 2% to the area under the receiver operator characteristic curve (AUC-ROC) when the filters are applied to sampled data rather than the whole data. Then Khoshgoftaar *et al.* [99] improve the performance by repeatedly applying the sampling and filtering several times then aggregating the results.

Gao *et al.* [61] studied seven filter feature selection techniques. They report that the studied classification models were either improved or remained unchanged while 85% of the original features were eliminated. Krishnan *et al.* [107] analyze whether change metrics remain good predictors during the evolution of Eclipse. In their analysis, they use J48 decision tree as the machine learning algorithm and refer to the top five features as the set of prominent predictors, then study the consistency of this set over the consecutive versions of Eclipse. They report that there is a small subset of change metrics that is consistently good at classifying software entities as buggy or clean across products and revisions. Wang *et al.* [204] study the effect of removing redundant or useless features from the PROMISE dataset[3]. They report that feature selection improves classification accuracy. Catal and Diri [33] explore which machine learning algorithm performs best before and after applying feature reduction. In these studies, researchers treat bug prediction as a classification problem while we study the effect of feature selection on bug prediction as a regression problem.

Turhan and Bener [199] have another opinion on the matter of feature selection. They argue that Naïve Bayes assumes the independence and the equal importance of features. These assumptions are not true in the context of bug prediction. Yet, Naïve Bayes is one of the best classifiers [134]. Turhan and Bener [199] empirically show that the independence assumption is not harmful for defect prediction using Naïve Bayes and assigning weights to features increases the performance and removes the need for feature selection. Turhan and Bener conclude that *"either weighted Naive Bayes or pre-processing data with PCA may produce better results to locate software bugs in source code"*[199].

We confirm that most of the existing literature treats bug prediction as a classification problem, and that studying the number of bugs is neglected in the field. For instance, Ostrand *et al.* [159] use negative binomial regression (NBR) to predict the number of bugs in software modules. They report that NBR fits the bug prediction problem and demonstrates high accuracy. Janes *et al.* [89] compare three count models to predict the number of bugs and find out that zero-inflated NBR performs better than Poisson regression and NBR. Rathore and Kumar [172] investigate six different fault prediction models for predicting the number of bugs and show that count models (*i.e.*, NBR and Zero-Inflated Poisson regression) underperform compared to linear regression, decision tree regression, genetic programming and multilayer perceptron. Graves *et al.* [69] build a generalized linear regression model to predict the number of bugs based on the various change metrics. Gao and Khoshgoftaar [60] compare the performance of several count models (*e.g.*, Poisson regression) and show that Hurdle Poisson regression has the most accurate predictions. Nevertheless, these studies do not apply feature selection and their results should be reassessed in the light of our findings.

[3]`http://openscience.us/repo/`

2.2.3 Prediction Models

Most studies comparing different machine learning models in bug prediction show no difference in performance [40, 111] [203, 135]. Menzies *et al.* [135] evaluate many models using the area under the curve of a probability of false alarm versus probability of detection "AUC(pd, pf)". They conclude that better prediction models do not yield better results. Similarly, Vandecruys *et al.* [203] compare the accuracy, specificity (true negative rate), and sensitivity (recall or true positive rate) between seven classifiers. Using the non-parametric Friedman test, as recommended in this type of problem [40], it is shown that there is no statistically significant difference at the 5% significance level. Lessmann *et al.* [111] study 22 classifiers and conclude that the classification model is less important than generally assumed, giving researchers more freedom in choosing models when building bug predictors. Actually simple models like naïve Bayes or C4.5 classifiers perform as well as more complicated models [44, 134].

Other studies suggest that there are certain models which perform better than others in predicting bugs. Elish and Elish [47] compare SVM against 8 machine learning and statistical models and show that SVM performs classification generally better. Guo *et al.* [70] compare Random Forest with other classifiers and show how it generally achieves better prediction. Ghotra *et al.* [63] show that there are four statistically distinct groups of classification techniques suggesting that the choice of the classification model has a significant impact on the performance of bug prediction. Our findings confirm the superiority of certain models over others. Specifically, we show that Random Forest is indeed the best machine learning model, followed by Support Vector Machines.

2.2.4 Hyperparameter Optimization

Model hyperparameters are often left set to their default values in the bug prediction literature. Very few studies inspect the effect of hyperparameter settings on the bug prediction accuracy.

Martino *et al.* [41, 179] use a genetic search algorithm to optimize the hyperparameter settings for SVM and compare it with six machine learning models. Their experiments are carried out on jEdit data from the PROMISE repository [181]. They report that the genetically optimized SVM gives the highest *F-measure*. In their study, Martino *et al.* do not optimize the other models making the comparisons unfair.

The response variable of probabilistic models, such as Naiv̈e Beyes, is the probability that a software entity is defective. When these models are used as classifiers, they take the threshold that separates the classes as a hyperparameter. It is usually 0.5 by default. Tosun and Bener [197] report that optimizing this threshold decreases false alarms by up to 11%.

Recently, Tantithamthavorn *et al.* [193] carried out a large study on the effect of hyperparameter optimization on the accuracy of 26 classification techniques in bug prediction. They show that tuning model hyperparameters increases the accuracy measures by up to 40%.

All previous studies treat bug prediction as a classification problem where the response variable is the class of a software entity (*i.e.*, buggy or clean). To the best of our knowledge, our study in chapter 7 is the first study on hyperparameter optimization in bug prediction as a regression problem, where the response variable is the number of bugs.

2.2.5 Experimental Setup

There are many other studies that look into the effect of experimental setup on bug prediction studies. Tantithamthavorn *et al.* show how different evaluation schemes can lead to different outcomes in bug prediction research [194]. In this study we focus on configuring bug predictors to be cost-effective. Thus, we fix the evaluation scheme to the one the reflects our purpose (*i.e.*, *CE*). In a systematic literature review, Hall *et al.* [75] define some criteria that makes a bug prediction paper and its results reproducible. Surprisingly, out of 208 surveyed papers, only 36 were selected. In our study, we adhere to these criteria by extensively describing our empirical setup. Shepperd *et al.* [184] raised concerns about the conflicting results in previous bug prediction studies. They surveyed 42 primary studies. They found out that the choice of the classification technique, the metric family, and the data set have small impact on the variability of the results. The variability comes mainly from the research group conducting the study. Shepperd *et al.* conclude that who is doing the work matters more than what is done, meaning that there is a high level of researcher bias. We agree with the authors that there are many factors that might affect the outcome of a bug prediction study. In fact this is the main motivation behind our study. However, Shepperd *et al.* looked at studies that treat bug prediction as a classification problem, ignoring the fact that the response variable is itself a factor that affects the outcome. In our study we include more factors and emphasize the interplay among them.

2.2.6 Cost-Aware Evaluation

In the literature, bug prediction models are evaluated using the standard measures, *i.e.*, confusion matrix, prediction error, and statistical correlation. However, many researchers argue that bug prediction should be evaluated based on their ability to steer software quality assurance activities efficiently. Mende and Koschke [130] studied the concept of effort-aware defect prediction. They compared models with respect to predicting defect density using only Random Forest trained only on source code metrics. They took the effort into account during the training phase by predicting bug density as a response variable. Although we agree with Mende and Koschke on the importance of building effort-aware prediction models, our results actually advise against using source code metrics and bug density as independent and dependent variables respectively. Canfora *et al.* also consider cost in the training phase [31]. Using genetic algorithms, they trained a multi-objective logistic regressor that, based on developer preferences, is either highly effective, with low cost, or balanced. They treated bug prediction as a classification problem and they used only source code metrics as independent variables. We agree with Canfora *et al.* that bug predictors should be tuned to be cost-effective, but our study shows that source code metrics are rarely a good choice of independent variables, and predicting bug count is more cost-effective than predicting bug proneness.

Kamei *et al.* [94] also evaluated the predictive power of history and source code metrics in an effort-sensitive manner. They used regression model, regression tree, and Random Forest as models. They found that history metrics significantly outperform source code metrics with respect to predictive power when effort is taken into consideration. Our results confirm their findings but also add that the use of both metrics is even more cost-effective.

Arisholm *et al.* [6] studied several prediction models using history and source code metrics in bug prediction. They also dealt with the class imbalance problem and performed effort-aware evaluation. They found that I) history metrics perform better than source code metrics and II) source code metrics are no better than random class selection. Our results confirm their first finding but contradict the second. We show that indeed using source code metrics is less cost-effective than using change metrics or a mix of both. However, using source code metrics with the right options of other configurations is certainly better than random class selection. This contradiction with our findings comes from the fact that the dependent variable in their study is bug proneness and, as opposed to our study, they did not consider other dependent variables.

Jiang *et al.* [90] surveyed many evaluation techniques for bug prediction models and concluded that the system cost characteristics should be taken into account to find the best model for that system. The cost Jiang *et al.* considered is the cost of misclassification because they dealt with the bug prediction problem as a classification problem. In our study, we treat bug prediction as a regression problem (*i.e.*, bug count and bug density) and as a classification problem (bug proneness and entity class) and the cost of the model is the actual cost of maintenance using LOC as a proxy.

In chapter 8, we use the *cost effectiveness* measure proposed by Arisholm *et al.* [6] to evaluate the efficiency of bug predictors as software quality tools.

2.3 Conclusions

In this chapter, we demonstrate existing studies in bug pattern discovery and what researchers propose to detect the most prevalent ones. In particular, we show that detecting bugs in method invocation is mainly based on API usage mining, while null-related bugs are detected be means of static analysis. Method nullness has not been considered as an API feature in any of the previous studies. In this thesis, we combine API usage mining with static analysis to build an efficient missing null check detector.

In this chapter, we also review the state of the art in bug prediction and show that building a bug predictor requires making decisions regarding independent variables, machine learning models, and response variables. We call this triple *bug prediction configurations*. From our review, we see that there are four main research gaps in the bug prediction literature:

1. Although bug prediction configurations are interconnected, studies in the literature focus on each one of them in isolation.
2. There have been many comparative studies in bug prediction, but there is no study comparing different response variables.
3. Useful machine learning techniques such as feature selection and hyperparameter optimization are rarely applied.
4. Bug predictors are evaluated and optimized mainly as machine learning models instead of software quality tools

In this thesis, we try to fill these gaps by empirically optimizing bug predictors as machine learning tools (chapter 6 & chapter 7) *and* as software quality tools (chapter 8).

3 Discovering the Missing Null Check Bug Pattern

The first step to building an efficient bug detector is to understand where, how, and when software bugs arise in the code and how programmers usually fix them. In this chapter, we employ an empirically-grounded analysis to explore bug fixes in a zoom-in approach by analyzing the types of files involved in fixes, the size of fixes, and the frequent bugs and fixes in various software projects. We mine the bug-fix code changes in two Java software corpora. We call the first one the Java Popular corpus, consisting of 717 unrelated popular Java projects crawled and cloned from GitHub. The second corpus is the whole Eclipse Ecosystem, which consists of 756 related projects. We mainly pose the following research questions:

RQ1: *What file types are involved in fixing bugs?*
We use the file extension of the textual files as the file type. As expected, Java files are involved the most in the fixes. However, a non negligible percentage of fixes involve XML files.

RQ2: *How large are bug fixes?*
We measure the size of fixes in terms of lines of code. We find that most fixes are one liners and the fix size distribution may be project-dependent.

RQ3: *What are the most recurrent bug and fix patterns?*
Throughout the study, the syntactically similar code changes corresponding to recurrent bug fixes are called *patterns*. For instance, a bug pattern can be a missing method invocation and a fix pattern can be inserting that method invocation. We find out that missing null checks and missing invocations are the most recurrent bugs in both corpora.

Answering these questions reveals many opportunities in the field of bug detection and automatic bug fixing. If we find bug-fix patterns in software systems, we can eventually apply fixes automatically in the right contexts. Such patterns can also reveal some good programming practices and some questionable ones.

3.1 Bug-Fix Analysis Procedure

The procedure we follow to identify our bug and fix patterns can be divided into three separate phases: building the software corpus, analyzing the source code, and inspecting instances of the top bug-fix code change patterns.

Table 3.1: Details of the studied corpora. The extracted commits are the ones corresponding to bug fixes.

Corpus	#Projects	#Commits	#Code Changes (Java)
Java Popular Projects	717	80,937	91,644
Eclipse Ecosystem	756	53,374	59,060
Total	1,473	134,311	150,704

3.1.1 Building The Software Corpora

Table 3.1 presents the two corpora we investigate; the Java Popular corpus and the Eclipse Ecosystem corpus. For the Java Popular corpus, we have built a crawler that queries GitHub for relatively large Java projects with good ratings and clones these projects into our server. The crawler's search criteria specified projects that were last updated at least on 01.01.2013 (active), had more than a five-star rating (popular), and were more than 100KB in size (relatively big). For the Eclipse Ecosystem corpus, we cloned all the Git source code repositories of Eclipse.[1]

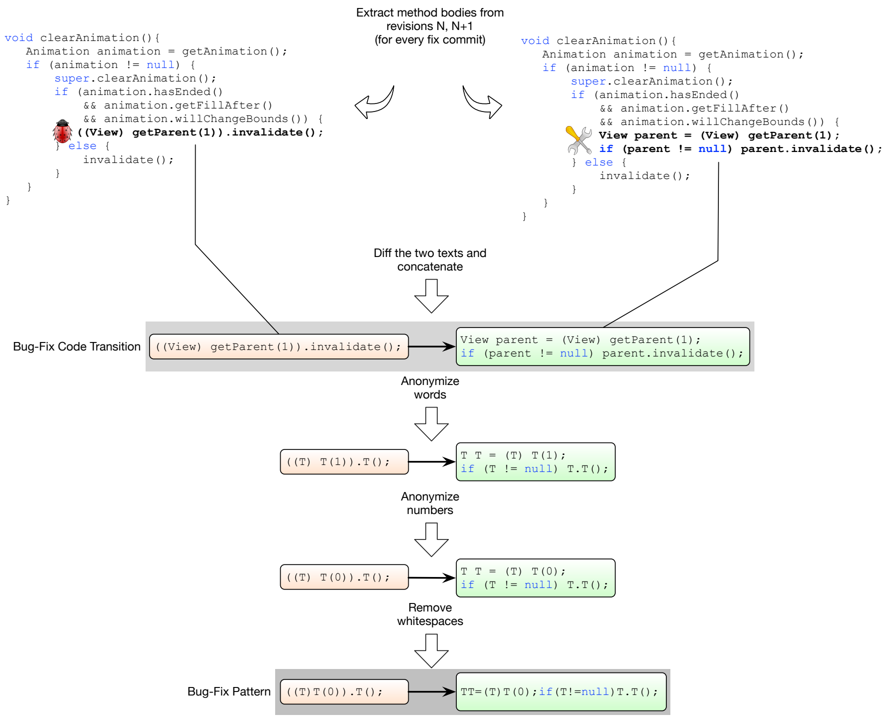
3.1.2 Analyzing Source Code and Change History

In the second phase, we analyze the change history and source code of the cloned projects. Figure 3.1 illustrates the main steps we perform for every project in our corpus:

Find bug-fix commits: We extract the commits corresponding to bug fixes from the Git repositories[2]. In our approach, every commit corresponds to a bug fix if the commit message contains some specific keywords, *i.e.*, "fix", "defect", "bug", or "patch" (following previous studies [94][37][102][190][141]). To increase the

[1]`https://git.eclipse.org/c/`

[2]We used the JGit library for parsing GIT history changes. `http://www.eclipse.org/jgit/`


Extract method bodies from revisions N, N+1 (for every fix commit)
Diff the two texts and concatenate
Bug-Fix Code Transition
Anonymize words
Anonymize numbers
Remove whitespaces
Bug-Fix Pattern

```
void clearAnimation(){
   Animation animation = getAnimation();
   if (animation != null) {
       super.clearAnimation();
       if (animation.hasEnded()
           && animation.getFillAfter()
           && animation.willChangeBounds()) {
         ((View) getParent(1)).invalidate();
       } else {
           invalidate();
       }
   }
}
```

```
void clearAnimation(){
   Animation animation = getAnimation();
   if (animation != null) {
       super.clearAnimation();
       if (animation.hasEnded()
           && animation.getFillAfter()
           && animation.willChangeBounds()) {
           View parent = (View) getParent(1);
           if (parent != null) parent.invalidate();
       } else {
           invalidate();
       }
   }
}
```

```
((View) getParent(1)).invalidate();  →  View parent = (View) getParent(1);
                                        if (parent != null) parent.invalidate();

((T) T(1)).T();  →  T T = (T) T(1);
                    if (T != null) T.T();

((T) T(0)).T();  →  T T = (T) T(0);
                    if (T != null) T.T();

((T)T(0)).T();  →  TT=(T)T(0);if(T!=null)T.T();
```

Figure 3.1: The pipeline which each project goes through. The final output of this analysis is the fix patterns and their concrete instances in the software projects.

precision of our results, we filtered out large commits touching more than five files, because they tend to include a fix and some other type of code change like refactoring or feature addition. As shown by Śliwerski *et al.* [190], the average number of changed files in fix commits tends to be small (2.73 in Eclipse and 4.39 in Mozilla). Also Thung *et al.* [119] show that bug fixes that span more than five files are very rare (7% in Rhino, 1% in AspectJ, 10% in Lucene).

Detect files with textual content: For each of the changed files, we read the top 100 bytes as characters, to form a string (S1), then, using regex, we delete the letters, digits, and punctuations and end up with another string (S2). If the ratio between the length of S2 and the length of S1 is less than a 0.5, we consider the file to be a textual one. Otherwise, we discard it.

Extract the buggy code & the fixed code: For each method that has been changed inside a commit representing a bug fix, we extract the version before the fix (buggy method) and the version after the fix (fixed method)[3].

Build the patterns: We call the minimal code that contains the bug a *bug hunk*, and we call the minimal code that fixed the bug a *fix hunk*, as in Figure 3.1. All pairs of bug hunks and fix hunks are concatenated to form a *bug-fix code transitions*, *e.g.*, *mImageViewer.close();* $\longrightarrow$ *mImageViewer.free();* The bug-fix transition can span several lines of code where some lines are changed, added, or deleted. The transition also can contain unchanged lines between the changed ones.

Following a similar procedure by Gabel and Su [59], we normalize the bug-fix code transitions to build syntactic patterns by applying the following three steps:

1. Each word (variable name, method name, *etc.*) is replaced by the letter "T" (except for Java keywords).
2. Each string literal is replaced by "T".
3. Each number is replaced by "0".
4. Each sequence of white space characters is removed.

In this way, we end up with anonymized patterns like: *T.T();* $\longrightarrow$ *T.T();*

3.1.3 Manual Inspection

In the third phase, after gathering all the necessary information, we group the code changes by fix pattern. For each bug-fix change pattern, we have the number of occurrences, the number of projects where the pattern occurred, and the concrete code snippets that adhere to the pattern.

We order the fix patterns based on the number of instances and the distribution across the projects. To get deeper insights into the semantics of the bugs and fixes, we manually inspected the instances of each of the 20 most significant patterns by reading the actual buggy code, how it was fixed, and the commit messages. Then we categorized them. Table 3.2 shows the manually-inspected patterns from both corpora and their corresponding categories. We noticed some similarities in the inspected syntactic patterns of each category. Using these similarities, we generalized the patterns to extract the exact number of occurrences of each category.

3.2 Results

We compare the results of analyzing the two corpora, the Java Popular corpus and the Eclipse Ecosystem. The analysis shows that both corpora are statistically similar regarding the most frequent bug-fix patterns, the types of changed files corresponding to bug fixes, and the number of source lines of code forming a fix.

[3]We used the JAPA library for parsing Java code.
`https://github.com/javaparser/javaparser`

3.2.1 The Types of Fixed Files

Although all the analyzed projects are Java systems, we found no single project that was pure Java. All the systems involved multiple programming languages (*e.g.*, Ruby, JavaScript, PHP, Scala) and configuration files (*e.g.*, XML).

To answer the first research question (**RQ1**: *What file types are involved in fixing bugs?*), for every commit corresponding to a fix, we extract the information about the types of the changed files. Figure 3.2 shows that, in both corpora, the fixed files are mostly Java files. This is expected due to the fact that both corpora are composed of Java projects. However, XML, JavaScript, and HTML files come in the second, third, and fourth order of importance correspondingly in both corpora.

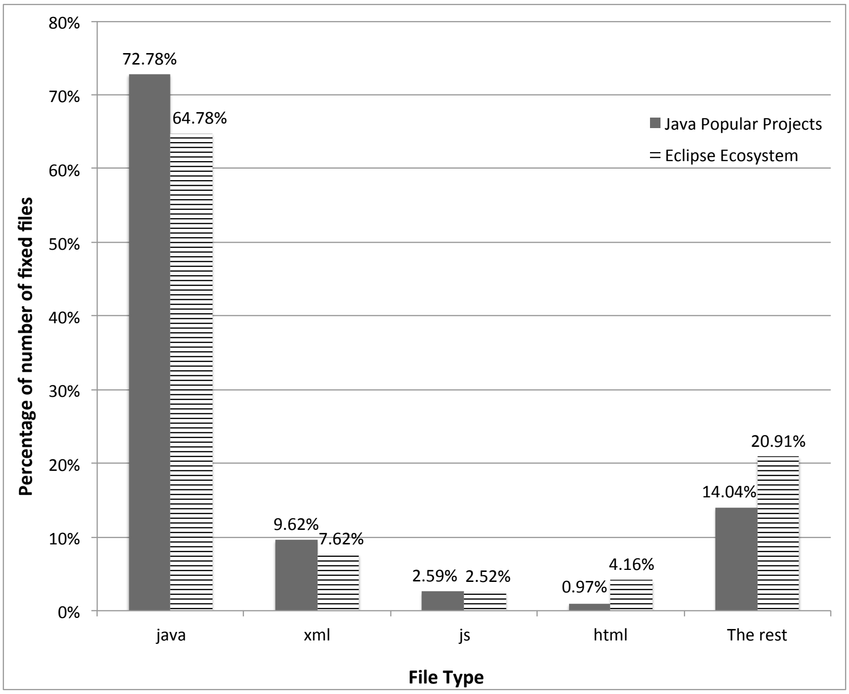


Figure 3.2: This figure shows the distribution of changes involved in bug fixes from the perspective of file types in the Java Popular corpus and the Eclipse Ecosystem. Looking at the files involved in bug fixes, one can notice the similarity between the Java Popular corpus and the Eclipse Ecosystem. Since both corpora contain Java projects, Java files are the most commonly fixed files through the evolution of systems. However, XML, JavaScript, and HTML files are the most frequently fixed files after Java files in both corpora.

3.2.2 Fix Size

Regarding the second research question (**RQ2**: *How large are bug fixes in terms of lines of code?*), we find that most fixes in both corpora involve one source line of code in both corpora, as shown in Figure 3.3. However, we need to compare the distribution of fix size in both corpora. Our null hypothesis is that the Java Popular corpus and the Eclipse Ecosystem have identical distribution. We test this hypothesis without assuming the data to be normally distributed. At 0.05 significance level, we use the *Mann-Whitney-Wilcoxon Test* to test our hypothesis. Applying the test, the p-value turns out to be approximately less than $2.2 * 10^{-16}$, which is less than the 0.05 confidence interval[4]. This means that we reject the null hypothesis and the two distributions, shown in Figure 3.3, are nonidentical.

The conclusion is that the fix size may be dependent on the projects. Different projects have different structures, external dependencies, and domains. This may lead to variations in fix sizes. However, one-line fixes still form the majority of the fixes in both corpora.

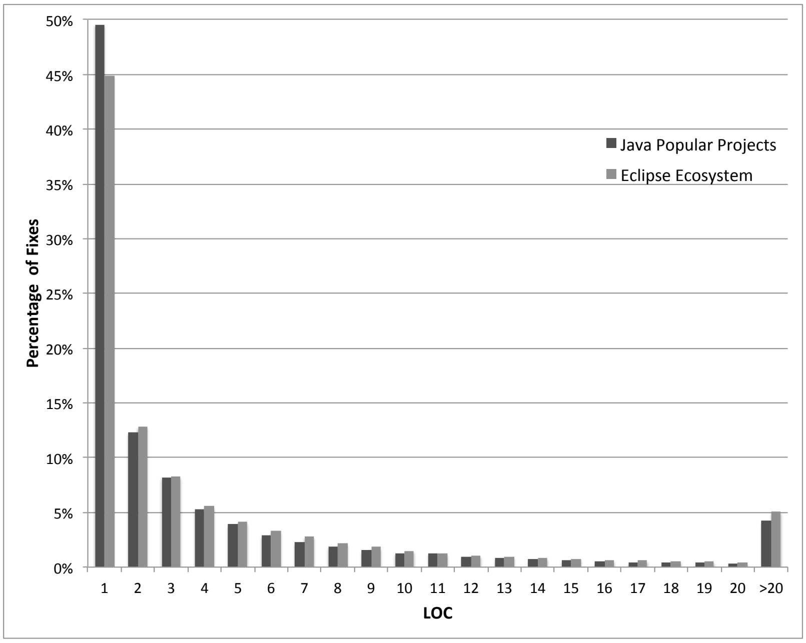


Figure 3.3: The distribution of bug fixes according to the number of changed lines of code in the Java Popular Corpus and the Eclipse Ecosystem. Both distributions are very similar and almost identical.

[4]The data set is large. Hence the Mann-Whitney-Wilcoxon Test can only approximate the p-value. However, the approximation is quite accurate and considered to be a standard.

Table 3.2: The Top 20 Syntactic Fix Patterns in The Popular Java Corpus and The Eclipse Ecosystem Corpus

Syntactic Pattern		Instances	Projects	Category
		Java Popular Corpus		
␣	⟶ T.T();	668	25 %	Missing Invocation
␣	⟶ T();	448	18 %	Missing Invocation
␣	⟶ T.T(T);	368	19 %	Missing Invocation
return T;	⟶ return T;	336	12 %	Wrong Name/Value
T.T(T);	⟶ T.T(T);	335	18 %	Wrong Name/Value
␣	⟶ T(T);	221	9 %	Undue Invocation
T.T();	⟶	191	12 %	Undue Invocation
␣	⟶ T.T(T, T);	162	10 %	Missing Invocation
T.T(T, T);	⟶ T.T(T, T);	151	11 %	Wrong Name/Value
T();	⟶	136	8 %	Undue Invocation
return T.T();	⟶ return T.T();	126	4 %	Wrong Name/Value
␣	⟶ if (T == null) return;	118	4 %	Missing Null Check
T.T(T);	⟶	115	9 %	Undue Invocation
T(T);	⟶ T(T);	102	5 %	Wrong Name/Value
T T = T.T(T);	⟶ T T = T.T(T);	99	7 %	Wrong Name/Value
}catch(TT){	⟶ }catch (TT){	93	6 %	Too General Exception
␣	⟶ if (T==null){return null;}	85	6 %	Missing Null Check
␣	⟶ T=null;	85	5 %	Missing Release
␣	⟶ if(T==null) {return;}	79	5 %	Missing Null Check
␣	⟶ if(T==null) return null;	68	5 %	Missing Null Check
		Eclipse Ecosystem		
␣	⟶ T();	368	25 %	Missing Invocation
␣	⟶ T.T(T);	307	25 %	Missing Invocation
T.T(T);	⟶ T.T(T);	255	23 %	Wrong Name/Value
␣	⟶ T.T();	242	22 %	Missing Invocation
return T;	⟶ return T;	192	12 %	Wrong Name/Value
␣	⟶ T(T);	176	15 %	Missing Invocation
␣	⟶ T.T(T.T());	132	12 %	Missing Invocation
T();	⟶	109	12 %	Undue Invocation
T.T(T);	⟶	109	12 %	Undue Invocation
T.T();	⟶	96	13 %	Undue Invocation
␣	⟶ T = null;	94	10 %	Missing Release
␣	⟶ if (T == null) return;	85	7 %	Missing Null Check
T(T);	⟶	73	11 %	Undue Invocation
␣	⟶ if (T==null){return null;}	71	10 %	Missing Null Check
␣	⟶ if (T==null) return null;	66	9 %	Missing Null Check
T();	⟶ T();	66	7 %	Wrong Name/Value
␣	⟶ T.T(true);	60	7 %	Missing Invocation
␣	⟶ if (T==null){return;}	56	8 %	Missing Null Check
T(T);	⟶ T(T);	56	8 %	Wrong Name/Value
return T.T(T);	⟶ return T.T(T);	54	3 %	Wrong Name/Value

Table 3.3: The most recurrent bug categories and corresponding examples of their bug-fix code-change patterns.

Bug	Java Popular Corpus		Eclipse Ecosystem		Example	
Category	#Instances	%Projects	#Instances	%Projects	Paterns	
Missing Null Check	5,328	65%	4,411	64%	␣	⟶ if (T == null) return
					␣	⟶ if (T == null) return null;
					T.T();	⟶ if (T != null) T.T();
					␣	⟶ if (T == null) return T
Wrong Name/Value	5,997	76%	2,872	72%	T.T(T);	⟶ T.T(T);
					return T;	⟶ return T;
					T();	⟶ T();
					T(T);	⟶ T(T);
Missing Invocation	2,618	66%	1,596	61%	␣	⟶ T.T();
					␣	⟶ T();
					␣	⟶ T.T(T);
					␣	⟶ T(T);
Undue Invocation	1,907	53%	1,293	54%	T.T();	⟶ ␣
					T();	⟶ ␣
					T.T(T);	⟶ ␣
					T(T);	⟶ ␣

3.2.3 The Most Frequent Bug-Fix Patterns

As an answer to the third research question (**RQ3:** *What are the most recurrent bug and fix patterns?*) we find that the most frequent bug patterns are missing null checks (null dereferencing), wrong name/value, missing method invocations, and undue invocations. These bug patterns are fixed by adding null checks, editing the name/value, adding the missing invocation, and removing the undue method call correspondingly. We elaborate on these patterns and report our findings from the manual investigation in the next section.

We compare the frequent bug-fix patterns in both corpora using Spearman Rank Correlation between the two lists[5] in Table 3.2. The Spearman Rank Correlation between the two lists is 0.58, meaning that the two lists are highly similar. Also considering the categories of the most recurrent bug patterns in Table 3.3, we see that the order of importance of these categories regarding the occurrences of the categories and the percentage of projects affected is identical in both corpora. These results indicate that the patterns are project-independent. Our intuition is that the patterns, as we define them, are Java-specific.

3.3 Bug-Fix Patterns

In the following subsections, we explain and demonstrate the most recurrent bug-fix patterns we find through our study in both corpora. Table 3.3 shows the bug

[5]The two lists are not identical. We mitigate this problem using the naïve approach of appending the missing values from one list to the other list and give them equal ranks.

categories, the number of instances, the percentage of affected projects, and some example bug-fix syntactic patterns in each category. These categories are generalized from our findings from the manual inspection phase.

3.3.1 Missing Null Checks

Dereferencing an object without checking if it is null or not is the most recurrent and critical bug pattern we found. The criticality of this bug comes from the fact that 65% of the studied projects suffered from this bug, as shown in Table 3.3, and it usually causes systems to crash due to the uncaught NullPointerException. In our manually investigated sample, the bugs of this category are fixed by inserting null checks on certain objects (*the checked object*) like *if (T==null)* or *if (T!=null)*, meaning that the fixes for the bugs in this category are all *addition* changes.

We generalize from the manually-inspected syntactic patterns of this category and consider a pattern to belong to this category if (1) the bug hunk does not have a null check, and (2) the fix hunk has a null check. Table 3.3 shows some example fix patterns that fall into this category.

In many cases, a blank *return* will follow the check. In other words, if the checked object is null then the method cannot continue its execution and should immediately return. One example of this is:

```
if (viewActionsContainer == null) return;
```

The checked objects are either the results of method invocations, parameters, or member variables. In almost 70% of the manually inspected cases the checked object comes from a method invocation. Moreover, this kind of bug often appears when chaining method calls, as in the illustrated example in Figure 3.1. In chapter 4, we elaborate on the root causes of null checks.

3.3.2 Wrong Name/Value

All the syntactic patterns that have identical bug and fix hunks fall into this category. Although the bugs of these patterns seem very frequent from Table 3.3, they are less important than the other categories because in most of the inspected instances, the fixes were corrections or edits to literal strings passed as parameters. This pattern appeared very often with logging methods (*log, print, println,* etc.).

However, we encountered many cases where the bug lies in the object names, method names, or parameter names. The reasons behind this kind of bug are either (1) using the wrong identifier due to name similarity, or (2) calling the wrong method based on mistaken name-driven assumptions about its functionality.

```
imageViewer.close(); ⟶ imageViewer.freeTextures();
key.rewind(); ⟶ key.flip();
```

Listing 3.1: Sometimes developers invoke wrong methods due to misunderstanding method names.

```
dragView.setPaint(mPaint); ⟶ dragView.setPaint(mHoverPaint);
visitedURLs.clear(); ⟶ visitedURIs.clear();
return hasClassAnnotations; ⟶ return hasRuntimeAnnotations;
```

Listing 3.2: In some cases developers confuse objects or parameters with similar names.

Listing 3.1 shows two fixes where the bugs were due to misunderstanding method names. Listing 3.2 shows that the names can be very similar and the programmer might mistakenly use one instead of the other.

3.3.3 Missing Invocation

When the fix hunk has a new method call that is not present in the bug hunk, then the pattern falls into this category. Table 3.3 shows example patterns. There are many scenarios in which this kind of bug can appear. The first is a **missing initialization or configuration** of an object immediately after its creation. For example:

```
ConnectionPool config=new ConnectionPool()
```

should be immediately followed by:

```
config.initialize()
```

The second scenario is what we can call a **missing refresh** where before doing something or after finishing something, a certain object should be brought to a consistent state or "*refreshed*". These missing method invocations are either at the very beginning or at the very end of a method body. Method names that we often encountered from this category are: *refresh, reset, clear, pack, repaint, redraw, etc.* In most of the cases, the "*refreshed*" object is some kind of a container or an aggregator class like a canvas, tree, view, *etc.*

The third scenario concerns a **missing release**. This type of invocation is always about freeing resources and always comes at the end of the method body in the manually inspected instances. Example methods are: *release, delete, dispose, close* etc.

