# A Novel Bug Prediction Dataset, Process Metrics, and a Public Dataset of JavaScript Bugs

**Péter Gyimesi**

Department of Software Engineering
University of Szeged

Szeged, 2023

Supervisor:

Dr. habil. Rudolf Ferenc

A THESIS SUBMITTED FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
OF THE UNIVERSITY OF SZEGED

University of Szeged
Ph.D. School in Computer Science

*"No thief, however skillful, can rob one of knowledge, and that is why knowledge is the best and safest treasure to acquire."*

— L. Frank Baum, The Lost Princess of Oz

# Preface

In 2000, I acquired my first personal computer as a generous gift from my half-brother. Looking back, it is strange to realize how unfamiliar I was with computers at that time. Up until then, my passion revolved around LEGO, and I derived great joy from constructing various creations. However, when I got my hands on a computer, I discovered a whole new realm of possibilities that captivated me. Subsequently, with the advent of LEGO Mindstorms, I was able to merge my two hobbies: building intricate structures and programming them using a computer. It was during this phase that I realized my inclination to pursue a career in the field of computer science. However, at that time, I did not have a specific end goal in mind, such as obtaining a PhD.

As time progressed, I enrolled in university and started working at the Department of Software Engineering. I embraced the opportunities that came my way and delved into research. Now, I find myself in the midst of writing my doctoral thesis, a culmination of the journey so far that started with my early fascination for computers and the subsequent pursuit of academic and professional opportunities.

I could not have reached this point without assistance, and I want to express my gratitude for the support I received. I am immensely thankful to my supervisor, Dr. Rudolf Ferenc, for his invaluable guidance and profound insights. He has shown me that with dedication and perseverance, I can overcome any challenge and achieve my goals. His mentorship has been extremely valuable, and I am truly grateful for his belief in my potential and the opportunities he has opened for me. I would also like to extend my sincere thanks to Dr. Tibor Gyimóthy, the former head of the Department of Software Engineering, for his support in my journey to become a PhD student. A special acknowledgment goes to my coworkers, Dr. Zoltán Tóth, Béla Vancsics, and Gábor Gyimesi, whose contributions and unwavering support have played a significant role in my progress. I would also like to express my deep appreciation to all of my co-authors for their valuable contributions, as well as to Edit Szűcs for her assistance in providing stylistic and grammatical comments on this thesis.

Last but not least, I would like to express my heartfelt gratitude to my family, especially to my wife, Évi, for believing in me and for their constant encouragement throughout my journey.

*Péter Gyimesi, 2023*

# Contents

**Preface** iii

**1 Introduction** 1

**2 Background** 5

2.1 Bug Taxonomies . . . 5
2.2 Bug Localization and Source Code Management . . . 6
2.3 Public Datasets . . . 7
2.4 Bug Prediction . . . 10
2.5 Source Code Analysis . . . 11
2.6 Software Metrics . . . 12
2.6.1 Software Product Metrics . . . 13
2.6.2 Software Process Metrics . . . 15
2.7 GitHub . . . 18

**3 A Novel Bug Prediction Dataset and its Validation** 21

3.1 Methodology . . . 23
3.1.1 Subject System Selection . . . 23
3.1.2 Data Collection . . . 28
3.1.3 Data Processing . . . 28
3.1.4 Source Code Analysis . . . 30
3.1.5 Extracting the Number of Bugs . . . 30
3.1.6 Combining CSV Files . . . 31
3.1.7 The GitHub Bug Dataset . . . 31
3.1.8 The BugHunter Dataset . . . 31
3.2 Computational Cost of Extending the Dataset . . . 32
3.3 Validation . . . 33
3.3.1 Relationship between Files and Classes . . . 34
3.4 Evaluation . . . 34
3.4.1 Filtering . . . 35
3.4.2 Research Question 1 . . . 38
3.4.3 Research Question 2 . . . 45
3.4.4 Research Question 3 . . . 46
3.5 Threats to Validity . . . 50
3.5.1 Threats to Construct Validity . . . 50
3.5.2 Threats to Internal Validity . . . 51
3.5.3 Threats to External Validity . . . 51
3.6 Summary . . . 51

**4 Calculation of Process Metrics and their Bug Prediction Capabilities 53**
4.1 Difficulty of Computing Process Metrics . . . 54
4.2 Methodology . . . 55
4.2.1 Database Schema . . . 55
4.2.2 Calculating Process Metrics . . . 56
4.2.3 Computing Bug Numbers . . . 58
4.2.4 The Open-source Toolchain . . . 60
4.3 Experimental Set-up . . . 61
4.4 Evaluation . . . 61
4.4.1 Research Question 1 . . . 62
4.4.2 Research Question 2 . . . 65
4.4.3 Research Question 3 . . . 67
4.5 Summary . . . 69

**5 A Public Dataset of JavaScript Bugs 71**
5.1 Methodology . . . 71
5.1.1 Subject Systems Selection . . . 72
5.1.2 Bugs Collection . . . 73
5.1.3 Manual Patch Validation . . . 75
5.1.4 Dynamic Validation . . . 76
5.1.5 Patch Creation . . . 77
5.1.6 Benchmark Infrastructure and Implementation . . . 78
5.1.7 Extending BugsJS . . . 80
5.2 Taxonomy of Bugs in BugsJS . . . 81
5.2.1 Manual Labeling of Bugs . . . 81
5.2.2 Taxonomy Construction . . . 82
5.2.3 Taxonomy Internal Validation . . . 82
5.2.4 The Final Taxonomy . . . 83
5.3 Evaluation . . . 89
5.3.1 Research Question 1 . . . 89
5.3.2 Research Question 2 . . . 91
5.4 Discussion . . . 98
5.4.1 Directing Developer Efforts . . . 99
5.4.2 Limitations . . . 100
5.4.3 Threats to Validity . . . 100
5.5 Summary . . . 101

**6 Conclusions 103**

**Appendices 105**

**A Summary in English 107**

**B Magyar nyelvű összefoglaló 111**

**Bibliography 117**

# List of Tables

2.1 Clone metrics used for characterization . . . . . 13
2.2 Source code metrics used for characterization . . . . . 14
2.3 The number of pushes created on GitHub in 2022 for the top 10 languages 19

3.1 Comparison of the two types of datasets . . . . . 22
3.2 Comparison of the datasets . . . . . 23
3.3 The selected projects and their descriptions . . . . . 25
3.4 Statistics about the selected projects . . . . . 26
3.5 The number of entries in the *GitHub Bug Dataset* . . . . . 32
3.6 *BugHunter Dataset* metadata . . . . . 33
3.7 Validation results . . . . . 34
3.8 Filtering results at method level . . . . . 36
3.9 Significance test results for method level filtering . . . . . 37
3.10 Filtering results at the class level . . . . . 37
3.11 Significance test results for class-level filtering . . . . . 37
3.12 Filtering results at the file level . . . . . 38
3.13 Significance test results for file-level filtering . . . . . 38
3.18 TOP 5 machine learning algorithms for the method level based on F-measure . . . . . 39
3.14 Significance test results for the method level - Algorithms . . . . . 40
3.15 Significance test results for the class level - Algorithms . . . . . 40
3.16 Significance test results for projected - Algorithms . . . . . 41
3.17 Significance test results for the file level - Algorithms . . . . . 41
3.19 The best F-measure values by projects at the method level . . . . . 42
3.20 TOP 5 machine learning algorithms for class level based on F-measure 42
3.21 The best F-measure values by projects at the class level . . . . . 43
3.22 TOP 5 machine learning algorithms for the file level based on F-measure 44
3.23 The best F-measure values by projects at the file level . . . . . 44
3.24 The results of projected learning . . . . . 45
3.25 Comparison of the size of the datasets . . . . . 47
3.26 Predictive capabilities . . . . . 48
3.27 Uncertainty in the traditional dataset . . . . . 49

4.1 The list of implemented process metrics . . . . . 58
4.2 Statistics about the graph databases that we created . . . . . 61
4.3 Average F-measure values at the class level . . . . . 62
4.4 Average F-measure values at the file level . . . . . 62
4.5 Average F-measure values at the method level . . . . . 63
4.6 Number of bug entries at the file (F), class (C) and method (M) levels . 63

4.7 F-measure values at the class level . . . . . . . . . . . . . . . . . . . . . 64
4.8 F-measure values at the file level . . . . . . . . . . . . . . . . . . . . . 64
4.9 F-measure values at the method level . . . . . . . . . . . . . . . . . . . . 65
4.10 F-measure values for the release versions with the highest number of bug entries . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 66
4.11 Statistical characteristics of the F-measure values . . . . . . . . . . . . 66

5.1 Bug-fixing commit inclusion criteria . . . . . . . . . . . . . . . . . . . . 74
5.2 Subjects included in BugsJS . . . . . . . . . . . . . . . . . . . . . . . . 75
5.3 Statistics about the projects included in BugsJS . . . . . . . . . . . . . 76
5.4 Manual and dynamic validation statistics per application for all considered commits . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 77
5.5 Bug-fixing change types found in BugsJS . . . . . . . . . . . . . . . . . 90
5.6 Bug-fixing patterns . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 92
5.7 Taxonomy and bug-fixing types . . . . . . . . . . . . . . . . . . . . . . . 98

A.1 Thesis contributions and supporting publications . . . . . . . . . . . . 110

B.1. A tézispontokhoz kapcsolódó publikációk . . . . . . . . . . . . . . . . . 115

# List of Figures

3.1 The components of the process . . . . . . . . . . . . . . . . . . . . . . . 24
3.2 The number of bug reports with the corresponding number of commits 27
3.3 The number of commits per projects . . . . . . . . . . . . . . . . . . . 27
3.4 The relationship between the bug reports and commits . . . . . . . . . 28
3.5 The relationship between the bugs and release versions . . . . . . . . . 29
3.6 Distribution of the number of classes in a Java file . . . . . . . . . . . . 35
3.7 Uncertainty included in the traditional approach . . . . . . . . . . . . . 49

4.1 The graph database schema . . . . . . . . . . . . . . . . . . . . . . . . 55
4.2 Example graph (green: project, purple: issue, red: user, blue: commit, pink: file, yellow: class, grey: method) . . . . . . . . . . . . . . . . . . 57
4.3 Example graph (purple: issue, blue: commit, grey: method) . . . . . . 59
4.4 Overview of processes involved . . . . . . . . . . . . . . . . . . . . . . . 60
4.5 Correlation of method-level metrics . . . . . . . . . . . . . . . . . . . . 68
4.6 Correlation of class-level metrics . . . . . . . . . . . . . . . . . . . . . . 68

5.1 Overview of the bug selection and inclusion process . . . . . . . . . . . 72
5.2 Overview of BugsJS architecture . . . . . . . . . . . . . . . . . . . . . 78
5.3 Taxonomy of bugs in the benchmark of JavaScript programs of BugsJS. 83
5.4 Bug-fixing patterns used in the Incomplete feature implementation category . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 94
5.5 Bug-fixing patterns used in the Incorrect feature implementation category 95
5.6 Bug-fixing patterns used in the Generic category . . . . . . . . . . . . . 97

# Listings

3.1 Unified diff format example . . . . . . . . . . . . . . . . . . . . . . . . 30

4.1 Cypher query for calculating the *Number of Modifications* metric . . . . 57
4.2 Cypher query for calculating method-level bug numbers . . . . . . . . . 59

*To my family...*

*"It's not a bug - it's an undocumented feature."*

— Anonymous

# 1 Introduction

Managing software bugs is an essential part of software development and companies tend to spend a large amount of resources on it. Programmers tend to make mistakes despite the assistance provided by different integrated development environments, and errors may also occur due to frequent changes in the code and inappropriate specifications; therefore, it is important to get more and/or better tools to help the automatic detection of errors [67]. Dealing with software bugs consists of tasks like preventing, finding, and fixing bugs.