The fourth scenario occurs when the missing invocation is a **missing step** during the building of an object. This pattern is seen mostly in the "Builder Design Pattern." For instance, take the following method found in the project *eclipse.pde.build*:

```
void generateBuildScript(AntScript script){
  generatePrologue(script);
  generateBuildUpdateJarTarget(script);
  generateGatherBinPartsTarget(script);
  generateBuildJarsTarget(script, model);
  generateBuildZipsTarget(script);
  generateGatherSourcesTarget(script);
  generateGatherLogTarget(script);
  generateCleanTarget(script);
  generateRefreshTarget(script);
  generateZipPluginTarget(script);
}
```

This method is buggy because it is missing this invocation at the end:

```
    generateZipFolderTarget(script);
```

3.3.4 Undue Invocation

A pattern is in the undue invocation category when the bug hunk has a method call that is not in the fix hunk. Table 3.3 shows example patterns of this type. This pattern is exactly the opposite of the missing invocation pattern and, surprisingly, corresponds to the same kind of methods like *flush, reindex, init, close, etc.*

A deeper investigation showed that this bug-fix pattern aims at either

- improving the performance of the system by removing unnecessary initializations or resettings, or
- avoiding premature resource freeing that causes NullPointerExceptions (NPEs) somewhere else. We identify this pattern when the bug-fix is in the undue invocation category and the commit message explicitly states that this is an NPE fix.

3.3.5 Other Patterns

As shown in Table 3.2, there are two further, less frequent bug-fix patterns that we inspected manually. The first, called **Missing Release**, assigns *null* to certain variables. This bug usually appears inside the methods responsible for freeing the resources such as *release, delete, dispose, close,* etc.

The second pattern is the **Too General Exception** bug. This bug-fix pattern usually is applied when a *"try"* block has only one *"catch"* block that catches only an object of *"Exception"*, which is the most general type of exception in Java. The fix changes the type of caught exceptions to more specific ones and adds more catch blocks to handle the different exceptional situations differently.

3.4 Implications

Changing Developer Practices: The frequent bug patterns impact programming style, programming language design, and common practices in software design. The *missing invocation*, *undue invocation*, and the *wrong name* categories shed light on the importance of API design. Several of the discussed causes of these bugs would have been eliminated with proper API design and method/variable naming. It would be interesting to see whether language-level support could be offered for these kinds of bugs. *Missing null checks*, on the other hand, shows that developers should be aware of the high risk associated with using null. Alternatives to null should be employed whenever possible. We elaborate on this issue in chapter 4.

Automatically Detecting Bug Patterns: The existence of "global" bug patterns reveals opportunities for bug localization and automatic bug fixing. However, the studied bug-fix patterns are syntactic. We believe that there is no unified context for all patterns but rather a specific context for each pattern. As we show in chapter 4, the context of the missing null check would be the source of the dereferenced object. If that object is coming from a method call and that method returns null in certain situations, this might be a likely candidate for a missing null check bug. On the other hand, the context of the missing invocation pattern can be the preceding and succeeding method calls, as explained in subsection 2.1.2. Defining an automated approach to extract the contexts and the bug patterns is essential to build bug detectors.

Bug-Detection Beyond Java: Finally, we found that a significant percentage of the bug-fix code changes in Java systems are made to non-Java files, especially XML files. Modern software frameworks (*e.g.*, Spring, Hadoop, *etc.*) rely heavily on configuration files making them the second most commonly fixed files after Java files. Studying the patterns of changes made to those configuration files would be useful for both practitioners and researchers alike. This has implications on the future development of bug detectors, which might have to consider different types of software artifacts that are common in modern applications.

3.5 Threats to Validity

We built our corpora to be as representative as possible for Java systems to increase our confidence in the results. The first corpus represents Java projects that are popular, big enough, and actively maintained. The second corpus is an entire ecosystem. However, even if the results were statistically similar between the two corpora, there is no guarantee that the results can be generalized to all Java systems.

In our extraction, parsing, anonymization, and normalization of the bug-fix code changes, we might have missed some instances or grouped some instances in the wrong categories. For instance, wrong requirement specification can result in a bug spanning multiple files and lines of code. However, due to our large number of code change snippets, as shown in Table 3.1, we argue that corner cases affect our statistics and pattern significance computation within acceptable limits.

3.6 Conclusions

In this chapter, our main aim is to explore different bugs in Java systems and see whether there are frequent bugs that are worth investigating. As we discussed in chapter 1, bugs that are detectable, conspicuously severe, and reasonably frequent are worth building detectors, efficient detectors that is.

Indeed, we find that frequent bug patterns do exist in Java systems. Namely, null-related and method-call-related bugs are the most frequent and automatically detecting them would be of great value to developers.

In this thesis, we choose to build a detector for the missing null check bug pattern. In the next chapter, we empirically analyze null usage in general and null checks in specific, in order to have a better understanding of the problem and identify possible approaches to tackle it.

4
Null Usage Analysis

Tony Hoare[1] considers null as his *"billion-dollar mistake"* [83]. As we found in chapter 3, *missing null check* is the most frequent pattern of bugs in Java systems. These results suggest that null dereferencing is a major source of bugs in Java programs, forcing developers to add guards (null checks) on objects before using them. However, besides the bugs it introduces in running systems, null usage hinders performance [167], increases maintenance costs [103], and decreases code readability.

Although many tools and techniques have been introduced to solve the null dereferencing problem, an analytical study is still missing on how null is used, why null checks are introduced, and where the checked-for-null objects come from in the source code. We argue that such a study is crucial for building an efficient detector of null-related bugs. In this chapter, we aim at understanding when, why, and how often developers introduce null checks. More concretely, we pose the following three research questions:

RQ1: *How common are null checks?* We answer this question by measuring the ratio between the overall number of conditional statements and those containing null checks.

RQ2: *What kind of objects are compared to null?* We consider whether the checked object is a method parameter, a member variable (field), or a local variable.

RQ3: *How are the checked-for-null objects initialized?* We analyze the kind of expression that was assigned to the checked-for-null object.

[1]A computing pioneer credited with introducing the concept of a null pointer in Algol W. `https://en.wikipedia.org/wiki/Tony_Hoare`

RQ4: *How is null used in Java programs?* We manually investigate 100 samples from our corpus to understand what null represents in Java programs (*e.g.*, errors, default values, or other special values).

To answer the posed research questions, we developed a tool, *NullTracker*, that statically analyzes Java source code files and approximates the statistics about null check usage. Applying our analysis to a large Java software corpus, we find that 35% of the conditional statements in our corpus are null checks.

We also find that 24% of the checked objects are member variables,[2] 23% are parameters, and 50% are local variables. Unsurprisingly, 71% of the checked objects come from method calls. In other words, developers insert null checks mainly when they use methods that may return null. Passing null values to methods and uninitialized member variables are recurrent reasons for introducing null checks.

We manually review 100 code samples from our corpus to understand the contexts and discover patterns of null check usage. In 76 samples null is used to represent errors, in 15 samples it is a representation of absence of a value, and in 9 samples null represents a meaningful value. Most of these instances of null usage can (or should) be replaced by proper exceptions or *special case objects* such as a *Null Object* [211][212].

4.1 Motivation

Pominville *et al.* achieved 2% to 10% performance gain in Java bytecode when they annotated Java class files with assumptions about the "nullness" and array bounds [167]. With respect to null, their framework, SOOT [202], performs intra-procedural null analysis to make assumptions about variables being null or not to be able to remove unnecessary null checks in the bytecode level. This means that null checks impose a non-negligible performance overhead on Java programs.

In a managed language like Java,[3] null is an unusable value. Its interpretation depends on the context and when it causes a problem, its value yields "no useful debugging information" [23]. For instance, the `listFiles()` method in the `File` class returns an array of the files, `File[]`, in the specified directory. However, if the directory does not exist, it returns null. This returned null value might mean that the `File` object does not exist, that it is not a directory, or that an I/O error has occurred. This inherent ambiguity of null leads to increased difficulties in code comprehension.

Missing null checks are the most recurrent bug pattern in Java systems [158]. This bug manifests itself as the Java NullPointerException. Debugging this kind of exception is particularly hard because the stack traces do not provide enough information about the source of the null. Acknowledging this problem, Bond *et al.* introduced *Origin Tracking* [23], which records code locations where unused values (such as null) are assigned. *Origin Tracking* gathers the necessary information

[2]Member variables are also known as fields or field variables.

[3]A program written in a managed language is executed under the management of a language runtime.

dynamically at run time so they can be reported at failure time to aid the debugging activities. This indirectly means that the overuse of null in program increases maintenance efforts.

To this end, we establish that the use of null in Java code often leads to performance degradation, code that is harder to read, more defective software, and increased project maintenance efforts. In the following sections we explore how often null is used in Java code and in what contexts. This knowledge can help software engineers to build better static code checkers and develop better practices for writing and reviewing Java code.

4.2 Null Check Analysis

4.2.1 Experimental Corpus

For our experiment, we used the same software corpus from a previous study [158]. This corpus was built using a crawler that queries Github[4] for Java projects that have more than 5 stars (popular) and are more than 100KB in size (relatively large). This corpus contains 810 Java projects, 371,745 Java source files, and 34,894,844 lines of code. We are making the corpus available for download through the Pangea infrastructure[5] [32].

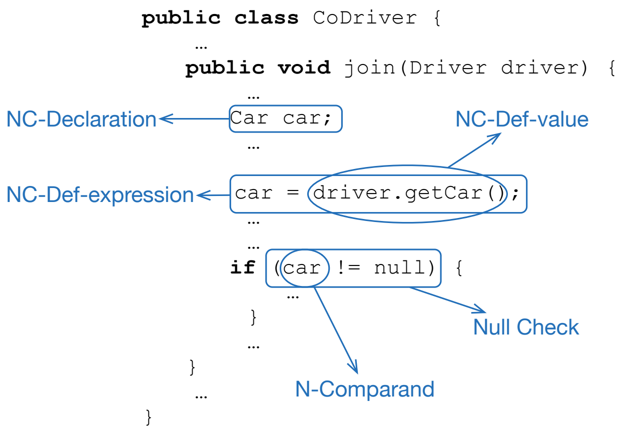
4.2.2 Terminology

Before we explain the analysis, we define the terms used in this thesis as follows (depicted in Figure 4.1):

- *A Conditional*: is a binary comparison expression that evaluates to a boolean value, such as:
  - `y > 0`
  - `x != null`
- *A Conditional Statement*: is a Java statement that contains a conditional, such as:
  - `if(y > 0) ...`
  - `isNull = (x != null);`
- *A Null Check*: is a conditional that contains the *null* literal as a left hand side or a right hand side operand. In other words, it is a comparison to null. *e.g.*,
  - `Person != null`
  - `iterator.next() != null`

[4]`http://www.github.com/`
[5]`http://scg.unibe.ch/research/pangea`

- *An N-Comparand*: is an expression that is compared to null in the null check (Usually an assignable l-value.)
- *An NC-Declaration*: is the declaration expression of the *N-Comparand*.
- *An NC-Def-expression*: is an assignment expression involving the *N-Comparand* as the assignable l-value.
- *An NC-Def-value*: is the value assigned to the comparand in an *NC-Def-expression* (*i.e.*, the right-hand side operand of the *NC-Def-expression*).



```
public class CoDriver {
  ...
  public void join(Driver driver) {
    ...
    Car car;
    ...
    car = driver.getCar();
    ...
    ...
    if (car != null) {
      ...
    }
    ...
  }
  ...
}
```

Figure 4.1: A code example showing the definitions of the terms used in the thesis.

4.2.3 Analysis

We implemented a tool, *NullTracker*[6], to extract null checks and analyze the kinds and definitions of the *N-Comparand*s. *NullTracker* is designed as a pipeline, following a pipes and filters architecture. *NullTracker* analyzes each Java source file as follows:

1. Parse the Java source file and extract the null checks.
2. For each null check, extract the *N-Comparand*.
3. Parse the *N-Comparand* and determine its kind (*e.g.*, name, method call, field access, *etc.*).
4. When the *N-Comparand* is a name expression, determine its kind (member variable, local variable, or parameter) by looking for its *NC-Declaration* within the current method for local variables and parameters, and within the current type declaration for member variables.

[6] `https://github.com/haidaros/NullTracker`

5. When the *N-Comparand* is assignable (name, array access, field access), extract all the *NC-Def-expression*s that appear lexically before the null check and within the same method as the null check itself. Then, parse them and extract the kind of the *NC-Def-value*s (method call, null literal, object creation, or any expression that can evaluate to a reference value).

6. Finally, the resulting data, which conforms to the model illustrated in Figure 4.2, is saved in the database for further analysis.

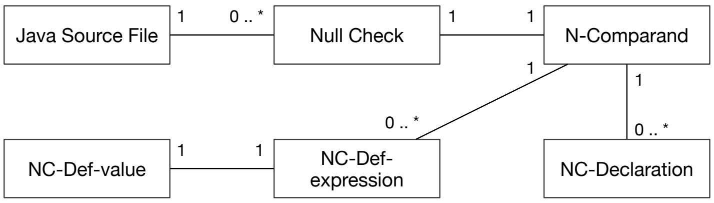


Figure 4.2: The data model of *NullTracker* analysis.

4.2.4 Manual Inspection

After the analysis phase, we manually inspect multiple instances of the null checks belonging to different categories and types. More concretely, we inspect ten random samples of each of the following categories to gain more insight:

1. Method call *N-Comparand*s.
2. Field access *N-Comparand*s.
3. local variable name *N-Comparand*s.
4. Parameter name *N-Comparand*s.
5. Member variable name *N-Comparand*s.[7]
6. Method invocation *NC-Def-value*.
7. Null literal *NC-Def-value*.
8. Cast *NC-Def-value*.
9. Object creation *NC-Def-value*.
10. Name *NC-Def-value*.

In this phase we aim at understanding how and why developers use null values and null checks. We look specifically at the following:

[7]Member variable name *N-Comparand* and field access are the same semantically but different lexically. For example, `this.name` is a field access whereas `name` is a member variable name.

1. The intended semantics of the null value.
2. Potentially missing null checks (*e.g.*, a member variable that is sometimes checked against null and sometimes not before dereferencing it).
3. The type of the checked-for-null object (String, List, Tree, Number, *etc.*).
4. The source of the null value. (*e.g.*, uninitialized local variables, a return null statement in a method body, *etc.*)

We do not derive any statistics from this phase, as we only want to gain deeper insights into how null and null checks are used in the code and for what reasons.

4.3 Results

We applied our analysis to the 810 Java projects in our dataset and we manually reviewed 100 code samples. In the following subsections which are organized around the research questions, we explain the results.

4.3.1 How Common Are Null Checks?

To our knowledge, only Kimura *et al.* have measured the density of null checks in source code [103]. They measured the ratio between the number of null checks and the number of lines of code. They found this ratio to be from one to four per 100 lines of code, depending on the project.

We, on the other hand, go one step further and measure the ratio of the conditional statements containing null checks with respect to all conditional statements. This will enable us in answering the first research question: **RQ1: How Common are Null Checks?**

We call the ratio between the null checks and the overall number of conditional statements the *null check ratio*. Analyzing our dataset, we found 2,329,808 conditionals, 818,335 of which contain null checks. This means that a staggering 35% of the conditional statements contain null checks.

As detailed in Table 4.1 and Figure 4.3, the null check ratio exhibits a bell-shaped distribution around the value of 32%. In other words, more than half of the projects have a null check ratio of more than 32%. Our results show evidence of an existing overuse (or abuse) of null checks by Java programmers, which indicates the overuse of the "null" value itself. As discussed in section 4.1, this practice affects the readability of code, the maintainability of the project, and the performance of the running system.

4.3.2 What Entities Do Developers Check For Null?

To answer the second research question (**RQ2: What kind of objects are checked against null?**), first, we analyze the kind of the *N-Comparand* itself. Second, we

Table 4.1: A summary of the per-project null check ratio.

	Number of Conditionals	Number of Null Checks	Null Check Ratio
Min	1.0	0.0	0%
1st quartile	117.5	33.0	23%
Median	531.5	138.5	32%
Mean	2,876.3	1,010.3	32%
3rd quartile	2,171.8	671.5	40%
Max	196,812.0	76,609.0	100%

look for the *NC-Declaration* of the *N-Comparand* when it is an l-value (*i.e.*, *name expression, array, or field access expression*).

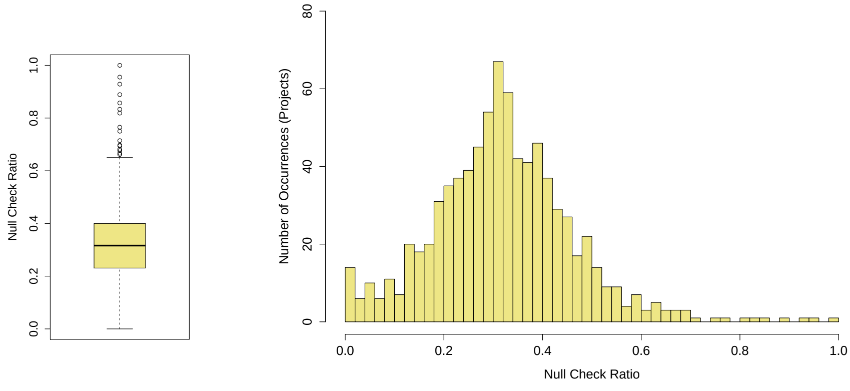
We find that *N-Comparand*s are mainly *name expressions* (78%) and *method call expressions* (15%), as Figure 4.4 shows. Figure 4.5 shows that 50% of the name *N-Comparand*s are local variables, 24% are member variables, and 23% are parameters.

Inspecting 10 code samples where the null check is against a method call, the method calls are all to getter methods either from the same class or from another class. This puts field access *N-Comparand*s, method call *N-Comparand*s, and member variable name *N-Comparand*s in the same category, which is member variable null check. Member variables are checked against null because they might not be initialized. This happens in the manually inspected code when one or more of the following is true:

- There exists a constructor that does not initialize the variable.
- There exists a constructor or a setter method that can accepts null as a parameter and sets the variable to null.
- There exists a method that explicitly sets the variable to null.
- The member variable is public or is returned by address in the getter method.

The code in all inspected 10 samples where the *N-Comparand* is a member variable can be improved to avoid having to check for null every time the member variable needs to be used. In our samples, there is no obvious reason why member variables are not explicitly initialized in every constructor. In fact in 5 of the inspected samples, the member variable should even be `final`. This suggests a failure in applying well-established object-oriented design principles, in particular that of establishing class invariants [136].

As we see in Figure 4.5, a considerable percentage of null checks are guards on method parameters. In the inspected code samples, we observe that developers



(a) (b)

Figure 4.3: The distribution of the per-project null check ratio. The boxplot in (a) shows that more than half the projects have null check ratios of more than 32%. The histogram in (b) shows that the null check ratio distribution demonstrates a bell-shaped curve around the value of 32%.

check parameters against null mainly to validate them. However, we differentiate between two recurring patterns of null checks on parameters. In the first pattern, a method throws an exception if a certain parameter is null. Listing 4.1 shows a real example of this pattern. The second, and more questionable, pattern is shown in Listing 4.2. The method skeletons in Listing 4.2 are the most recurrent usage scenarios of a parameter null check. In these scenarios, the method does not accept null as a parameter. However, instead of throwing a proper exception, the method does nothing and returns silently without informing the caller of any problem.

```
public File writeToFile(final HttpEntity entity)
  throws ClientProtocolException, IOException {
        if (entity == null) {
          throw new LibRuntimeException(LibResultCode.
    E_PARAM_ERROR);
        }
  ...
}
```

Listing 4.1: Throwing a proper exception when the parameter is null.

```
SOME_TYPE method1(Param p){
  if(p==null){
    return null;
  }
  METHOD_BODY
}
```

```
void method2(Param p){
  if(p==null){
    return;
  }
  METHOD_BODY
}


void method3(Param p){
  if(p!=null){
    METHOD_BODY
  }
}
```

Listing 4.2: The most recurrent usage scenarios of a parameter null check.

More often than not, this indicates a poor API design. One can use, for instance, the *specification pattern* [50][49] to extract the validation code, which throws a proper exception, in a dedicated method or class. In any case, it is widely acknowledged that passing null as an argument to methods is a bad practice and *"the rational approach is to forbid passing null by default"* [124]. Nevertheless, developers often add null checks on parameters because they expect null to be passed as a parameter. Our results show a clear gap between what is considered a good practice and how software is implemented in reality.

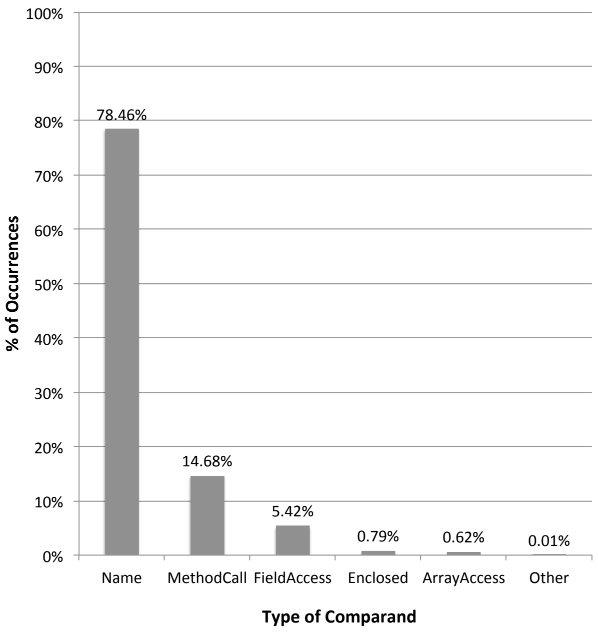


Figure 4.4: The immediate kind of the *N-Comparand*s as a parsing expression.

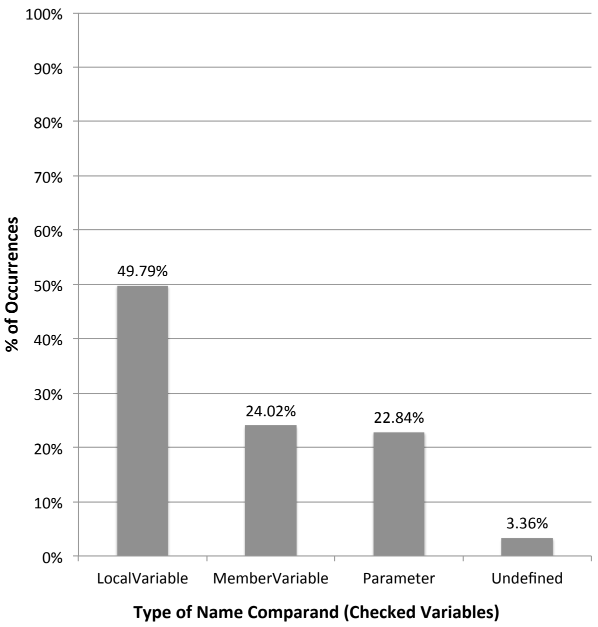


Figure 4.5: The type of name expression comparands. *Undefined* indicates that 3.36% cannot be determined, as explained in section 4.5

4.3.3 Where Does Null Come From?

When the *N-Comparand* is a name expression, we analyze the *NC-Def-value*s assigned to it in all the *NC-Def-expression*s preceding the null check within the same method. Figure 4.6 shows the kinds of the assigned *NC-Def-value*s. As an answer to the third research question (**RQ3: How are the checked-for-null objects initialized?**) we find that 71% of the time the *NC-Def-value* is a method call expression, which means that null checks are mostly applied to values returned from method invocations. In other words, when methods possibly return null, they tend to cause NullPointerExceptions in the invoking methods forcing developers to add null checks.

There is a long debate about whether methods should return null or not. In a previous study [158], we found that missing null checks represent the most frequent bug in Java programs. In this study, we show that null checks are applied to the results of method invocations. Both studies combined provide evidence that returning null in methods is a major cause of bugs. Hence, we side with the opinion that developers should avoid returning null in their method implementations and either throw an exception or return a *special case object* [124] such as a *Null Object* [211][212].

Surprisingly, some Java standard libraries exhibit this questionable design [124]. In our manually inspected code samples, we find 5 null checks because of methods from the Java standard API *e.g.*, Map.get(...), List.get(...), Iterator.next().

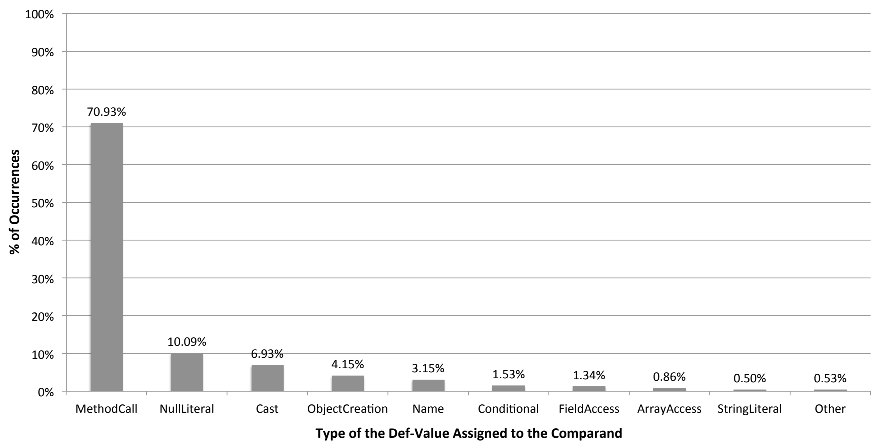


Figure 4.6: This diagram shows that the checked-for-null objects are mainly set or initialized using method invocations.

Another less frequent reason for checking a local variable for null is when it is initialized within a method call or a constructor that might throw an exception before completion. We observe this pattern in our manually-inspected code samples, as the code skeleton shows in Listing 4.3. The variable is set to null first, initialized in a `try-catch` block, then checked for null to make sure that initialization completed and no exception was thrown.

```
...
Object obj = null;
try{
  obj = METHOD_INVOCATION or CONSTRUCTOR_CALL
}catch(...){
  ....
}
if(obj != null){
  ...
}
...
```

Listing 4.3: When the initialization is in a try-catch block, a null check usually follows to make sure no exception is thrown.

4.3.4 What Does Null Mean?

In the manual inspection phase we find that developers use null values for three different reasons.

As an answer to the fourth research question (**RQ4: How is null used in Java programs?**), the most recurrent usage of null is to encode or represent an error. 76 of 100 inspected samples fall into this category. For instance, if a method does not accept null as a parameter, it returns null or just returns when it encounters a null parameter. Another example is when a method returns null in a `catch` block.

The second usage of null is to represent the absence of a value (or the *nothingness*). 15 out of the 100 inspected samples fall into this category. For example, left and right branches in a leaf node are null in a binary tree. Another example is a `find(...)` method that returns null in case it cannot find the item.

The third usage of null is when it is a proxy for a meaningful value. For instance, in many code samples, we find non-boolean variables used as ones (null means `false` and instantiated means `true`).

We argue that the first and third usage of null are bad practices as there are other language constructs to represent the corresponding semantics. For errors and failures, one should use well-defined exceptions as they are easier to read and act upon. For the third usage, one can have a dedicated `enum`, `class`, or other data type to represent the semantics in a more readable and maintainable form. In fact, 19 out of the 100 inspected classes contain potential bugs in the form of potentially missing null checks. All of these 19 potential bugs exist when null represents an error or a meaningful value.

We also argue that second usage scenario where null means the absence of a value is only tolerable and not ideal, as one should use *special case object* [124] like the *Null Object* [211][212] when applicable. Actually, even when the representation of absence of value is available for free (such as an empty string for a `String` variable), developers tend to not use it. Eight out of the inspected *N-Comparand*s are of type `String` and seven are of type `List`. Carefully reading the code, we find out that uninitialized strings and lists are equivalent to empty strings and lists correspondingly in these samples.

4.4 Discussion

Null usage often leads to various maintenance problems. Missing null check bugs, useless NullPointerException stack traces, and vague null semantics are often the consequences of the careless use of null value. A disciplined usage of null is highly recommended [58], and alternatives, like the Null Object Pattern [211][212], should be considered. However, the results of this study suggest that null is often misused and overused in Java code.

To demonstrate how null is often misused, we put the observations from this study and from our previous study on bug-fix patterns [158] into a hypothetical story that represents the most recurrent pattern of wrong null usage.

A developer, *A*, adds a `return null` statement in the method body of `m(...)` to indicate that a problem has occurred and the method `m(...)` cannot continue its normal execution. Another developer, *B*, uses the method `m(...)` and assigns its result to a local variable `obj`. When developer *B* runs his code, a NullPointerException is thrown. Developer *B* looks at the stack traces struggling to understand the problem and finally identifies the place of the null dereferencing. It is the local variable `obj` and the null comes from the method invocation of `m(...)`. Unable to understand the meaning of null in this situation, developer *B* adds a null check before using the local variable *obj*.

When such scenarios accumulate in a codebase, the code starts to get harder to maintain and reason about, leading to the problems discussed in section 4.1. Our study also reveals some actionable recommendations to reduce null checks and null usage:

1. A method should not return null in case of errors. A method should always throw a proper exception that explains the exact reason and even possible solutions in case of errors.

2. Null should not be passed to public methods and public methods should not accept null as a parameter. In other words, public method arguments should be non-null by default.

3. Member variables should be initialized either in all constructors or through the use of the *Builder* pattern. The point here is that objects should be fully constructed before being created and class invariants should be explicitly established.

4. String instances should be initialized to empty strings `""`.

5. List instances should be initialized to empty lists.

Following the aforementioned practices can prevent or at least mitigate the problems coming from null usage. These practices can be ensured manually during code review or automatically using static code analyzers and annotations. An even more radical approach is to forbid the usage of null altogether in the language and observe the effects on the code quality, but this is a topic for a further study.

4.5 Threats to Validity

The internal threats to validity come from the known limitations of static analysis itself on one hand, and the limitations of our heuristics on the other hand.

- As can be seen in Figure 4.5, *NullTracker* cannot trace 3.36% of the variables back to their declarations, as can be seen in Figure 4.5, because we parse and analyze one Java source file at a time. For instance, inherited member variables cannot be discovered.

- *NullTracker* extracts all the *NC-Def-expression*s that appear lexically before the null check regardless of the actual data flow taking all the possible *NC-Def-expression*s into account.

- *NullTracker* cannot detect whether an *N-Comparand* is changed by passing it as a parameter to other methods. Only *NC-Def-expression*s are considered.

- When an *N-Comparand* appears within the same method in multiple null checks, every time it is considered a different *N-Comparand* leading to possible duplicates in the analysis of the *N-Comparand* kinds.

The external threats to validity come from the fact that we only analyzed 810 Java open-source projects. The results might not generalize to all open source projects or to industrial closed-source projects

4.6 Conclusions

In this chapter, we aim at understanding the missing null check problem by empirically analyzing when, how, and why developers use null. We conduct a census of the null checks in Java systems showing that 35% of all conditionals are null checks. Our analysis reveals many bad practices in terms of null usage. Returning null in methods, passing null as arguments, and uninitialized member variables are the most frequent, and questionable, null usage patterns causing the high null check density. However, the main source of null in null checks is returning null in methods.

These findings reveal that not only missing null checks pose a problem, but excessive null usage also does. Although we believe that a disciplined use of null would solve most of the null-related problems, a tool that tells developers where to add null checks and where to remove them would help developers produce better code in terms of maintainability and quality.

In the next chapter, we design an empirical solution to the null-related problem based on the key finding in this chapter: methods that return null are the main cause of null checks.

5 An Empirical Solution to the Missing Null Check Bug

In chapter 3, we show that missing null checks are among the most frequent bug patterns in Java, leading to `NullPointerException`s (NPEs) and causing systems to crash. Then in chapter 4, we show that methods that return null appear to be the main source of this problem, as 70% of null-checked values are returned from method calls. This indicates that developers are often unaware of the nullness of the invoked methods. Method nullness denotes whether a method might return null (nullable) or never returns null (non-null). In this chapter, we propose an empirical approach to deduce method nullness to help developers add null checks in the right place.

5.1 Harvesting the Wisdom of the Crowd

5.1.1 Motivation

Although it might be fairly easy for developers to reason about the nullness of methods in their own projects, analyzing methods in external dependencies is not. When developers want to dereference the return value from methods in external APIs, they often face three scenarios:

1. They assume that the method does not return null, then they deal with an NPE later if it occurs. Although this technique can be used during development with NPEs being detected and fixed through testing, some null dereferences can make it into production code causing system crashes.

2. They defensively add a null check to eliminate the risk of getting an NPE. This technique clutters the code with excessive null checks hindering code comprehension and maintainability [103].

3. They check the source code and read the documentation of the external method and try to reason about its return value, then add a null check when necessary. However, as we show later, documentation is often missing and reverse engineering unknown methods can be complicated.

All three scenarios are suboptimal. Method nullness is an important post-condition of methods for developers to be aware of in order to make the right decisions.

5.1.2 The Nullability Measure

We devise an empirical approach to infer the nullness of library methods relying on the wisdom of the crowd. Given a particular library, we collect and analyze its clients and track how they handle the returned values from method calls, *i.e.*, how often they are checked to be non-null. The results are then aggregated per library method in a *nullability* measure defined as follows:

$$Nullability(Method) = \frac{CheckedDereferences(Method)}{Dereferences(Method)}$$

The nullability measure serves as a proxy for method nullness and expresses the confidence that a particular method returns null. A nullability of zero indicates that a method never returns null, *i.e.*, the returned value is always dereferenced in clients without a null check. The method's nullness is therefore non-null. Conversely, a non-zero nullability indicates that a method is nullable.