Finding software bugs is usually done by checking the source code manually and looking for the root of the problem based on bug reports. It is a time- and resource-consuming activity, and minimizing the required effort would help to reduce the cost of the development. Unit tests can also help us detect and localize software faults, but it requires writing test cases in parallel with development, and it is also a resource-intensive task. Another way of assisting bug localization is to characterize the known ones with some appropriate metrics and try to predict which source code elements have the highest probability of containing a bug. The most important step in facilitating error detection is to analyze already known errors to identify patterns or trends.

Analyzing known bugs requires a source code change history and a bug tracking system. Nowadays, many developers use a versioning system - like Subversion or Git -, hence the source code history is often available. The use of bug tracking systems is also quite common in software development. There are numerous commercial and open-source software systems available for these purposes. The bug reports are recorded within these systems and all changes related to the bugs are also tracked, including the source code fixes. Furthermore, different web services are built to meet these needs. The most popular ones, like SourceForge, Bitbucket, and GitHub, fulfill the above-mentioned functionalities. They usually provide several services, such as source code hosting and user management. Their APIs make it possible to retrieve various kinds of data, e.g., they provide support for the examination of the behavior or the cooperation of users, or even for the analysis of the source code itself. Since most of these services include bug tracking, the idea of using this information in the characterization of buggy source code parts is raised [132]. To do so, the bug reports managed by these source

code hosting providers must be connected to the appropriate source code parts [127]. A common practice in version control systems is to describe the changes in a comment belonging to a commit (log message) and often to provide the identifier of the associated bug report that the commit is supposed to fix [71]. This can be used to identify the faulty versions of the source code [39, 40]. GitHub contains more than 330 million repositories[1] and has a readily usable API[2] to access these projects, which are accessible via Git[3]; hence it is a convenient choice as a data source for the studies.

In terms of programming languages, some of the most popular languages in use today include Java, Python, C++, JavaScript, and PHP[4]. According to the TIOBE Index[5], the most popular programming language in 2023 was Python, followed by C, C++, and Java. Java has been a popular programming language for many years and is widely used in enterprise software development due to its scalability, reliability, and portability[6]. JavaScript (JS) is the de-facto web programming language globally[7], and the most adopted language on GitHub[8]. JavaScript is massively used in the client side of web applications to achieve high responsiveness and user-friendliness. In recent years, due to its flexibility and effectiveness, it has also been increasingly adopted for server-side development, leading to full-stack web applications [13]. Platforms such as Node.js[9] allow developers to conveniently develop both the front- and back-end of the applications entirely in JavaScript. Despite its popularity, the intrinsic characteristics of JavaScript—such as weak typing, prototypal inheritance, and run-time evaluation—make it one of the most error-prone programming languages. As such, a large body of software engineering research has focused on the analysis and testing of JavaScript web applications [23, 121, 122, 94, 46, 11, 83, 12]. For these reasons, although there are many different programming languages, we made a conscious decision to focus our efforts on analyzing bugs reported in Java or JavaScript projects.

Although a vast amount of raw data is available regarding software bugs, collecting, filtering, and processing it can be a time-consuming and resource-intensive task. Many papers have dealt with bug databases using many kinds of approaches, such as bug prediction, fault localization, or testing techniques [31, 98, 95, 81, 133]. Researchers often use a database created for their own purposes, but these datasets are rarely published for the community. Despite the abundance of research on software bugs, the availability of bug databases that are accessible to the public is notably inadequate and overlooked. Additionally, subject programs or accompanying experimental data are rarely made available in a detailed, descriptive, curated, and coherent manner. This not only hampers the reproducibility of the studies themselves but also makes it difficult for researchers to assess the state-of-the-art of related research and compare existing solutions. Specifically, testing techniques are typically evaluated with respect to their effectiveness at detecting faults in existing programs, however, real bugs are hard to isolate, reproduce, and characterize. Therefore, the common practice relies on manually seeded faults or mutation testing [66]. Each of these solutions has limitations.

[1] `https://github.com/about`
[2] `https://docs.github.com/en/rest`
[3] `https://git-scm.com/`
[4] `https://octoverse.github.com/2022/top-programming-languages`
[5] `https://www.tiobe.com/tiobe-index/`
[6] `https://github.com/readme/featured/java-programming-language`
[7] `https://insights.stackoverflow.com/survey/2019`
[8] `https://octoverse.github.com`
[9] `https://nodejs.org/en/`

Manually injected faults can be biased toward researchers' expectations, undermining the representativeness of the studies that use them. Mutation techniques, on the other hand, allow for generating a large number of "artificial" faults. Although research has shown that mutants are quite representative of real bugs [55, 70, 14], mutation testing is computationally expensive to use in practice. For these reasons, publicly available benchmarks of bugs are of paramount importance for devising novel debugging, bug prediction, fault localization, or program repair approaches.

The characterization of buggy source code elements through various methods is still a popular research area. For automatic recognition of unknown faulty code elements, it is a prerequisite to characterize the already known ones. There are many good studies on bug characterization [56, 110, 30, 41]. Processing the diff files of a commit can help us obtain the exact code sections affected by the bug [128]. The most commonly used methods for bug characterization include textual similarities with faulty code parts [18], source code analysis, product metrics [37, 100], or process metrics. There are numerous tools, some of which are free, that are capable of analyzing source code written in different programming languages and producing product metrics for the code elements. During our studies, we used the OpenStaticAnalyzer[10] tool for source code analysis, because it is able to process source code written in either Java or JavaScript programming languages and it can extract detailed information about the source code elements.

The thesis is composed of three thesis points and is structured as follows. Chapter 2 discusses the related work and background information of our study. In the following three chapters, we delve into each of the three thesis points. In Chapter 3, we present the *first thesis point*, which is a novel method for constructing a bug database. Using this method we construct a dataset and evaluate its usefulness for bug prediction, while also comparing it with a database made using the traditional approach. Chapter 4 covers the *second thesis point*, where we present a method for computing software process metrics using a graph database. We assess the metrics' predictive power for bugs and compare them with product metrics. In Chapter 5, we present the *third thesis point*, which is a benchmark of real, manually-verified JavaScript bugs, and we discuss the results of our quantitative and qualitative analyses of these bugs. Finally, in Chapter 6, we provide a summary of the thesis. In Appendices A and B, we present concise summaries of the thesis in English and Hungarian, respectively, outlining the concrete thesis points, as well as the author's contributions and supporting publications.

[10] `https://github.com/sed-inf-u-szeged/OpenStaticAnalyzer`

*"I grew up thinking that a research scientist was a natural thing to be."*

— Stephen Hawking

# 2 Background

## 2.1 Bug Taxonomies

There are several industry standards for categorizing software bugs, such as the IEEE Standard Classification for Software Anomalies [9], or IBM's Orthogonal Defect Classification [34]. Also, there are countless categorization schemes proposed by various testing and defect management tool and service vendors, which are also less relevant to our research.

Hanam et al. [59] discuss 13 cross-project bug patterns occurring in JavaScript pertaining to six categories, which are the following: *Dereferenced non-values* (e.g., Uninitialized variables), *Incorrect API config* (e.g., Missing API call configuration values), *Incorrect comparison* (e.g., `===` and `==` used interchangeably), *Unhandled exceptions* (e.g., Missing `try-catch` block), *Missing arguments* (e.g., Function call with missing arguments), and *Incorrect `this` bounding* (e.g., Accessing a wrong `this` reference). The most common pattern according to the Hanam et al. scheme, *Dereferenced non-values*, can also be identified in other related work. Previous work showed that this pattern also occurs frequently in client-side JavaScript applications [94]. Developers could avoid these syntax-related bugs by adopting appropriate coding standards. Moreover, IDEs can be enhanced to alert programmers to possible effects or bad practices. They could also aid in prevention by prohibiting certain actions or by recommending the creation of stable constructs.

Catolino et al. [32] analyzed 1,280 bug reports of 119 popular projects with the aim of building a taxonomy of the types of reported bugs. They devised and evaluated the automated classification model, which is able to classify the reported bugs according to the defined taxonomy. The authors defined a three-step manual method to build the taxonomy. The final taxonomy defined in this work contains nine main common bug types over the considered systems: configuration, network, database-related, GUI-related, performance, permission/deprecation, security, program anomaly, and test code-related issues. This classification is less suitable to apply to source code because it is a very high-level one.

Li et al. [79] used natural language text classification techniques to automatically

analyze 29,000 bugs from the Bugzilla databases of the Mozilla and Apache HTTP Server projects. The authors classified the bugs along three dimensions: root cause (RC), impact (I), and software component (SC). According to RC, bugs can be classified into three disjoint groups (and subgroups): semantic, memory, and concurrency.

Tan et al. [116] proposed a work that is related to the previous study. They examined more than 2,000 randomly sampled real-world bugs in three large projects (Linux kernel, Mozilla, and Apache) and manually analyzed them according to the three dimensions defined by Li et al. [79]. They created a bug type classification model, which used machine learning techniques to automatically classify the bug types.

Zhang et al. [130] investigated the symptoms and root causes of TensorFlow bugs. They identified the bugs from the GitHub issue tracker using commit and pull request messages. The authors collected the common root causes (that were based on structure, model tensor, and API operation) and symptoms (based on error, effectiveness, and efficiency) into categories and classified each bug accordingly.

Thung et al. [117] presented a semi-supervised defect prediction approach (LeDEx - Learning with Diverse and Extreme Examples) to minimize manual bug labeling. The researchers used a benchmark that contains 500 defects from three projects that have been manually labeled based on ODC. In their approach, hand-labeled samples were used to learn and build the model, which uses non-labeled elements to refine the model.

In another study, Thung et al. [118] proposed a classification-based approach used to categorize the bugs into control and data flow, structural or non-functional groups. They performed NLP pre-processing and feature extraction operations on the text mined from JIRA. The resulting data was used to build the model based on Support Vector Machine (SVM).

Nagwani et al. [92] used the bug tracking system to collect textual information and several attributes of bugs. They presented a methodology to bug classification, which is based on a generative statistical model (LDA - Latent Dirichlet allocation) in natural language processing.

## 2.2 Bug Localization and Source Code Management

Numerous approaches have been proposed to tackle bug characterization and localization [107, 123, 42]. For instance, Zhou et al. introduced *BugLocator* [132], a tool that identifies the relevant source code files that need to be modified to fix a bug by utilizing textual similarities between the initial bug report and the source code to rank potentially problematic files. *BugLocator* employs a bug database that stores information about previous bug reports. The ranking is based on the assumption that descriptions with high similarities indicate highly similar related files. Similarly, *Rebug-Detector* by Wang et al. [120] is a tool that detects related bugs from source code using bug information, focusing on overridden and overloaded method similarities.

The goal of developing *ReLink* was to address the issue of missing links between code changes in version control systems and fixed bugs, and its potential usefulness for software engineering research that relies on linkage data, such as software defect prediction [127]. *ReLink* works by analyzing bug tracking database information, such as bug reporter, description, comments, and date, and then pairing the bug with the appropriate source code files based on source code information extracted from the version control system. While most studies focus on linking information retrieved solely from version control and bug tracking systems [49, 87, 72, 126], this approach relies

on file-level textual features to extract additional information about the relationship between bugs and source code.

Kalliamvakou et al. [71] conducted a study on the characteristics and qualities of GitHub repositories. Their analysis covered various project features, such as (in)activity, and raised additional research questions, such as whether a project is an independent entity or part of a larger system. The study found that the extracted data could be a valuable input for various investigations, however, one must always verify the usefulness and reliability of the data. While it is recommended to select projects with many developers and commits, the most important factor is to choose projects that suit one's specific purpose.