In this chapter, we investigate the nullability of methods in Apache Lucene, which is the de facto standard library in Java for full-text searching. We collect 4,197 versions of 186 Lucene clients by exploiting the Maven dependency management system with KOWALSKI, a tool developed particularly for this purpose. We compute the nullability for the 42,092 detected methods belonging to 75 versions of Lucene. We formulate and answer the following research questions:

RQ1: How is nullability distributed and what are the factors affecting it?
We partition the collected methods into methods that expose object state while keeping encapsulation intact (getters) and methods that compute their return value (processors). Then we calculate the nullability distribution for the methods when they are called within Lucene itself (internal usage) and from external clients (external usage). The majority of methods are never checked against null, but those that are account for most of the usage. Getters do not document nullness at all, whereas there is some nullness documentation for nullable processors. While the shapes of the distributions in internal and external usages are similar, in external usage there are many methods checked that are not checked in internal usage, hinting at unnecessary null checks.

RQ2: How does nullability reflect method nullness?
We select 600 methods that range in nullability from 0 to 1, and try to reverse-engineer the nullness by manually inspecting the documentation and the source code of the method. For about half the methods, this context is not broad enough to decide whether a method returns null or not. Nonetheless, for some methods with a nullability of 0 we can deduce that they do not return null, whereas most methods with a non-zero nullability either document that they potentially return null or they contain a `return null` statement.

RQ3: How can the nullability measure be used in practice?
We present an IDE plugin that adds nullability information to the method's documentation as a usage recommendation, and we can add a corresponding annotation to the return type of the method to assist null analysis tools. The nullability documentation enables developers to make a more informed decision about adding or removing a null check. The null analysis points developers at locations where a null check is unnecessary or potentially missing.

5.2 Bytecode Collection and Analysis

5.2.1 Bytecode Collection

We collect binaries of Lucene releases and Lucene clients, so that we can compare internal and external usage of Lucene methods. We find and download the Lucene related binaries by exploiting the *Maven* dependency management system. *Maven* projects declare their dependencies in a meta-data file and binary releases are published on a central package repository. Collecting binaries related to a specific library means that we need to extract a sub-graph of the dependency graph spanned by all projects. First, we need to find the libraries for which we want to collect clients. Second, we need to find clients of the matched libraries. Third, we need to download the binaries we want to analyze.

We implement this client collection process in a tool designed for this purpose called KOWALSKI (detailed in Appendix A). KOWALSKI takes a project name as an input, Lucene in our case, and finds all releases of Lucene by querying *Maven Central Search.*[1] Then it scrapes *mvnrepository* to find projects depending on Lucene.[2] In the third step it uses *Maven* to fetch the clients, including their dependencies, from the package repository. More information about KOWALSKI can be found in the dedicated paper [113].

KOWALSKI finds and collects 7,123 versions of 294 artifacts belonging to 174 groups related to Lucene. The whole process is highly parallelized and completes within two hours on a multi-core machine. The machine runs a 64 bit Ubuntu OS, has 32 cores at 1.4 GHz, and 128 GB of RAM.

[1]`https://search.maven.org/#search%7Cga%7C1%7Cg%3A%22org.apache.lucene%22`

[2]`https://mvnrepository.com/artifact/org.apache.lucene/lucene-core/usages`

5.2.2 Static Analysis

```
1  public void traverse(Node child) {
2    Node parent = child.getParent();
3    child.mark();
4    if (parent != null) {
5      parent.getSibling().mark();
6    }
7  }
```

Listing 5.1: Example method for null dereference analysis.

We analyze the collected binaries to detect which return values of Lucene methods are checked for null and which are not. The static analysis is based on SOOT's null analysis [202, 167], an intra-procedural and path-sensitive data-flow analysis that tracks the nullness of local variables. For each path in the control flow graph it tracks whether a local variable is known to be null, non-null, or unknown. The analysis supports variable aliasing and learns about their nullness from null checks, `instanceof` checks, assignments of newly instantiated objects, and dereferences in field accesses, array accesses, method invocations, *etc.* For instance, in Listing 5.1, SOOT finds the nullness of the variable `parent` to be non-null on line 5 as it was checked in the condition of the wrapping `if`. Our analysis extends the SOOT nullness analysis by extracting the following three features from all method implementations in the collected clients of Lucene. First, we identify the *first* dereference of every value. Those are the locations where a null pointer exception could potentially happen. The method in Listing 5.1 contains three potential null dereferences: `child` on line 2, `parent` and `parent.getSibling()` on line 5. Note that the subsequent dereference of `child` on line 3 can never throw a null pointer exception, as the same value has been dereferenced on line 2 before. Second, we track where the dereferenced value originates from. For `child` this is the parameter, for `parent` this is `child.getParent()`, and for `parent.getSibling()` it is `parent.getSibling()` itself. Third, we note whether or not the dereferenced value is checked for null before, indicating whether its origin is nullable or not. For `child` the nullness of the parameter is unknown, for `parent` we know that `child.getParent()` is non-null as it was checked, and for `parent.getSibling()` it is unknown.

```
if (child.getParent() != null) {
  child.getParent().mark();
}
```

Listing 5.2: Null check of a method return value.

A conservative null analysis only works on local variables. However, not all null checks use a local variable that is checked and dereferenced later. Especially fields accessed through getters and values that are only used once may not be assigned to a local variable, but checked for null using an expression. For example, Listing 5.1 uses a local variable, but Listing 5.2 uses a method call. A conservative analysis detects a potential null dereference of `child.getParent()` on line 2 in Listing 5.2, but for a human reader the same dereference looks safe, as the method call is checked before dereferencing it. Our analysis assumes lexically identical expressions to evaluate to the same value, thus classifying this dereference as safe. This assumption is unsound as methods can return different objects for each invocation, but it is a cheap heuristic compared to a more precise, but computationally much more expensive, object-sensitivity [138].

Different methods might have conceptually different intents that can reveal further insight into what exactly is checked for null, therefore we distinguish between getters and processors. We introduce a heuristic to classify a method as a getter if its name is `get` followed by the name of a field of the declaring class, otherwise we classify the method as a processor. Note that this classification is merely a tag on a method. It does not interfere with the analysis.

The analysis of the 7,123 artifacts takes approximately 12 hours on the same 32-core machine on which we collect them. Of all artifacts, 1,627 Lucene artifacts and 2,570 artifacts of external clients contain a dereference of Lucene methods. External clients include Solr, Elasticsearch, Neo4j, and OrientDB amongst others. Overall we find 292,871 dereferences of return values of 42,092 methods belonging to 75 different Lucene releases.

5.2.3 Validation

We inspect the precision of our analysis and heuristics by manually inspecting a sample of the results and the related code.

Dereference Analysis

We check how precise our potentially unsafe dereference analysis is. Over half of all analyzed methods do not contain potentially unsafe dereferences. About a quarter of the methods contain a single dereference. About 96% of the methods contain no more than eight dereferences. We ignore the remaining methods with more dereferences in the inspection, as they are potentially complex and hard to reason about. For 98% of all dereferences there is only a single possible origin. This allows us to precisely reason about the intention of a null check, as it can only check a single value. We inspect 50 randomly selected methods in which no dereferences are detected and 50 randomly selected methods for which we detected at least one dereference. Our analysis reliably finds 109 potentially unsafe dereferences and their originating values. Only in two methods do we mistakenly classify as potentially unsafe. For the originating values we found no errors. All detected originating values are true possible origins and all possible originating values are detected.

Nullness Analysis

We inspect the accuracy of the nullness analysis. We inspect another 100 randomly selected methods for which we detected at least one dereference. Of 117 dereferences we correctly classified 15 as non-null and 98 as unknown. One instance is misclassified as non-null, another one as unknown. Three dereferences are mistakenly detected as potentially unsafe dereferences. In one instance the dereferenced

```
 1  public void register(XMLReader parser) {
 2    try {
 3      // 1 unsafe, is unsafe
 4      parser.setFeature("...", true);
 5    } catch (SAXException e) {
 6      log.warn("...");
 7    }
 8    try {
 9      // 2 unsafe, but is safe
10      parser.setFeature("...", false);
11    } catch (SAXException e) {
12      log.warn("...");
13    }
14    // 3 unsafe, but is safe
15    parser.setContentHandler(this);
16    // 4 safe, is safe
17    ErrorHandler handler = parser.getErrorHandler();
18  }
```

Listing 5.3: Dereferences of `parser` with dereferences 2 and 3 falsely classified as potentially unsafe. String literals shortened and method trailer omitted from `AbstractTopicMapContentHandler` in `net.ontopia:ontopia-engine:5.3.0`.

field is initialized with a new object immediately before the dereference. As we do not track assignments to fields, but only locals, we cannot detect this initialization. The other two misclassifications are the second and third dereference of the parameter `parser` on line 10 and line 15 in the method listed in Listing 5.3. They are caused by the way the control flow graph is constructed. A try block creates a new branch in the control flow graph and we have to merge two branches after the try statement. In the first branch the try block succeeds, in which case `parser` is now safe to dereference. In the second branch the try block fails and it is assumed that no statement was executed, in which case we do not know if `parser` is safe to dereference on line 10 and line 15. Assuming that pushing the method parameters and receiver of the method `setFeature()` on the stack on line 4 always succeeds (since they are constants and a reference), the actual method invocation is the first operation that could fail. Correcting these misclassifications would require more complex control flow graphs that replicate the execution model of the virtual machine more precisely.

As we have a low rate of non-null instances, we inspect another 100 randomly selected potentially unsafe dereferences classified as non-null. In 95 cases we correctly classify it as non-null. In 5 cases the classification is wrong. They happen in complex loops, try-catch statements, and casting situations. Here we also find interesting patterns where values are checked for null. Many null checks for fields occur within `hashcode()` and `equals()` methods, which can be auto-generated by an IDE like Eclipse. The pattern to check a resource for null before closing it in a finally block is also prevalent. This can be refactored to use Java 7 try-with-resource

Table 5.1: Methods used both internally by Lucene and external clients, reduced to the Lucene major version.

major version	**methods**	**dereferences**
1	6	35
2	57	12,038
3	105	34,128
4	209	64,502
5	117	31,992
6	74	14,022
	568	156,717

blocks[3] to improve code readability. We find null checks for lazy initialization of fields, and setting defaults for null parameters. It is also common to throw an error if a value is null, or exit the method with a `return` or a `return null;`.

Getter Heuristic

We validate our getter heuristic, which classifies methods as getters if their name starts with `get` followed by the name of the field of the class. Therefore, we randomly select 100 methods and inspect them, 50 of which classified as getters, 50 of which are not. Only one method classified as a getter does not return the field with the getter name, but instead computes the return value. Eight methods which are not classified as getters, are getters. They return the field with a name that is very similar to the method name, or do not use a preceding `get`. Most getters directly return the associated field. In some cases the field is lazily initialized. In case of a collection type a shallow copy is sometimes returned. The getter heuristic we use is an under-approximation, as we can precisely identify only getters, but we miss some getters.

5.3 Nullability Distribution

To answer our first research question, *RQ1: How is nullability distributed and what are the factors affecting it?*, we measure the nullability for all 42,092 methods whose return values are dereferenced. In Figure 5.1 we observe that 79% of the methods are never checked for null, 12% are always checked, and 9% are sometimes checked and sometimes not. Note that the nullability peaks at $\frac{1}{2}$, $\frac{2}{3}$, $\frac{3}{4}$, *etc.* are due to methods for which only a handful of dereferences are found.

We look at the dataset from two different perspectives. First, we contrast internal and external usage of Lucene regarding nullability to identify potential disagreement. Second, we distinguish between the usage of getters and processors, as they serve different intents. We filter the dataset to only include methods that are used internally by Lucene itself as well as by external clients. We reduce the selected

[3]`https://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html`

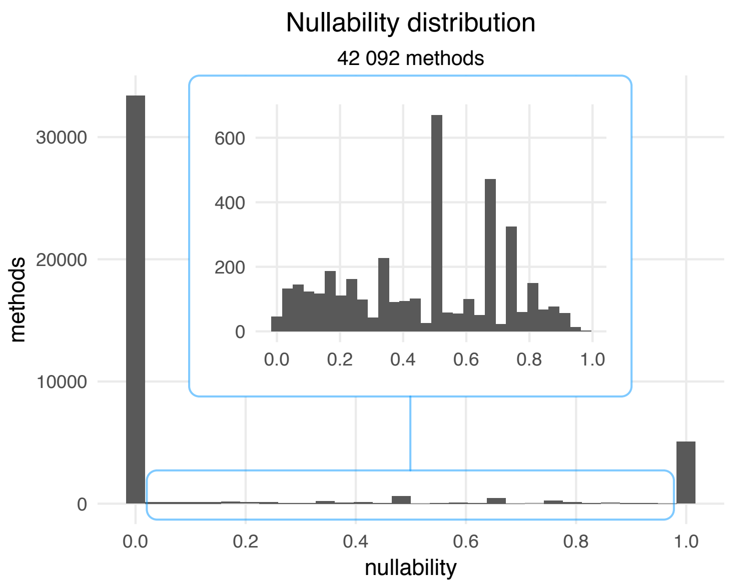


Figure 5.1: The nullability distribution of the 42,092 Lucene methods that are dereferenced. The nested distribution zooms in to the nullability range excluding the extremes $]0, 1[$.

methods belonging to 75 releases to the six major versions of Lucene by their signature (qualified class name, method name, argument types) to increase the support for each method. This reduction assumes that the nullness of a method remains unchanged for all different releases of a major version. In Table 5.1 we find the 568 methods spread across the major releases dereferenced 156,717 times. We partition the dereferences and methods by the getter/processor and external/internal dimensions. For each of the four partitions we find a median of around 30 dereferences per method. The full spectrum ranges from methods only dereferenced once to some dereferenced 5,071 times.

Figure 5.2 shows the distribution of the selected methods over the whole nullability range from 0 (returned value is never checked for null before dereferencing) to 1 (returned value is checked for null before every dereference). We find 86 getters and 482 processors. Across all four partitions most methods are never checked (external getter 90%, external processor 65%, internal getter 95%, internal processor 74%). Some processors are always checked while this category is not significant for getters. For getters the share of checked methods is much smaller than for processors. This could indicate that fields are rarely null. Evidence for this hypothesis is also reported by Chalin *et al.*, as they show that fields are eventually non-null [34]. In the validation of our getter heuristic we also find many cases of lazy initialization of fields, which supports this hypothesis.

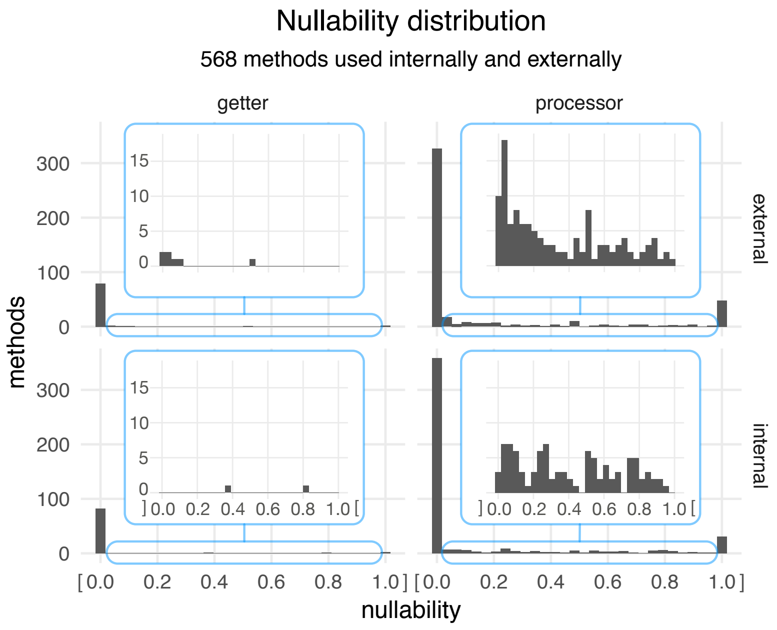


Figure 5.2: The nullability distribution of the 568 Lucene methods that are dereferenced both internally and externally. The nested distribution zooms in to the nullability range excluding the extremes $]0, 1[$.

5.3.1 Documentation

Documentation could be another factor that affects whether or not the return value of method is checked before dereferencing it. For this purpose we collect and process the JavaDoc for all selected methods. If either the word `null` occurs in the return section of the method documentation, or both words `null` and `return` occur in the general method description, we classify the method as having its nullness documented. If the method documentation is found but the heuristic does not pass, we classify the method as not mentioning the nullness of the return value. The measure does not extract whether the documentation states that the method is non-null or nullable, but only if the nullness is mentioned at all. We do not differentiate between internal and external usage, as the documentation is an attribute of a method. Figure 5.3 shows that nullness is not documented at all for getters and often undocumented for processors. Table 5.2 contains the exact counts for each category, as we could not always find the documentation. Regarding nullability, nullness is rarely documented for processors when the nullability is 0. For processors with a nullability higher than zero, the method documentation often makes a statement about the nullness. Getters do not document nullness at all, which can again be seen as support for eventually non-null fields [34].

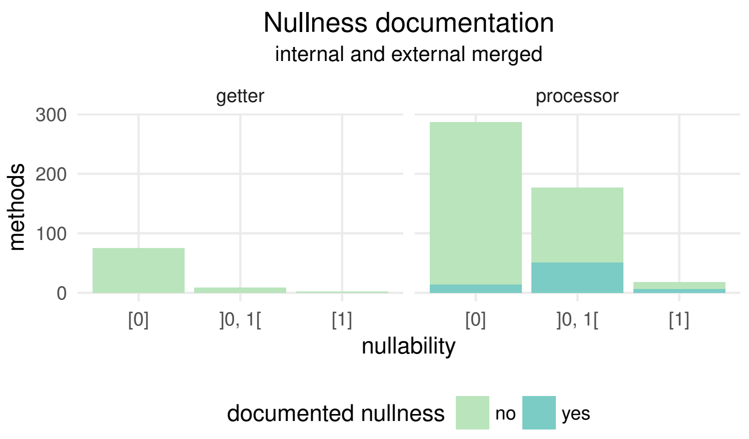


Figure 5.3: Nullness documentation of methods, split into nullability classes. See Table 5.2 for exact counts.

Table 5.2: Nullness documentation of methods, split into nullability classes.

	nullability	class doc. not found	method doc. not found	nullness not mentioned	nullness mentioned
getter	[0]	4	11	60	0
	]0, 1[	0	4	5	0
	[1]	0	1	1	0
processor	[0]	14	63	196	14
	]0, 1[	1	47	78	51
	[1]	0	6	5	7

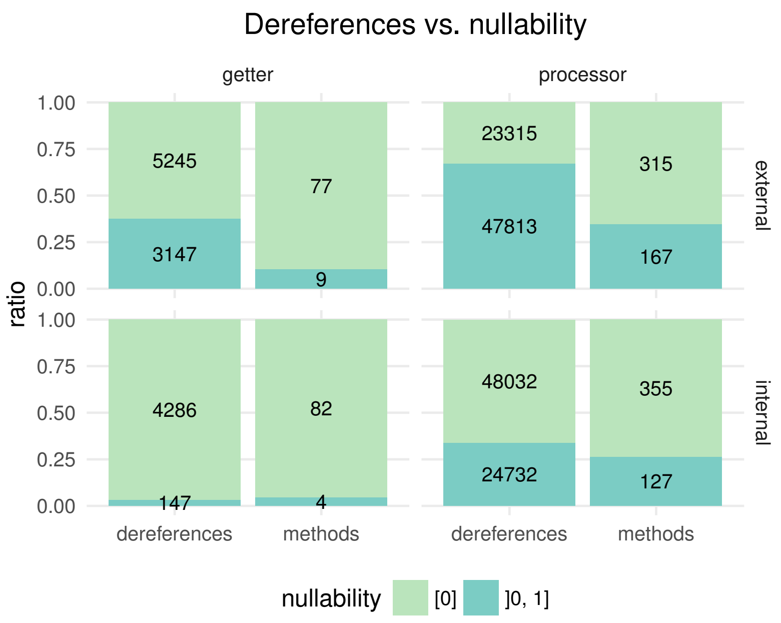


Figure 5.4: Dereferences contrasted with the originating methods of the dereferenced values.

5.3.2 Disagreement between Internal and External Usage

The sets of checked and unchecked methods only partially overlap between internal and external usage. In Figure 5.2 we see that in external usage there are many processors with a low nullability whereas the internal nullability distribution in $]0, 1[$ is more balanced. The disagreement in usage is also visualized by contrasting the number of dereferences and associated methods in Figure 5.4. In external usage there are generally more methods checked and they are more used than unchecked ones. The checked processors are more often dereferenced, relative to all externally dereferenced processors as well as relative to the internal usage. Table 5.3 shows that there are 7 getters and 68 processors that are never checked internally but are checked externally. Only 2 getters and 28 processors are checked internally but not

Table 5.3: Classification differences regarding nullability of methods between internal and external usage.

	nullability (internal → external)	**getter**	**processor**
agree	$[0] \rightarrow [0]$	75	287
	$]0, 1] \rightarrow]0, 1]$	2	99
disagree	$[0] \rightarrow]0, 1]$	7	68
	$]0, 1] \rightarrow [0]$	2	28

Table 5.4: Inspected methods with classification differences regarding nullability between internal and external usage.

	nullability	manual classification		
	(internal → external)	**unknown**	**non-null**	**nullable**
getter	$[0] \rightarrow]0,1]$	4	2	1
	$]0,1] \rightarrow [0]$	2	0	0
processor	$[0] \rightarrow]0,1]$	46	15	7
	$]0,1] \rightarrow [0]$	15	3	10

externally. We inspect the 105 methods where internal and external usage disagree. From the method source code and documentation we try to reverse-engineer the method nullness. We classify a method as non-null if either the documentation states so or all return paths always return an object. We classify a method as nullable if either the documentation states so or at least one return path returns null. The results in Table 5.4 show that for most of those methods our reverse-engineering approach cannot deduce the nullability of the majority of the methods. However, we find that the internal usage is more often in concordance with our classification than external usage. For example, we classify 15 processors that are never checked internally as non-null, whereas 7 are nullable.

We learn three things from the inspection of disagreement of internal and external usage. First, external usage is more defensive than internal usage. Second, reverse-engineering the nullness of a method is non-trivial, even with the source code at hand. Third, internal usage is a better indicator for method nullness than external usage, yet, it is not precise.

5.4 Manual Inspection

We answer our second research question, *RQ2: How does nullability reflect method nullness?*, by manually inspecting Lucene methods across the whole nullability range and trying to reverse-engineer the nullness of the method. For this purpose we partition the nullability spectrum into six intervals and randomly select 100 methods with a nullability in each interval. The nullability intervals are $[0]$, $]0, 0.2]$, $]0.2, 0.4]$, $]0.4, 0.6]$, $]0.6, 0.8]$, and $]0.8, 1]$. We select those methods from the 11,754 Lucene methods with a minimal support of five dereferences, so that their nullability is computed from at least a handful of dereferences. For each method we find their implementation and documentation, then we try to decide on the nullness of the return value. We classify a method as non-null if either the documentation states so or all return paths always return an object. We classify a method as nullable if either the documentation states so or at least one return path returns null. If none of the above conditions holds, we classify the method as unknown. The results of this inspection are shown in Figure 5.5. For about half of the methods the nullness is unknown using our procedure. It is quite rare that the documentation states that the method is non-null. Even rarer are methods for which we can deduce non-nullness from

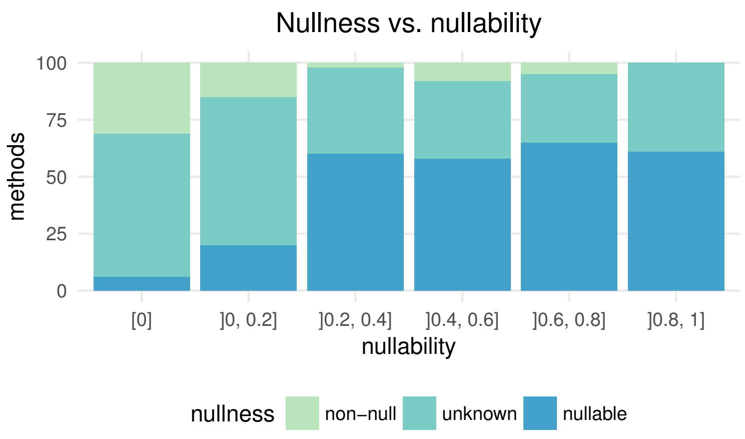
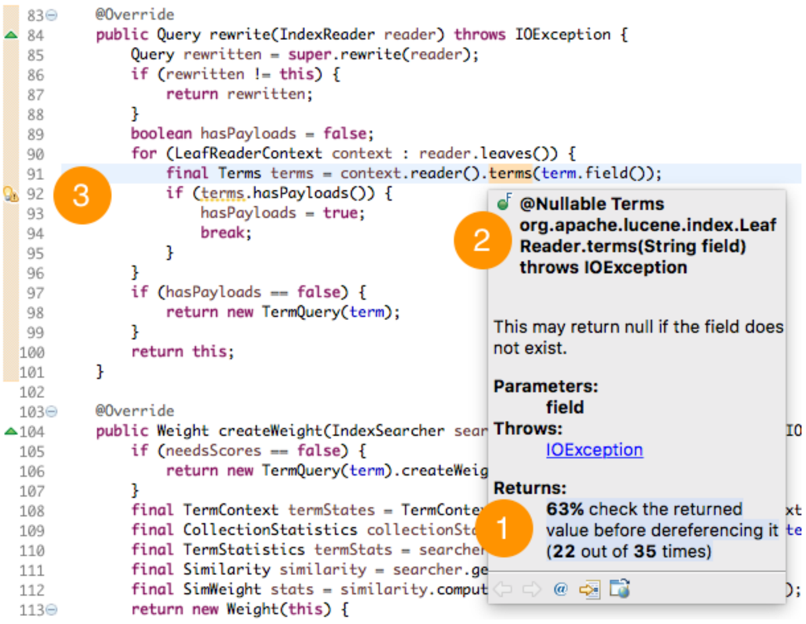


Figure 5.5: Manual classification of sources and documentation of 600 Lucene methods, partitioned by their nullability into five intervals of equal length, except the `[0]` category.

their implementation. In those cases the return value is mostly a collection. We can see that the inferred nullability correlates with the reverse-engineered nullness. The majority of methods with low nullability are non-null. Even more convincing is the situation at the other end of the spectrum. The vast majority with a high nullability are actually nullable. We observe non-null methods with a non-zero nullability, indicating unnecessary null checks, as well as nullable methods with nullability of zero, indicating potentially missing null checks. The direction of causality is unclear though: Do developers check return values because of the documentation or is the nullness documented because null pointers have been observed? Nonetheless, we see that nullability reflects the nullness quite well.

5.5 Transferring Nullability to the IDE

Inspecting the nullability distribution reveals that non-null and nullable methods can be partially separated. Manual inspection suggests that the nullness of methods is often undocumented or non-trivial to reverse-engineer. These findings lead us to the third research question, *RQ3: How can the nullability measure be used in practice?* As a showcase of how nullability information can be harvested, we have implemented an Eclipse plugin that integrates Lucene nullability information in the IDE. The goal of the plugin is to give hints to developers about potential null dereferences and unnecessary null checks. If we can detect potential null pointers, we can avoid bugs. If we can detect unnecessary null checks, we can reduce cyclomatic complexity [126]. The latter point is relevant as we find unnecessary null checks in the manual inspection and null checks account for 35% of all conditional statements [157].



```
    @Override
    public Query rewrite(IndexReader reader) throws IOException {
        Query rewritten = super.rewrite(reader);
        if (rewritten != this) {
            return rewritten;
        }
        boolean hasPayloads = false;
        for (LeafReaderContext context : reader.leaves()) {
            final Terms terms = context.reader().terms(term.field());
            if (terms.hasPayloads()) {
                hasPayloads = true;
                break;
            }
        }
        if (hasPayloads == false) {
            return new TermQuery(term);
        }
        return this;
    }

    @Override
    public Weight createWeight(IndexSearcher sear                  IO
        if (needsScores == false) {
            return new TermQuery(term).createWeig
        }
        final TermContext termStates = TermConte                   xt
        final CollectionStatistics collectionSt                    te
        final TermStatistics termStats = searcher
        final Similarity similarity = searcher.ge
        final SimWeight stats = similarity.comput                  );
        return new Weight(this) {
```

@Nullable Terms org.apache.lucene.index.LeafReader.terms(String field) throws IOException

This may return null if the field does not exist.

Parameters:
field
Throws:
IOException
Returns:
63% check the returned value before dereferencing it (**22** out of **35** times)

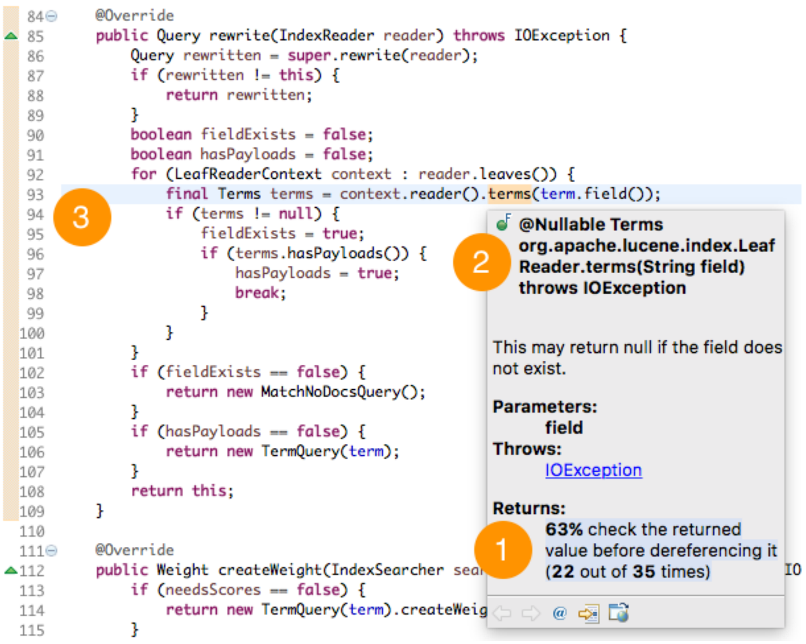
Figure 5.6: Augmented JavaDoc with nullability (1), nullness annotation (2) for a Lucene method with a high nullability giving a warning for a potential bug (3).

First, the plugin adds the nullability information to the JavaDoc of a method, with both confidence and support. The nullability gives the developer the static frequency of a null return value. Confidence and support are a proxy for the trust that can be put into the nullability. If the nullability is computed from only a handful of samples, it might not be trustworthy as not all usage scenarios of the method might be covered. Figure 5.6 shows our plugin applied on a code excerpt of Elasticsearch 2 for which a null pointer bug was reported.[4] The cause of the bug is the unconditional dereference of the return value of the Lucene method `LeafReader.terms(String)`. Our plugin adds nullability documentation with a blue background at position (1). Besides the nullability of 63% it also tells us that this is computed from 22 out of 35 dereferences that were checked for null. Documenting nullness is certainly important, but it still requires developers to read the documentation to detect the nullness. For example, in Figure 5.6 the nullness is described in the original documentation, but the developer nevertheless missed it. To raise awareness for potential nullness, the tools should give more obvious hints to developers about potential null pointers.

Second, the plugin generates external nullness annotations[5] from the nullability measured for Lucene methods. For methods with a nullability of zero we gener-

[4]`https://github.com/elastic/elasticsearch/pull/12495/commits/d7491515b21fb4b3c94956c75bcb74b8a5c863ae`

[5]`http://help.eclipse.org/neon/index.jsp?topic=/org.eclipse.jdt.doc.user/tasks/task-using_external_null_annotations.htm`



```java
@Override
public Query rewrite(IndexReader reader) throws IOException {
    Query rewritten = super.rewrite(reader);
    if (rewritten != this) {
        return rewritten;
    }
    boolean fieldExists = false;
    boolean hasPayloads = false;
    for (LeafReaderContext context : reader.leaves()) {
        final Terms terms = context.reader().terms(term.field());
        if (terms != null) {
            fieldExists = true;
            if (terms.hasPayloads()) {
                hasPayloads = true;
                break;
            }
        }
    }
    if (fieldExists == false) {
        return new MatchNoDocsQuery();
    }
    if (hasPayloads == false) {
        return new TermQuery(term);
    }
    return this;
}

@Override
public Weight createWeight(IndexSearcher sea
    if (needsScores == false) {
        return new TermQuery(term).createWeig
    }
```

@Nullable Terms org.apache.lucene.index.LeafReader.terms(String field) throws IOException

This may return null if the field does not exist.

Parameters:
field
Throws:
IOException

Returns:
63% check the returned value before dereferencing it (**22** out of **35** times)

Figure 5.7: Augmented JavaDoc with nullability (1), nullness annotation (2) for a Lucene method with a high nullability giving no warning for a potential bug (3) due to a missing null check.

ate a `@NonNull` annotation. For methods with a non-zero nullability we generate a `@Nullable` annotation. These annotations are stored as Eclipse External Annotations that do not require bytecode manipulation of the Lucene binary, but they are stored alongside the library and linked through the Eclipse project configuration. The internal Eclipse JDT Null Analysis[6] considers these external annotations. In Figure 5.6 the annotation is visible in the documentation at position (2). The warning generated for the potential null pointer is at position (3). The bug in Elasticsearch was eventually fixed by guarding the dereference with a null check (see Figure 5.7). After the bug fix, the null analysis issues no warning anymore, as the dereference is now safe. Detecting potential bugs is not the only use case for the null analysis, as it can also detect dead code caused by unnecessary null checks.