Mining software repositories can be a harsh task when an automatic mechanism is used to construct a large set of data based on the information gathered from a distributed software repository. As we used GitHub to address our research questions, we paid extra attention to prevent and avoid pitfalls. Bird et al. [24] presented a study on distributed version control systems – focusing mainly on Git – that examined their usage and the available set of data (such as whether the commits are removable, modifiable, or movable). The main purpose of the paper was to draw attention to pitfalls and help researchers avoid them during the processing and analysis of a mined information set.

The value of bug tracking systems in improving software quality has been well-documented in numerous research papers. Bangcharoensap et al. [18] propose a method that leverages bug reports stored in such systems to quickly locate buggy files in a software system. Their approach offers three distinct methods for ranking fault-prone files: (a) Text mining, which evaluates files based on their textual similarity to the bug report, (b) Code mining, which predicts potential buggy modules using source code product metrics, and (c) Change history, which predicts fault-prone modules using process metrics. The authors test their approach using data from the Eclipse platform and demonstrate its effectiveness in identifying buggy files. They find that bug reports with brief descriptions and specific language can be particularly helpful in identifying weaknesses in the system. Overall, the results suggest that bug tracking systems provide a rich source of data for identifying and addressing software issues. By combining multiple methods for ranking fault-prone files, the proposed approach offers a comprehensive approach to bug localization that can be easily applied to a range of software systems.

Apart from the methods mentioned earlier, changes in source code metrics can also suggest the presence of potential bugs in relevant source code files [57]. Couto et al. [37] published a study demonstrating a possible link between changed source code metrics and bugs. The study aimed to find more reliable evidence regarding the causality between software metrics and bug occurrence.

## 2.3 Public Datasets

Bug prediction is an intensively studied research area [84, 85, 82]. The techniques and approaches used for bug prediction can be presented and compared in different ways, however, there are some basic points that can serve as common components [78]. One common element can be a dataset used for the evaluation of the various approaches. The biggest of these datasets is the *tera-PROMISE* [112, 86] repository. It is a repository of datasets out of which several contain bugs gathered from open-source and

also from closed-source industrial software systems[1]. Amongst others, it includes the *NASA MDP* dataset, which was used in many research studies and also criticized for containing erroneous data [109, 97]. The *tera-PROMISE* repository also contains an extensively referenced dataset created by Jureczko[68], which provides object-oriented metrics as well as bug information for the source code elements (classes). This latter one includes open-source projects, such as Apache Ant, Apache Camel, JEdit, and Apache Lucene, forming a dataset containing 48 releases of 15 projects. The main purpose of these datasets is to support prediction methods and summarize bugs and their characterizations extracted from various projects. Many research papers used datasets from the *tera-PROMISE* repository as input for their investigations, but unfortunately, the latest version of the repository is not available anymore.

A similar dataset for bug prediction by D'Ambros et al. [41] came to be commonly known as the *Bug prediction dataset*[2]. The reason for creating this dataset was mainly inspired by the idea of measuring the performance of the different prediction models and also comparing them to each other. As part of their research, they created a benchmark database from several open-source projects (Eclipse, Mylyn, Lucene). This database contains bug numbers at the class level with 15 change metrics and 17 product metrics. According to their findings, change metrics could improve the performance of the fault prediction methods. The bug information was extracted from the commit messages and bug tracking systems by using pattern matching, as others did in earlier studies [134, 49]. This dataset handles the bugs and the relevant source code parts at the class level, i.e., the bugs are assigned to classes. They describe the whole process of building such a database, but the links to the tools used do not work anymore.

Zimmermann et al. [134] used Eclipse as the input for a study dealing with defect prediction. They investigated whether the complexity metrics have the power to detect fault-prone points in the system at package and file level. During the study, they constructed a public dataset, called *Eclipse Bug Dataset*[3]. It contains different source code metrics and a subset of the files/packages is marked as "buggy" if it contained any bugs in the interval between two releases. The dataset is still available, but the website is marked as archived, and it has not been maintained since 2010.

*iBUGS* [40, 39] provides a complex environment for testing different automatic defect localization methods. Information describing the bugs comes from both version control systems and bug tracking systems. *iBUGS* used the following three open-source projects to extract the bugs from (the numbers of extracted bugs are in parentheses): AspectJ – an extension for the Java programming language to support aspect-oriented programming (223); Rhino – a JavaSript interpreter written in Java (32); and Joda-Time – a quality replacement (extension) for the Java date and time classes (8). The authors attempted to generate the *iBUGS* dataset in an automatic way and they compared the generated set to the manually validated set of bugs. *iBUGS* is a framework that is rather aimed towards bug localization instead of being a standalone dataset containing source code elements and their characterizations (i.e., metrics).

The *Bugcatchers* [58] dataset, created by Hall et al., is not only a bug dataset but it also contains bad smells detected in the subject systems. They showed that code smells also play a significant role in bug prediction. The selected three systems are Eclipse JDT Core, ArgoUML, and Apache Commons. The dataset is built and evaluated at

[1] `http://promise.site.uottawa.ca/SERepository/index.html`
[2] `http://bug.inf.usi.ch/`
[3] `https://www.st.cs.uni-saarland.de/softevo/bug-data/eclipse/`

the file level.

The *ELFF* dataset [111] is a recent dataset proposed by Shippey et al. They found that only a few method-level datasets exist, thus they created a dataset whose entries are methods. Additionally, they also made class-level datasets publicly available[4]. They used Boa [45] to mine SourceForge repositories and collect as many candidates as they could, selecting 23 projects out of 50,000 that fulfilled their criteria (number of closed bugs, bugs are referenced from commits, etc.). They only kept projects with SVN version control systems, which narrows down their candidate set. They used the classic and well-defined SZZ algorithm [114] to find the linkage between bugs and the corresponding source code elements.

The *Had-oops!* dataset was created using a novel approach introduced by Harman et al [60]. In their study, they examined eight consecutive versions of Hadoop and investigated how the chronology of versions impacts the performance of fault prediction. They constructed prediction models for a given version using data from all previous versions, as well as models using only the current version, and compared their performance. Interestingly, the results were not straightforward, as they found that earlier versions often outperformed recent versions in many cases. Additionally, they discovered that using all versions was not always superior to using only the current version when building prediction models.

The *Mutation-aware fault prediction dataset* was created as a result of an experiment conducted by Bowes et al., where they explored the use of mutation metrics as independent variables for fault prediction [27]. The experiment utilized three software systems, including two open-source projects (Eclipse and Apache) and one closed-source project. The widely-used PITest [36] tool was employed to obtain a set of mutation metrics at the class level that were incorporated into the final dataset. Additionally, static source code metrics calculated by JHawk [10] were included in the dataset for comparison purposes.

*Software-artifact Infrastructure Repository* [44] is a repository of software-related artifacts meant to support rigorous, controlled experimentation with program analysis and software testing techniques, and education in controlled experimentation. The repository holds a collection of software systems written in Java, PHP, C#, C, C++, and other programming languages, spanning across multiple versions. It also houses various supporting artifacts like test suites, fault data, and scripts. Additionally, the repository includes documentation on how to utilize these objects for experimentation, as well as supporting tools that aid in the experimentation process. It also contains information on the methods employed for artifact selection, organization, and enhancement, along with tools that facilitate these processes.

*Defects4J* [69] (Defects for Java) is a publicly available curated dataset of real-world software defects specifically designed for software testing and debugging research. It is a collection of reproducible bugs or defects extracted from open-source Java projects, with accompanying test cases that trigger the defects. It was created to facilitate the development and evaluation of automated software testing and debugging techniques by providing a standardized and diverse set of real-world software defects for researchers and practitioners to use in their experiments. *Defects4J* is maintained as an open-source project[5] and contains 835 bugs from 17 projects.

[4] `https://github.com/tjshippey/ESEM2016`

[5] `https://github.com/rjust/defects4j`

*BugZoo*[6] is a decentralized platform for distributing, reproducing, and interacting with historical software bugs [119]. It is designed to support both software engineering researchers as well as developers of tools for program testing, analysis, and repair.

*ManyBugs*[7] and *IntroClass*[8] are sets of C programs that contain defects and are associated with test suites. They are intended to support reproducible and comparative studies of research techniques in automatic patch generation for defect repair [77]. *ManyBugs* and *IntroClass* are seamlessly integrated with *BugZoo*.

*BugSwarm*[9] is a toolset that enables the creation of a scalable, diverse, real-world, continuously growing set of reproducible build failures and fixes from open-source projects [43]. It mines GitHub projects that utilize Travis-CI as a continuous integration service. It collects pairs of build results where the first build in the pair fails, and the second build, which is the subsequent one in Git history on each branch, passes. It has over 3000 entries from projects written in Java or Python.

Furthermore, purpose-specific test and bug datasets also exist to support studies in program repair [80], test generation [50], and security [54].

*Awesome-MSR*[10] is a curated repository named after the conference series known as Mining Software Repositories (MSR). It serves as a comprehensive collection of repositories, tools, and datasets to conduct evidence-based, data-driven research on software systems.

## 2.4 Bug Prediction

In the aforementioned studies, various statistical models and machine learning approaches were utilized to help predict the occurrence of bugs by identifying patterns and relationships in the data. Logistic regression is a commonly used statistical technique that is used to model the relationship between a binary outcome variable (e.g., the occurrence of a bug) and one or more predictor variables (e.g., software metrics, historical bug data, etc.). Machine learning approaches, including supervised, unsupervised, and semi-supervised learning algorithms, are also commonly used in bug prediction. These techniques often involve training a model on labeled data (e.g., bug reports with labeled bug/non-bug instances) and then using the trained model to predict the likelihood of bugs in new, unlabeled data.

For evaluating our results during the study, we applied the following algorithms:

- NaiveBayes [129]
- NaiveBayesMultinomial
- Logistic [38]
- SGD [106]
- SimpleLogistic [76]
- VotedPerceptron [51]
- DecisionTable [73]
- OneR [65]
- J48 (C4.5) [99]

[6] `https://github.com/squaresLab/BugZoo`
[7] `https://github.com/squaresLab/ManyBugs`
[8] `https://github.com/ProgramRepair/IntroClass`
[9] `https://github.com/BugSwarm/bugswarm`
[10] `https://github.com/dspinellis/awesome-msr`

- RandomForest [64]
- RandomTree

For the training, we used 10-fold cross-validation and compared the results based on precision, recall, and F-measure metrics where these metrics are defined in the following way:

$$precision = \frac{TP}{TP + FP} \qquad recall = \frac{TP}{TP + FN}$$

$$F - measure = 2 \cdot \frac{precision \cdot recall}{precision + recall},$$

where *TP* (True Positive) is the number of entries that were predicted as faulty and observed as faulty, *FP* (False Positive) is the number of entries that were predicted as faulty but observed as not faulty, *FN* (False Negative) is the number of entries that were predicted as non-faulty but observed as faulty. We carried out the training with the popular machine learning library called Weka[11]. It contains algorithms from different categories, for instance, Bayesian methods, support vector machines, and decision trees.

## 2.5 Source Code Analysis

There are two main approaches to analyzing source code in software development: static analysis and dynamic analysis.

Static analysis tools analyze the source code without actually running the code, thus providing feedback before the code is executed, which allows developers to identify and fix issues early in the development process. They scan the code for potential issues, such as coding standard violations, security vulnerabilities, and performance optimizations, by examining the code's syntax, structure, and patterns. They use predefined or customizable rules or patterns to identify potential issues in the code. These rules are typically based on coding standards, best practices, and security guidelines. Static analysis tools are often integrated into popular development environments and build systems, such as IDEs and CI/CD pipelines, to provide real-time feedback during development.

Dynamic analysis tools analyze the behavior of the code during runtime, by actually executing the code and observing its behavior. This allows for insights into the actual execution of the code and its runtime behavior, thus providing feedback during code execution, which allows developers to observe the code's behavior in different scenarios and identify issues that may not be evident through static analysis. Dynamic analysis tools can also assess test coverage, profiling, and performance during runtime.