Third, our plugin can be configured to only generate annotations for methods with nullability within a certain range. For example we can set the upper limit for the generation of non-null annotations to 0 and the lower limit for nullable annotations to 0.8. For all methods with a nullability in between 0 and 0.8 no annotations are generated and conversely the null analysis does not generate any warnings. This tuning can be used to reduce the number of warnings.

In the future, we plan to evaluate the IDE plugin with developers in the industry

[6]`https://wiki.eclipse.org/JDT_Core/Null_Analysis`

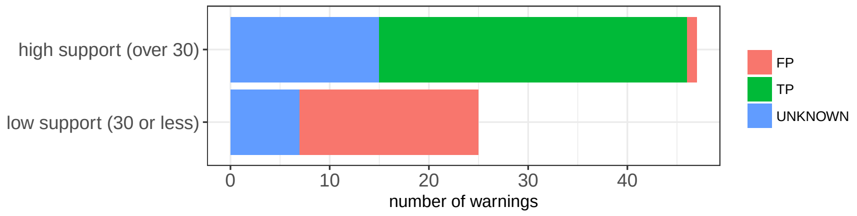


Figure 5.8: The accuracy of the nullness warnings on Elasticsearch code.

to assess its usefulness and further improve it. The plugin sources are available online,[7] including a sample project and sample nullability data.

5.5.1 Evaluation

We inspect the warnings generated by our plugin on the sources of the Elasticsearch project at version 1.7.3 from October 15th, 2015. This is the newest version of Elasticsearch we analyzed and uses Lucene 4, for which we collect the most usage data. Our analysis identifies 764 potentially unsafe dereferences of Lucene methods. The plugin is configured to generate `@Nullable` annotations for methods with a nullability of at least 0.2, as our manual inspection reveals that there is a high agreement of nullability and nullness above this theshold. For methods with a nullability of at most 0.1, the plugin generates `@NonNull` annotations, so that a few unnecessary null checks detected in the analysis still lead to a generated annotation. When our plugin is activated, we find 72 problems, caused by 21 different Lucene methods. We classify the warnings according to the nullness of the causing method using the same procedure as in section 5.4. Figure 5.8 shows the results of the manual inspection. Interestingly, warnings generated from methods with high support have only one confirmed false positive, but 31 confirmed true positive. On the other hand, low support methods do not generate any confirmed true positive. We conclude that nullability is correct for methods with a high support. In other wards, we can trust the wisdom of the crowd, with respect to inferring nullness, only when the crowd is large enough.

5.6 Threats To Validity

5.6.1 Construct Validity

The results we gain are only as accurate as the analysis. We validate all parts of the analysis we suspect to be critical. We report some wrongly classified samples that hint at bugs in the analysis implementation. Nonetheless, most samples are correctly classified. Our confidence in the analysis is supported by the manual inspection

[7] `https://github.com/maenu/method-nullability-plugin`

which reveals that methods with a high nullability are mostly nullable, whereas many methods with a low nullability are non-null.

To inspect where internal and external usage disagree on nullability we group methods by their Lucene major version. This is an over-approximation, as Lucene does not strictly adhere to semantic versioning and introduces some breaking changes in minor releases. If a method changes from non-null to nullable without changing its signature between two minor versions of the same major release, this approximation mixes the nullability for two different methods.

Based on nullability we generate nullness annotations that lead to warnings. Those warnings can also be false positives in some cases. We find both checked methods that never return null according to the method's documentation and implementation, as well as unchecked methods that document a potential nullness. The nullability measure of a method hints at the frequency of null checked return values. Nullability cannot express in which context a method returns null or not. To reduce the number of false positive warnings, our plugin can be configured to generate annotations only for methods in a specific range.

5.6.2 Generalizability

The composition of our dataset influences the generalizability of our results. KOWALSKI collects only Lucene and other OSS projects that depend on Lucene and are published on Maven Central. First, this excludes all closed source projects. If given access to a company repository, KOWALSKI can also be used to collect a dataset of clients of a company-internal library. The static analysis can be reapplied as well. Second, all open source projects that are not published on Maven Central are excluded. There are other popular Maven repositories that may contain other Lucene clients, for example jcenter[8] and clojars.[9] However, Maven Central is a large repository that serves 1,935,045 versions of 185,693 artifacts.[10] Some software projects are not published in a repository at all. We lack a measure to estimate how many Lucene clients are only distributed as sources, for example on GitHub. As our analysis is tailored to run on binaries, it would require a build of these projects. Package repositories are primarily used to distribute reusable libraries, therefore our dataset has a strong bias towards libraries as clients. Libraries may use Lucene differently than projects further down the dependency hierarchy. Third, we only analyze the Lucene ecosystem. The results may not be generalizable to other ecosystems.

5.7 Conclusions

In this chapter, we demonstrate an empirical approach to detect null-related bugs. We harvest the wisdom of the crowd to infer the nullness of methods of Apache Lucene. The inference is based on nullability that measures the frequency of null

[8] `https://bintray.com/bintray/jcenter`
[9] `https://clojars.org/`
[10] `https://search.maven.org/#stats`, date of access May 3, 2017

checks of the method return value before it is dereferenced. We find that most methods are used as if they were non-null, *i.e.*, they never return null. The nullness of the non-null methods is rarely documented, nullable methods document the nullness more often. Getters are generally non-null and do not document nullness at all. We present an IDE plugin that utilizes method nullability to augment documentation and to integrate with static analysis tools. The plugin points to potential null dereferences and unnecessary null checks. With tuneable configurations, developers can limit the number of false positives and detect potential problems in the code with high accuracy, leading to an efficient and adaptable bug detection for this family of bugs.

In chapter 3, chapter 4, and this chapter, we show how we employ empirically-grounded analysis to discover the missing null bug pattern, analyze it, and build an efficient bug detector for it. In the following chapters, we demonstrate how we empirically optimize bug prediction as a machine learning model and a software quality tool.

6 Optimizing Bug Prediction by Applying Feature Selection

There are several optimizations that can be applied to a bug predictor from the machine learning perspective. The first optimization we explore is feature selection,[1] which is the process of determining the smallest subset of features that exhibits the strongest effect. Feature selection often decreases model complexity and increases prediction accuracy, especially when there is a large number of features. There are two well-known types of feature selection methods: filters and wrappers. Filters select features based on their relevance to the response variable independently of the prediction model. Wrappers select features that increase the prediction accuracy of the model.

While there has been extensive research on the impact of feature selection on prediction models in different domains, our investigation reveals that it is a rarely studied topic in the domain of bug prediction. Few studies explore how feature selection affects the accuracy of classifying software entities into buggy or clean [187, 61, 33, 107, 204, 100, 99, 64], but to the best of our knowledge no dedicated study exists on the impact of feature selection on the accuracy of predicting the number of bugs. As a result of this research gap, researchers often overlook feature selection and provide their prediction models with all the metrics they have on a software project or in a dataset. We argue that feature selection is an important step in the bug prediction pipeline and its application might alter previous findings in the literature, especially when it comes to comparing different machine learning models or different software metrics.

[1]Feature selection is also known as variable selection, attribute selection, and variable subset selection.

In this chapter, we treat bug prediction as a regression problem where a bug predictor predicts the number of bugs in software entities as opposed to classifying software entities as buggy or clean. We investigate the impact of filter and wrapper feature selection methods on the prediction accuracy of five machine learning models: K-Nearest Neighbour, Linear Regression, Multilayer Perceptron, Random Forest, and Support Vector Machine. More specifically, we carry out an empirical study on five open source Java projects: Eclipse JDT Core, Eclipse PDE UI, Equinox, Lucene, and Mylyn to answer the following research questions:

RQ1: How does feature selection impact the prediction accuracy? Our results show that applying correlation-based feature selection (CFS) improves the prediction accuracy in 32% of the experiments, degrades it in 24%, and keeps it unchanged in the rest. On the other hand, applying the wrapper feature selection method improves prediction accuracy by up to 33% in 76% of the experiments and never degrades it in any experiment. In fact, after applying wrapper feature selection, the simple Linear Regression performs at least as well as more advanced models like Random Forest and Support Vector machine.

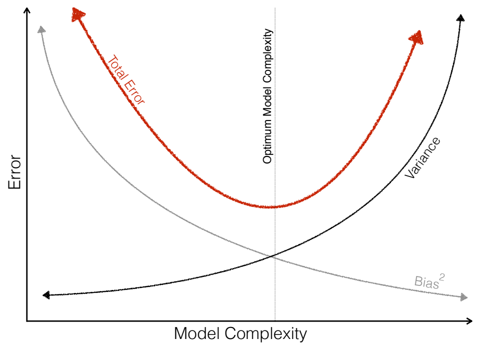
RQ2: Are wrapper feature selection methods better than filters? Wrapper feature selection methods are consistently either better than or similar to CFS. Applying wrapper feature selection eliminates noisy and redundant features and keeps only relevant features for that specific project, increasing the prediction accuracy of the machine learning model.

RQ3: Do different methods choose different feature subsets? We realize there is no optimal feature subset that works for every project and feature selection should be applied separately for each new project. We find that not only different methods choose different feature subsets on the same projects, but also the same feature selection method chooses different feature subsets when applied on different projects. Interestingly however, all selected feature subsets include a mix of change and source code metrics.

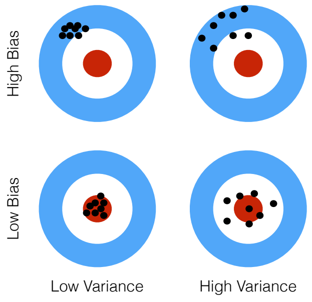
6.1 Technical Background

Trained on bug data and software metrics, a bug predictor is a machine learning model that predicts defective software entities using software metrics. The software metrics are called the independent variables or the features. The prediction itself is called the response variable or the dependent variable. If the response variable is the absence/presence of bugs then bug prediction becomes a classification problem and the machine learning model is called a classifier. If the response variable is the number of bugs in a software entity then bug prediction is a regression problem and the model is called a regressor.

Feature selection is an essential part in any machine learning process. It aims at removing irrelevant and correlated features to achieve better accuracy, build faster models with stable performance, and reduce the cost of collecting features. Model



(a)



(b)

Figure 6.1: This illustrative figure shows the relationship between model complexity and model error in (a) and demonstrates the concepts of variance and bias in (b) [22].

error is known to be increased by both noise [8] and feature multicollinearity [3]. Different feature selection algorithms eliminate this problem in different ways. For instance, correlation based filter selection chooses features with high correlation with the response variable and low correlation with each other.

Also when we build a prediction model, we often favour less complex models over more complex ones due to the known relationship between model complexity and model error, as shown in Figure 6.1(a). Feature selection algorithms try to reduce model complexity down to the sweet spot where the total error is minimal. This point is called the optimum model complexity. Model error is computed via the mean squared error (MSE) as: $MSE = \frac{1}{N}\sum_{i=1}^{N}(\hat{Y}_i - Y_i)^2$ where $\hat{Y}_i$ is the predicted value and Y_i is the actual value. MSE can be decomposed into model bias and model variance as:
$MSE = Bias^2 + Variance + IrreducibleError$ [78]

As seen in Figure 6.1(b), Bias is the difference between the average prediction of our model to the true unknown value we are trying to predict. Variance is the variability of a model prediction for a given data point. As can be seen in Figure 6.1(a), reducing model complexity increases the bias but decreases the variance. Feature selection sacrifices a little bit of bias in order to reduce variance and, consequently, the overall MSE.

Every feature selection method consists of two parts: a search strategy and a scoring function. The search strategy guides the addition or removal of features to the subset at hand and the scoring function evaluates the performance of that subset. This process is repeated until no further improvement is observed.

Table 6.1: The CK Metrics Suite [36] and other object-oriented metrics included as the source code metrics in the bug prediction dataset [38]

Metric Name	Description
CBO	Coupling Between Objects
DIT	Depth of Inheritance Tree
FanIn	Number of classes that reference the class
FanOut	Number of classes referenced by the class
LCOM	Lack of Cohesion in Methods
NOC	Number Of Children
NOA	Number Of Attributes in the class
NOIA	Number Of Inherited Attributes in the class
LOC	Number of lines of code
NOM	Number Of Methods
NOIM	Number of Inherited Methods
NOPRA	Number Of PRivate Atributes
NOPRM	Number Of PRivate Methods
NOPA	Number Of Public Atributes
NOPM	Number Of Public Methods
RFC	Response For Class
WMC	Weighted Method Count

Table 6.2: The change metrics proposed by Moser *et al.* [143] included in the bug prediction dataset [38]

Metric Name	Description
REVISIONS	Number of reversions
BUGFIXES	Number of bug fixes
REFACTORINGS	Number Of Refactorings
AUTHORS	Number of distinct authors that checked a file into the repository
LOC_ADDED	Sum over all revisions of the lines of code added to a file
MAX_LOC_ADDED	Maximum number of lines of code added for all revisions
AVE_LOC_ADDED	Average lines of code added per revision
LOC_DELETED	Sum over all revisions of the lines ofcode deleted from a file
MAX_LOC_DELETED	Maximum number of lines of code deleted for all revisions
AVE_LOC_DELETED	Average lines of code deleted per revision
CODECHURN	Sum of (added lines of code - deleted lines of code) over all revisions
MAX_CODECHURN	Maximum CODECHURN for all revisions
AVE_CODECHURN	Average CODECHURN for all revisions
AGE	Age of a file in weeks (counting backwards from a specific release)
WEIGHTED_AGE	Sum over age of a file in weeks times number of lines added during that week normalized by the total number of lines added to that file

6.2 Motivation

In this section, we shortly discuss the importance of predicting the number of bugs in software entities. Then, we highlight the impact of feature selection on bug prediction and particularly motivate the need for studying the wrapper methods.

6.2.1 Regression vs Classification

Most of the previous research treats bug prediction as a classification problem where software entities are classified as either buggy or clean. In a recent study of existing bug prediction techniques only 5 out of the 64 surveyed papers apply regression techniques to predict the number of bugs [121]. However, software bugs are not evenly distributed and tend to cluster [160], and some software entities commonly have larger numbers of bugs compared to others. A classifier would draw no distinction between a module containing ten bugs and one with a single bug, resulting in a less useful feedback for quality assurance. Predicting the number of bugs in each entity provides more insights about the quality of these software entities [152]. First, it helps in prioritizing software entities to increase the efficiency of related development tasks such as testing and code reviewing [98]. This is an important quality of a bug predictor especially for cost-aware bug prediction [129, 6, 94, 105, 79]. In fact, predicting the number of bugs in software entities and then ordering these entities based on bug density is the most cost-effective option [156]. Second, software quality assurance teams may be interested in ranking software entities based on other factors such as bug severity or the number of users affected by the bugs. Although these aspects are out of the scope of this study, regression would also fit these problems better than classification because of the numeric response variable. Finally, the defectiveness of a software entity can be derived from the number of bugs, but not vice versa [170].

6.2.2 Dimensionality Reduction

When the dimensionality of data increases, distances grow more and alike between the vectors and it becomes harder to detect patterns in data. This phenomenon is known as the curse of dimensionality [19]. Feature selection not only eliminates the confounding effects of noise and feature multicollinearity, but also reduces the dimensionality of the data to improve accuracy. However, feature selection does not seem to be considered as important as it should be in the field of bug prediction. For instance, only 25 out of the 64 studied techniques in a recent research apply feature selection before training a machine learning model [121]. Interestingly, only 2 out of the 25 are applied to bug prediction as a regression problem.

6.2.3 Filters vs Wrappers

Feature selection methods are of two types: wrappers and filters [106]. With wrappers, the scoring function is the accuracy of the prediction model itself. Wrappers look for the feature subset that works best with a specific machine learning model. They are called wrappers because the machine learning algorithm is wrapped into the selection procedure. With filters (*e.g.*, CFS, InfoGain, PCA), the scoring function is independent of the machine learning model. They are called filters because the attribute set is filtered before the training phase. Generally, filters are faster than wrappers but less powerful because wrappers address the fact that different learning algorithms can achieve best performance with different feature subsets.

Bug prediction datasets are relatively small regardless of the size of software systems. In terms of machine learning, the problem size is composed of the number of features and the number of data points. The number of features does not depend on the size of the system and the number of data points depends on the granularity of prediction (*e.g.*, file, class, package). If defect prediction is done on the file level, then the number of files is what matters and not the LOC, and usually this number is small. For instance, the Linux kernel has fewer than 20'000 C source files, and yet, it is one of the largest systems in the world. This means that although wrappers are more resource intensive, they are easily applicable to bug prediction. Nevertheless, our investigation reveals that wrapper methods are rarely applied in the literature. Only 2 out of 64 major bug prediction studies apply wrapper methods [121].

6.3 Empirical Study

In this section, we investigate the effect of feature selection on the accuracy of predicting the number of bugs in Java classes. Specifically, we compare five widely-used machine learning models applied to five open source Java projects to answer the following research questions:
RQ1: How does feature selection impact the prediction accuracy?
RQ2: Are wrapper feature selection methods better than filters?
RQ3: Do different methods choose different feature subsets?

Table 6.3: Details about the systems in the studied dataset, as reported by D'Ambros *et al.* [38]

System	Release	KLOC	#Classes	% Buggy	% classes with more than one bug
Eclipse JDT Core	3.4	≈ 224	997	$\approx 20\%$	$\approx 7\%$
Eclipse PDE UI	3.4.1	≈ 40	1,497	$\approx 14\%$	$\approx 5\%$
Equinox	3.4	≈ 39	324	$\approx 40\%$	$\approx 15\%$
Mylyn	3.41	≈ 156	1,862	$\approx 13\%$	$\approx 4\%$
Lucene	2.4.0	≈ 146	691	$\approx 9\%$	$\approx 3\%$

6.3.1 Experimental Setup

Dataset

We adopt the "Bug Prediction Dataset" provided by D'Ambros *et al.* [38] which serves as a benchmark for bug prediction studies. We choose this dataset because it is the only dataset that contains both source code and change metrics at the class level, in total 32 metrics listed in Table 6.1 and Table 6.2; and also provides the number of post-release bugs as the response variable for five large open source Java systems listed in Table 6.3. The other dataset that has the number of bugs as a response variable comes from the PROMISE repository, but contains only 21 source code metrics [93].

Prediction Models

We use Multi-Layer Perceptron (MLP), Random Forest (RF), Support Vector Machine (SVM), Linear Regression (LR), and an implementation of the k-nearest neighbour algorithm called IBK. Each model represents a different category of statistical and machine learning models that is widely used in the bug prediction research [121]. Table 6.4 provides a brief description of these machine learning models.

We use the correlation-based feature selection (CFS) method [73], the best [35, 64] and the most commonly-used filter method in the literature [121]. For the wrapper feature selection method we use the corresponding wrapper applicable to each prediction model. In other words, we use MLP wrapper for MLP, RF wrapper for RF, SVM wrapper for SVM, LR wrapper for LR, and IBK wrapper for IBK. Every feature selection method also needs a search algorithm. We use the *Best First* search algorithm which searches the space of feature subsets using a greedy hill-climbing procedure with a backtracking facility.

We use the Weka data mining tool [72] to build prediction models for each project in the dataset. Following an empirical method similar to the method of Hall and Holmes [74], we apply each prediction model to three feature sets: the full set, the subset chosen by CFS, and the subset chosen by the wrapper. The prediction model is built and evaluated following the 10-fold cross validation procedure. The wrapper feature selection is applied using a 5-fold cross validation on the training set of each fold, then the best feature set is used. The CFS algorithm is applied on the whole training set of each fold. Then the whole process is repeated 30 times. We evaluate the predictions by means of the root mean squared error (RMSE). In total, we have 25 experiments. Each experiment corresponds to a specific project and a specific prediction model trained on the three feature sets.

We use the default hyperparameter (*i.e.*, configuration) values of Weka 3.8.0 for the used machine learning models. Although hyperparameters can be tuned [193, 154], we do not perform this optimization because we want to isolate the effect of feature selection. Besides, Linear Regression does not have hyperparameters and the gained improvement of optimizing SVM and RF is negligible [193, 154].

Table 6.4: A brief explanation of the used machine learning models.

Model	Description
Multi-Layer Perceptron	A feed forward neural network with one input layer, one hidden layer, and one output layer.
Random Forest	An ensemble technique that comprises a multitude of decision trees generated to fit random samples from the training set. The prediction of the unseen sample is the average of the predictions of the random trees.
Support Vector Machine	A geometrically-inspired machine learning technique that can, in the feature space, separate data points by choosing the best separating hyperplane (*i.e.*, with highest separation margin that is).
Linear Regression	The simplest regression algorithm. It tries to fit the training data on a line then uses this line as a prediction function for unseen data samples.
IBK	An implementation of the k-nearest neighbour algorithm where the value of the dependent variable of a new data point is calculated based on the average values of the dependent variable of the k-nearest data points in the feature space.

6.3.2 Results

Figure 7.1 shows standard box plots for the different RMSE values obtained by the different feature sets per prediction model per project. Within each box, there exists 50% of that specific population.[2] We can see that the wrapper populations are almost always lower than the full set ones, have smaller boxes, and have fewer outliers. This means that applying the wrapper gives better and more consistent predictions. On the other hand, we cannot make any observations about applying CFS because the difference between the CFS populations and the full set populations are not apparent.

While box plots are usually good to get an overview of the different populations and how they compare to each other, they do not provide any statistical evidence. To get more concrete insights, we follow the two-stage statistical test: Kruskal-Wallis + Dunn post-hoc analysis, both at the 95% confidence interval. We apply the Kruskal-Wallis test on the results to determine whether different feature subsets have different prediction accuracies (*i.e.*, different RMSE). Only when this test indicates that the populations are different, can we quantify such differences with a post-hoc analysis. We perform Dunn post-hoc pairwise comparisons and analyze the effect size between each two populations. Figure 6.3 shows on the y-axis the detailed effect size between the two compared RMSE populations on the x-axis. In this plot, there are two possible scenarios:

1. The Kruskal-Wallis test indicates that there is no statistical difference between the populations. Then all the bars are red to show that there is no effect between any two populations.

[2]By population we mean the RMSE values of a specific experiment with a specific feature set. Each population consists of $10 \times 50 = 500$ data items (10-fold cross validation done 50 times)

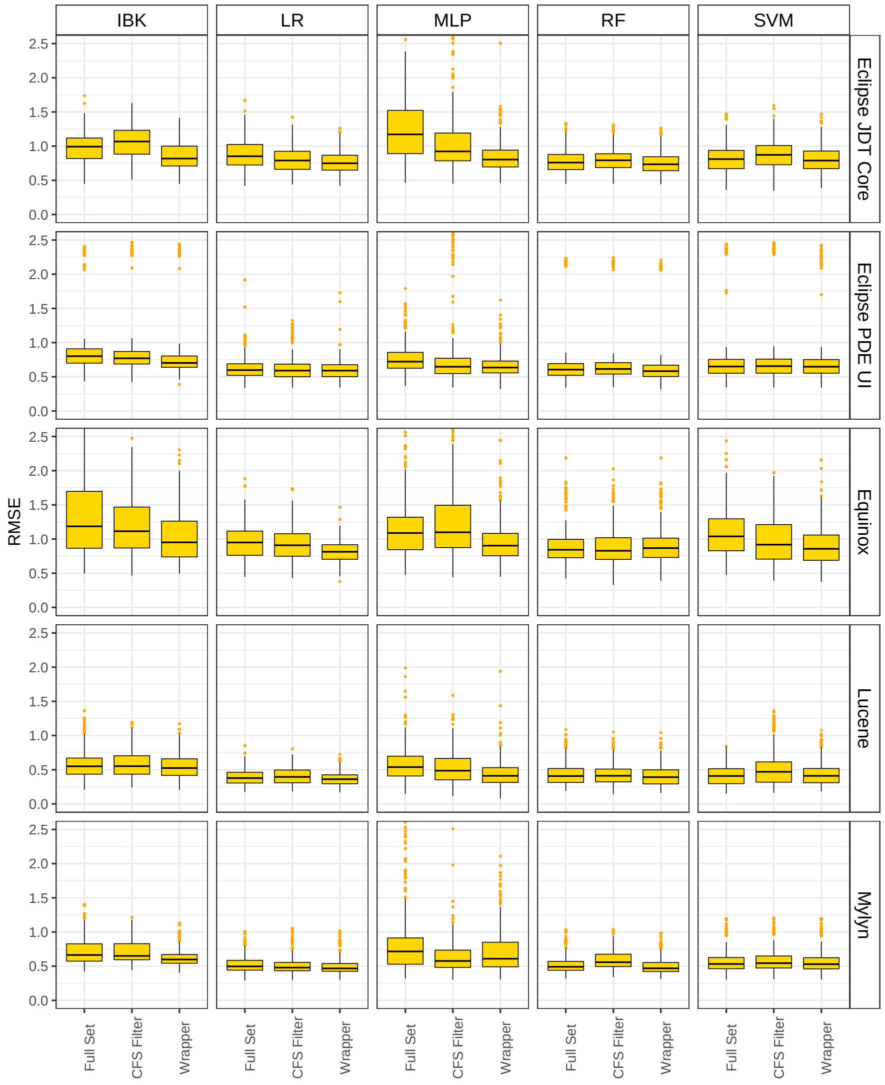


Figure 6.2: Boxplots of all the experiments in our empirical study. The y-axis represents the root mean squared error (RMSE). For each project/model, we examine three feature sets: the full set, the subset chosen by the CFS filter, and the subset chosen by the wrapper corresponding to the model.

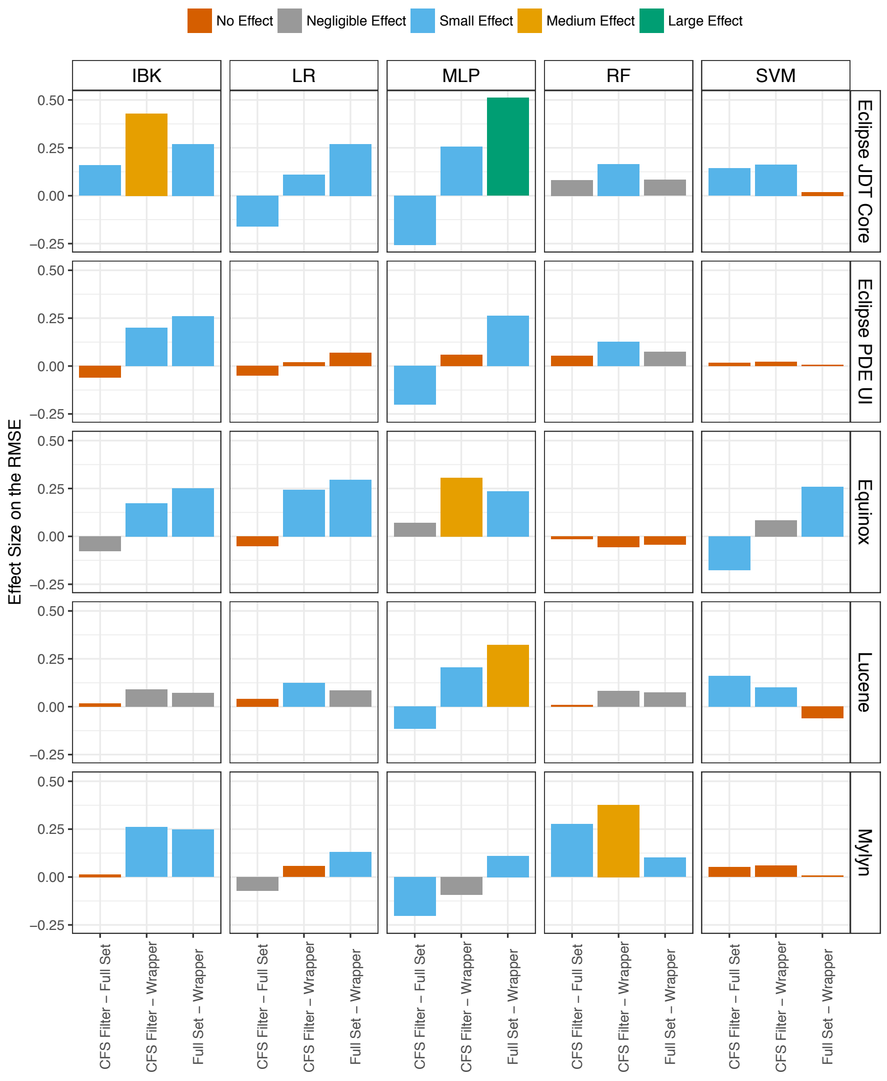


Figure 6.3: This figure shows the bar plots of the effect size of the Dunn post-hoc analysis, which is carried out at the 95% confidence interval. The x-axis indicates the pairwise comparison and the y-axis indicates the effect size. The bars are color-coded. If the bar is red, this means that the difference is not statistically significant. Grey means that there is a statistical significant difference, but the effect is negligible. Blue, golden, and green indicate a small, medium, and large statistically significant effect, respectively.

2. The Kruskal-Wallis test indicates a statistically significant difference between the populations. Then the color of the bars encode the pairwise effect size. Red means no difference and the two populations are equivalent. Grey means that there is a significant difference but can be ignored due to the negligible effect size. Blue, golden, and green mean small, medium, and large effect size respectively.

To see how feature selection methods impact the prediction accuracy (*RQ1*), we compare the RMSE values obtained by applying CFS and wrappers with those obtained by the full feature set. We observe that the RMSE value obtained by the CFS feature subset is statistically lower than the full set in 8 experiments (32%),[3] statistically higher in other 6 experiments (24%),[4] and statistically equivalent in 11 experiments (44%).[5] Although CFS can decrease the RMSE by 24% on average (MLP with Mylyn), it can increase it by up to 24% (SVM with Lucene). We also notice that, applying CFS is not consistent within experiments using the same model. It does not always improve, or always degrade, or always retain the performance of any model throughout the experiments. We conclude that CFS is unreliable and gives unstable results. Furthermore, even when CFS reduces the RMSE, the effect size is at most small.

On the other hand, the RMSE value of the wrapper feature subset is statistically lower than that of the full set in 19 experiments (76%) and statistically equivalent in the rest. Applying the wrapper feature selection method can decrease RMSE of a model by up to 33% (MLP with Eclipse JDT). We also observe that the impact of the wrapper feature selection method on the accuracy is different from one model to another. It has a non-negligible improvement on the prediction accuracy of IBK, LR, MLP, RF, and SVM in 80%, 60%, 100%, 20%, and 20% of the experiments, respectively. This is due to the fact that different machine learning models are different in their robustness against noise and multicollinearity. MLP, IBK, and LR were improved significantly almost always in our experiments. On the other hand, SVM and RF were not improved as often, because they are known to be resistant to noise, especially when the number of features is not too high. RF is an ensemble of decision trees created by using bootstrap samples of the training data and random feature selection in tree induction [25]. This gives RF the ability to work well with high-dimensional data and sift the noise away. The SVM algorithm is also designed to operate in a high-dimensional feature space and can automatically select relevant features [88]. In fact, this might be the reason behind the proven record of Random Forest and Support Vector Machine in bug prediction [70, 47]. However, in our experiments, applying the wrapper brings the RMSE value of Linear Regression to equivalent or even lower levels compared to that of Random Forest. In fact, after applying feature selection by wrappers, Linear Regression is the best model. This is particularly surprising and interesting because it shows that applying feature selection makes the simplest regression model perform potentially better than advanced

[3]negative non-red effect size in Figure 6.3
[4]positive non-red effect size in Figure 6.3
[5]red effect size in Figure 6.3

ones in bug prediction.

The wrapper method is statistically better than CFS in 18 experiments, statistically equivalent in 6 experiments, and worse in one experiment, but with a negligible effect size. These results along with the fact that CFS sometimes increases the RMSE, clearly show that the wrapper selection method is a better choice than CFS (*RQ2*).

Figure 6.4 shows the details about the features selected by each method using the whole data of each project in the dataset. To answer the third research question (*RQ3*), we use the Fleiss' kappa statistical measure [57] to evaluate the level of agreement between the different feature selection methods for each project and the level of agreement of each feature selection method over the different projects. The Fleiss' kappa value, called k, is interpreted as described in Table 6.5.