Symbolic analysis is a type of dynamic analysis that involves using symbolic execution techniques to analyze the behavior of software during runtime. Symbolic execution is a technique where variables and inputs are represented symbolically, allowing for the exploration of different paths and conditions in the code to understand how the software behaves in different scenarios. It can be used for various purposes, such as identifying

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

bugs, vulnerabilities, or unexpected behavior in software. However, it can also be computationally expensive and may require specialized tools and expertise to effectively implement and interpret the results.

Both static source code analysis and dynamic analysis have their advantages and limitations. There are many tools available for static code analysis[12] as well as symbolic analysis[13]. For our studies, we used the OpenStaticAnalyzer tool, which is a static analysis tool. We chose it because it is able to process source code written in either Java or JavaScript programming languages and it can extract detailed information about the source code elements, including various source code metrics. The output of the static source code analysis using OpenStaticAnalyzer is presented in the form of a graph, which contains the source code elements (files, classes, methods) as nodes and the corresponding relationships between these elements as edges. Additionally, the tool includes a Java library that offers an API for working with the graph file generated from the static source code analysis. These features make this tool an ideal choice for performing the task of static source code analysis.

## 2.6 Software Metrics

A software metric is a quantified measure of a property of a software project. By using a set of different metrics, we can measure the properties of a project objectively from various points of view. Metrics can be obtained from the source code, from the project management system, or even from the execution traces of the source code. We can deduce higher-level software characteristics from lower-level ones [15], such as the maintainability of the source code or the distribution of defects, but they can also be used to build a cost estimation model, apply performance optimization, or to improve activities supporting software quality [25, 17, 16]. Software metrics can be classified into different categories based on their characteristics and purposes. A common classification is based on whether they measure the characteristics of the resulting software product (product metrics) or the efficiency and effectiveness of the software development process (process metrics).

Software product metrics are extracted from the structure of the source code. Some examples are lines of code, cyclomatic complexity, and the number of methods. These metrics are frequently used for bug characterization[100] because they are easy to compute. Product metrics do not rely on the software's project history as they are based on a single state of the software and hence do not incorporate temporal characteristics. Typically, these metrics are computed for files or classes, although there are now more tools that also support methods.

Software process metrics are derived from developer activities, and many of them incorporate temporal information. Common examples include metrics based on the number of previous modifications, the number of different contributors, the number of modified lines, and the timing of modifications. These metrics have diverse applications and can be used for various purposes. They are particularly useful for examining developers' behavior, as the computation is based on their activities. Additionally, these metrics can aid in identifying key source code parts that are frequently or recently modified, providing valuable insights into the software development process.

[12] https://github.com/analysis-tools-dev/static-analysis
[13] https://github.com/ksluckow/awesome-symbolic-execution

### 2.6.1 Software Product Metrics

The area of object-oriented source code metrics has been researched for many years [33, 19, 28], thus, no wonder that several tools exist for measuring them. These tools are suitable for the detailed examination of systems written in various programming languages. The source code metrics provide information about the size, inheritance, coupling, cohesion, or complexity of the code. As mentioned before, during our studies, we used the OpenStaticAnalyzer tool to analyze the source code of the selected systems. It is capable of computing various software product metrics. The full list of the object-oriented metrics we used is shown in Table 2.2. The last three columns of the table indicate the kind of elements the given metric is calculated for, namely method, class, and file. The presence of 'X' indicates that the metric is calculated for the given source code level. Most of the blanks in the table come from the fact that the metric is defined only for a given level. For instance, *Weighted Methods per Class* cannot be interpreted for methods and files. Other blanks come from the limitations of OpenStaticAnalyzer.

One special metric category is provided by source code duplication detection [105]. OpenStaticAnalyzer is able to detect Type-1 (exact copy of code, not considering white spaces and comments) and Type-2 clones (syntactically identical copy of code where variable, function, or type identifiers can be different; also not considering white spaces and comments) in software systems [20] and also supports clone management tasks, such as:

- Clone tracking: clones are tracked during the source code analysis of consecutive revisions of the analyzed software system.
- Calculating clone metrics: a wide set of clone-related metrics is calculated to describe the properties of a clone in the system (for example, the risk of a clone or the effort needed to eliminate the clone from the system).

Basic clone-related metrics that are calculated for methods and classes are presented in Table 2.1.

Table 2.1: Clone metrics used for characterization

| **Abbreviation** | **Full name** |
|---|---|
| CC | Clone Coverage |
| CCL | Clone Classes |
| CCO | Clone Complexity |
| CI | Clone Instances |
| CLC | Clone Line Coverage |
| CLLC | Clone Lines of Code |
| LDC | Lines of Duplicated Code |
| LLDC | Logical Lines of Duplicated Code |

OpenStaticAnalyzer also provides a coding rule violation detection module. The occurrence of rule violations in a source code element can potentially result in errors later [26], which could be akin to a ticking time bomb. Therefore, the count of distinct rule violations identified in the source code element can serve as a valuable predictor; thus, this information is also encapsulated in the dataset.

Table 2.2: Source code metrics used for characterization

| Abbreviation | Full name | Method | Class | File |
|---|---|---|---|---|
| CLOC | Comment Lines of Code | X | X | X |
| LOC | Lines of Code | X | X | X |
| LLOC | Logical Lines of Code | X | X | X |
| NL | Nesting Level | X | X | |
| NLE | Nesting Level Else-If | X | X | |
| NII | Number of Incoming Invocations | X | X | |
| NOI | Number of Outgoing Invocations | X | X | |
| CD | Comment Density | X | X | |
| DLOC | Documentation Lines of Code | X | X | |
| TCD | Total Comment Density | X | X | |
| TCLOC | Total Comment Lines of Code | X | X | |
| NOS | Number of Statements | X | X | |
| TLOC | Total Lines of Code | X | X | |
| TLLOC | Total Logical Lines of Code | X | X | |
| TNOS | Total Number of Statements | X | X | |
| McCC | McCabe's Cyclomatic Complexity | X | | X |
| PDA | Public Documented API | | X | X |
| PUA | Public Undocumented API | | X | X |
| HCPL | Halstead Calculated Program Length | X | | |
| HDIF | Halstead Difficulty | X | | |
| HEFF | Halstead Effort | X | | |
| HNDB | Halstead Number of Delivered Bugs | X | | |
| HPL | Halstead Program Length | X | | |
| HPV | Halstead Program Vocabulary | X | | |
| HTRP | Halstead Time Required to Program | X | | |
| HVOL | Halstead Volume | X | | |
| MIMS | Maintainability Index (Microsoft version) | X | | |
| MI | Maintainability Index (Original version) | X | | |
| MISEI | Maintainability Index (SEI version) | X | | |
| MISM | Maintainability Index (SourceMeter version) | X | | |
| NUMPAR | Number of Parameters | X | | |
| LCOM5 | Lack of Cohesion in Methods 5 | | X | |
| WMC | Weighted Methods per Class | | X | |
| CBO | Coupling Between Object classes | | X | |
| CBOI | Coupling Between Object classes Inverse | | X | |
| RFC | Response set For Class | | X | |
| AD | API Documentation | | X | |
| DIT | Depth of Inheritance Tree | | X | |
| NOA | Number of Ancestors | | X | |
| NOC | Number of Children | | X | |
| NOD | Number of Descendants | | X | |
| NOP | Number of Parents | | X | |
| NA | Number of Attributes | | X | |
| NG | Number of Getters | | X | |
| NLA | Number of Local Attributes | | X | |
| NLG | Number of Local Getters | | X | |
| NLM | Number of Local Methods | | X | |
| NLPA | Number of Local Public Attributes | | X | |
| NLPM | Number of Local Public Methods | | X | |
| NLS | Number of Local Setters | | X | |
| NM | Number of Methods | | X | |
| NPA | Number of Public Attributes | | X | |
| NPM | Number of Public Methods | | X | |
| NS | Number of Setters | | X | |
| TNA | Total Number of Attributes | | X | |
| TNG | Total Number of Getters | | X | |
| TNLA | Total Number of Local Attributes | | X | |
| TNLG | Total Number of Local Getters | | X | |
| TNLM | Total Number of Local Methods | | X | |
| TNLPA | Total Number of Local Public Attributes | | X | |
| TNLPM | Total Number of Local Public Methods | | X | |
| TNLS | Total Number of Local Setters | | X | |
| TNM | Total Number of Methods | | X | |
| TNPA | Total Number of Public Attributes | | X | |
| TNPM | Total Number of Public Methods | | X | |
| TNS | Total Number of Setters | | X | |

### 2.6.2 Software Process Metrics

There are many software process-related metrics [90, 91, 61, 21] and they were mainly used in studies concerning bug prediction. In these studies, the authors evaluated the predictive capability of these metrics and they often compared this capability with that of product metrics. Other studies [101, 89, 88, 75, 82] have shown that while software process metrics are generally better bug predictors than product metrics, tools that can compute these metrics are still quite rare. The studies primarily focus on defining process metrics and presenting their results. However, the method of computing these metrics is not always thoroughly described, which can make reproducing the results challenging.

Rahman et al. [101] analyzed the properties of process metrics from the perspective of performance, stability, portability, and stasis. They found that product metrics have a higher stasis - which means they do not change much compared to the process metrics -, thus the same elements were predicted as defective over and over. Also, product metrics are less stable and less portable across projects.

Hassan [61] went further. In his paper, he proposed complexity metrics that are based on process metrics. He analyzed 6 projects written in C and C++ and computed process metrics at the file level, but he did not give a detailed description of the method of processing and how to compute these metrics. He concluded that the proposed change complexity metrics are better fault predictors than the well-known process metrics. He also said that we should consider using these metrics instead of simple metrics like the number of prior modifications and the number of prior faults.

*Buginfo* is a tool that is used for collecting bug information from source code repositories [68]. It uses regular expressions on the commit messages to count the number of bugs in classes, as other studies did [134, 49]. It is also capable of computing process metrics, but unfortunately, the tool is not maintained.

D'Ambros et al. [41] computed change metrics and bug information on the file level. They found that the *Weighted Churn* and *Linearly Decayed Entropy* metrics perform the best (around 90%) for bug prediction, but the computation of these metrics is quite complex. Furthermore, they concluded that multiple metrics should be used for this purpose in order to achieve good results across multiple systems.

The Eclipse project is used quite often for studies on bug prediction. Bernstein et al. [21] used this project to examine whether temporal features are suitable for bug prediction. They gathered change information from CVS and bugs from Bugzilla and they computed several temporal features. They built non-linear models for the bug database they created and they achieved a high accuracy score (99%) in predicting defects. They concluded that temporal features (process metrics) and non-linear models are suitable for bug prediction. Moser et al. [88] also used the Eclipse project to investigate the characteristics of change metrics in bug prediction. They calculated 18 change metrics at the file level. They achieved better results with these metrics than with product metrics [89] and they showed that 3 out of 18 change metrics can achieve good results, and they are as stable as the model with all the metrics. These three metrics are the following: number of revisions, number of bug fixes, and maximum size of all of its change sets.

Shihab et al. [110] also examined whether the number of predictors can be reduced. As a data source, they used the Eclipse data set [134]. They showed that the 34 product and process metrics can effectively be reduced to 4 with very little difference in the overall prediction accuracy. They found that the most stable independent metrics were:

total prior changes, number of pre-release defects, and TLOC.

The study made by Krishnan et al. [75] sought to answer the research question of whether the process metrics are good bug predictors for the family of products in the evolving Eclipse product line. They replicated the results previously achieved by Moser et al. [89] and extended them with their observations. They concluded that process metrics are good bug predictors for the Eclipse product line. Furthermore, they found that a small subset of these metrics are stable and consistent across multiple projects. They are called maximum changeset, number of revisions, and number of authors.