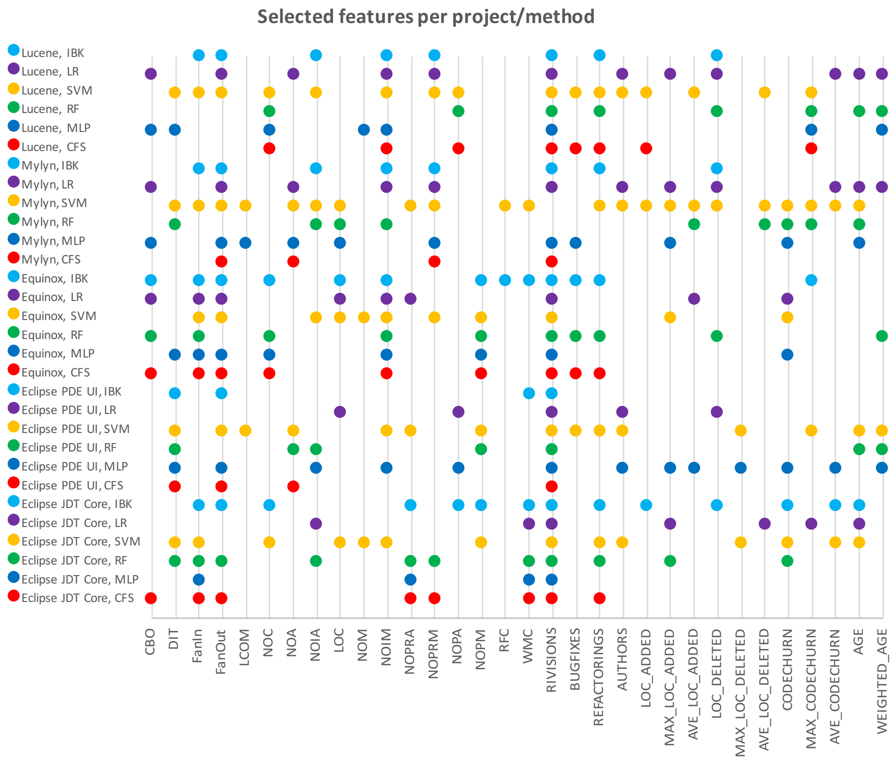
Table 6.5: The interpretation of the possible values of Fleiss-kappa (k)

k	Agreement
< 0	Poor
0.01 - 0.20	Slight
0.21 - 0.40	Fair
0.41 - 0.60	Moderate
0.61 - 0.80	Substantial
0.81 - 1.00	Almost perfect

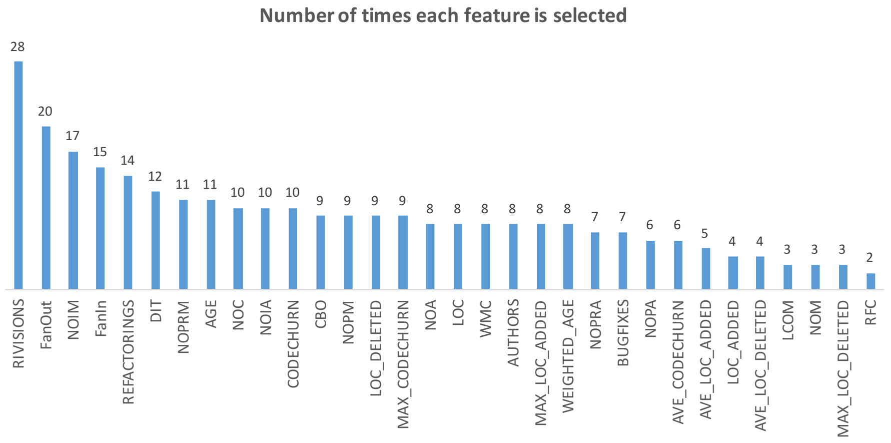
Table 6.6: The level of agreement between different feature selection methods in each project

Project	k	Agreement
Eclipse JDT Core	0.18	Slight
Eclipse PDE UI	0.17	Slight
Equinox	0.40	Fair
Mylyn	0.08	Slight
Lucene	0.18	Slight

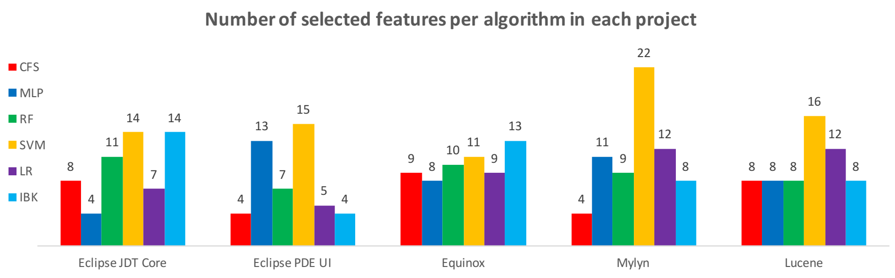
Figure 6.4(a) shows that different methods choose different features in each project. The level of agreement between the different methods is *slight* in four projects and *fair* in only one, as detailed in Table 6.6. Also the same method chooses different features in different projects. Table 6.7 shows that the level of agreement between the feature subsets selected by the same method in different projects is at most *fair*. However, there exists some agreement on some features. Figure 6.4(b) shows that *REVISIONS, FanOut, NOIM,* and *FanIn* are chosen most often. *REVI-*


Selected features per project/method
Lucene, IBK
Lucene, LR
Lucene, SVM
Lucene, RF
Lucene, MLP
Lucene, CFS
Mylyn, IBK
Mylyn, LR
Mylyn, SVM
Mylyn, RF
Mylyn, MLP
Mylyn, CFS
Equinox, IBK
Equinox, LR
Equinox, SVM
Equinox, RF
Equinox, MLP
Equinox, CFS
Eclipse PDE UI, IBK
Eclipse PDE UI, LR
Eclipse PDE UI, SVM
Eclipse PDE UI, RF
Eclipse PDE UI, MLP
Eclipse PDE UI, CFS
Eclipse JDT Core, IBK
Eclipse JDT Core, LR
Eclipse JDT Core, SVM
Eclipse JDT Core, RF
Eclipse JDT Core, MLP
Eclipse JDT Core, CFS
CBO
DIT
FanIn
FanOut
LCOM
NOC
NOA
NOIA
LOC
NOM
NOIM
NOPRA
NOPRM
NOPA
NOPM
RFC
WMC
RIVISIONS
BUGFIXES
REFACTORINGS
AUTHORS
LOC_ADDED
MAX_LOC_ADDED
AVE_LOC_ADDED
LOC_DELETED
MAX_LOC_DELETED
AVE_LOC_DELETED
CODECHURN
MAX_CODECHURN
AVE_CODECHURN
AGE
WEIGHTED_AGE

(a)


Number of times each feature is selected
RIVISIONS 28
FanOut 20
NOIM 17
FanIn 15
REFACTORINGS 14
DIT 12
NOPRM 11
AGE 11
NOC 10
NOIA 10
CODECHURN 10
CBO 9
NOPM 9
LOC_DELETED 9
MAX_CODECHURN 9
NOA 8
LOC 8
WMC 8
AUTHORS 8
MAX_LOC_ADDED 8
WEIGHTED_AGE 8
NOPRA 7
BUGFIXES 7
NOPA 6
AVE_CODECHURN 6
AVE_LOC_ADDED 5
LOC_ADDED 4
AVE_LOC_DELETED 4
LCOM 3
NOM 3
MAX_LOC_DELETED 3
RFC 2

(b)



(c)

Figure 6.4: Subfigure (a) shows the features selected by each method using the whole data of each project. Subfigure (b) shows the number of times each feature is selected out of the 30 (1 CFS feature set + 5 wrapper sets per project). The more times a feature is selected the more important it is for making accurate predictions. Subfigure (c) shows how different selection methods vary in the number of selected features. Details about the features (metrics) are in Table 6.1 and Table 6.2

Table 6.7: The level of agreement between the feature subsets selected by each method over all projects

Feature Selection Method	k	Agreement
CFS	0.23	Fair
IBK Wrapper	0.26	Fair
LR Wrapper	0.16	Slight
MLP Wrapper	0.04	Slight
RF Wrapper	0.04	Slight
SVM Wrapper	-0.01	Poor

SIONS in particular is chosen by all methods almost all the time. It is selected in 28 out of 30 feature subsets,[6] meaning that it has a high predictive power. On the other hand, *RFC, MAX_LOC_DELETED, NOM*, and *LCOM* are picked the least number of times, which means they have little to no predictive power.

The number of discarded features varies from 10 to 28 features out of 32 in total, as detailed in Figure 6.4(c). In fact, the best performing model, linear regression, selects between 5 (Eclipse PDE UI) and 12 features (Lucene and Mylyn) only. This means that most of the features can be removed while enhancing (in 48% of the experiments), or at least retaining (in 52% of the experiments), the prediction accuracy.

Another important observation is that no feature subset contains only source

[6]For each one of the 5 projects in the dataset, there are 6 feature subsets: 1 CFS subset and 5 wrapper subsets.

code metrics or only change metrics, but a mix of both. This means that no category of metrics (*i.e.*, change and source code) alone is good for predicting the number of bugs, but they should be combined to achieve better performance. Previous studies show that change metrics are better than source code metrics in bug prediction and combining the two sets either does not bring any benefit [143] or hinders the performance [6]. However, these studies either did not employ feature selection at all [143] or employed only CFS [6].

6.3.3 Threats to Validity

Threats to internal validity are concerned with systematic errors or bias. Although we use a well-known benchmark as our dataset, the quality of our results is very dependent on the quality of that dataset. Also our dependence on WEKA for building the machine learning models, makes the quality of the models dependent solely on the quality of WEKA implementation itself. Since we are dealing with machine learning, the risk of overfitting always exists. In general, we mitigate this risk by repeating each feature selection/training/testing process 50 times. Also for the wrapper method, we apply train-test-validate (TTV) approach to mitigate any overfitting bias in favour of wrappers.[7]

Threats to external validity are the threats that affect the generalizability of the results. In our study, the dataset contains metrics only from open source software systems. It is hard to generalize the results to all systems due to the differences between industrial and open source projects. Also all the systems in the dataset are developed in Java and the findings might not generalize to systems developed in other languages. Another important threat to external validity comes from the chosen prediction models. Although we try to pick a representative set of prediction models, our findings might not generalize to other machine learning models such as hybrid and genetic ones.

Another point is that machine learning methods can be dependant on the underlying data. Although our findings are statistically strong, they may not generalize to all open source Java systems. Practitioners and researchers are encouraged to run comparative studies, similar to ours, before making decisions regarding the machine learning techniques employed for bug prediction.

6.4 Conclusions

Generalizing bug prediction findings is hard. Software projects have different teams, cultures, frameworks, and architectures. Consequently, software metrics have different correlations with the number of bugs in different projects. These correlations can be captured differently by distinct prediction models. In this chapter we show that wrapper feature selection methods fit this problem best because they not only choose features relevant to the response variable but also to the prediction model itself. Indeed, our results show that wrapper feature selection is always better than CFS and

[7]The train-test-validate is not applicable to CFS due to how CFS operates

improves the performance of a model (by up to 47%) while eliminating most of the features (up to 87%). We also show that the same feature selection method chooses different features in different projects. We cannot generalize what feature subset to use, but we can recommend combining both change and source code metrics and letting the wrapper feature selection method choose the right subset and adapt the bug predictor to the project at hand.

In the next chapter, we perform another optimization from the machine learning field, which is hyperparameter optimization, to adapt the bug predictor further and improve its accuracy.

On a side note, while carrying out this study, we realized that datasets providing the number of bugs as the response variable are scarce, which could be a hurdle to studies predicting the number of bugs in software systems. We therefore built a tool that extracts source code and change metrics along with the number of bugs in software projects. This tool is further explained in Appendix B.

7 Optimizing Bug Prediction by Tuning Hyperparameters

In chapter 6, we investigate optimizing bug prediction by applying feature selection. In this chapter, we explore another technique for machine learning optimization: hyperparameter optimization. Hyperparameters are the parameters that are set for a machine learning model and affect its learning, construction, and evaluation. These parameters need to be set before training the model. Example hyperparameters are the *complexity* parameter in support vector machines and the number of neurons in the hidden layer in a feed-forward neural network. Different machine learning problems have different characteristics and the hyperparameters need to be optimized accordingly.

In the past three decades, researchers have analyzed the performance of various machine learning models in bug prediction. However, more often than not, model hyperparameters were set to the default values of machine learning frameworks, which are not necessarily the optimal ones.

We investigate the effect of the hyperparameter optimization[1] of a model on its prediction accuracy. We study two machine learning models: support vector machines (SVM), and an implementation of the k-nearest neighbours algorithm called IBK. Using a grid search algorithm, the search space of the hyperparameter values of each model is traversed and the optimal values are reported. We evaluate the prediction accuracy of each model before and after hyperparameter optimization on five open source Java systems (same dataset as in section 6.3.1).

Our results reveal that tuning model hyperparameters has a statistically signif-

[1]Hyperparameter optimization is also known as model selection and hyperparameter tuning.

icant positive effect on the prediction accuracy of the models. The prediction accuracy is improved by up to 20% in IBK and by up to 10% in SVM. However, we notice that IBK is more sensitive to its hyperparameter values than SVM in our experiments. We also observe that most of the hyperparameter values are changed during the tuning phase, indicating that default values are suboptimal. Our findings suggest that researchers in bug prediction need to take hyperparameter optimization into account in their bug prediction pipelines, as it potentially improves the prediction accuracy of the machine learning models.

7.1 Empirical Study

We carry out our empirical investigation using the same dataset that we use in chapter 6. More details about the dataset are in section 6.3.1.

7.1.1 Machine Learning Algorithms

For this experiment, we pick two machine learning algorithms: k-nearest neighbours (IBK) and support vector machines (SVM). SVM is a geometrically-inspired machine learning technique that can, in the feature space, separate data points by choosing the best separating hyperplane (*i.e.*, with highest separation margin that is). IBK is an implementation of the k-nearest neighbours algorithm where the value of the dependent variable of a new data point is calculated based on the average values of the dependent variable of the k-nearest data points in the feature space.

We choose these two machine learning algorithms because they operate differently, have many hyperparameters, and have two different track records in the field of bug prediction. While SVM has been extensively used and shown to be one of the best performing models [47][189][122][179][41], IBK has not been reported to excel in the bug prediction literature [121]. Studying these two models reveals whether the tuning process affects how they compare to each other. We use the WEKA[2] data mining framework [72] to train and test the models.

7.1.2 Parameter Tuning

For IBK, we tune three hyperparameters: the number of neighbours, the evaluation criterion, and the neighbour search algorithm, as detailed in Table 7.1. For SVM, we tune the complexity parameter and the used kernel and its parameters, as detailed in Table 7.2 and Table 7.3.

The search space for the optimal hyperparameter values is large and it is impractical to try every possible combination of values. We use *Multisearch-weka*[3] to search for the optimal hyperparameter values. It implements a hill-climbing grid search algorithm. An initial point in the search space is considered the center, then the algorithm performs 10-fold cross validation on the adjacent parameter values.

[2] `http://www.cs.waikato.ac.nz/~ml/weka/`

[3] `https://github.com/fracpete/multisearch-weka-package`

The best one is considered as the new center. This process is repeated until no better values are found or the navigation hits the search space borders.

Table 7.1: IBK hyperparameters that are tuned.

Parameter	Default Value	Search Range
Number of neighbours	1	1 to 5 by step of $+1$
Evaluation Criterion	Mean Absolute Error	$\{$Mean Absolute Error, Mean Squared Error$\}$
Neighbour Search Algorithm	Linear Search	$\{$Linear, BallTree, CoverTree, KDTree, Filtered Neighbour Search$\}$

Table 7.2: SVM hyperparameters that are tuned.

Parameter	Default Value	Search Range
Complexity	1	10^{-4} to 10^{2} by step of $\times 10$
Kernel	Polynomial	$\{$Polynomial, RBF, PUK, Normalized Polynomial$\}$

7.1.3 Procedure

For each project, we split the data into two sets using stratified sampling: tuning set (10%) and experimentation set (90%). The hyperparameters are tuned using the tuning set only. Then, for each machine learning model, we compare the prediction error between the model with the default hyperparameter values and the model with the tuned values. For this comparison we use stratified 10-fold cross-validation on the experimentation set and the root mean squared error (RMSE) is calculated for each fold. This 10-fold cross-validation is repeated 30 times.

Up to this point, for each project/model, we have 300 RMSEs for the model with default hyperparameter values and 300 RMSEs for it with the optimal ones. We use paired student's t-test with 95% confidence interval to compare the two populations and determine whether the tuning process improves the prediction accuracy of the model.

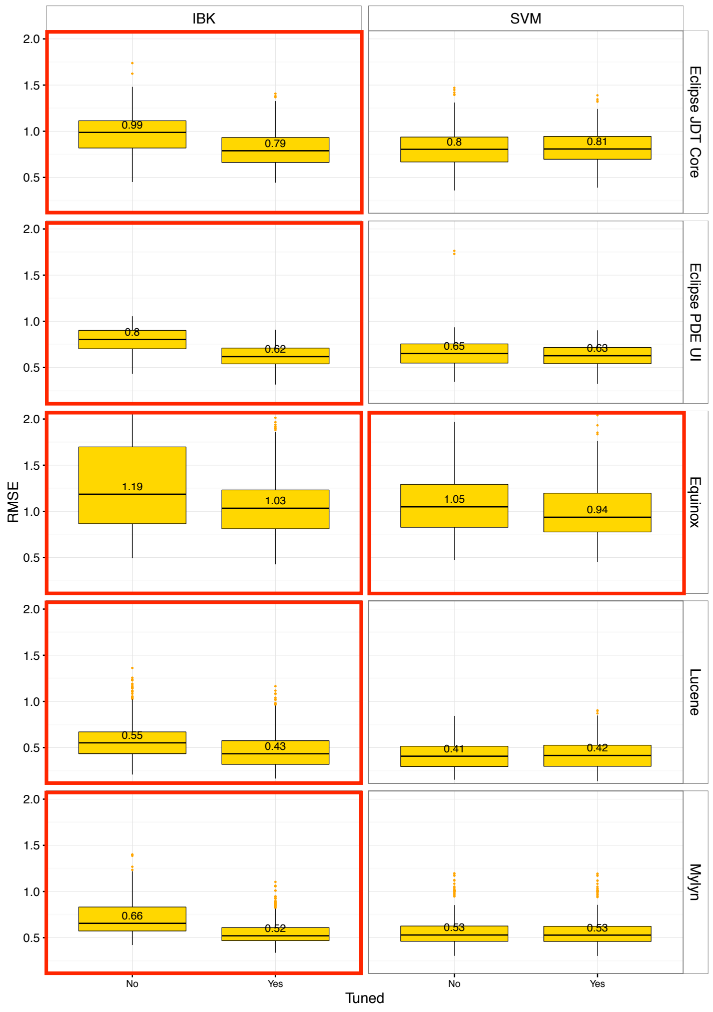


Figure 7.1: Boxplots of all the experiments in our empirical study. The y-axis is the root mean squared error (RMSE). For each project, we compare each model before and after tuning. We carried out the student's *t*-test at 95% confidence interval. Red bold frames indicate the statistically significant results, where the tuning significantly reduced the RMSE of the model.

Table 7.3: SVM kernel hyperparameters that are tuned.

Kernel	Parameter	Default Value	Search Range
Polynomial	Exponent	1	1 to 10 by step of $+0.5$
Normalized Polynomial	Exponent	2	1 to 10 by step of $+0.5$
RBF	Gamma	0.01	$10^{-}4$ to 10 by step of $\times 10$
PUK	Omega	1	1 to 10 by step of $+0.5$
	Sigma	1	1 to 10 by step of $+0.5$

Table 7.4: The tuning results for the hyperparameters for all projects.

	Eclipse JDT Core	Eclipse PDE UI	Equinox	Lucene	Mylyn
IBK					
#Neighbours	5	5	2	5	5
Evaluation Criterion	Mean Squared Error	Mean Squared Error	Mean Squared Error	Mean Squared Error	Mean Squared Error
Search Algorithm	CoverTree	Linear Search	Linear Search	Linear Search	Linear Search
SVM					
Complexity	10	1	10	10	10
Kernel	PUK{Omega=1, Sigma=4.1}	Normalized Polynomial {Exponent=5}	RBF {Gamma=0.01}	RBF {Gamma=0.1}	Polynomial {Exponent=1}

7.1.4 Results

Figure 7.1 shows boxplots of the RMSEs of each model before and after tuning. Red frames indicate statistically significant results (the null hypothesis of student's *t*-test is rejected).

For IBK, tuning hyperparameters improves prediction accuracy significantly for all projects. The RMSE is reduced between 13% (in Equinox) and 22% (in Lucene). The results for SVM are not as significant as in IBK. Only in Equinox does hyperparameter tuning reduce RMSE significantly by 10%, while prediction accuracy of the tuned and untuned models are similar in the other four projects. This means that IBK is more sensitive to its hyperparameter settings than SVM in bug prediction.

We also compare IBK and SVM. A paired student's *t*-test with 95% significance interval shows that the RMSE of SVM is always statistically lower than the RMSE of IBK. However, after tuning, the two models are actually statistically equivalent for four projects and SVM is more accurate than IBK only in Equinox. This result shows that tuning not only potentially improves the prediction accuracy of a model, but also changes how different machine learning models compare to each other.

Finally, our results show that the values of model hyperparameters often change from the default after tuning. Table 7.4 shows the parameter values after tuning. The tuning phase always changes the value of at least one hyperparameter from the default. Some hyperparameter values actually always change, such as the number of neighbours and the evaluation criterion in IBK. This leads to the conclusion that default values in machine learning frameworks are suboptimal for bug prediction.

In summary, we conclude the following:

1. Tuning hyperparameters significantly improves the prediction accuracy of IBK. Previously, IBK has not been in the top performing models for bug prediction [121], but with tuning it can be as performant as SVM.

2. IBK is more sensitive to hyperparameter tuning than SVM. We recommend that researchers experiment with the hyperparameters of the machine learning models they use before using them, as some require tuning while others are less susceptible to the hyperparameter values.

3. Many studies have been conducted to compare different machine learning models in the context of bug prediction. Some studies conclude that classifiers perform similarly in bug prediction [203][135][111][44][134] while others suggest that the choice of the classification model has a significant impact on the performance of bug prediction [70][47][63]. We show that hyperparameter tuning can have an effect on how machine learning models compare to each other. This raises doubts in the outcome of studies where no tuning of or at least experimentation with the hyperparameter values has been conducted.

7.1.5 Threats to Validity

The main threats to the validity of our study are threats to generalizability. First, the dataset we use contains only open-source Java projects. Replicating the same experiments on industrial projects or projects written in other languages may give different results.

We only experiment on two machine learning models only. Hyperparameter optimization might have different effects on other machine learning models. Also we carry out the tuning process on only on 10% of the dataset with bounded value ranges for the hyperparameters. A larger tuning set and wider value ranges might provide better hyperparameter values.

7.2 Conclusions and Future Work

In this chapter, we study the effect of optimizing model hyperparameters to improve the accuracy of predicting the number of bugs. We show that the k-nearest neighbours algorithm (IBK) is always significantly improved and the prediction accuracy of support vector machines (SVM) is either improved or at least retained. We conclude that hyperparameter optimization should be conducted before using a machine learning model and default hyperparameter values are often suboptimal.

In this chapter and the previous one (chapter 6), we show how to empirically optimize a bug predictor from the machine learning perspective. Hyperparameter optimization and feature selection adapt a bug predictor to the software project at hand and improve the prediction accuracy. In the next chapter, we optimize bug prediction as a software quality tool.

8 Empirically-Grounded Optimization of Bug Prediction as a Quality Tool

In chapter 6 and chapter 7, we optimize bug prediction as a machine learning model. However, the main promise of bug prediction is to help software engineers focus their testing and reviewing efforts on those software parts that most likely contain bugs. Under this promise, for a bug predictor to be useful in practice, it not only needs to be accurate, but it also must be *efficient*, that is, it must be optimized to locate the maximum number of bugs in the minimum amount of code [129][130][6].[1] Optimizing a bug predictor requires making the right decisions for (i) the independent variables, (ii) the machine learning model, and (iii) the response variable.[2] We call this triple, *bug prediction configurations*.

These configurations are interconnected. The entire configuration should be evaluated in order to provide individual answers for each aspect reliably. However, the advice found in the literature focuses on each aspect of bug prediction in isolation and it is unclear how previous findings hold in a holistic setup. In this study, we adopt the *Cost-Effectiveness* measure (CE), introduced by Arisholm *et al.* [6], to empirically evaluate the different options of each of the bug prediction configurations all at once, shedding light on the interplay among them. Consequently, we pose and answer the following research questions:

[1]Efficient bug prediction as we define it, is sometimes referred to as effort-aware bug prediction in the literature

[2]Also known as the dependent variable or the output variable

RQ1: What type of software metrics are cost-effective?

We find that using a mix of source code metrics and change metrics yields the most cost-effective predictors for all subject systems in the studied dataset. We observe that change metrics alone can be a good option, but we advise against using source code metrics alone. These findings contradict the advice found in the literature that object-oriented metrics hinders the cost-effectiveness of models built using change metrics [6]. In fact although source code metrics are the worst metrics set, it can still be used when necessary, but with the right configuration combination.

RQ2: What prediction model is cost-effective?

In this study we compare five machine learning models: Multilayer Perceptron, Support Vector Machines, Linear Regression, Random Forest, and K-Nearest Neighbour. Our results show that Random Forest stands out as the most cost-effective one. Support Vector Machines come a close second. While some previous studies suggest that Random Forest performs generally better than other machine learning models [70], other studies note that Random Forest does not perform as well [75]. Our findings suggest that Random Forest performs the best with respect to cost-effectiveness.

RQ3: What is the most cost-effective response variable to predict?

We establish that predicting the number of bugs in a software entity is the most cost-effective approach and predicting bug proneness is the least cost-effective one. To our knowledge, this research question has not been investigated before in the literature.

RQ4: Is there a configuration combination that consistently produces highly cost-effective bug predictors?

Here we evaluate all configurations at once to provide more reliable guidelines for building cost-effective bug predictors. We conclude that *both source code and change metrics as independent variables combined, Random Forest as the prediction model, and bug count as the response variable*, form the configuration combination of the most cost-effective bug predictor across all subject systems in the studied dataset.

8.1 Empirical Setup

8.1.1 Evaluation Scheme

There is a strong relationship between what is expected from a model and how the model is evaluated. In the field of bug prediction, the desired value expected from a bug predictor as a software quality tool is to enhance the efficiency of the quality assurance procedure by directing it to the buggy parts of a software system. This is possible only when the bug predictor can find most of the bugs in the least amount of code. Intuitively, the efficiency of a predictor increases inversely proportional to the number of lines of code in which it suspects a bug might appear because writing unit tests for large software entities or inspecting them requires more effort.

Arisholm *et al.* state that *"... the regular confusion matrix criteria, although popular, are not clearly related to the problem at hand, namely the cost-effectiveness of using fault-proneness prediction models to focus verification efforts to deliver software with less faults at less cost"*[6]. Consequently, they proposed a cost-aware evaluation scheme called *Cost-Effectiveness* (CE) [6]. CE measures the benefit of using a certain bug prediction model. It summarizes the accuracy measures and the usefulness of a model by measuring how close the prediction model is to the optimal model, taking the random order as the baseline. This scheme assumes the ability of the prediction model to rank software entities in an ordered list. To demonstrate CE, we show in Figure 8.1 an example cumulative lift curves (Alberg diagrams [150]) of three orderings of software entities:

1. **Optimal Order:** The green curve represents the ordering of the software entities with respect to the bug density from the highest to the lowest.

2. **Random Order:** The dashed diagonal line is achieved when the percentage of bugs is equal to the percentage of lines of code. This is what one gets, on average, with randomly ordering the software entities.

3. **Predicted Order:** The blue curve represents the ordering of the software en-

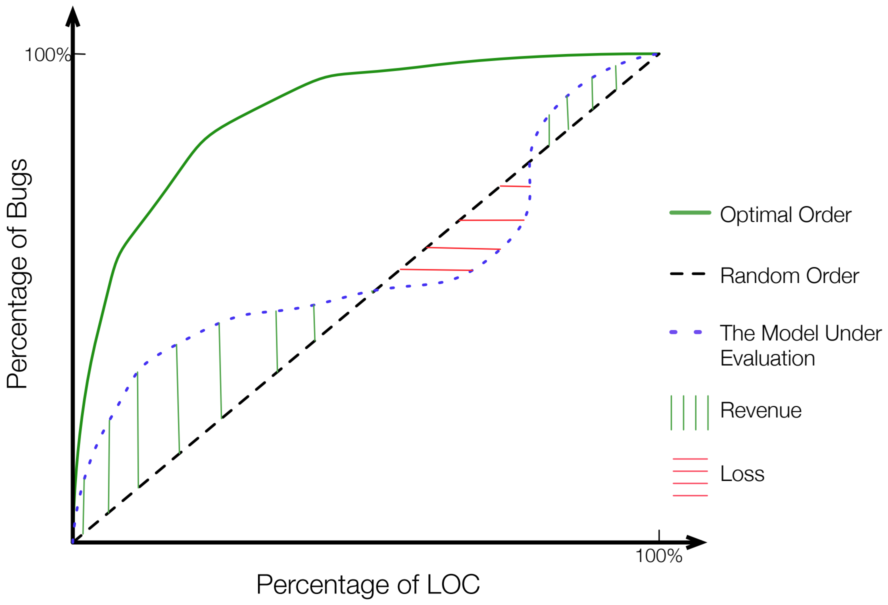


Figure 8.1: An overview of the CE measure as defined by Arisholm *et al.* [6].

tities based on the predicted dependent variable.

The area under each of these curves is called the Cost-Effectiveness (CE) area. The larger the CE area, the more cost-effective the model. However, two things need to be taken into account in this scenario. First, optimal models are different for different datasets. Second, the prediction model should perform better than the random ordering model to be considered valuable. That's why Arisholm *et al.* [6] took the optimal ordering and the random ordering into consideration in the Cost-Effectiveness measure as:

$$CE(model) = \frac{AUC(model) - AUC(random)}{AUC(optimal) - AUC(random)}$$

where $AUC(x)$ is the area under the curve x.

CE assesses how good the prediction model is in producing a total order of entities. The value of CE ranges from -1 to +1. The larger the CE measure is, the more cost-effective the model is. There are three cases:

1. When CE is close to 0, it means that there is no gain in using the prediction model.

2. When CE is close to 1, it means that the prediction model is close to optimal.

3. When CE is between 0 and -1, it means that the cost of using the model is more than the gain, making the use of the model actually harmful.

In our experiments, we use CE to compare prediction results and draw conclusions.

8.1.2 Dataset

We carry out our empirical investigation using the same dataset that we use in chapter 6. More details about the dataset are in section 6.3.1.

8.1.3 Response Variable

The chosen dependent variable is an important design decision because it sometimes determines the difference between a usable and an unusable model. We have to keep in mind that the final goal of the prediction is to prioritize the software entities into an ordered list to be able to apply the Cost-Effectiveness (CE) measure. There are multiple possible schemes to do it:

1. predict the number of bugs as the response variable then order the software entities based on the *calculated* bug density.

2. predict the bug density directly then order the software entities based on this *predicted* bug density.

3. predict the bug proneness and order the software entities based on it. Small classes come before large ones in case of ties.

4. classify software entities into buggy and bug-free, then order them as follows: buggy entities come before bug-free ones and small classes come before large ones (with respect to LOC).

The optimal prediction of number of bugs (first scheme) is equivalent to the optimal prediction of bug density (second scheme), since bug density is calculated from the number of bugs as $bugdensity = (\#bugs/LOC)$. Also the optimal prediction of bug proneness (third scheme) is an optimal classification (fourth scheme). Obviously the optimal regression in the first or second schemes is more cost-effective than the optimal classification in the third or fourth scheme because it reflects exactly the optimal solution in the cost-effectiveness (CE) evaluation method. However, we need to verify whether classification is a valid approach in bug prediction and whether we should include the third and fourth schemes in the empirical study. In Figure 8.2, we evaluate the cost-effectiveness of the optimal classifier following the fourth scheme, for all five systems in our corpus. The cost-effectiveness of the fourth scheme is excellent for Equinox and JDT, and almost optimal for Mylyn, Lucene, and PDE. As shown in Table 6.3, the percentage of buggy classes with more than one bug ranges from 28% to 38%. This leads to the conclusion that in the set of buggy classes, the number of bugs is proportional to the number of lines of code. This is particularly interesting because it makes classification as good as predicting the number of bugs, in the ideal case. We empirically verify which response variable is better when we train our prediction models.

8.1.4 Machine Learning Models

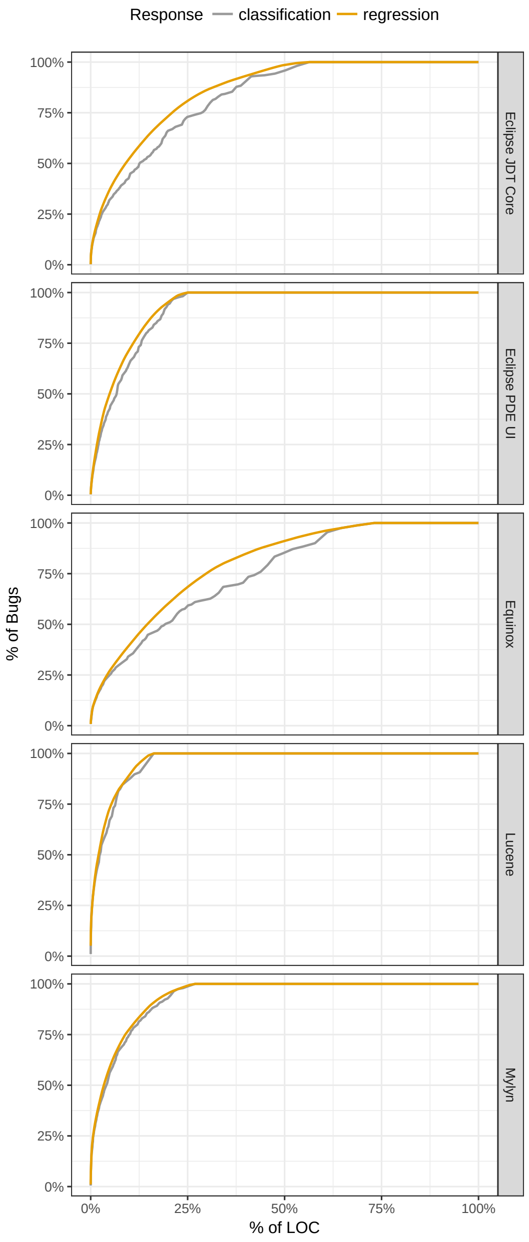
We investigate the following machine learning models: Random Forest (RF), Support Vector Machine (SVM), Multilayer Perceptron (MLP), an implementation of the K-nearest neighbours algorithm called IBK, and Linear Regression (LinR) / Logistic Regression (LogR)[3]. We choose these machine learning models for two reasons: First, they are extensively used in the bug prediction literature [121]. Second, each one of them can be used as a regressor and as a classifier, making comparisons across different configurations possible. Classifiers are used to predict the bug proneness or the class (buggy, bug-free) and regressors are used to predict the bug count and bug density. We use the Weka data mining tool [72] to build these prediction models.