Graves et al. [56] also made a study on the bug prediction capabilities of process metrics. They computed the metrics at the module level and analyzed systems written in C. Their observation was that the best model used the weighted time damp metric and the best linear models used the number of changes and the age metrics. They found that the number of developers and the changeset metrics did not influence the accuracy of the fault prediction.

In the literature, process metrics are usually defined for files or modules (collection of files). Here, an overview of the most commonly used metrics gathered from the aforementioned studies is provided, along with a general definition for each that can be applied to various source code elements, such as classes or methods:

- **Number of Modifications**: The number of previous modifications of the source element.
- **Number of Bug Fixes**: The number of previous modifications of the source element that reflect an intention to fix a bug.
- **Number of Versions**: The number of software versions (revisions) since the source element was created. In other words, the number of commits on the whole project since the creation of the element.
- **Number of Refactorings**: The number of previous modifications of the source element that were committed in order to perform refactoring.
- **Age**: The age of the source element in days, weeks, or months.
- **Weighted Age**: The weighted age [89] is calculated using the age and size of the previous modifications. It may be expressed in days, weeks, or months. The formal definition is the following:

$$WeightedAge(e) = \frac{\sum_v Age(v) \times NumberOfAddedLines(e,v)}{\sum_v NumberOfAddedLines(e,v)} \tag{2.1}$$

  In this formula, we compute the metric for the source element $e$. *Age* is the age of the software version $v$ (days, weeks, or months), where $v$ is earlier than the version for which we want to calculate. *NumberOfAddedLines* represents the number of lines added for source element $e$ in version $v$.
- **Number of Contributors**: How many different developers contributed to the source element.
- **Number of Contributor Changes**: The number of developer changes in the code history. A developer change occurs when the next sequential modification on the same source element was performed by a different developer.

- **Sum of Added Lines**: The total sum of the lines of code added to the source element.
- **Maximum Number of Added Lines**: The maximum number of lines of code added with one commit to the source element.
- **Average Number of Added Lines**: The average number of lines of code added to the source element.
- **Number of Additions**: The number of previous commits in which new lines were added to the source element.
- **Sum of Deleted Lines**: The total sum of the lines of code deleted from the source element.
- **Maximum Number of Deleted Lines**: The maximum number of lines of code deleted with one commit from the source element.
- **Average Number of Deleted Lines**: The average number of lines of code deleted from the source element.
- **Number of Deletions**: The number of previous commits containing lines that were deleted from the source element.
- **Code Churn**: The sum of lines added minus the lines deleted from the source element[90].
- **Relative Code Churn**: The normalized Code Churn metric. Normalization can be achieved with, for example, lines of code, file count, or time period [91].
- **Maximum Number of Elements Modified Together**: The maximum number of distinct elements that were modified with one commit.
- **Average Number of Elements Modified Together**: The average number of distinct elements that were modified together with the source element.
- **Average Time Between Changes**: The average number of days, weeks, or months that passed between consecutive modifications of the source element.
- **Author**: The identity of the original author of the source element. It may include other information about the developer, such as the total number of commits of the author and the number of projects.
- **Number of Referenced Issues**: The number of distinct issues referenced in the comments of commits that introduce modifications to the source element.
- **Number of Commits Without Message**: The number of previous modifications without any comment message.

Many of these metrics rely on versioning information and issue-tracking data, while others may also consider software management data. However, since software management data sources were not processed in this study, metrics that depend on them are not described here. Additionally, more specific characteristics, such as the *Number*

*Of Referenced High Priority Issues* metric, which considers the count of high-priority issues referenced in the commit message, may also be taken into account if issue priority information is available. However, due to the potential variations of these details across different systems, these variations were omitted from the list, and the focus was on the most commonly used metrics.

Other metrics also can be formed like *Change Activity Rate* [102], which is defined as the overall number of modifications relative to the age of the source element in months. This metric can be computed by a simple division. The calculation of these combined metrics is relatively straightforward, and thus we won't go into further detail about them.

Furthermore, most of these metrics can be specified with a time period. For instance, the calculation interval can be limited to the last six months, allowing for the production of new metrics, such as *Number Of Modification In The Last Six Months.*

## 2.7 GitHub

GitHub is one of today's most popular source code hosting services. It is used by several major open-source teams for managing their projects like Node.js, Ruby on Rails, Spring Framework, Zend Framework, and Jenkins, among others. GitHub offers public and private Git repositories for its users, with some collaborative services, e.g., built-in bug and issue tracking systems.

Bug reporting is supported by the fact that any GitHub user can add an issue, and collaborators can even label these issues for further categorization. The system provides some basic labels, such as "bug", "duplicate", and "enhancement", but these tags can be customized if required. In an optimal case, the collaborators review these reports and label them with the proper labels, for instance, the bug reports with the "bug" label. The most important feature of bug tracking is that we can refer to an issue from the log message of a commit by using the unique identifier of the issue, thereby identifying a connection between the source code and the reported bug. GitHub has an API[14] that can be used for managing repositories from other systems, or query information about them. This information includes events, feeds, notifications, gists, issues, commits, statistics, and user data.

With the GitHut[15] project that also uses this API, we can get up-to-date statistics about the public repositories. For instance, Table 2.3 presents the number of pushes created in 2022, grouped by the main programming languages they use (only the top 10 languages are shown). It shows that Java and JavaScript are still among the most often used programming languages in the open-source world.

Although extracting basic information from GitHub is easy, some version control features are hard to deal with, especially during the linking process when we try to match source code elements to bugs. For example, Git provides a powerful branching mechanism by supporting the creation, deletion, and selection of branches. A fixing commit of a bug most often occurs on other – so-called "topic" – branches and not on the master branch. During the merge, isomorphic commits are generated and placed on the master branch, thus all the desired analysis can be done by taking only the master branch with a given version as input. Another example is forking a repository, which

[14] `https://docs.github.com/en/rest`
[15] `https://githut.info/`

Table 2.3: The number of pushes created on GitHub in 2022 for the top 10 languages

| **Language** | **Number of Pushes** |
|---|---:|
| Python | 1,069,484 |
| Java | 572,890 |
| JavaScript | 534,972 |
| C++ | 478,294 |
| PHP | 361,926 |
| TypeScript | 367,136 |
| C | 245,035 |
| Go | 281,807 |
| Shell | 191,369 |
| Ruby | 200,872 |

is used worldwide. In our experiments, we do not handle forks, since it would have encumbered the above-mentioned linking process and we would not gain significant additional information since bugs are often corrected in the original repository. These details can be viewed as our underlying assumptions regarding the usage of GitHub.

*"A ship in port is safe, but that is not what ships are for. Sail out to sea and do new things."*
— Grace Hopper

# 3 A Novel Bug Prediction Dataset and its Validation

Previously published datasets follow a so-called traditional concept to create a dataset that serves as a benchmark for testing bug prediction techniques [41, 68]. These datasets include all code elements – both buggy and non-buggy – from one or more versions of the analyzed system. In this chapter, we present a new approach that collects before-fix and after-fix snapshots of source code elements (along with their characteristics) that were affected by bugs and does not consider those code elements that were not impacted by bugs. This kind of dataset helps us capture the changes in software product metrics when a bug is being fixed, thus, we can learn from the differences in source code metrics between faulty and non-faulty code elements. As far as we know, there exists no other bug dataset yet that tries to capture this before-fix and after-fix state.

This new dataset is called *BugHunter Dataset* and it is freely available (see Section 3.1.8). It can serve as a new kind of benchmark for testing different bug prediction methods since it includes a wide range of source code metrics to describe the previously detected bugs in the chosen systems. To construct this dataset, we have taken all reported bugs from the bug tracking system into consideration. We used the usual methodology of connecting commits to bugs by analyzing the log messages and by looking for clues that would unambiguously identify the bug that was the intended target of the corresponding fixing commit(s). Commit diffs helped us detect which source code elements were modified by a given change set, thus the code elements that had to be modified in order to fix the bug. We have analyzed 15 projects with more than 3.5 million lines of code, and more than 114 thousand commits in total. The subject systems that we selected differ in many ways from each other (size, domain, number of bugs reported) to cover a wide and general set of systems.

We have also performed experiments to check whether our novel dataset is suitable for bug prediction purposes. We collected bug characterization metrics at three source code levels: file, class, and method. After the dataset was constructed, we used different machine learning algorithms to analyze its usefulness.

Table 3.1: Comparison of the two types of datasets

| **Feature** | **Traditional** | **Novel** |
|---|---|---|
| *Included time interval* | Usually 6 months | Entire project history |
| *Included source code elements* | All the elements from a single version | Only the modified elements right before and after bug-fixes |
| *Assumptions* | Source code elements that are not included in any bug-fix are non-faulty | No assumptions needed |
| *Uncertainty* | The source code elements are faulty in the latest release version before the bug fix and non-faulty after the fix | The source code elements are faulty right before the bug fix and fixed afterwards |

We also performed a novel kind of experiment in which we assessed whether the method level metrics are better predictors when projected to class level than the class level metrics themselves.

An important aspect to investigate is how the bug prediction models built from the novel dataset compare to the ones which used the traditional datasets as corpus. However, this comparison is hard in its nature due to the variability in multiple factors. One major problem is the difference in the corpus itself. The list of the included projects varies from dataset to dataset. To overcome this issue, we utilized our traditional dataset, which was generated from the same 15 projects and referred to as the *GitHub Bug Dataset* [7]. This traditional dataset was constructed using release versions of the systems, with snapshots taken at approximately six-month intervals. This way, we could assess whether there was any difference in the bug prediction capabilities of these two types of datasets.

To emphasize the research artifact contributions and the research questions, we list them here:

- **Research Artifact:** A freely available novel dataset, called *BugHunter Dataset*, containing source code metrics of buggy Java source code elements (files, classes, methods) before and after bug fixes were applied to them.
- **Research Question 1:** Is the *BugHunter Dataset* usable for bug prediction purposes?
- **Research Question 2:** Are the method-level metrics projected to the class level better predictors than the class-level metrics themselves?
- **Research Question 3:** Is the *BugHunter Dataset* more powerful and expressive than the *GitHub Bug Dataset*?

In Table 3.2, we compare the main characteristics of our two datasets with those that are publicly available.

Our goal was to pick the strong aspects of all the previous datasets and put them together. Although the works discussed in Chapter 2 successfully made use of their datasets, an extended dataset can serve as a good basis for further investigations. Our

Table 3.2: Comparison of the datasets

| Project | Level of bugs | Bug characteristics | Projects |
|---|---|---|---|
| *NASA MDP Dataset* | class | static source code metrics | 11 |
| *Jureczko Dataset* | class | static source code metrics | 15 |
| *Bug prediction dataset* | class | static source code metrics, process metrics | 5 |
| *Eclipse dataset* | file, package | complexity metrics | 1 |
| *iBUGS* | N/A | bug-fix size properties, AST fingerprints | 3 |
| *Bugcatchers* | file | code smells | 3 |
| *ELFF* | class, method | static source code metrics | 23 |
| *Had-oops!* | class | static source code metrics | 1 |
| *Mutation-aware fault prediction dataset* | class | static source code metrics, mutation metrics | 3 |
| *GitHub Bug Dataset* | file, class, method | static source code metrics, code duplication metrics, code smell metrics | 15 |
| *BugHunter dataset* | file, class, method | static source code metrics, code duplication metrics, code smell metrics | 15 |

datasets include various projects from GitHub, include numerous static source code metrics, and store a large number of entries in fine granularity (file, class, and method level as well). Furthermore, we also experimented with chronology, although in a different way compared to Harman et al [60]. The differences between the traditional datasets and the proposed novel *BugHunter Dataset* are summarized in Table 3.1. See Section 3.1.3 for details about the process of selecting the bug-related data for the novel dataset. The detailed comparison can be found in Section 3.4.4.