8.1.5 Hyperparameter Optimization

Machine learning models may have configurable parameters that should be set before starting the training phase. This process is called hyperparameter optimization or model tuning, and can have a positive effect on the prediction accuracy of

[3]Linear Regression and Logistic Regression are equivalent, but with different types of response variables. Linear Regression is a regressor and Logistic Regression is a classifier.

Figure 8.2: Commutative lift curves (Alberg diagrams [150]) comparing the optimal regressor (bug density predictor) and the optimal classifier with ranking based on LOC (smallest to largest). These diagrams show that optimal classification performs almost as well as optimal regression.



the models. However, different models have different sensitivities to this process. While model tuning improves IBK and MLP substantially, it has a negligible effect on SVM and RF [193][154]. In this study, we follow the same procedure as discussed in chapter 7. The used model parameters are detailed in Table 8.1.

Table 8.1: The tuning results for the hyperparameters

RF	Number of Trees= 100

SVM	Kernel= RBF {Gamma=0.1}
	Complexity=10

MLP	Learning Rate=0.6 Momentum=0.6 Hidden Layer Size= 32
IBK	#Neighbours=5 Search Algorithm= Linear Search Evaluation Criterion=Mean Squared Error
LinR/LogR	No parameters to tune

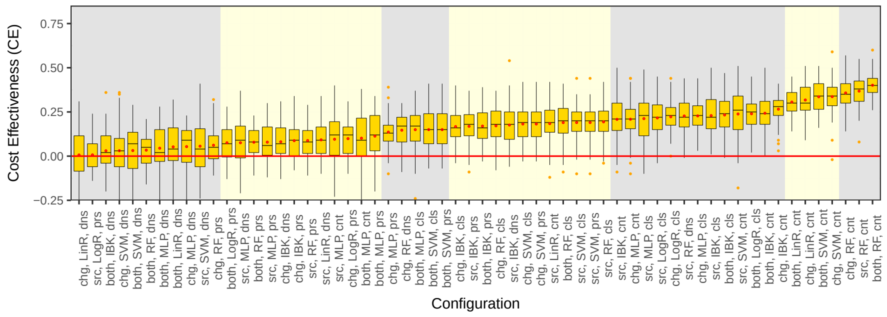
8.1.6 Feature Selection

The prediction accuracy of machine learning models is highly affected by the quality of the features used for prediction. Irrelevant and correlated features can increase prediction error, increase model complexity, and decrease model stability. Feature selection is a method that identifies the relevant features to feed into machine learning models. We apply wrapper feature selection for SVM, MLP, and IBK as it has been shown that it leads to higher prediction accuracy [155]. We do not apply feature selection for RF because it performs feature selection internally. Following the guidelines by Osman *et al.* [153], we apply $l2$ regularization (Ridge) on LinR/LogR as the feature selection method.

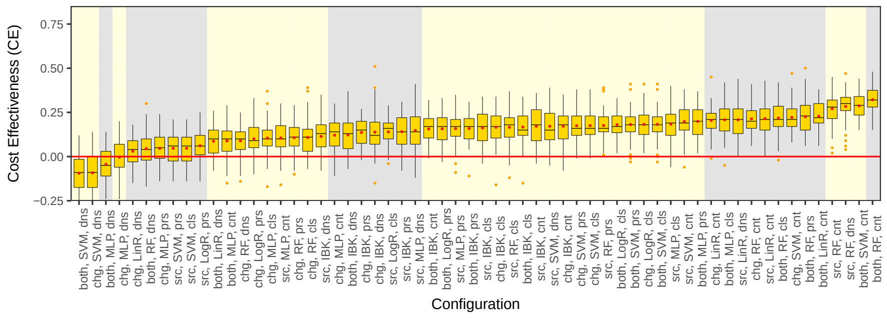
8.1.7 Data Pre-Processing

Bug datasets are inherently imbalanced where most software entities are bug-free. This is called the class-imbalance problem and can negatively affect the performance of machine learning models [206][4]. To cope with this problem, we divide the data set for each project into two sets: test set (25%) and training set (75%). The samples in each set are taken at random but maintain the distribution of buggy classes to be similar to the one in the full data set. We then balance the training set by oversampling. This is important for training and for evaluating the prediction models. The machine learning models are then trained using the balanced training set and evaluated on the unseen test set.

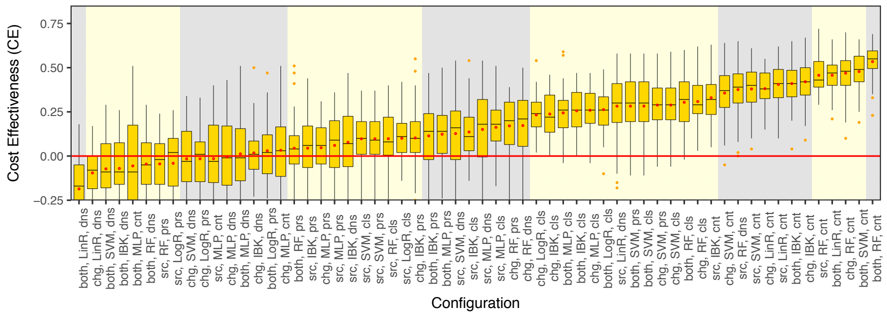
Figure 8.3: Boxplots of the CE outcome. Each box plot represents the CE obtained by 100 runs of the corresponding configuration combination on the x-axis. Different background colors indicate the statistically distinct groups obtained by applying the Scott-Knott clustering method with 95% confidence interval. The configurations on the x-axis are of the form `Metrics-Model-Response`.



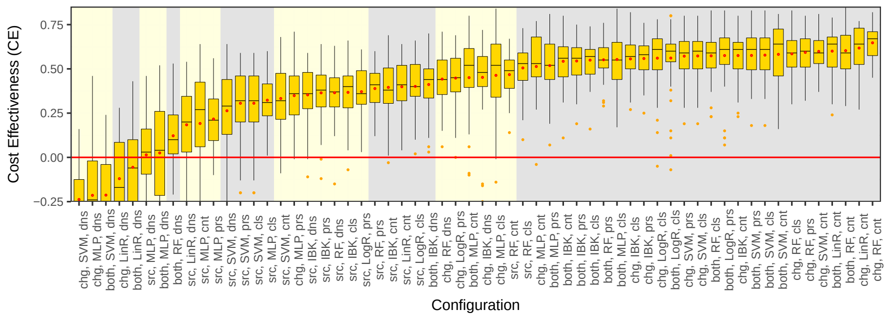
(a) Eclipse JDT Core



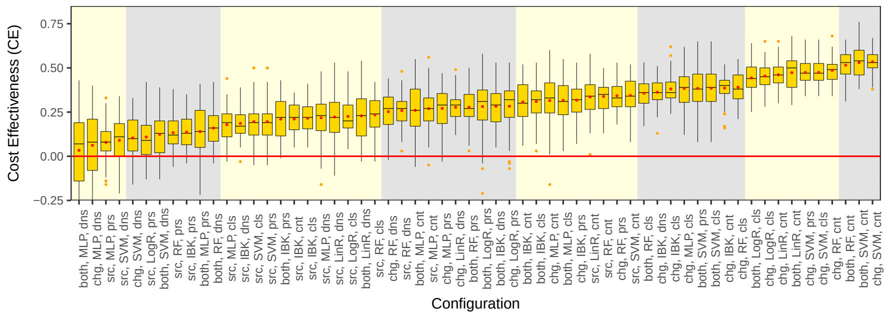
(b) Eclipse PDE UI



(c) Equinox



(d) Lucene



(e) Mylyn

8.2 Results

In this study, we consider the following configurations:

1. Independent Variables:
   (a) source code metrics (`src`)
   (b) change metrics (`chg`)
   (c) both of them combined (`both`).
2. Machine Learning Model:
   (a) Support Vector Machines (`SVM`)
   (b) Random Forest (`RF`)
   (c) Multilayer Perceptron (`MLP`)
   (d) K-Nearest Neighbours (`IBK`)
   (e) Linear Regression (`LinR`) or Logistic Regression (`LogR`)
3. Response Variable:
   (a) bug count (`cnt`)
   (b) bug density (`dns`)
   (c) bug proneness (`prs`)
   (d) classification (`cls`)

There are 60 possible configuration combinations. For each one, we pre-process the data, train the model on the training set, perform the predictions on the test data, and calculate the Cost Effectiveness measure (CE). We repeat this process 50 times to mitigate the threat of having outliers because of the random division of the dataset into a training set and a test set. We do not perform k-fold cross-validation method because calculating CE over a small set of classes can be misleading. Instead, we perform the repeated hold-out validation because it is known to have lower variance than k-fold cross validation making it more suitable for small datasets [18].

In this experiment, statistically speaking, the treatment is the configuration combination and the outcome is the CE score. Hence, we have 60 different treatments and one outcome measure. To answer the posed research questions, we need to compare the CE of different configuration combinations. Since there is a large number of treatments, traditional parametric tests (*e.g.*, ANOVA) or non-parametric tests (*e.g.*, Friedman) have the overlapping problem, *i.e.*, the clusters of treatment overlap. Therefore, we use the Scott-Knott (SK) cluster analysis for grouping of means [183], which is a clustering algorithm used as a multiple comparison method for the analysis of variance. SK clusters the treatments into statistically distinct non-overlapping groups (*i.e.*, ranks), which makes it suitable for this study. We apply SK with 95% confidence interval to cluster the configuration combinations for each project in the dataset.

Figure 8.3 shows box plots of the CE outcomes for each configuration combination. Each box plot represents the population of the 100 runs of the corresponding configuration. The box plots are sorted in an increasing order of the means of CE, represented by the red points. Alternating background colors indicate the Scott-Knott statistically distinct groups (*i.e.*, clusters or ranks).

The results in Figure 8.3 clearly demonstrate the interplay between the design choices in bug prediction. Changing one value in the configuration can transform a bug predictor from being highly cost-effective, to being actually harmful. For instance, while `both-RF-cnt` is the most cost-effective configuration in Figure 8.3(a), `both-RF-dns` is in the least cost-effective cluster. This means that although `RF` is the best machine learning model and `both` is the best choice of metrics, using them with the wrong response variable renders a bug predictor useless. There are many examples where changing one configuration parameter brings the bug predictor from one cluster to another. Examples for each configuration variable include:

1. In Figure 8.3(a), `both-RF-cnt` is ranked 1^{st} whereas `both-RF-dns` is ranked 7^{th}.

2. In Figure 8.3(b), `both-SVM-cnt` is ranked 2^{nd} whereas `both-MLP-cnt` is ranked 6^{th}.

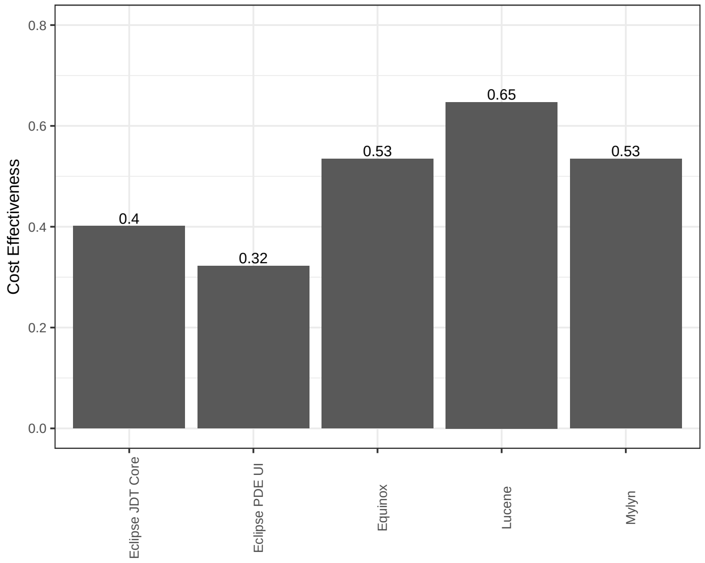
3. In Figure 8.3(e), `chg-SVM-cls` is ranked 2^{nd} where as `src-SVM-cls` is ranked 6^{th}.

These examples constitute a compelling evidence that bug prediction configurations are interconnected.

To answer the first research question (*RQ1*) regarding the choice of independent variables, we analyze the top cluster of configurations in Figure 8.3. We observe that for Eclipse PDE UI, Equinox, and Mylyn, the software metrics value in the top rank is either `both` or `chg`. In Eclipse JDT Core, `src` appears in one configuration out of three in the top rank. In Lucene, *src* appears in one configuration out of 27 in the top rank. These results suggest that the use of both source code and change metrics together is the most cost-effective option for the independent variables. Using only change metrics is also a good choice, but using only source code metrics rarely is. It was shown in the literature that adding source code metrics to change metrics hinders the cost-effectiveness and using source code alone is not better than random guessing [6]. Our results show that although less cost-effective, source code metrics can be used alone when necessary (*e.g.*, change metrics cannot be computed). There is always a cost-effective configuration combination with the source code metrics as the independent variables (*e.g.*, `src-RF-cnt`).

For the second research question (*RQ2*) regarding the choice of the machine learning model, we observe that `RF` is *the only* option in the top rank in Eclipse JDT Core, Eclipse PDE UI, and Equinox, and it is in the top rank of Lucene and Mylyn. `SVM` also performs well. It appears in the top rank in Lucene and Mylyn, and in the second rank in the rest of the projects. These results indicate the superiority of Random Forest and Support Vector Machines in producing cost-effective bug

Figure 8.4: The max mean values of CE obtained for each project



predictors. On the other hand, `MLP` and `IBK` made it to the top two clusters only in Lucene, suggesting that Multilayer Perceptron and K-Nearest Neighbour do not fit the bug prediction problem well.

For the third research question (*RQ3*) regarding the most cost-effective response variable, we observe that `cnt` is in the top rank in Lucene and is *the only* response variable in the top rank in the other projects. It is clear that predicting the bug count results in the most cost-effective bug predictors. Another observation is that the response variable configuration in the bottom two clusters is almost conclusively either `dns` or `prs`. This means that predicting bug density or bug proneness actually hinders the cost-effectiveness of the bug prediction.

Overall, one result that stands out is that the configuration `both-RF-cnt` is in the top cluster across projects (*RQ4*). In fact, it is *the most* cost-effective configuration in Eclipse JDT Core, Eclipse PDE UI, and Equinox and it is in the top cluster in Lucene and Mylyn. This finding suggests that this configuration seems to be the best from the cost-effectiveness point of view.

Software projects differ in their domains, development methods, used frameworks, and developer experiences. Consequently, software metrics differ in the correlation with the number of bugs among projects. This is the reason why using both metrics came out as the best choice of independent variable. However, to deal with the inevitable noise and redundancy in using both metrics, the best configurations includes Random Forest as the machine learning model. Random Forest is an ensemble of decision trees created by using bootstrap samples of the training data and random feature selection in tree induction [25]. This gives RF the ability to work well with high-dimensional data and sift the noise away. This is the reason why feeding `both` types of metrics into Random Forest actually makes sense. Also bug count came out as the best option for response variable because it reflects the "gain" in CE better than classification or proneness. Bug density also reflects the "gain" but it is better to calculate it from bug count than to leave it to the prediction model to deduce. Therefore, bug count is a simpler and more appropriate response variable than bug density. All these factors contribute to the fact that `both-RF-cnt` is the most cost-effective configuration for bug prediction.

Finally, the results in Figure 8.4 also show that the cost-effectiveness of the best bug predictor varies among projects. Although the best bug predictor is never harmful to use (no negative CE) in our experiments, it can still be of little value for some projects, *e.g.*, $CE = 0.32$ for Eclipse PDE UI. This means that bug prediction should be evaluated as a technique in the context of a software project before putting it in use in that specific project.

8.3 Threats to Validity

To minimize the threats to validity in our empirical study, we follow the guidelines of Mende [128] by

- using a large dataset to avoid large variance of performance measures,
- maintaining the same fault distribution in the test set and training set as the original set, to minimize bias,
- repeating the experiment 50 times to minimize the effect of outliers,
- and reporting on the dataset, data preprocessing procedure, and model configurations to enable replication.

In our study we use the "Bug Prediction Dataset" provided by D'Ambros *et al.* [38] as a benchmark. Although it is a well-known and studied dataset, the quality of our results is very dependent on the quality of that dataset.

Our dependence on WEKA [72] for building the artificial intelligence models, makes the quality of the models dependent solely on the quality of WEKA itself.

The fact that this dataset contains metrics only from open source software systems makes it hard to generalize to all Java systems. In the future, we plan to apply our study on more datasets and to use other data mining tools.

Another threat to validity comes from the use of LOC as a proxy for cost. As we explained before, reviewing code and writing unit tests take much more effort for large modules than small ones. However, we are aware of the fact that this proxy might introduce some bias. We use it because it has been used in several previous studies as such (*e.g.*, [129][6]) and it is widely accepted in the community as a good measure of effort.

Our study is on the Java-class level. Hence, our findings may not apply on other granularity levels such as method level or commit level.

Finally, in this study, we assume that the purpose of bug prediction is to locate the maximum number of bugs in the minimum amount of code in order to be a useful support to quality assurance activities. However, defect prediction models can be used for other purposes. For example, they can be used as tools for understanding common pitfalls and analyzing factors that affect the quality of software. In these cases, our findings do not necessarily apply. In the future, we plan to extend this study to the broader context of several defect prediction use cases.

8.4 Conclusions

Bug prediction is used to reduce the costs of testing and code reviewing by directing maintenance efforts towards the software entities that most likely contain bugs. From this point of view, a successful bug predictor should not only be accurate in its prediction, but also find the largest number of bugs in the least amount of code. In chapter 6 and chapter 7, we optimize bug prediction from the machine learning perspective. In this chapter, we optimize bug predictors as software quality tools. Using the cost effectiveness evaluation scheme, we carry out a large-scale empirical study to find the most efficient bug prediction configurations, as building a bug predictor entails many design decisions, each of which has many options.

Our findings reveal a compelling evidence that bug prediction configurations are interconnected. Changing one configuration can render a bug predictor useless. A mix of source code and change metrics as independent variables, Random Forest as the machine learning model, and bug count as the response variable result in the most cost-effective bug predictor in all systems in the used dataset. Without the empirically-grounded analysis of the different configuration options, it is hard to reach this conclusion.

9 Conclusions and Future Work

Bug prediction and detection tools are auxiliary software quality assurance (AQSA) tools that help developers carry out the main quality assurance activities of testing and code reviewing efficiently. AQSA tools should direct developers to the parts that are more likely to contain bugs. However, for these tools to be useful in practice, they need to be adapted to the peculiarities of the projects they are applied to, and they need to be efficient in locating the maximum number of bugs in the minimum amount of code. In this chapter we conclude this thesis by summarizing its contributions and pointing for possible future directions.

9.1 Contributions of This Dissertation

In this thesis, we present empirically-grounded analysis (EGA) as a unified approach to improve the adaptability and efficiency of ASQA tools. The contributions of this thesis can be summarized as follows:

1. We use EGA to discover bug patterns that are common in Java projects so that they are worth building detection tools for. Among others, we find that missing null checks are the most frequent bugs in Java.

2. In order to build an efficient detection tool, we use EGA to analyze null checks and null usage in Java systems. We find that methods that return null are the root cause of null checks.

3. We propose an empirical solution to the missing null problem. For every method in an API, we calculate its nullability, *i.e.*, how often the returned

value from this method is checked in the ecosystem of the API. An Eclipse plugin uses this nullability measure to annotate method nullness in a software project and display warnings of missing and excessive null checks.

4. We use EGA to find out whether certain optimizations are beneficial to apply in bug prediction. We empirically show that feature selection and hyperparameter optimization adapt bug prediction to software projects and improve its accuracy

5. We use EGA to optimize bug prediction as a quality tool. We empirically evaluate the 60 configuration options and find that training Random Forest on both change and source code metrics to predict the number of bugs results in the most cost-effective bug predictor.

Throughout the thesis, we learn that there is no silver bullet when it comes to building bug prediction and detection tools. Every project is unique and what works for one does not necessarily work for others. Studies in bug prediction and detection cannot and should not be generalized. AQSA tools need to be carefully designed, evaluated, and tailored to projects before they can be adopted. As we show, EGA provides a suitable framework for extracting requirements, analyzing problems, designing solutions, and evaluating results. Furthermore, EGA is sometimes the only option. For instance, there is no analytical method other than EGA to find the most efficient bug prediction configurations (as in chapter 8). Trial and error, statistical comparisons, and repeated experimentation are proven to be powerful methods that software developers can use systematically to build efficient software quality tools.

9.2 Future Research Directions

This dissertation opens the door for better tool crafting and integration.

9.2.1 Ad-hoc Nullability Analysis

We envision a tool that extends our nullability plugin to implement an ad-hoc analysis. This tool can be an IDE plugin that does the following:

1. Analyze the dependencies and APIs used in a project.

2. Collect clients of these APIs from several sources other than mvnrepository such as GitHub and Bitbucket.

3. Calculate the nullability of each method in the API and annotate the project code with the proper nullness annotations.

9.2.2 Integrated Bug Prediction

All previous studies in bug prediction, including ours, are detached from the IDE. We argue that insights from bug prediction should be embedded in the IDE. We envision a bug prediction plugin that automatically collects metrics, empirically finds the best machine learning optimizations and bug prediction configurations, and then makes predictions on the current state of the project. These predictions are then presented to the developer in the IDE as insights on where to focus attention.

However, there is still a missing step to implement such a plugin: the link between the predictions and developer actions. Currently, bug prediction studies do not take into account the amount of testing and reviewing invested in a certain software entity. When developers act upon the suggestions of a bug predictor, future predictions should take these actions into account. This feedback loop is missing from state-of-the-art bug prediction studies. One way to implement this feedback loop is to include testing metrics as dependent variables, *e.g.*, coverage metrics and number of tests for each software module. We believe that such metrics would improve the accuracy and usability of a bug predictor, but further research is needed to verify this claim. We actually have already started started building the necessary infrastructure to carry out such studies, as explained in Appendix B.

9.2.3 Putting AQSA Tools to Work

The main goal of auxiliary software quality tools is to improve the efficiency of software testing and code reviewing. This goal should also be evaluated by putting AQSA tools to work. One possible direction is to integrate bug prediction insights into test case generation. Generating test cases automatically and executing them is a resource-intensive process that needs to be carried out efficiently. We believe that the ranked list of software modules produced by a bug predictor can act as a test case generation strategy. We plan to pursue this direction in the future.

9.2.4 Quality Tool Evolution

Ensuring software quality is a continuous activity. Quality tools that work efficiently on a project during early phases of development may not work in later phases. We argue that software quality tools should also evolve to stay relevant. In the context of this thesis, there are many questions that are interesting to investigate:

1. Do we get similar bug patterns if we analyze early phases of projects and advanced phases?
2. How often does a bug predictor need to be (re)trained? What metrics should it be trained with in different phases?
3. How does the efficiency of bug prediction tools evolve as projects evolve?
4. Should software developers use different tools in different phases?

We believe that answering these questions may reveal interesting findings that can change the way we develop software quality tools and adopt them.

Bibliography

[1] M. Acharya, T. Xie, J. Pei, and J. Xu. Mining API patterns as partial orders from source code: From usage scenarios to specifications. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 25–34, New York, NY, USA, 2007. ACM.

[2] A. F. Ackerman, L. S. Buchwald, and F. H. Lewski. Software inspections: an effective verification process. *IEEE software*, 6(3):31–36, 1989.

[3] M. P. Allen. The problem of multicollinearity. *Understanding Regression Analysis*, pages 176–180, 1997.

[4] H. Altinger, S. Herbold, F. Schneemann, J. Grabowski, and F. Wotawa. Performance tuning for automotive software fault prediction. In *2017 IEEE 24th International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2017.

[5] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y.-G. Guéhéneuc. Is it a bug or an enhancement?: a text-based approach to classify change requests. In *Proceedings of the 2008 conference of the center for advanced studies on collaborative research: meeting of minds*, page 23. ACM, 2008.

[6] E. Arisholm, L. C. Briand, and E. B. Johannessen. A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. *J. Syst. Softw.*, 83(1):2–17, Jan. 2010.

[7] D. Astels. *Test-Driven Development — A Practical Guide*. Prentice Hall, 2003.

[8] A. Atla, R. Tada, V. Sheng, and N. Singireddy. Sensitivity of different machine learning algorithms to noise. *Journal of Computing Sciences in Colleges*, 26(5):96–103, 2011.

[9] N. Ayewah and W. Pugh. Null dereference analysis in practice. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '10, pages 65–72, New York, NY, USA, 2010. ACM.

[10] A. Bacchelli and C. Bird. Expectations, outcomes, and challenges of modern code review. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 712–721, Piscataway, NJ, USA, 2013. IEEE Press.

[11] A. Bachmann, C. Bird, F. Rahman, P. Devanbu, and A. Bernstein. The missing links: bugs and bug-fix commits. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 97–106. ACM, 2010.

[12] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *Construction and analysis of safe, secure, and interoperable smart devices*, pages 49–69. Springer, 2005.

[13] V. Basili, L. Briand, and W. Melo. A validation of object-oriented design metrics as quality indicators. *Software Engineering, IEEE Transactions on*, 22(10):751–761, Oct. 1996.

[14] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison Wesley, 2000.

[15] A. Begel and R. DeLine. Codebook: Social networking over code. In *ICSE Companion*, pages 263–266, 2009.

[16] A. Begel, Y. P. Khoo, and T. Zimmermann. Codebook: discovering and exploiting relationships in software repositories. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 125–134, New York, NY, USA, 2010. ACM.

[17] A. Begel and T. Zimmermann. Keeping up with your friends: Function foo, library bar.dll, and work item 24. In *Proceedings of the 1st Workshop on Web 2.0 for Software Engineering*, Web2SE '10, pages 20–23, New York, NY, USA, 2010. ACM.

[18] C. Beleites, R. Baumgartner, C. Bowman, R. Somorjai, G. Steiner, R. Salzer, and M. G. Sowa. Variance reduction in estimating classification error using sparse datasets. *Chemometrics and intelligent laboratory systems*, 79(1):91–100, 2005.

[19] R. Bellman. *Adaptive control processes: A guided tour.* Princeton University Press, St Martin's Press, 1960.

[20] A. Bernstein, J. Ekanayake, and M. Pinzger. Improving defect prediction using temporal features and non linear models. In *Ninth international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting*, IWPSE '07, pages 11–18, New York, NY, USA, 2007. ACM.

[21] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM*, 53(2):66–75, Feb. 2010.

[22] Understanding the bias-variance tradeoff, accessed June 9, 2016. http://scott.fortmann-roe.com/docs/BiasVariance.html.

[23] M. D. Bond, N. Nethercote, S. W. Kent, S. Z. Guyer, and K. S. McKinley. Tracking bad apples: reporting the origin of null and undefined value errors. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications (OOPSLA'07)*, pages 405–422, New York, NY, USA, 2007. ACM.

[24] A. Bosu and J. C. Carver. Impact of peer code review on peer impression formation: A survey. In *Empirical Software Engineering and Measurement, 2013 ACM/IEEE International Symposium on*, pages 133–142. IEEE, 2013.

[25] L. Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[26] L. C. Briand, J. Wüst, J. W. Daly, and D. V. Porter. Exploring the relationships between design measures and software quality in object-oriented systems. *Journal of Systems and Software*, 51(3):245–273, 2000.

[27] L. C. Briand, J. Wüst, S. V. Ikonomovski, and H. Lounis. Investigating quality factors in object-oriented designs: An industrial case study. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 345–354, New York, NY, USA, 1999. ACM.

[28] R. P. L. Buse and W. Weimer. Synthesizing API usage examples. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 782–792, Piscataway, NJ, USA, 2012. IEEE Press.

[29] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Softw. Pract. Exper.*, 30(7):775–802, June 2000.

[30] J. Cahill, J. M. Hogan, and R. Thomas. Predicting fault-prone software modules with rank sum classification. In *2013 22nd Australian Software Engineering Conference*, pages 211–219. IEEE, 2013.

[31] G. Canfora, A. De Lucia, M. Di Penta, R. Oliveto, A. Panichella, and S. Panichella. Multi-objective cross-project defect prediction. In *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*, pages 252–261, Mar. 2013.

[32] A. Caracciolo, A. Chiş, B. Spasojević, and M. Lungu. Pangea: A workbench for statically analyzing multi-language software corpora. In *Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on*, pages 71–76. IEEE, Sept. 2014.

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

[34] P. Chalin, P. R. James, and F. Rioux. Reducing the use of nullable types through non-null by default and monotonic non-null. *IET Software*, 2(6):515–531, December 2008.

[35] V. U. B. Challagulla, F. B. Bastani, I.-L. Yen, and R. A. Paul. Empirical assessment of machine learning based software defect prediction techniques. *International Journal on Artificial Intelligence Tools*, 17(02):389–400, 2008.

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

[37] V. Dallmeier and T. Zimmermann. Extraction of bug localization benchmarks from history. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 433–436, New York, NY, USA, 2007. ACM.

[38] M. D'Ambros, M. Lanza, and R. Robbes. An extensive comparison of bug prediction approaches. In *Proceedings of MSR 2010 (7th IEEE Working Conference on Mining Software Repositories)*, pages 31–40. IEEE CS Press, 2010.

[39] A. B. De Carvalho, A. Pozo, and S. R. Vergilio. A symbolic fault-prediction model based on multiobjective particle swarm optimization. *Journal of Systems and Software*, 83(5):868–882, 2010.

[40] J. Demšar. Statistical comparisons of classifiers over multiple data sets. *The Journal of Machine Learning Research*, 7:1–30, 2006.

[41] S. Di Martino, F. Ferrucci, C. Gravino, and F. Sarro. A genetic algorithm to configure support vector machines for predicting fault-prone components. In *International Conference on Product Focused Software Process Improvement*, pages 247–261. Springer, 2011.

[42] W. Dietl, S. Dietzel, M. D. Ernst, K. Muşlu, and T. W. Schiller. Building and using pluggable type-checkers. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 681–690, New York, NY, USA, 2011. ACM.

[43] I. Dillig, T. Dillig, and A. Aiken. Static error detection using semantic inconsistency inference. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 435–445, New York, NY, USA, 2007. ACM.

[44] P. Domingos and M. Pazzani. On the optimality of the simple bayesian classifier under zero-one loss. *Mach. Learn.*, 29(2-3):103–130, Nov. 1997.

[45] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 422–431, Piscataway, NJ, USA, 2013. IEEE Press.

[46] T. Ekman and G. Hedin. Pluggable checking and inferencing of non-null types for Java. *Journal of Object Technology*, 6(9):455–475, 2007.

[47] K. O. Elish and M. O. Elish. Predicting defect-prone software modules using support vector machines. *Journal of Systems and Software*, 81(5):649–660, 2008.

[48] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, pages 57–72, New York, NY, USA, 2001. ACM.

[49] E. Evans. *Domain-Driven Design: Tacking Complexity In the Heart of Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[50] E. Evans and M. Fowler. Specifications. In *Proceedings of the 1997 Conference on Pattern Languages of Programming*, pages 97–34, 1997.

[51] M. Fagan. Design and code inspections to reduce errors in program development. *IBM Journal of Research and Development*, 15(3):182, 1976.

[52] M. Fähndrich and R. Leino. Declaring and checking non-null types in an object-oriented language. In *Proceedings of OOPSLA '03, ACM SIGPLAN Notices*, 2003.

[53] M. Fischer, M. Pinzger, and H. Gall. Analyzing and relating bug report data for feature tracking. In *Proceedings IEEE Working Conference on Reverse Engineering (WCRE 2003)*, pages 90–99, Los Alamitos CA, Nov. 2003. IEEE Computer Society Press.

[54] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *Proceedings International Conference on Software Maintenance (ICSM 2003)*, pages 23–32, Los Alamitos CA, Sept. 2003. IEEE Computer Society Press.

[55] C. Flanagan and K. R. M. Leino. *Houdini, an Annotation Assistant for ESC/Java*, pages 500–517. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[56] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 234–245, New York, NY, USA, 2002. ACM.

[57] J. L. Fleiss. Measuring nominal scale agreement among many raters. *Psychological bulletin*, 76(5):378, 1971.

[58] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

[59] M. Gabel and Z. Su. A study of the uniqueness of source code. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 147–156. ACM, 2010.

[60] K. Gao and T. M. Khoshgoftaar. A comprehensive empirical study of count models for software fault prediction. *IEEE Transactions on Reliability*, 56(2):223–236, 2007.

[61] K. Gao, T. M. Khoshgoftaar, H. Wang, and N. Seliya. Choosing software metrics for defect prediction: an investigation on feature selection techniques. *Software: Practice and Experience*, 41(5):579–606, 2011.

[62] M. Ghafari, K. Rubinov, and M. M. Pourhashem K. Mining unit test cases to synthesize API usage examples. *Journal of Software: Evolution and Process*, pages e1841–n/a, 2017. e1841 smr.1841.

[63] B. Ghotra, S. McIntosh, and A. E. Hassan. Revisiting the impact of classification techniques on the performance of defect prediction models. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 789–800. IEEE Press, 2015.

[64] B. Ghotra, S. Mcintosh, and A. E. Hassan. A large-scale study of the impact of feature selection techniques on defect classification models. In *Proceedings of the 14th International Conference on Mining Software Repositories*, MSR '17, pages 146–157, Piscataway, NJ, USA, 2017. IEEE Press.

[65] E. Giger, M. D'Ambros, M. Pinzger, and H. 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.

[66] E. Giger, M. Pinzger, and H. C. Gall. Comparing fine-grained source code changes and code churn for bug prediction. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, pages 83–92. ACM, 2011.

[67] E. Giger, M. Pinzger, and H. C. Gall. Can we predict types of code changes? an empirical analysis. In *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on*, pages 217–226. IEEE, 2012.

[68] I. Gondra. Applying machine learning to software fault-proneness prediction. *Journal of Systems and Software*, 81(2):186–195, 2008.

[69] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, 26(2), 2000.

[70] L. Guo, Y. Ma, B. Cukic, and H. Singh. Robust prediction of fault-proneness by random forests. In *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on*, pages 417–428. IEEE, 2004.

[71] N. Haenni, M. Lungu, N. Schwarz, and O. Nierstrasz. A quantitative analysis of developer information needs in software ecosystems. In *Proceedings of the 2nd Workshop on Ecosystem Architectures (WEA'14)*, pages 1–6, 2014.

[72] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.

[73] M. A. Hall. Correlation-based feature selection for discrete and numeric class machine learning. In *Proceedings of the Seventeenth International Conference on Machine Learning*, pages 359–366. Morgan Kaufmann Publishers Inc., 2000.

[74] M. A. Hall and G. Holmes. Benchmarking attribute selection techniques for discrete class data mining. *IEEE transactions on knowledge and data engineering*, 15(6):1437–1447, 2003.

[75] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell. A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering*, 38(6):1276–1304, 2012.

[76] Haskell 98 Report.

[77] A. E. Hassan and R. C. Holt. The top ten list: Dynamic fault prediction. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 263–272, Washington, DC, USA, 2005. IEEE Computer Society.

[78] T. Hastie, R. Tibshirani, J. Friedman, and J. Franklin. The elements of statistical learning: data mining, inference and prediction. *The Mathematical Intelligencer*, 27(2):83–85, 2005.

[79] H. Hata, O. Mizuno, and T. Kikuno. Bug prediction based on fine-grained module histories. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 200–210, Piscataway, NJ, USA, 2012. IEEE Press.

[80] S. Herbold. Training data selection for cross-project defect prediction. In *Proceedings of the 9th International Conference on Predictive Models in Software Engineering*, PROMISE '13, pages 6:1–6:10, New York, NY, USA, 2013. ACM.

[81] K. Herzig, S. Just, and A. Zeller. It's not a bug, it's a feature: how misclassification impacts bug prediction. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 392–401. IEEE Press, 2013.

[82] A. Hindle, D. M. German, and R. 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.

[83] T. Hoare. Null references: The billion dollar mistake. *Presentation at QCon London*, 2009.

[84] A. Hora, N. Anquetil, S. Ducasse, and S. Allier. Domain specific warnings: Are they any better? In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 441–450, Sept. 2012.

[85] D. Hovemeyer and W. Pugh. Finding bugs is easy. *ACM SIGPLAN Notices*, 39(12):92–106, 2004.

[86] D. Hovemeyer and W. Pugh. Finding more null pointer bugs, but not too many. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '07, pages 9–14, New York, NY, USA, 2007. ACM.

[87] D. Hovemeyer, J. Spacco, and W. Pugh. Evaluating and tuning a static analysis to find null pointer bugs. In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '05, pages 13–19, New York, NY, USA, 2005. ACM.

[88] C.-W. Hsu, C.-C. Chang, C.-J. Lin, et al. A practical guide to support vector classification. Technical report, Department of Computer Science, National Taiwan University, 2003.

[89] A. Janes, M. Scotto, W. Pedrycz, B. Russo, M. Stefanovic, and G. Succi. Identification of defect-prone classes in telecommunication software systems using design metrics. *Information sciences*, 176(24):3711–3734, 2006.

[90] Y. Jiang, B. Cukic, and Y. Ma. Techniques for evaluating fault prediction models. *Empirical Softw. Engg.*, 13(5):561–595, Oct. 2008.

[91] Jlint home page. http://jlint.sourceforge.net/.

[92] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge. Why don't software developers use static analysis tools to find bugs? In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 672–681. IEEE Press, 2013.

[93] M. Jureczko and L. Madeyski. Towards identifying software project clusters with regard to defect prediction. In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, PROMISE '10, pages 9:1–9:10, New York, NY, USA, 2010. ACM.

[94] Y. Kamei, S. Matsumoto, A. Monden, K.-i. Matsumoto, B. Adams, and A. Hassan. Revisiting common bug prediction findings using effort-aware models. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–10, Sept. 2010.

[95] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering*, 39(6):757–773, 2013.

[96] S. Kanmani, V. R. Uthariaraj, V. Sankaranarayanan, and P. Thambidurai. Object-oriented software fault prediction using neural networks. *Information and software technology*, 49(5):483–492, 2007.

[97] A. Kaur and R. Malhotra. Application of random forest in predicting fault-prone classes. In *2008 International Conference on Advanced Computer Theory and Engineering*, pages 37–43. IEEE, 2008.

[98] T. M. Khoshgoftaar and E. B. Allen. Ordering fault-prone software modules. *Software Quality Journal*, 11(1):19–37, 2003.

[99] T. M. Khoshgoftaar, K. Gao, A. Napolitano, and R. Wald. A comparative study of iterative and non-iterative feature selection techniques for software defect prediction. *Information Systems Frontiers*, 16(5):801–822, 2014.

[100] T. M. Khoshgoftaar, K. Gao, and N. Seliya. Attribute selection and imbalanced data: Problems in software defect prediction. In *2010 22nd IEEE International Conference on Tools with Artificial Intelligence*, volume 1, pages 137–144. IEEE, 2010.

[101] T. M. Khoshgoftaar, N. Seliya, and N. Sundaresh. An empirical study of predicting software faults with case-based reasoning. *Software Quality Journal*, 14(2):85–111, 2006.

[102] S. Kim, T. Zimmermann, E. J. W. Jr., and A. Zeller. Predicting faults from cached history. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 489–498, Washington, DC, USA, 2007. IEEE Computer Society.

[103] S. Kimura, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto. Does return null matter? In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week - IEEE Conference on*, pages 244–253, Feb. 2014.

[104] M. Klas, F. Elberzhager, J. Munch, K. Hartjes, and O. von Graevemeyer. Transparent combination of expert and measurement data for defect prediction: an industrial case study. In *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, volume 2, pages 119–128, May 2010.

[105] K. Kobayashi, A. Matsuo, K. Inoue, Y. Hayase, M. Kamimura, and T. Yoshino. ImpactScale: Quantifying change impact to predict faults in large software systems. In *Proceedings of the 2011 27th IEEE International Conference on Software Maintenance*, ICSM '11, pages 43–52, Washington, DC, USA, 2011. IEEE Computer Society.

[106] R. Kohavi and G. H. John. Wrappers for feature subset selection. *Artificial intelligence*, 97(1):273–324, 1997.

[107] S. Krishnan, C. Strasburg, R. R. Lutz, and K. Goševa-Popstojanova. Are change metrics good predictors for an evolving software product line? In *Proceedings of the 7th International Conference on Predictive Models in Software Engineering*, page 7. ACM, 2011.

[108] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue. Do developers update their library dependencies? *Empirical Software Engineering*, pages 1–34, 2017.

[109] R. Lämmel, E. Pek, and J. Starek. Large-scale, AST-based API-usage analysis of open-source Java projects. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, SAC '11, pages 1317–1324, New York, NY, USA, 2011. ACM.

[110] X. B. D. Le, D. Lo, and C. Le Goues. History driven program repair. In *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, volume 1, pages 213–224. IEEE, 2016.

[111] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Trans. Softw. Eng.*, 34(4):485–496, July 2008.

[112] M. Leuenberger, H. Osman, M. Ghafari, and O. Nierstrasz. Harvesting the wisdom of the crowd to infer method nullness in Java. In *Proceedings of the 17th International Working Conference on Source Code Analysis and Manipulation*, SCAM 2017. IEEE, 2017.

[113] M. Leuenberger, H. Osman, M. Ghafari, and O. Nierstrasz. KOWALSKI: Collecting API clients in easy mode. In *Proceedings of the 33rd International Conference on Software Maintenance and Evolution*, ICSME 2017. IEEE, 2017.

[114] Z. Li and Y. Zhou. Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 306–315. ACM, 2005.

[115] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk. Mining energy-greedy API usage patterns in Android apps: an empirical study. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 2–11. ACM, 2014.

[116] B. Livshits and T. Zimmermann. Dynamine: finding common error patterns by mining software revision histories. *SIGSOFT Softw. Eng. Notes*, 30(5):296–305, Sept. 2005.

[117] A. Loginov, E. Yahav, S. Chandra, S. Fink, N. Rinetzky, and M. Nanda. Verifying dereference safety via expanding-scope analysis. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 213–224, New York, NY, USA, 2008. ACM.

[118] M. Lowry, M. Boyd, and D. Kulkami. Towards a theory for integration of mathematical verification and empirical testing. In *Automated Software Engineering, 1998. Proceedings. 13th IEEE International Conference on*, pages 322–331. IEEE, 1998.

[119] Lucia, F. Thung, D. Lo, and L. Jiang. Are faults localizable? In *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on*, pages 74–77, June 2012.

[120] R. Madhavan and R. Komondoor. Null dereference verification via over-approximated weakest pre-conditions analysis. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 1033–1052, New York, NY, USA, 2011. ACM.

[121] R. Malhotra. A systematic review of machine learning techniques for software fault prediction. *Applied Soft Computing*, 27:504–518, 2015.

[122] R. Malhotra, A. Kaur, and Y. Singh. Empirical validation of object-oriented metrics for predicting fault proneness at different severity levels using support vector machines. *International Journal of System Assurance Engineering and Management*, 1(3):269–281, 2010.

[123] R. Malhotra and Y. Singh. On the applicability of machine learning techniques for object oriented software fault prediction. *Software Engineering: An International Journal*, 1(1):24–37, 2011.

[124] R. C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1 edition, 2008.

[125] M. Martinez and M. Monperrus. Mining software repair models for reasoning on the search space of automated program fixing. *Empirical Software Engineering*, 20(1):176–205, 2015.

[126] T. J. McCabe. A measure of complexity. *IEEE Transactions on Software Engineering*, 2(4):308–320, Dec. 1976.

[127] S. McIntosh, Y. Kamei, B. Adams, and A. 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.

[128] T. Mende. Replication of defect prediction studies: problems, pitfalls and recommendations. In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, page 5. ACM, 2010.

[129] T. Mende and R. Koschke. Revisiting the evaluation of defect prediction models. In *Proceedings of the 5th International Conference on Predictor Models in Software Engineering*, PROMISE '09, pages 7:1–7:10, New York, NY, USA, 2009. ACM.

[130] T. Mende and R. Koschke. Effort-aware defect prediction models. In *Proceedings of the 2010 14th European Conference on Software Maintenance and Reengineering*, CSMR '10, pages 107–116, Washington, DC, USA, 2010. IEEE Computer Society.

[131] D. Mendez, B. Baudry, and M. Monperrus. Empirical evidence of large-scale diversity in API usage of object-oriented software. In *Source Code Analysis and Manipulation (SCAM), 2013 IEEE 13th International Working Conference on*, pages 43–52. IEEE, 2013.

[132] A. Meneely, L. Williams, W. Snipes, and J. Osborne. Predicting failures with developer networks and social network analysis. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, pages 13–23, New York, NY, USA, 2008. ACM.

[133] T. Menzies and B. Cukic. When to test less. *IEEE Software*, 17(5):107–112, 2000.

[134] T. Menzies, J. Greenwald, and A. Frank. Data mining static code attributes to learn defect predictors. *Software Engineering, IEEE Transactions on*, 33(1):2–13, Jan. 2007.

[135] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and A. Bener. Defect prediction from static code features: Current results, limitations, new approaches. *Automated Software Engg.*, 17(4):375–407, Dec. 2010.

[136] B. Meyer. Applying design by contract. *IEEE Computer (Special Issue on Inheritance & Classification)*, 25(10):40–52, Oct. 1992.

[137] A. K. Midha. Software configuration management for the 21st century. *Bell Labs Technical Journal*, 2(1):154–165, 1997.

[138] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 1–11, New York, NY, USA, 2002. ACM Press.

[139] A. T. Mısırlı, A. B. Bener, and B. Turhan. An industrial case study of classifier ensembles for locating software defects. *Software Quality Journal*, 19(3):515–536, 2011.

[140] A. Mockus. Organizational volatility and its effects on software defects. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 117–126, New York, NY, USA, 2010. ACM.

[141] A. Mockus and L. Votta. Identifying reasons for software change using historic databases. In *Proceedings of the International Conference on Software Maintenance (ICSM 2000)*, pages 120–130. IEEE Computer Society Press, 2000.

[142] M. Monperrus, M. Bruch, and M. Mezini. Detecting missing method calls in object-oriented software. *ECOOP 2010–Object-Oriented Programming*, pages 2–25, 2010.

[143] R. Moser, W. Pedrycz, and G. Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 181–190, New York, NY, USA, 2008. ACM.

[144] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 284–292, New York, NY, USA, 2005. ACM.

[145] N. Nagappan, T. Ball, and A. 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.

[146] N. Nagappan, B. Murphy, and V. Basili. The influence of organizational structure on software quality: An empirical case study. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 521–530, New York, NY, USA, 2008. ACM.

[147] M. G. Nanda and S. Sinha. Accurate interprocedural null-dereference analysis for Java. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 133–143, Washington, DC, USA, 2009. IEEE Computer Society.

[148] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Graph-based mining of multiple object usage patterns. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 383–392, New York, NY, USA, 2009. ACM.

[149] M. Odersky. Scala language specification v. 2.4. Technical report, École Polytechnique Fédérale de Lausanne, 1015 Lausanne, Switzerland, Mar. 2007.

[150] N. Ohlsson and H. Alberg. Predicting fault-prone software modules in telephone switches. *IEEE Trans. Softw. Eng.*, 22(12):886–894, Dec. 1996.

[151] A. Okutan and O. T. Yıldız. Software defect prediction using bayesian networks. *Empirical Software Engineering*, 19(1):154–181, 2014.

[152] H. Osman. On the non-generalizability in bug prediction. In *Post Proceedings of the Ninth Seminar on Advanced Techniques and Tools for Software Evolution (SATToSE 2016)*, 2016.

[153] H. Osman, M. Ghafari, and O. Nierstrasz. Automatic feature selection by regularization to improve bug prediction accuracy. In *1st International Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE 2017)*, pages 27–32, Feb. 2017.

[154] H. Osman, M. Ghafari, and O. Nierstrasz. Hyperparameter optimization to improve bug prediction accuracy. In *1st International Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE 2017)*, pages 33–38, Feb. 2017.

[155] H. Osman, M. Ghafari, and O. Nierstrasz. The impact of feature selection on predicting the number of bugs. *Information and Software Technology*, page in review, 2017.

[156] H. Osman, M. Ghafari, O. Nierstrasz, and M. Lungu. An extensive analysis of efficient bug prediction configurations. In *Proceedings of the The 13th International Conference on Predictive Models and Data Analytics in Software Engineering*, PROMISE 2017. ACM, 2017.

[157] H. Osman, M. Leuenberger, M. Lungu, and O. Nierstrasz. Tracking null checks in open-source Java systems. In *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Mar. 2016.

[158] H. Osman, M. Lungu, and O. Nierstrasz. Mining frequent bug-fix code changes. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week - IEEE Conference on*, pages 343–347, Feb. 2014.

[159] T. Ostrand, E. Weyuker, and R. Bell. Predicting the location and number of faults in large software systems. *Software Engineering, IEEE Transactions on*, 31(4):340–355, Apr. 2005.

[160] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. Where the bugs are. In *ACM SIGSOFT Software Engineering Notes*, volume 29, pages 86–96. ACM, 2004.

[161] K. Pan, S. Kim, and E. J. Whitehead, Jr. Toward an understanding of bug fix patterns. *Empirical Softw. Engg.*, 14(3):286–315, June 2009.

[162] A. Panichella, R. Oliveto, and A. 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, Feb. 2014.

[163] M. M. Papi, M. Ali, T. L. Correa, Jr., J. H. Perkins, and M. D. Ernst. Practical pluggable types for java. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 201–212, New York, NY, USA, 2008. ACM.

[164] M. M. Papi and M. D. Ernst. Compile-time type-checking for custom type qualifiers in Java. In *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*, OOPSLA '07, pages 809–810, New York, NY, USA, 2007. ACM.

[165] M. Pinzger, N. Nagappan, and B. Murphy. Can developer-module networks predict failures? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, pages 2–12, New York, NY, USA, 2008. ACM.

[166] Pmd home page. http://pmd.sourceforge.net/.

[167] P. Pominville, F. Qian, R. Vallée-Rai, L. Hendren, and C. Verbrugge. A framework for optimizing Java using attributes. In *Proceedings of the 2000 conference of the Centre for Advanced Studies on Collaborative research*, page 8. IBM Press, 2000.

[168] M. Pradel, C. Jaspan, J. Aldrich, and T. R. Gross. Statically checking API protocol conformance with mined multi-object specifications. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 925–935, Piscataway, NJ, USA, 2012. IEEE Press.

[169] R. Purushothaman and D. E. Perry. Towards understanding the rhetoric of small changes-extended abstract. In *International Workshop on Mining Software Repositories (MSR 2004), International Conference on Software Engineering*, pages 90–94. IET, 2004.

[170] G. K. Rajbahadur, S. Wang, Y. Kamei, and A. E. Hassan. The impact of using regression models to build defect classifiers. In *Proceedings of the 14th International Conference on Mining Software Repositories*, pages 135–145. IEEE Press, 2017.

[171] M. K. Ramanathan, A. Grama, and S. Jagannathan. Path-sensitive inference of function precedence protocols. In *Proceedings of the 29th international conference on Software Engineering*, pages 240–250. IEEE Computer Society, 2007.

[172] S. S. Rathore and S. Kumar. An empirical study of some software fault prediction techniques for the number of faults prediction. *Soft Computing*, pages 1–18, 2016.

[173] L. Rising and N. S. Janoff. The scrum software development process for small teams. *IEEE Software*, 17(4):26–32, July 2000.

[174] C. D. Roover, R. Laemmel, and E. Pek. Multi-dimensional exploration of API usage. In *2013 21st International Conference on Program Comprehension (ICPC)*, pages 152–161, May 2013.

[175] C. Rosen, B. Grawi, and E. Shihab. Commit guru: analytics and risk prediction of software commits. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 966–969. ACM, 2015.

[176] J. R. Ruthruff, J. Penix, J. D. Morgenthaler, S. Elbaum, and G. Rothermel. Predicting accurate and actionable static analysis warnings: An experimental approach. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 341–350, New York, NY, USA, 2008. ACM.

[177] M. A. Saied, O. Benomar, H. Abdeen, and H. Sahraoui. Mining multi-level api usage patterns. In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, pages 23–32. IEEE, 2015.

[178] H. E. Salman. Identification multi-level frequent usage patterns from apis. *Journal of Systems and Software*, 130:42–56, 2017.

[179] F. Sarro, S. Di Martino, F. Ferrucci, and C. Gravino. A further analysis on the use of genetic algorithm to configure support vector machines for inter-release fault prediction. In *Proceedings of the 27th annual ACM symposium on applied computing*, pages 1215–1220. ACM, 2012.

[180] A. A. Sawant and A. Bacchelli. A dataset for API usage. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, pages 506–509. IEEE Press, 2015.

[181] J. Sayyad Shirabad and T. Menzies. The PROMISE repository of software engineering databases. School of Information Technology and Engineering, University of Ottawa, Canada, 2005.

[182] K. Schwaber and M. Beedle. *Agile Software Development with Scrum*. Alan R. Apt, first edition, 2001.

[183] A. J. Scott and M. Knott. A cluster analysis method for grouping means in the analysis of variance. *Biometrics*, pages 507–512, 1974.

[184] M. Shepperd, D. Bowes, and T. Hall. Researcher bias: The use of machine learning in software defect prediction. *IEEE Transactions on Software Engineering*, 40(6):603–616, 2014.

[185] S. A. Sherer. Software fault prediction. *Journal of Systems and Software*, 29(2):97–105, 1995.

[186] E. Shihab, A. E. Hassan, B. Adams, and Z. M. Jiang. An industrial study on the risk of software changes. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 62. ACM, 2012.

[187] S. Shivaji, E. J. Whitehead, R. Akella, and S. Kim. Reducing features to improve code change-based bug prediction. *IEEE Transactions on Software Engineering*, 39(4):552–569, 2013.

[188] P. Singh and S. Verma. An investigation of the effect of discretization on defect prediction using static measures. In *Advances in Computing, Control, & Telecommunication Technologies, 2009. ACT'09. International Conference on*, pages 837–839. IEEE, 2009.

[189] Y. Singh, A. Kaur, and R. Malhotra. Prediction of fault-prone software modules using statistical and machine learning methods. *International Journal of Computer Applications*, 1(22):8–15, 2010.

[190] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *Proceedings of International Workshop on Mining Software Repositories — MSR'05*, Saint Lous, Missouri, USA, 2005. ACM Press.

[191] R. Subramanyam and M. Krishnan. Empirical analysis of ck metrics for object-oriented design complexity: implications for software defects. *Software Engineering, IEEE Transactions on*, 29(4):297–310, Apr. 2003.

[192] M.-H. Tang, M.-H. Kao, and M.-H. Chen. An empirical study on object-oriented metrics. In *Software Metrics Symposium, 1999. Proceedings. Sixth International*, pages 242–249, 1999.

[193] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto. Automated parameter optimization of classification techniques for defect prediction models. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 321–332, New York, NY, USA, 2016. ACM.

[194] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto. An empirical comparison of model validation techniques for defect prediction models. *IEEE Transactions on Software Engineering*, 43(1):1–18, 2017.

[195] F. Thung, S. Wang, D. Lo, and L. Jiang. An empirical study of bugs in machine learning systems. In *Software Reliability Engineering (ISSRE), 2012 IEEE 23rd International Symposium on*, pages 271–280, Nov. 2012.

[196] A. Tomb, G. Brat, and W. Visser. Variably interprocedural program analysis for runtime error detection. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07, pages 97–107, New York, NY, USA, 2007. ACM.

[197] A. Tosun and A. Bener. Reducing false alarms in software defect prediction by decision threshold optimization. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, pages 477–480. IEEE Computer Society, 2009.

[198] B. Turhan and A. Bener. A multivariate analysis of static code attributes for defect prediction. In *Seventh International Conference on Quality Software (QSIC 2007)*, pages 231–237. IEEE, 2007.

[199] B. Turhan and A. Bener. Analysis of naive bayes' assumptions on software fault data: An empirical study. *Data & Knowledge Engineering*, 68(2):278–290, 2009.

[200] B. Twala. Software faults prediction using multiple classifiers. In *Computer Research and Development (ICCRD), 2011 3rd International Conference on*, volume 4, pages 504–510. IEEE, 2011.

[201] R.-G. Urma. Tired of null pointer exceptions? Consider using Java SE 8's optional! Technical report, Oracle, Mar. 2014.

[202] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot — a Java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM Press, 1999.

[203] O. Vandecruys, D. Martens, B. Baesens, C. Mues, M. De Backer, and R. Haesen. Mining software repositories for comprehensible software fault prediction models. *Journal of Systems and software*, 81(5):823–839, 2008.

[204] P. Wang, C. Jin, and S.-W. Jin. Software defect prediction scheme based on feature selection. In *Information Science and Engineering (ISISE), 2012 International Symposium on*, pages 477–480. IEEE, 2012.

[205] S. Wang, D. Chollak, D. Movshovitz-Attias, and L. Tan. Bugram: bug detection with n-gram language models. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 708–719. ACM, 2016.

[206] S. Wang and X. Yao. Using class imbalance learning for software defect prediction. *IEEE Transactions on Reliability*, 62(2):434–443, 2013.

[207] A. Wasylkowski, A. Zeller, and C. Lindig. Detecting object usage anomalies. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 35–44, New York, NY, USA, 2007. ACM.

[208] E. J. Weyuker, T. J. Ostrand, and R. M. Bell. Using developer information as a factor for fault prediction. In *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, PROMISE '07, pages 8–, Washington, DC, USA, 2007. IEEE Computer Society.

[209] E. J. Weyuker, T. J. Ostrand, and R. M. Bell. Do too many cooks spoil the broth? using the number of developers to enhance defect prediction models. *Empirical Softw. Engg.*, 13(5):539–559, Oct. 2008.

[210] L. Williams, G. Kudrjavets, and N. Nagappan. On the effectiveness of unit test automation at microsoft. In *Software Reliability Engineering, 2009. ISSRE'09. 20th International Symposium on*, pages 81–89. IEEE, 2009.

[211] B. Woolf. The null object pattern. In *Design Patterns, PLoP 1996*. Robert Allerton Park and Conference Center, University of Illinois at Urbana-Champaign, Monticello, Illinois, 1996.

[212] B. Woolf. Null object. In R. Martin, D. Riehle, and F. Buschmann, editors, *Pattern Languages of Program Design 3*, pages 5–18. Addison Wesley, 1998.

[213] A. Ying, G. Murphy, R. Ng, and M. Chu-Carroll. Predicting source code changes by mining change history. *Software Engineering, IEEE Transactions on*, 30(9):574 – 586, Sept. 2004.

[214] C. Zhang, J. Yang, Y. Zhang, J. Fan, X. Zhang, J. Zhao, and P. Ou. Automatic parameter recommendation for practical api usage. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 826–836, Piscataway, NJ, USA, 2012. IEEE Press.

[215] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. MAPO: Mining and recommending API usage patterns. In S. Drossopoulou, editor, *ECOOP 2009 - Object-Oriented Programming*, volume 5653 of *Lecture Notes in Computer Science*, pages 318–343. Springer Berlin Heidelberg, 2009.

[216] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy. Cross-project defect prediction: A large scale experiment on data vs. domain vs. process. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 91–100, New York, NY, USA, 2009. ACM.

A
Collecting API Clients with KOWALSKI

Understanding API usage is important for upstream and downstream developers. For upstream developers it is important to know how their APIs are used, so that they can estimate the impact of changes. Downstream developers require a self-explanatory API in the best case or at least documentation [71]. With the lack of documentation, usage examples of an API serve as a good entry point to learn and explore the API [28], but finding the clients of a specific API to extract usage patterns is a non-trivial task. Many studies therefore analyze a few hand-selected projects to mine the API usage [1][115]. Others collect usage patterns by analyzing a large corpus of projects and select those patterns with the highest support [174][109]. Few studies mine unit test cases to synthesize API usage examples when other sources of client code are rare [62]. Nevertheless, high diversity exists in API usage [131] since an API can provide many callable methods, while different clients make use of different subsets of them. Therefore, to find different possible usages of an API, one should find enough client code to cover as many usage scenarios as possible.

In this chapter, we present KOWALSKI, a tool to collect clients of specific Java APIs. KOWALSKI exploits the wide-spread use of Maven as a dependency management system. KOWALSKI takes the name of an API as an input, crawls Maven repositories, and outputs JAR (Java ARchive) artifacts of the API clients, including their dependencies. While most existing large-scale miners operate on sources with limited type-awareness only [45], KOWALSKI enables type-aware analysis of API clients as it collects them in bytecode format. The classes referenced in the client bytecode can be resolved, so that typed call graphs can be constructed. The call graphs could be used to extract protocols, *i.e.*, methods that need to be called

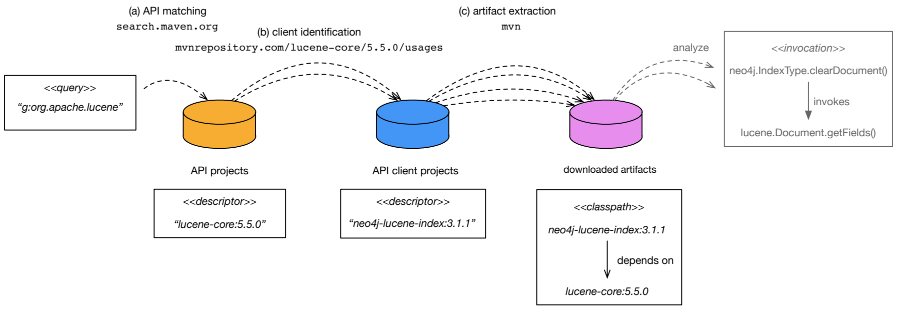


Figure A.2: Dataflow in KOWALSKI from the initial query to the class path of dowloaded binaries to analyze.

in a certain order [148][168][171]. Moreover, KOWALSKI facilitates tracking of the evolution of clients and APIs, as collected artifacts are tagged with their version numbers. The source code of KOWALSKI is available on GitHub.[1]

We use KOWALSKI to collect clients of Apache Lucene, the de facto standard for full-text search, available in the Maven Central repository.[2] Within one hour KOWALSKI collects 7,755 client artifacts of Lucene, for which we extract call graphs in six hours. From the call graphs we determine in how many clients a Lucene method is used and how often a method is used in those clients. API developers can use this information to distinguish between API hotspots, which affect many clients if changed, and cold spots, which can be changed with little impact. We find hotspots in the high-level API methods to create queries and documents, and to read from and write to the full-text index. Cold spots are API methods that deal with file format of the index. The dataset, KOWALSKI binaries, and setup scripts for this experiment can be fetched from Figshare.[3]

We also use KOWALSKI to collect clients of Apache Lucene to infer the nullness of API methods [112], *i.e.*, whether a method may return a null value or not.

A.1 API Client Collection

We want to find clients of a specific API, so that our downstream analysis finds many API calls. We also require the called methods to be precisely identifiable. For that we need type information about the called methods, namely method signatures and declaring types. As APIs evolve over time, methods may be added, removed, or change the contract. Different versions of an API may co-exist. For example, the method `org.apache.lucene.search.Weight.scorer()` never returns null in an early version of Lucene, but does so in later versions.[4] Hence a method invocation

[1] `https://github.com/maenu/kowalski`
[2] `http://search.maven.org/`
[3] `https://figshare.com/projects/KOWALSKI_ICSME_Tools_2017/22756`
[4] `https://issues.apache.org/jira/browse/JCR-3481`

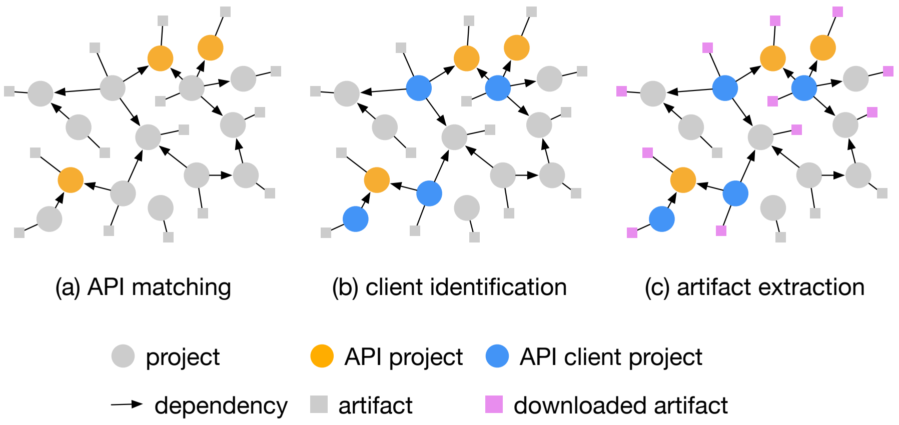


Figure A.1: Dependency subgraph extraction steps with project nodes marked as API, API client and artifacts.

must be traceable to the method signature, the declaring class, the declaring API, and the API's version. This information constructs a *universally unique method identifier*. We need a way to find the API clients and the universally unique identifier of the called API methods.

Collecting clients of a specific API means that we need to extract a subgraph of the dependency graph spanned by all projects, as shown in Figure A.1. First, we need to match the APIs for which we want to collect clients (a). Second, we need to identify clients of the matched APIs (b). Third, we need to extract the API client artifacts we want to analyze (c). These artifacts can be sources, binaries, or documentation.

Many Java projects use Maven as a dependency management system, which we can exploit for our purpose. Maven projects declare their dependencies in a meta-data file. For example, one version of Neo4j declares the artifact descriptor `org.neo4j:neo4j-lucene-index`, version `3.1.1`, and a dependency to Lucene version `5.5.0`. Neo4j developers use Maven to automatically collect the required Lucene dependency from a package repository. Just as Lucene, Neo4j itself is published to this package repository. Therefore, both the API and its client are stored in the same repository and linked through the declared dependency.

A.2 Implementation

We implement the aforementioned dependency subgraph extraction in KOWALSKI. KOWALSKI is designed for high concurrency and collects artifacts rapidly. In Figure A.2 we show how the extraction is implemented using the three *tasks* (a, b, c) as introduced in Figure A.1. Each task is run in a *job* that pipes the input and output of the tasks between *streams* of cached intermediate results.

A.2.1 Tasks

The three tasks are decoupled from each other, so that multiple instances of the same task run in parallel. The output of one task is the input of another task, which enables piping different tasks in sequence.

The *API matching* task (a) finds projects that match a query for *Maven Central Search*.[5] For example, to find all Apache Lucene versions, the task takes a query in the form `g:org.apache.lucene` as input. The task then collects all matching artifacts and outputs their descriptors, *e.g.*, `org.apache.lucene:lucene-core:5.5.0` for Lucene 5.5.0.

The *client identification* task (b) accepts an artifact descriptor and collects their clients. Clients of an artifact are found by scraping the *mvnrepository* website.[6] The task outputs artifact descriptors again, *e.g.*, `org.neo4j:neo4j-lucene-index:3.1.1` for Neo4j 3.1.1, as it is a client of Lucene.