# 3.1 Methodology

In this section, we will outline the methodology that was employed to construct the dataset. We carried out the data processing in multiple steps using the toolchain shown in Figure 3.1. Each of these steps – and their corresponding components – are detailed in their dedicated sections below.

## 3.1.1 Subject System Selection

We considered several criteria when searching for appropriate projects on GitHub. First of all, we searched for projects written in Java, especially larger ones, because those are more suitable for this kind of analysis. It was also important to have an adequate number of issues labeled as bugs, and the number of references from the log messages to certain commits is also a crucial factor (this is how we can link source code elements to bugs). Additionally, we preferred projects that are still actively maintained. Logged-in

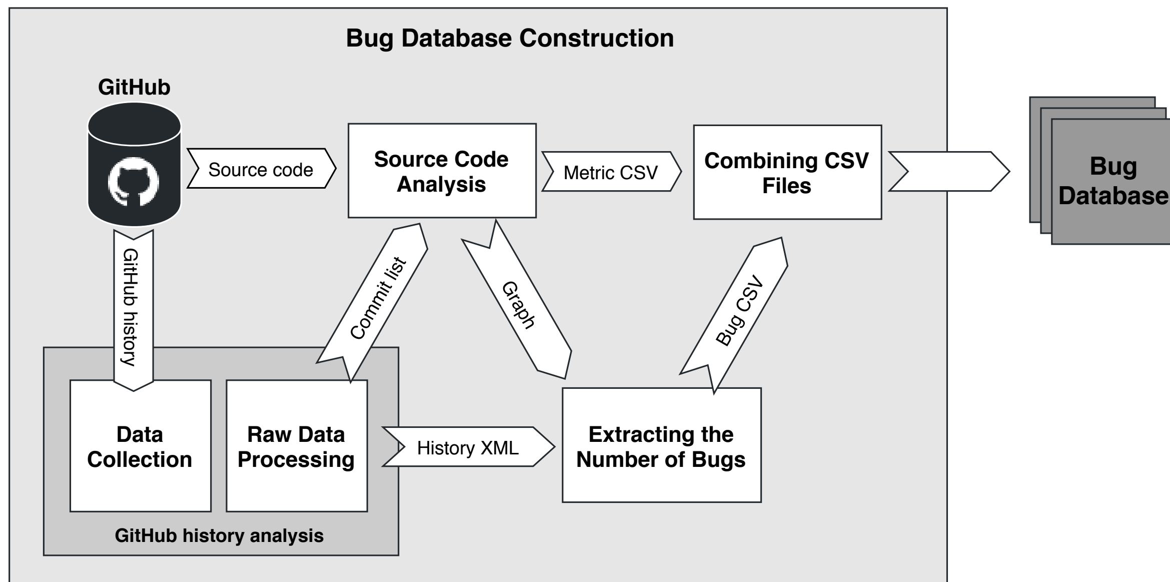


Figure 3.1: The components of the process

users can give a star for any repository and bookmark selected ones to follow. The number of stars and watches applied to repositories forms a ranking between them, which we will refer to as "popularity" from now on. We performed our search for candidate projects mainly based on popularity and activity. During our search, we discovered several projects that could have met most of our criteria, were it not for the developers' use of an external bug tracking system, which we were unable to support at that time.

In the end, we selected the 15 projects listed in Table 3.3 based on the previously mentioned criteria. As the descriptions show, these projects cover different domains; a good practice when the goal is creating a general dataset. The table contains the following additional data about the projects:

**Stars** the number of stars a project received on GitHub

**Forks** the number of forks of a project on GitHub

**kLOC** the thousand lines of code a project had in September, 2017

Recently, the repository of the Eclipse Ceylon project was moved to a new location and the old repository is not available anymore. Due to this reason, we could not obtain the total number of stars and the total number of forks of this repository, which resulted in the low values in the table.

Besides knowing each project's domain, further descriptors can help us get a more precise understanding. Table 3.4 provides a more accurate picture of the projects by showing different characteristics (related to the repositories) for each project. This table sums up the occurrences of various bug reports and commits of the projects present in September 2017. Considering the total number of commits (TNC) is a good starting point to show the scale and activity of the projects. The number of commits referencing a (closed) bug (NCRB) shows how many commits out of TNC referenced a bug by using the pattern '#$x$' in their commit log messages, where $x$ is a number that uniquely identifies the proper issue that is labeled as a bug [87]. NCBR (Number of Closed Bug Reports) is also important since we only consider closed bug reports and

Table 3.3: The selected projects and their descriptions

| Project | Stars | Forks | kLOC |
|---|---|---|---|
| Android Universal Image Loader[1]<br>An Android library that assists with the loading of images. | 16,521 | 6,357 | 13 |
| ANTLR v4[2]<br>A popular piece of software in the field of language processing. It is a powerful parser generator for reading, processing, executing, or translating structured text or binary files. | 6,030 | 1,559 | 68 |
| Broadleaf Commerce[3]<br>A framework for building e-commerce websites. | 1,266 | 1,020 | 322 |
| Eclipse plugin for Ceylon[4]<br>An Eclipse plugin that provides a Ceylon IDE. | 56 | 30 | 181 |
| Elasticsearch[5]<br>A popular RESTful search engine. | 42,685 | 14,303 | 995 |
| Hazelcast[6]<br>A platform for distributed data processing. | 3,211 | 1,169 | 949 |
| jUnit[7]<br>A Java framework for writing unit tests. | 7,536 | 2,826 | 43 |
| MapDB[8]<br>A versatile, fast, and easy-to-use database engine in Java. | 3,700 | 745 | 68 |
| mcMMO[9]<br>An RPG game based on Minecraft. | 511 | 448 | 42 |
| Mission Control Technologies[10]<br>Originally developed by NASA for space flight operations. It is a real-time monitoring and visualization platform that can be used for monitoring any other data as well. | 818 | 280 | 204 |
| Neo4j[11]<br>The world's leading graph database with high performance. | 6,643 | 1,636 | 1,027 |
| Netty[12]<br>An asynchronous event-driven networking framework. | 20,006 | 9,128 | 380 |
| OrientDB[13]<br>A popular document-based NoSQL graph database. Mainly famous for its speed and scalability. | 3,919 | 792 | 621 |
| Oryx 2[14]<br>An open-source software with machine learning algorithms that allows the processing of huge data sets. | 1,633 | 388 | 34 |
| Titan[15]<br>A high-performance, highly scalable graph database. | 4,931 | 1,015 | 108 |

the corresponding commits in this context. The abbreviations we used stand for the following:

**TNC** Total Number of Commits
**NCRB** Number of Commits Referencing a Bug
**NBR** Number of Bug Reports
**NOBR** Number of Open Bug Reports
**NCBR** Number of Closed Bug Reports

Table 3.4: Statistics about the selected projects

| Project name | TNC | NCRB | NBR | NOBR | NCBR | ANCBR |
|---|---|---|---|---|---|---|
| Android U. I. L. | 1,025 | 52 | 90 | 15 | 75 | 0.69 |
| ANTLR v4 | 6,526 | 162 | 179 | 23 | 156 | 1.04 |
| Broadleaf Comm. | 14,920 | 1,051 | 703 | 28 | 675 | 1.56 |
| Eclipse Ceylon | 7,984 | 316 | 923 | 82 | 841 | 0.38 |
| Elasticsearch | 28,815 | 2,807 | 4,494 | 207 | 4,287 | 0.65 |
| Hazelcast | 24,380 | 3,030 | 3,882 | 120 | 3,762 | 0.81 |
| jUnit | 2,192 | 72 | 90 | 6 | 84 | 0.86 |
| MapDB | 2,062 | 167 | 244 | 16 | 228 | 0.73 |
| mcMMO | 4,765 | 268 | 664 | 8 | 656 | 0.41 |
| Mission Control T. | 977 | 15 | 46 | 9 | 37 | 0.40 |
| Neo4j | 49,979 | 781 | 1,268 | 116 | 1,152 | 0.68 |
| Netty | 8,443 | 956 | 2,240 | 33 | 2,207 | 0.43 |
| OrientDB | 15,969 | 722 | 1,522 | 250 | 1,272 | 0.57 |
| Oryx | 1,054 | 69 | 67 | 2 | 65 | 1.06 |
| Titan | 4,434 | 93 | 135 | 8 | 127 | 0.73 |

**ANCBR** Average Number of Commits per closed Bug Reports (NCRB/NCBR)

It is apparent that the projects are quite different according to the number of bug reports and the lines of code they have. NCRB is always lower than NCBR except in three cases (ANTLR v4, Oryx, Broadleaf Commerce), which means that not all bug reports have at least one referencing commit to fix the bug. This is possible since closing a bug is viable not only from a commit but directly from GitHub's Web user interface without committing anything.

Figure 3.2 depicts the number of commits for each closed bug report. One insight here is that the rate of closed bug reports is high where not even a single commit is present to fix the bug. There are several possible causes for this, for example, the bug report is not referred to from the commit's log message, or the error has already been fixed.

Figure 3.3 shows the ratio of the number of commits per project, illustrating the activity and the size of the projects. When considering the number of commits, Neo4j appears to be dominant. However, bug report-related activities are relatively average

[1] `https://github.com/nostra13/Android-Universal-Image-Loader`
[2] `https://github.com/antlr/antlr4`
[3] `https://github.com/BroadleafCommerce/BroadleafCommerce`
[4] `https://github.com/eclipse/ceylon-ide-eclipse`
[5] `https://github.com/elastic/elasticsearch`
[6] `https://github.com/hazelcast/hazelcast`
[7] `https://github.com/junit-team/junit4`
[8] `https://github.com/jankotek/MapDB`
[9] `https://github.com/mcMMO-Dev/mcMMO`
[10] `https://github.com/nasa/openmct`
[11] `https://github.com/neo4j/neo4j`
[12] `https://github.com/netty/netty`
[13] `https://github.com/orientechnologies/orientdb`
[14] `https://github.com/OryxProject/oryx`
[15] `https://github.com/thinkaurelius/titan`

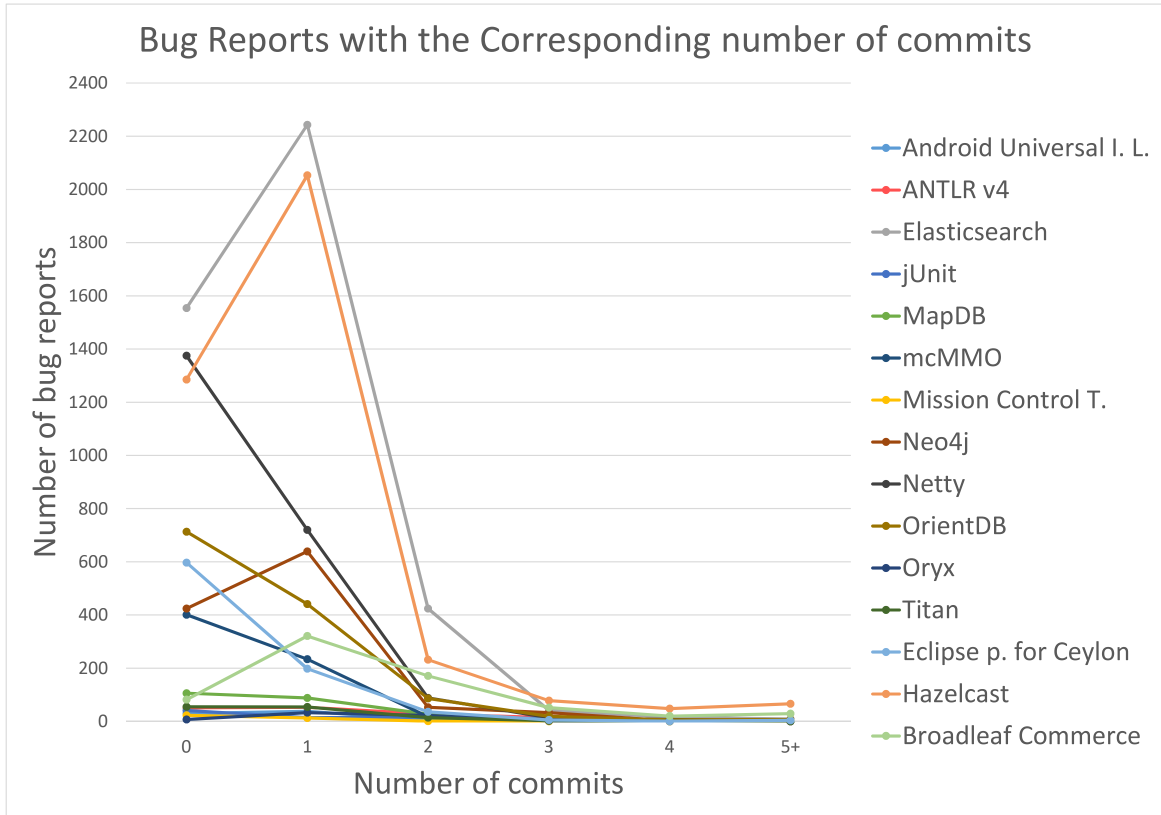


Figure 3.2: The number of bug reports with the corresponding number of commits

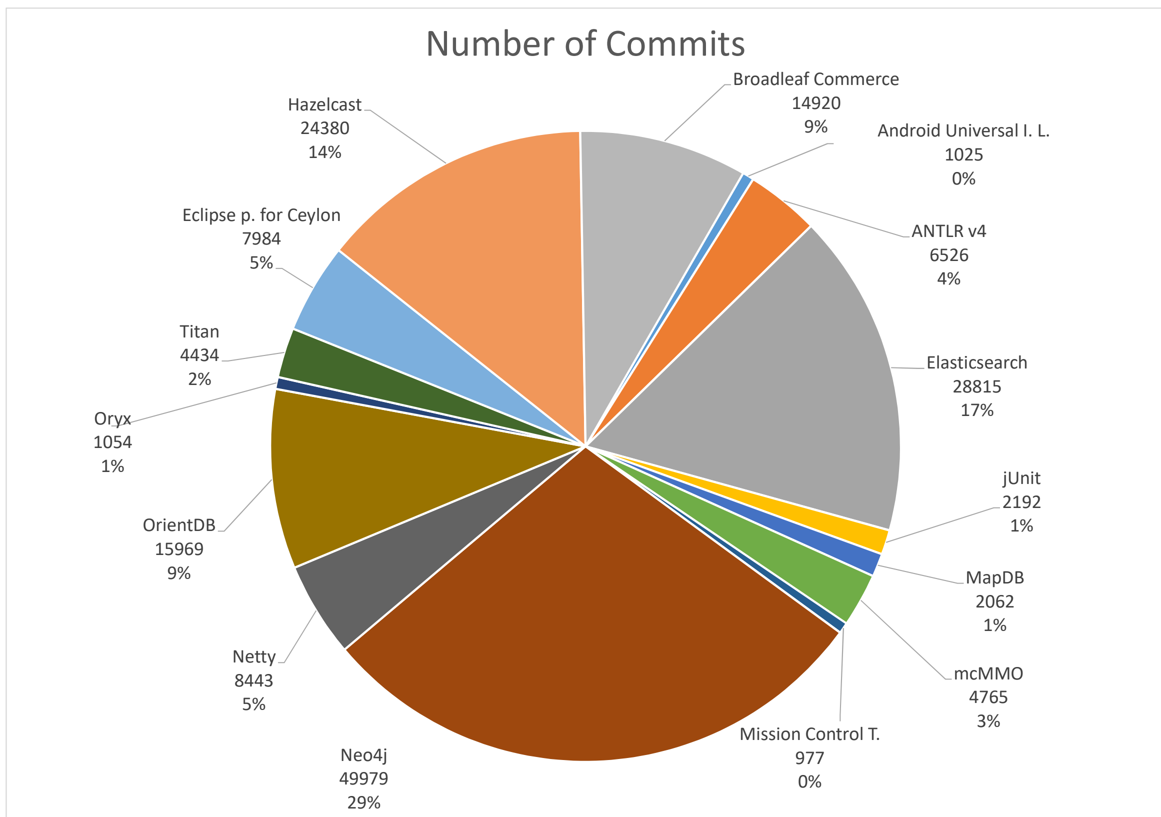


Figure 3.3: The number of commits per projects

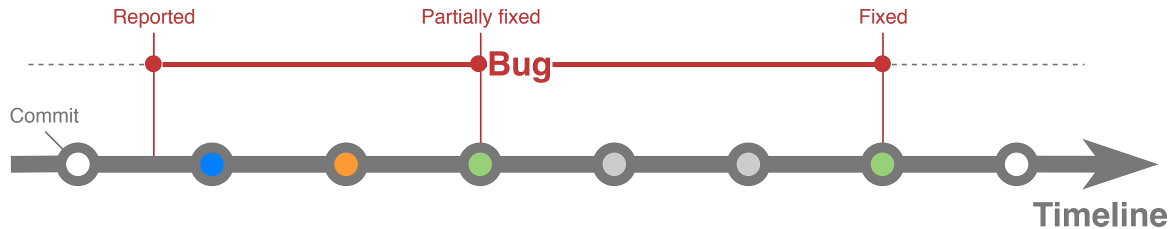


Figure 3.4: The relationship between the bug reports and commits

in comparison. The figures presented demonstrate the variability among the selected software systems, which contributes to the construction of a diverse and heterogeneous dataset.

### 3.1.2 Data Collection

As the first step, we save data about the selected projects via the GitHub API. This is necessary because while the data is continuously changing on GitHub due to the activities in the projects, we need a consistent data source for the analysis. The data we save includes the list of users assigned to the repository (Contributors), the open and closed bug reports (Issues), and all of the commits. For open issues, we store only the date of their creation. For closed issues, we store the creation date, closing date, and the hash of the fixing commits with their commit dates. Additionally, we focus exclusively on bug-related issues, so closed bugs that were not referenced from any commit were not stored. This filtering is based on the issue labels provided by GitHub and the set of labels we manually selected for each project. The data we store about the commits includes the identifier of the contributor, the parent(s) of the commit, and the affected files with their corresponding changes. All this raw information is stored in an XML format, ready for further processing.

### 3.1.3 Data Processing

While the data extracted from GitHub includes all commits, we only need the ones that relate to the bug reports. Amongst these commits, some can occur that need to be removed because they are no longer available through Git (deleted, merged). Moreover, we do not only search for links from the direction of commits but also from the direction of issues (bug reports). When considering a bug report, we can find a commit id showing that the bug was closed in that specific commit.

The selected commits are then divided into different subsets, as depicted in Figure 3.4. Green nodes are directly referencing the bug report (fixing intention). Gray nodes are commits applied between the first fix and the last fix but not referencing the bug id in their commit log messages. We have taken one extra commit into consideration, which is the one right before the first fix (colored with orange). This commit holds the state when the source code is buggy (not fixed yet), thus a snapshot (source code analysis) will be performed at that point too. Although the orange node represents the latest state where the bug is not fixed yet, the blue nodes also contain the bug so we mark the source code elements as buggy in these versions as well. These blue markings are important for distinguishing commits that are involved in multiple bugs at the same time. The white nodes are considered free from the bug.

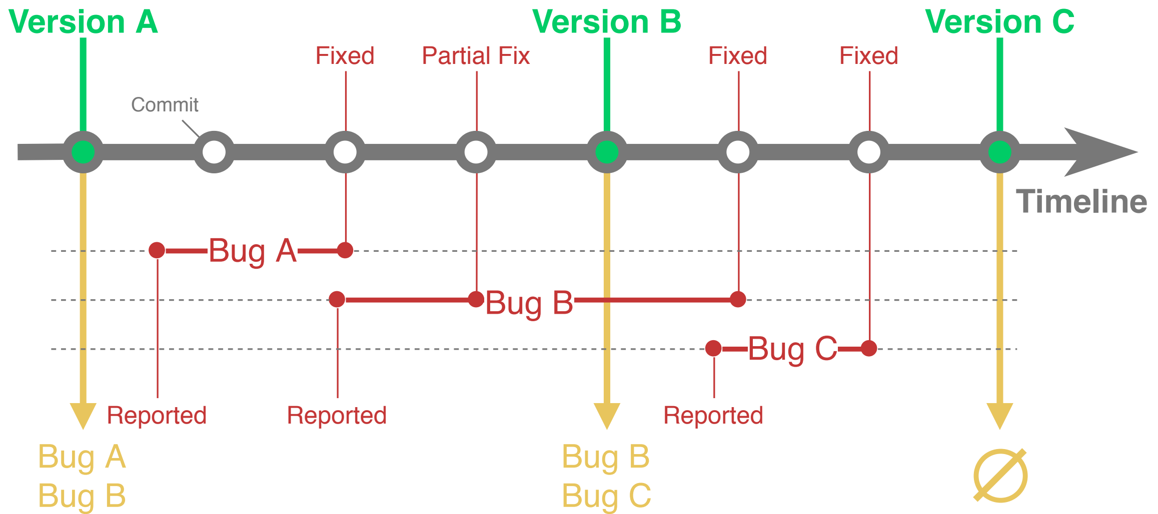


Figure 3.5: The relationship between the bugs and release versions

For the *GitHub Bug Dataset*, we create bug databases from release versions. Let us consider a few bugs that were later fixed (see Figure 3.5). There are 3 release versions of the system: Version A, Version B, and Version C, and we have 3 bugs in the software: Bug A, Bug B, and Bug C. Bug A was reported after Version A and was resolved prior to Version B, indicating that Bug A was only present in Version A of the system. By the time of Version B, Bug A had already been fixed, and therefore it is not included in that version or any subsequent versions. Bug B was also reported after Version A; however, Bug B was eventually fixed after Version B, resulting in Bug B being included in the output for Version A and Version B as well. Bug C was reported after Version B and was resolved prior to Version C, therefore it is included only in Version B.

Since the faulty elements are determined from the viewpoint of reported issues, and the issues are independent of the release versions that we selected, this means that the bug information is scattered across different points in time. If a bug was reported after a specific release version and fixed before the subsequent selected version, then the bug does not appear in either of the databases. To solve this issue, a common solution is to aggregate the bug information to the selected release versions. For every issue, we determine the preceding release version from those that we selected and mark the buggy source code elements.

For the construction of the database, we use the traditional approach, which means that we collect release versions with approximately six-month-long time intervals for every project. We use six-month-long intervals since enough bugs and versions are present for such a long time interval. Based on the age of a project, the number of selected release versions could differ for each project. We select the release versions manually from the list of releases located on the projects' GitHub pages. It is common practice that projects use the release tag on a newly branched (from the master) version of the source code. Since we only use the master branch as the main source of information, we have to perform a mapping when the hash id of the selected release is not representing a commit located in the master branch. Developers usually branch from the master and then tag the branched version as the release version, so our mapping algorithm detects when (timestamp) the release tag was applied on a version and searches for the last commit in the master branch that was made right before this timestamp.

```
1 --- /path/to/original ''timestamp''
2 +++ /path/to/new  ''timestamp''
3 @@ -1,4 +1,4 @@
4 +Added line
5 -Deleted line
6  This part of the
7  document has stayed the
8  same
```

Listing 3.1: Unified diff format example

For creating the *BugHunter Dataset*, we use two versions for each issue: the one right before the (first) fix was applied (orange), and the one where the bug was fully fixed (last green).