The *artifact extraction* task (c) takes an artifact descriptor and collects the corresponding JAR binaries, including dependencies. This task can be configured using a traditional Maven *setting.xml* to declare the repositories the JARs should be fetched from. It is also possible to configure the dependency scopes that should be used to resolve the necessary dependencies. For instance, a dependency to a unit testing framework is declared in the *test* scope. If we were to analyze tests, we could configure the artifact extraction task to include the test dependencies.

A.2.2 Jobs

The tasks are executed in jobs, denoted by the dashed arrows in Figure A.2. Jobs are responsible for providing input for the wrapped tasks and deciding what to do with the task's output. The output stream of one job is the input stream of another job. The chaining of jobs through streams acts as the composition mechanism to build the pipeline required to collect the dataset. A job reads the task's input from an input stream, delegates the input to the task to process, and writes the task's output to an output stream. Since tasks are independent, and the access to input and output streams is synchronized, jobs can be executed concurrently. Streams are named, so that a job can be configured by the type of the task it executes together with the names of the input and output streams.

A.2.3 Collector

The collector runs multiple workers that are responsible for carrying out the jobs. A worker is merely a thread running a job. All workers are executed in parallel by the collector. The collector can be configured by specifying how many workers for each job are instantiated. At startup, all jobs are created based on this configuration and executed with the number of workers desired.

[5]See `http://search.maven.org/#api` REST API

[6]For example `https://mvnrepository.com/artifact/org.apache.lucene/lucene-core/5.5.0/usages`

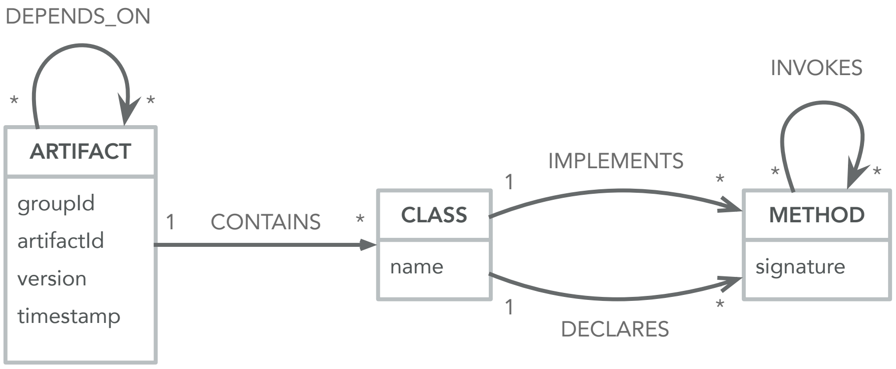


Figure A.3: Neo4j database schema for the analysis call graph.

A.3 Experiment

For our experiment we extract call graphs of Apache Lucene clients and identify Lucene API hotspots, which are the most used methods in the API. We deploy KOWALSKI to a multi-core Ubuntu server to collect the clients. The collection and analysis runs on a 64 bit Ubuntu machine with 32 cores at 1.4 GHz and 128 GB of RAM. We use Apache Artemis as a JMS server to persist the streams of intermediate results and we deploy Neo4j as the database to store the extracted call graphs. The artifact extraction task from KOWALSKI writes downloaded JAR and POM artifacts to the local Maven repository, so that they can be read by the downstream analysis. We run 16 worker threads to collect the dependencies of multiple API clients in parallel. The collected clients are analyzed in 4 concurrent processes.

The analysis applied to the API clients processes all methods that are defined in classes of the client artifact. All invocations of Lucene methods are extracted. The invoked methods are tracked to the defining classes and Lucene version, so that for each analyzed artifact, a typed call graph is stored in the database. The schema of the database is shown in Figure A.3. As each artifact is stored exactly once in the database, different clients calling the same API method in the same API artifact are reflected by multiple incoming invocations of the same method. Therefore, it is not just that we have a call graph for each analyzed client, but we have an aggregated call graph over the whole dataset.

It takes one hour to collect the 7,755 identified clients and six hours to analyze them. The analysis of a single client takes eleven seconds on average. As the analysis starts as soon as the binary artifacts of the first client are extracted, the whole process terminates within six hours. 1,685 binaries are part of Lucene itself, as it is a multi-module project. In 3,009 of non-Lucene binaries we find invocations to Lucene. In the remaining 3,061 binaries we cannot find Lucene usage, as Lucene is a transitive dependency and not directly used by the analyzed methods. The binaries belong to 186 different projects identified by the unique artifact descriptor. We group Lucene and client releases by major version to get an overview of the usage, as shown in

Table A.1: Client major versions per Lucene major version, invoked Lucene methods, and their invocations.

Lucene version	clients	methods	invocations
1	1	23	37
2	48	999	8,819
3	87	1,141	11,619
4	109	1,446	12,830
5	45	1,794	24,773
6	35	1,299	6,720
7	3	4	7

Table A.1. For Lucene versions 1 and 7 we observe very small usage, so we ignore them in the experiment hereafter.

Figure A.4 shows for each major Lucene version how widely a method is used among clients and how often it is invoked when used. The product of these two usage metrics results in the number of total invocations of a method, denoted by the color of a method point. This evaluation can serve as an estimation of the impact when Lucene changes its API methods. From the plots we can read how many call sites need to be refactored when a Lucene method changes its signature and how many clients are affected. We find hotspots in the high-level API methods to create queries and documents, and to read from and write to the full-text index. Cold spots are API methods that deal with file format of the index. We observe that the general usage changes over time. While some methods are used in 40 out of 48 clients of Lucene 2, the most widespread methods are used in only 17 out of 35 clients for Lucene 6. In newer Lucene versions, methods are generally invoked more often. Two projects, Elasticsearch and Solr, which have grown together with Lucene, are responsible for this phenomenon.

A.4 Limitations and Threats to Validity

For our experiment KOWALSKI collects only Lucene and other OSS projects that depend on Lucene and are published on Maven Central. First, this excludes all closed source projects. If given access to a company repository, KOWALSKI can also be used to collect a dataset of clients of a company-internal library. The static analysis can be reapplied as well. Second, all open source projects that are not published on Maven Central are excluded. There are other popular Maven repositories that may contain other Lucene clients, for example jcenter[7] and clojars.[8] However, Maven

[7] `https://bintray.com/bintray/jcenter`
[8] `https://clojars.org/`

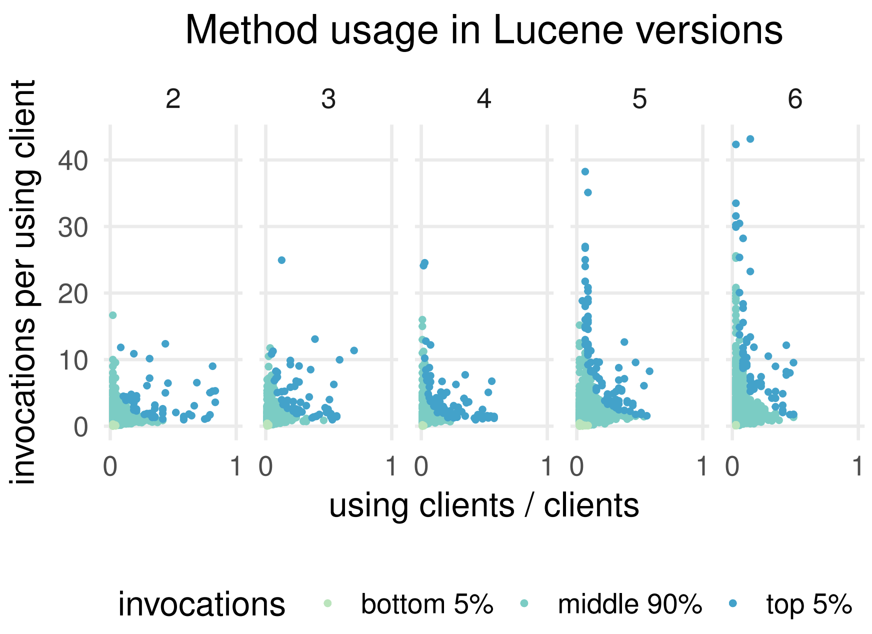


Figure A.4: Method usage in Lucene versions 2 to 6. We show in how many clients a method is used, and how often it is called in those clients.

Central is a large repository that serves 1,935,045 versions of 185,693 artifacts.[9] Package repositories are primarily used to distribute reusable libraries, therefore our dataset has a strong bias towards libraries as clients. Libraries may use Lucene differently than projects further down the dependency hierarchy. Third, we lack a measure to estimate how many Lucene clients are only distributed as sources, for example on GitHub. As our analysis is tailored to run on binaries, it would require a build of these projects. While building arbitrary projects from source is non-trivial [109], we conjecture the widespread use of Travis CI among active projects might facilitate this issue. The identification of clients of a dependency on GitHub requires a searchable index similar to *mvnrepository* for Maven, but for GitHub projects. Parsing POMs alone will not detect clients that rely on a transitive dependency [108]. Identifying transitive dependencies requires resolving all dependencies of a projects, which is an time-intensive task. Fourth, we only analyze the Lucene ecosystem, and the results may not generalize to other ecosystems.

[9]`https://search.maven.org/#stats`, date of access May 3, 2017

A.5 Related Work

There are several datasets over large parts of GitHub. Google's BigQuery GitHub dataset[10] can be queried for contents of source files. It even runs static analysis remotely,[11] but type information is not provided and must be reconstructed. Dyer *et al.* provide ASTs that include partial type information from sources in GitHub projects in the Boa dataset [45]. The binaries collected by KOWALSKI provide more type information as we can track method invocations to the invoked method and library version without ambiguity. By choosing Maven repositories as our datasource we are restricted to a smaller set of OSS projects than GitHub, but we gain type precision.

Lämmel *et al.* check out 6,286 SourceForge projects and manage to build 1,476 of them with Ant to analyze them for API usage [109]. They manually search for missing dependencies to fix build errors in 15% of the built projects. Instead of building projects from source, we collect binary Maven artifacts with resolved dependencies. We use the SOOT analysis framework that creates phantom references for unresolvable classes.

Sawant *et al.* build a typed dataset for five APIs and their usages in 20,263 GitHub projects using Maven [180]. They use partial compilation to work around unresolvable classes. By compiling from source, projects can be inspected for any revision of a source file in a version control system. Our dataset includes only built binary artifacts, yet they are versioned as well, therefore we can track the evolution of a project as well, although on a coarser level of releases.

A.6 Conclusions

KOWALSKI is a tool to collect API clients for API usage analysis. Our experiment shows that KOWALSKI is performant and produces datasets that can be analyzed to find hotspots.

The KOWALSKI pipeline can be used to collect datasets about any API that is hosted in the supported Maven repositories, from large ecosystems around Apache, Eclipse, or Mozilla artifacts to more focused sets of clients of a single product such as Hibernate, JUnit, or Guava. KOWALSKI can also be used on a company internal repository. The collected clients of closed source frameworks and libraries can be analyzed to identify hotspots.

To conquer the bias in the collected datasets towards libraries, one future direction is to utilize Travis CI for extending KOWALSKI with a task to build projects from GitHub.

[10]https://cloud.google.com/bigquery/

[11]https://medium.com/google-cloud/static%2Djavascript%2Dcode%2Danalysis%2Dwithin%2Dbigquery%2Ded0e3011732c

B Analyzing Commits With BiCo

Developers commit changes to the code base of a certain project in order to, for instance, fix bugs, add features, or refactor the code. In empirical studies, researchers often need to link commits with issues in issue trackers to audit the purpose of code changes. Unfortunately, there exists no general-purpose tool that can fulfill this need for different studies. For instance, while in theory each commit should serve one purpose, in practice developers may include several goals in one commit. Also, issues in issue trackers are often miscategorized.

BiCo (BIg COmmit analyzer) is a tool that links the source code management system with the issue tracker. BiCo presents the information in a navigable form, in order to make it easier to analyze and reason about the evolution of a certain project. It takes advantage of the fact that developers include issue IDs in commit messages to link them together. BiCo also provides dedicated analytics to detect *big commits*, *i.e.*, multi-purpose and miscategorized commits, using statistical outlier detection. In an initial evaluation, we use BiCo to analyze bug-fix commits in Apache Kafka. Our tool reports 9.6% of the bug-fixing commits as miscategorized or multipurpose commits with a precision of 85%. This high precision demonstrates the applicability of the outlier detection method implemented in BiCo.
Screencast: `https://vimeo.com/223184016`
Sourcecode & Instructions: `https://github.com/papagei9/scg-bico`

B.1 Introduction

Developers commit source code changes for many reasons, *e.g.*, to fix bugs, add features, or clean up the code. The purposes of these commits are usually documented in the commit messages themselves. Developers also include issue IDs in commit messages to link their commits with the issues they resolve. This ad-hoc style of linking source code management systems with issue trackers motivates researchers to mine both repositories and deduce knowledge from the evolution of systems, like discovering bug-fix patterns [161][158][110], predicting changes in the code [213][67], and building datasets for bug prediction [38]. Such studies face three main challenges:

Linking commits with issues

Although many techniques have been proposed in the past to approach this problem (*e.g.*, [190]), there exists no general-purpose tool that can be used off the shelf. Each study has its own implementation.

Finding clean commits

Ideally, commits should not be big and each commit should serve one purpose. This is not necessarily true in practice. There are commits that serve multiple purposes at the same time. Studies can produce more reliable results if they rely on clean commits, *i.e.*, uni-purpose commits.

Categorizing issues correctly

It has been shown previously that issues in issue trackers are sometimes miscategorized [5][81]. This threatens the external validity of the studies that rely on accurate categorization.

To address these challenges, we have developed BiCo, a general-purpose tool that links commits and issues, and provides further analytics to detect suspicious commits that either combine multiple purposes or are miscategorized, which we refer to as *big commits*. For instance, issue number 6271 in Apache Lucene[1] is categorized as a bug. After linking this issue with the commit that fixes it, it turns out that this commit changes 3'945 lines of code in 73 files and it is actually a refactoring commit. Eliminating such commits from analysis, or at least the awareness of their controversy, improves the reliability of further empirical studies in software evolution. BiCo implements statistical outlier detection to detect big commits like this one. An initial evaluation to extract and analyze bug-fix commits in Elasticsearch shows that BiCo could categorize 1'489 commits as fixes, 7% of which are detected as big commits with a precision of 85%.

BiCo represents a first step to build a general-purpose infrastructure for analyzing software evolution. To demonstrate its usefulness, we built a metric suite extractor on top of BiCo for calculating change metrics [143], source code metrics [36], and defect counts [190] for any analyzed system at the point in time of any commit. BiCo, with its big commit analysis, provides a usable infrastructure that facilitates this metric extraction.

[1] `https://issues.apache.org/jira/browse/LUCENE-6271`

B.2 BiCo

B.2.1 Linking Commits with Issues

BiCo requires the user to provide the Git URL and the issue tracker URL of the project to be analyzed. BiCo detects issue IDs in commit messages and uses them to link commits with issues. This technique is known in the literature and has been used in several studies [190][38] because developers often include in the commit messages the IDs of the issues resolved by the current commit. For instance, one of the commit messages in Apache Flume states:

"FLUME-774. Move HDFS sink into a separate module"

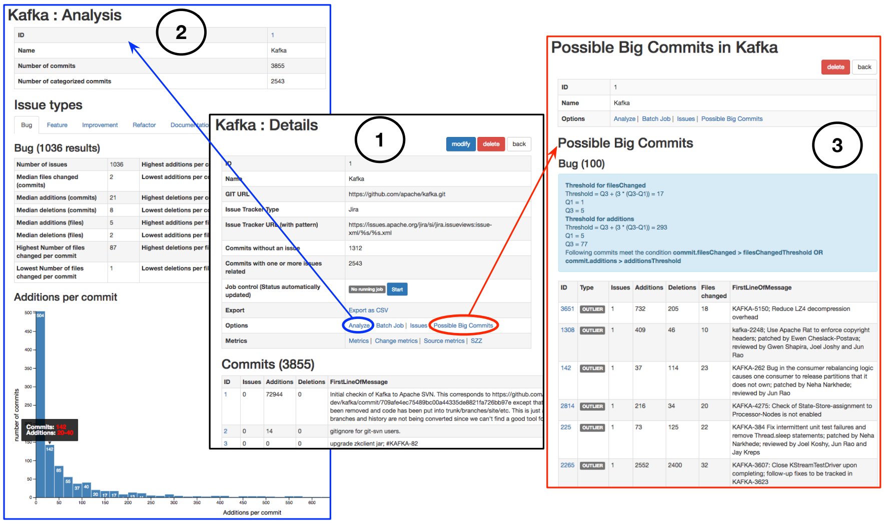
This means that this commit resolves the issue "FLUME-774". BiCo uses these IDs to fetch the issues from issue trackers. However, different projects can have different URLs for their issues and the user needs to provide the template URL for the projects they need to analyze. Also different issue trackers have different issue IDs. Currently, BiCo supports three issue trackers:

1. JIRA[2] is one of the most used issue trackers in the open-source community, provided by Atlassian. Usually, issue reports can be obtained using the following URL template: `https://issues.apache.org/jira/si/jira.issueviews:issue-xml/%s/%s.xml` where the user puts "%s" to instruct BiCo that this is the placeholder for the issue ID. When a project uses this tracker, BiCo looks in the commit messages for the pattern *WORD-NUMBER* and considers it as an issue ID, as this is the format JIRA gives IDs to issues, *e.g.*, ZOOKEEPER-2688, KAFKA-4744, and TIKA-2261.

2. Bugzilla[3] is another widely-used issue tracker provided by Mozilla. Similarly to JIRA, the template is usually `https://bugzilla.mozilla.org/show_bug.cgi?ctype=xml&id=%s`. Since issue IDs are just numbers in Bugzilla, for projects using Bugzilla as an issue tracker, BiCo looks for any number in the commit message and tries to find an issue with this number as an ID.

3. Github Issues[4] is an issue tracker for the projects hosted on Github. More and more projects are moving to Github Issues because of its convenience. The user just needs to provide the Github URL as the issue tracker, *e.g.*, `https://github.com/elastic/elasticsearch`. Similarly to Bugzilla, Github uses plain numbers as issue IDs.

[2]`https://www.atlassian.com/software/jira`
[3]`https://www.bugzilla.org/`
[4]`https://github.com/`



Kafka : Analysis

ID	1
Name	Kafka
Number of commits	3855
Number of categorized commits	2543

Issue types

Bug | Feature | Improvement | Refactor | Documentatio

Bug (1036 results)

Number of issues	1036	Highest additions per c
Median files changed (commits)	2	Lowest additions per c
Median additions (commits)	21	Highest deletions per c
Median deletions (commits)	8	Lowest deletions per c
Median additions (files)	5	Highest additions per fi
Median deletions (files)	2	Lowest additions per fi
Highest Number of files changed per commit	87	Highest deletions per fi
Lowest Number of files changed per commit	1	Lowest deletions per fil

Additions per commit

Kafka : Details

modify | delete | back

ID	1
Name	Kafka
GIT URL	https://github.com/apache/kafka.git
Issue Tracker Type	Jira
Issue Tracker URL (with pattern)	https://issues.apache.org/jira/si/jira.issueviews:issue-xml/%s/%s.xml
Commits without an issue	1312
Commits with one or more issues related	2543
Job control (Status automatically updated)	No running job Start
Export	Export as CSV
Options	Analyze | Batch Job | Issues | Possible Big Commits
Metrics	Metrics | Change metrics | Source metrics | SZZ

Commits (3855)

ID	Issues	Additions	Deletions	FirstLineOfMessage
1	0	72944	0	Initial checkin of Kafka to Apache SVN. This corresponds to https://github.com dev/kafka/commit/709afe4ec75489bc00a44335de8821fa726bb97e except that been removed and code has been put into trunk/branches/site/etc. This is just branches and history are not being converted since we can't find a good tool fo
2	0	14	0	gitignore for git-svn users.
3	0	0	0	upgrade zkclient jar; #KAFKA-82

Possible Big Commits in Kafka

delete | back

ID	1
Name	Kafka
Options	Analyze | Batch Job | Issues | Possible Big Commits

Possible Big Commits

Bug (100)

Threshold for filesChanged
Threshold = Q3 + (3 * (Q3-Q1)) = 17
Q1 = 1
Q3 = 5
Threshold for additions
Threshold = Q3 + (3 * (Q3-Q1)) = 293
Q1 = 5
Q3 = 77
Following commits meet the condition **commit.filesChanged > filesChangedThreshold OR commit.additions > additionsThreshold**

ID	Type	Issues	Additions	Deletions	Files changed	FirstLineOfMessage
3651	OUTLIER	1	732	205	18	KAFKA-5150; Reduce LZ4 decompression overhead
1308	OUTLIER	1	409	46	10	kafka-2248; Use Apache Rat to enforce copyright headers; patched by Ewen Cheslack-Postava; reviewed by Gwen Shapira, Joel Joshy and Jun Rao
142	OUTLIER	1	37	114	23	KAFKA-262 Bug in the consumer rebalancing logic causes one consumer to release partitions that it does not own; patched by Neha Narkhede; reviewed by Jun Rao
2814	OUTLIER	1	216	34	20	KAFKA-4275: Check of State-Store-assignment to Processor-Nodes is not enabled
225	OUTLIER	1	73	125	22	KAFKA-384 Fix intermittent unit test failures and remove Thread.sleep statements; patched by Neha Narkhede; reviewed by Joel Koshy, Jun Rao and Jay Kreps
2265	OUTLIER	1	2552	2400	32	KAFKA-3607: Close KStreamTestDriver upon completing; follow-up fixes to be tracked in KAFKA-3623

Figure B.1: Screenshots of the BICO showing the main functionality. From the overview of project details (1), one can navigate to an analysis of the project (2), or to a list of possible big commits (3).

B.2.2 Under the Hood

Under the hood, BICO is mainly a batch job tool. Starting from the UI, when a user adds a new project to be analyzed, BICO starts a batch job in the backend that does the actual heavy lifting. Each batch job consists of two steps. In the first step BICO clones the repository, parses the commits, extracts issue IDs from commit messages, and saves the results in the database. As soon as the first step is successfully completed, the second step starts, where BICO retrieves all the issues for all the IDs extracted in the previous step and links them with the commits in the database. After this step, the user can explore the extracted data via the web interface. BICO also provides a means to control the batch jobs themselves. Users can stop, restart, pause, and resume the jobs at anytime. The batch job backend is implemented in Spring Batch and the web front-end is implemented in Spring MVC.[5]

B.2.3 Big Commit Detection

BICO implements an outlier detection method to find big commits, as in Figure B.1(3). Since BICO keeps data about the number of changed files and the number of changed lines in each commit, it can calculate the first quartile Q1 and the third quartile Q3 of these metrics in each category of commits. Using these quartiles, BICO uses the definition of *extreme outliers* in statistics to detect big commits. Such outliers are

[5]https://spring.io/

data points that are smaller than $Q1 - 3 \times IQR$ (*i.e.*, lower outer fence) or larger than $Q3 + 3 \times IQR$ (*i.e.*, upper outer fence), where $IQR = Q3 - Q1$ (intermediate quartile range). In BICO, big commits are commits that have more changed files or lines of code than the extreme outlier threshold within that commit category. The main rationale behind this approach is that different types of source code changes have different characteristics. For instance, code refactoring changes tend to be large whereas bug-fix changes are known to be small [119].

B.2.4 User Interface

All extracted data is saved in a database and can be used separately from BICO itself. However, BICO provides its own UI for users to explore the extracted data. Figure B.1 shows the main functionality of BICO. Screenshot (1) shows the details of the analyzed projects where the user can see all the commits of the projects and can click on a commit to see its details. Also, the user can click "analyze" to perform statistical analysis on the commits, as in screenshot (2), where the commits of each issue type are analyzed as a group. Simple statistics as well as charts are displayed about the commits of the following issue types: bug, feature, improvement, refactoring, and documentation.

B.3 Evaluation

B.3.1 Big Commit Analysis

As an initial evaluation, we use BICO to extract and analyze the bug-fix commits in Apache Kafka, a popular publish/subscribe distributed infrastructure implemented on top of Hadoop.[6] Kafka uses *Github* as the source code management system and *JIRA* as an issue tracker. BICO was able to categorize 1036 commits as fixes using the categories of the linked issues. However, BICO detects that 100 of the fixing commits (9.6%) are big commits. We manually investigated the reported big commits in the bug-fix category and observe that only 15 of them are false positives, *i.e.*, they are actually bug-fix commits. The remaining 85 commits are correctly classified as big commits: 33 improvement, 19 multipurpose, 15 feature addition, 9 refactoring, 8 test addition, and 2 documentation addition. This precision of 85% suggests that the statistical outlier detection is a reliable method for detecting big commits and BICO can be used off-the-shelf to analyze project repositories and aid researchers in related empirical studies.

B.3.2 Software Metrics Extraction Use Case

The extraction and accurate categorization of commits in BICO makes it possible to implement a wide range of software evolution analyses. We have implemented a feature in BICO to extract software metrics of any analyzed system. The supported

[6] `https://github.com/apache/kafka`

metrics are the change metrics proposed by Moser *et al.* [143], source code metrics [36], and defect counts from the SZZ algorithm [190]. SZZ can be run once to label Java classes with bug counts across the evolution of the system. Change and source code metrics can be calculated for the system on every commit, on every n^{th} commit, or on a specific commit specified by the commit hash. This functionality allows researchers in the domain of defect prediction to build defect predictors and carry out empirical studies on any git-based system, and not only on the publicly-available datasets such as the bug prediction benchmark [38] or the Tera-PROMISE repository.[7]

B.3.3 Test Repository

We have created a git repository for a synthesized Java project.[8] This repository contains commits of different categories and acts as the ground truth to test BiCo. We are currently in the process of expanding this repository to cover more cases and scenarios to be able to test BiCo more extensively in the future.

B.3.4 Current Limitations and Future Work

In Github issues and Bugzilla, issue IDs are just numbers. Although a commit message may contain numbers that are not issue IDs (*e.g.*, dates, commit hashes), it is highly unlikely to have an issue ID with that same number. However, there might still be some false links between commits and issues. Besides, BiCo only extracts the issues that are linked by commits, because the focus of the tool is on commit analysis. In the future, we plan to extend BiCo to pull all issues of a certain projects regardless of the possibility to link them with commits. Also commits that do not contain issue IDs, are not categorized, but still kept in the database. These commits can be categorized based on the content of the commit messages, but this is not yet implemented.

Several open-source projects (*e.g.*, OpenStack) use the Gerrit code review system.[9] In these projects, commit messages may contain IDs to link commits to code reviews in the Gerrit system. We plan to extend BiCo to analyze code review data, since there is a growing interest in analyzing this type of data.

Although the initial evaluation reveals the high precision of outlier detection in BiCo, we plan to carry out a larger and more thorough evaluation, where the precision of big commit detection in each category is analyzed on multiple projects. Besides, we now only measure the precision of BiCo. We need to estimate the recall, as the outlier detection might miss some big commits.

BiCo is in its initial stage and we plan to extend it in multiple directions. We plan to support other outlier detection methods such as statistical tests, depth-based approaches, deviation-based approaches, and distance-based approaches. Different

[7] http://openscience.us/repo/
[8] `https://github.com/papagei9/AcmeStore`
[9] https://www.gerritcodereview.com/

projects might require different methods and we plan to let the user choose the appropriate method of outlier detection. Another direction is to analyze the textual content of the commit message and issue itself to determine the purpose of the commit using natural language processing techniques. The long-term goal is to provide an infrastructure that can extract commits and categorize them reliably to facilitate future related studies.

B.4 Related Work

Mockus and Votta [141] categorize the changes of software based on the textual content in transaction log messages in the Extended Change Management System (ECMS) [137].

Śliverski *et al.* [190] identify the commits that induce fixes in the future. They start from the bug report in an issue tracker, then navigate to the commit that fixes the issue, identify the changed code in that commit, and finally track that changed code to the commit that introduces it. Analyzing Eclipse and Mozilla, Śliverski *et al.* reveal that the average number of changed files in a fix commit tends to be small: 2.73 in Eclipse and 4.39 in Mozilla. Purushothaman and Perry [169] analyze small changes in software using change and defect history data. They find that nearly 10% of changes are one-liners and the maximum number of changes are adaptive (*i.e.*, new features). After a manual inspection of 374 bugs from three systems, Lucia *et al.* [119] find that bug fixes that span more than five files are very rare (7% in Rhino, 1% in AspectJ, 10% in Lucene). Herzig *et al.* [81] manually examined more than 7'000 issues from issue trackers of five open-source projects. They report that between 37% and 47% of issue reports are wrongly typed in issue trackers. This type of study can benefit from an off-the-shelf tool such as BICO.

Fischer *et al.* combine data from Mozilla CVS and the Bugzilla issue tracker into one database called the release history database as a part of a software evolution analysis framework [54]. Then Fischer *et al.* use this combination of data sources to pinpoint and track features in the source code and reveal relationship between features in Mozilla [53]. BICO provides similar functionality but on a wider scale where many issue trackers are supported and any git repository can be analyzed.

Dallmeier and Zimmermann [37] propose `iBUGS`, a tool that extracts bug localization data semiautomatically from software change history. Using patterns like "Fixed 1234" or "Bug #1234" in commit messages, they discover bug-fix commits, but without linking them to issue trackers. The authors made many datasets available using `iBUGS`. BICO provides a step further by linking the commit messages to issues in issue trackers.

Begel *et al.* develop a framework called *Codebook* [15][16][17], that extracts data from several types of software repositories and combines them into people-artifacts graphs. However, *Codebook* is tailored to the infrastructure at Microsoft and cannot be used in other contexts. BICO on the other hand is publicly available and can work with most of open-source project setups.

Rosen *et al.* present Commit Guru [175], a tool that can provide commit analytics for any publicly accessible git repository. The main goal is to identify risky commits (*i.e.*, bug-introducing commit) [186][95]. Commit Guru calculates commit-level metrics (*e.g.*, number of modified files, Age from last change) and use them to predict how risky every commit is. However, using the approach suggested by Hindel *et al.* [82], Commit Guru relies only the commit messages in order to categorize commits. It also assumes that each commit belongs to one category. BiCo is different from Commit Guru in that it links issues and commits, identifies big commits, and extracts change and source code metrics on the Java file level.

Hindle *et al.* [82] define the term *Large Commits* as the commits that include large numbers of files. Hindel *et al.* carried a case study on in 9 open source projects, and manually analyzed the 1% of the commits that contain the largest number of files, any files and not only source code files. They concluded that large commits are likely to be perfective while small commits are likely to be corrective. In this study we define big commits differently, as the commits that are miscategorized or serve multiple purposes. We also detect categories of large commits, rather than just listing all of them.

Bachmann *et al.* introduced Linkster [11], a tool that connects version control history and bug report history to identify defect-fix commits. The main intent of Linkster is to help developers and researchers navigate and annotate commits. BiCo uses similar analysis for linking commits and issues. BiCo also can be used for exploration but is mainly intended for analysis. BiCo identifies categories of commits, identifies big commits, and extract software metrics the analyzed system at any commit.

Although the techniques for linking commits with issues are already explored in the literature, there exists no tool or implementation that can be used. BiCo aims at filling this gap and facilitating reproducible empirical research in software engineering.

B.5 Conclusions

Analyzing software evolution often requires the purposes of code changes to be determined. BiCo is a tool that links information from software code management systems and issue trackers to determine the purposes of source code changes, *i.e.*, commits. BiCo can be used off-the-shelf to analyze software projects, build datasets on software changes, extract software metrics for defect prediction, and explore the collected data.

Declaration of consent

on the basis of Article 28 para. 2 of the RSL05 phil.-nat.

Name/First Name:	OSMAN / Haidar
Matriculation Number:	10-980-670
Study program:	PhD Program in Computer Science
	Bachelor ○ Master ○ [✓ Dissertation]
Title of the thesis:	Empirically-Grounded Construction of Bug Prediction and Detection Tools
Supervisor:	Prof. Dr. Oscar Nierstrasz

I declare herewith that this thesis is my own work and that I have not used any sources other than those stated. I have indicated the adoption of quotations as well as thoughts taken from other authors as such in the thesis. I am aware that the Senate pursuant to Article 36 para. 1 lit. r of the University Act of 5 September, 1996 is authorised to revoke the title awarded on the basis of this thesis. I allow herewith inspection in this thesis.

Place/Date

Signature

Haidar Osman

Personal Information

E-mail:	haidarpi@gmail.com
Nationality:	Syrian
Birthdate:	05.06.1984 (Latakia, Syria)

Research Interests

Empirical Software Engineering, Mining Software Repositories, Bug Prediction, Software Evolution.

Languages

Arabic (Native), English (Fluent), German (A1)

Education

PhD in Computer Science **2013 - 2017**
University of Bern, Bern, Switzerland.
Thesis Title: Empirically-Grounded Construction of Bug Prediction and Detection Tools
Advisor: Prof. Oscar Nierstrasz

MSc in Informatics **2010 - 2013**
Università della Svizzera Italiana (USI), Lugano, Switzerland.
Thesis Title: Web-Based Collaborative Software Modeling
Advisor: Prof. Michele Lanza

BSc in Informatics Engineering **2002 - 2007**
Tishreen University, Latakia, Syria.
Project Title: Topic Maps API
Advisor: Prof. Nasser Ali Nasser

Awards

Swiss Engineering Ticino Award (2nd prize) **2014**
for an outstanding master thesis.

AL-BASEL Top Student Awards **2003, 2005, 2006, 2007**
Top student awards during the bachelor studies.

Presidential Certificate of Honour **2002**
3rd top high school graduate nationwide in Syria.

Work Experience

PhD Researcher **2013 - 2017**
University of Bern, Switzerland

Big Data Engineering Intern **2015 - 2016**
Swisscom, Switzerland

Software Engineer **2008 - 2010**
CyberCode Consulting, Syria

Teaching Assistant **2007 - 2010**
Tishreen University, Syria