We have to perform code analysis on the orange and green commits and also on the release versions to construct the two datasets. At this step, we assemble a list that contains all the commit ids (hash) for each selected project to undergo static analysis.

### 3.1.4 Source Code Analysis

After gathering the appropriate versions of the source code for a given project, feature extraction could begin. This component wraps up the results of the OpenStaticAnalyzer tool. The output of the analysis contains the files, classes, and methods along with source code positions and various software product metrics (as described in Section 2.6). At this point, we have obtained all the desired raw data, including the source code elements and bug-related information from the project.

### 3.1.5 Extracting the Number of Bugs

The next step is to link the two data sets – the results of the code analysis and the data gathered from GitHub – and extract the characteristics of the bugs. We identify the source code elements impacted by the commits, as well as the number of bugs associated with each commit at the file, class, and method levels.

To determine the affected source code parts, we use an approach similar to the SZZ algorithm [126]. However, we do not want to detect the fix-inducing commits, only the mapping between the fixing code snippets and source code elements. For this purpose, we use the diff files – from the GitHub data – that contain the differences between two source code versions in a unified diff format. An example unified diff file snippet is shown in Listing 3.1. Each diff contains header information specifying the starting line number and the number of affected lines. Using this information, we can get the range of the modification (for a given file pair: original and new). To obtain a more accurate result, we subtract the unmodified code lines from this range. Although the diff files generated by GitHub contain additional information about which method is affected, it does not carry enough information because the difference can affect multiple source code elements (overlapping cases that are not handled by GitHub). Thus, there is no further task but to examine the source code elements in every modified file and identify which ones are affected by the changes. This method uses the source code elements' position, i.e., source line mappings from the output of the OpenStaticAnalyzer tool. We identify the source code elements by their fully qualified names that involve the

name of the package, the class, the method, the type of the parameters, and the type of the return value.

Next, we take the commits that were selected by the “Data Processing” step and mark the code sections affected by the bug in these commits. We do this by accumulating the modifications on the issue-level and collecting the fully qualified names of the elements. Then, the algorithm marks the source code elements in the appropriate versions that would then be entries in the dataset (modified by a bug fix). If a source code element in a specific version is marked by multiple issues, then it contains multiple bugs in that version. Finally, the data for files, classes, and methods are exported into three different files in a simple CSV format.

For the traditional *GitHub Bug Dataset*, we create a database for each of the release versions. The first row of these files contains the header information, namely the qualified name and the bug cardinality. Since bug tracking was not always used from the beginning of the projects, we could not assign any bug information to some of the earlier release versions. Also, the changing developer activity could result in a lack of bug reports, consequently making bug-fixing commits rare. All of these factors play roles in that the created databases vary in the number of bugs.

For the novel *BugHunter Dataset*, the first row of these files contains the header information, namely the commit id, the qualified name, and the bug cardinality. Further lines store the data of the source code elements according to the header. One entry is equivalent to one source code element at a given time (the same source code element can occur more than once with a different commit id – hash).

### 3.1.6 Combining CSV Files

As a final step, we merge the CSV outputs of OpenStaticAnalyzer and the previously described CSV output. In this phase, we match the source code elements that are entries in the dataset to the metrics and rule violations calculated during the static analysis. The output of this step is also a CSV file for each type of source code element, extended with the additional columns.

### 3.1.7 The GitHub Bug Dataset

Regarding the traditional *GitHub Bug Dataset*, in total we selected 105 release versions for the 15 projects with six-month-long intervals. The three columns: Method, Class, and File in Table 3.5 present the number of entries for each project at the three levels. The last column shows the number of release versions included in the dataset. The *GitHub Bug Dataset* is available as an online appendix at:
`https://www.inf.u-szeged.hu/~ferenc/papers/GitHubBugDataSet`

In the *database* folder, there are subfolders for each of the subject projects that contain a directory for each release version selected from that project. These directories contain the CSV files of the dataset. There are separate CSV files for the file, the class, and for the method level.

### 3.1.8 The BugHunter Dataset

As the main result and contribution of this thesis point, we constructed a novel kind of bug dataset that contains before/after fix states of source code elements at file, class,

Table 3.5: The number of entries in the *GitHub Bug Dataset*

| Project | Method | Class | File | Release versions |
|---|---|---|---|---|
| Android U. I. L. | 4,007 | 639 | 478 | 6 |
| ANTLR v4 | 18,049 | 2,353 | 2,029 | 5 |
| Broadleaf Commerce | 164,294 | 17,433 | 14,703 | 11 |
| Eclipse Ceylon | 25,031 | 4,512 | 2,129 | 5 |
| Elasticsearch | 313,767 | 54,562 | 23,252 | 12 |
| Hazelcast | 156,885 | 25,130 | 14,791 | 9 |
| jUnit | 18,820 | 5,432 | 2,266 | 8 |
| MapDB | 20,418 | 2,740 | 962 | 6 |
| mcMMO | 11,103 | 1,393 | 1,348 | 6 |
| Mission Control T. | 32,664 | 6,091 | 1,904 | 3 |
| Neo4j | 169,544 | 32,156 | 18,306 | 9 |
| Netty | 85,428 | 11,528 | 8,349 | 9 |
| OrientDB | 106,576 | 11,643 | 9,475 | 6 |
| Oryx | 10,142 | 2,157 | 1,400 | 4 |
| Titan | 32,443 | 5,312 | 3,713 | 6 |
| **Total** | 1,169,171 | 183,078 | 105,105 | 105 |

and method levels. We produced a dataset for every project (see Table 3.3) and also a combined one with all the projects included. In Table 3.6, we collected the general metadata about the dataset, which is scattered throughout the chapter.

The resulting *BugHunter Dataset* 1.0 is available as an online appendix at:
`https://www.inf.u-szeged.hu/~ferenc/papers/BugHunterDataSet`

The `BugHunterDataset-1.0.zip` file contains the dataset in CSV format as described above. The directory named `full` contains the unfiltered database. The remaining four directories, namely `gcf`, `remove`, `single`, and `subtract` contain the results of the different filtering methods. Each of these directories contains 15 subdirectories – one for each subject system – and an additional directory named `all`, which contains the summarized dataset. Three CSV files are placed in these directories for file, class, and method levels respectively. There is also a fourth CSV file in each directory, called `method-p.csv`, which contains the method-level dataset extended with an additional column, the name of the parent class (see Section 3.4.3). Additionally, the `appendix.zip` file contains the analysis results presented in Section 3.4 in spreadsheet files.

## 3.2 Computational Cost of Extending the Dataset

Extending the dataset comes with computational costs that depend on multiple factors. Adding a new project to the dataset requires the project to have bug reports along with bug-fixing commits. Finding such projects is time-consuming, because it mostly requires manual work to select good candidates (see Section 3.1.1).

The most critical step is to collect appropriate bugs. For this initial dataset, we have collected projects from GitHub, since its API makes it easy to gather the required information about bugs automatically. The actual runtime of this step depends on the size of the project, e.g. number of commits and number of bug reports, but for the selected 15 projects, it took just a few hours to save the required data. Selecting bugs manually would take considerably more time. GitHub has a limit on the number of

Table 3.6: *BugHunter Dataset* metadata

| | |
|---|---|
| **Type** | bug prediction dataset |
| **Granularity** | file, class, method |
| **Number of projects** | 15 |
| **Number of metrics** | static source code metrics (66),<br>code duplication metrics (8),<br>code smell metrics (35) |
| **Number of entries** | file: 70,088<br>class: 84,562<br>method: 159,078 |
| **Ratio of the faulty entries** | file: 1.03 (35,507/34,581)<br>class: 0.95 (41,098/43,475)<br>method: 0.59 (58,810/100,268) |

API requests per hour, which increased the total runtime. It is possible to collect data from other sources, although it may require a different amount of work.

The most time- and resource-consuming task is the source code analysis. We used the OpenStaticAnalyzer tool, which performs deep static analysis, therefore, it requires more resources than a simple parser tool. During this step, we extract the static source code metrics and the source code positions of the classes and methods, as described in Sections 3.1.3 and 3.1.4, respectively. The computational cost of this step highly depends on the number of bug-fixing commits, the size of the source code, and the analyzer tool. It took days to analyze each of the nearly 10,000 bug-fixing commits. There are other tools that could be used to extract the source code positions and other tools to compute metrics with a potentially shorter runtime, but the tool we used produces a wide range of metrics and rule violations accurately in a well-processable format.

The next step, determining the buggy source code elements, is a simple algorithm that does not require many resources. The runtime here mostly depends on the number of bug-fixing commits. This step took only a few hours for the 15 projects.

For example, processing a smaller project such as jUnit took around 2 hours of machine time: 10 minutes to download the data from GitHub, 110 minutes to analyze 107 versions of the project (on average 1 minute per version), and around 2 minutes to produce the bug dataset entries. Regarding a larger project, Elasticsearch, it took around 6 hours to download the data from GitHub, around 1,600 hours to analyze 4,881 versions of the project (on average 20 minutes per version), and it took around 90 minutes to produce the bug dataset.

At this point, the data is ready to be added to the dataset. The last step is to match the format of the dataset (see Section 3.1.7 and Section 3.1.8). Since the dataset consists of CSV files, it is very easy to extend it with new projects or with additional bugs for the projects that are already present.

## 3.3 Validation

When constructing a dataset in an automatic way, one always has to validate the constructed set. As seen previously, this kind of generated data should always be handled

with mistrust [71]. We chose the mid-sized project JUnit for such manual validation, which contains 90 bug reports (6 open and 84 closed) and a total of 72 referencing commits to the bugs, thus this project seems to be suitable for manual validation investing a reasonable amount of effort. We validated the 84 closed bug reports manually to verify whether the bugs are valid and whether the matching algorithm works well. Table 3.7 summarizes our findings.

Table 3.7: Validation results

| **Closed bugs** | **Bugs in dataset** | **Commits** | **Java code** | **Commit mismatch** |
|---|---|---|---|---|
| 84 | 37 | 72 | 61 | 5 |

From the total number of closed bugs, only 37 are present in the dataset because many fixing commits are not related to the source code (e.g., documentation) or Java code (e.g., bug in build XML). This is shown in the Java code column of the table that summarizes the number of commits that contain Java language code (61). These commits that include code are referencing 37 bugs, thus at least one bug exists that is referenced from multiple commits according to the pigeonhole principle.

We found 5 “commit mismatch” cases in total, where only comments were modified in the source code – this means 7 entries in the dataset. The dataset created for JUnit has 734 entries in total (92 files, 216 classes, and 426 methods), thus a very small (0.95%) number of entries was incorrectly included. Out of the 734 entries, 286 are not related to test code (43 files, 77 classes, and 166 methods). Based on this, we can presume that our validated dataset can be an appropriate corpus for further investigations and that our bug extraction mechanism is working quite reliably.

### 3.3.1 Relationship between Files and Classes

In the case of Java, there is usually one class per *.java* file. We examined how true this is for our subject projects. We randomly chose 100 commits that we analyzed from each project and we counted the number of classes in each file, not including test files. After we calculated the frequency of these values for each commit, we calculated the average frequency (see Figure 3.6). The diagram shows that most of the files contain only a single class (865), but there is a significant number of files with more than one class (120 files with 2 classes, 46 files with 3 classes, etc.).

Although we have a larger set of metrics on the class level than on file level, we cannot associate a file to a single class due to the one-to-many relationship between them.

## 3.4 Evaluation

In this section, we give answers to research questions related to the *BugHunter Dataset* by evaluating it using machine learning algorithms. As trying to predict the exact number of bugs in a given source code element would be much more difficult – and would presumably require much larger datasets – we chose to restrict our study to predicting a boolean “flag” for the presence of any bugs. Thus, we only applied classification algorithms, and to do so, classes needed to be formed from the bug numbers. For the

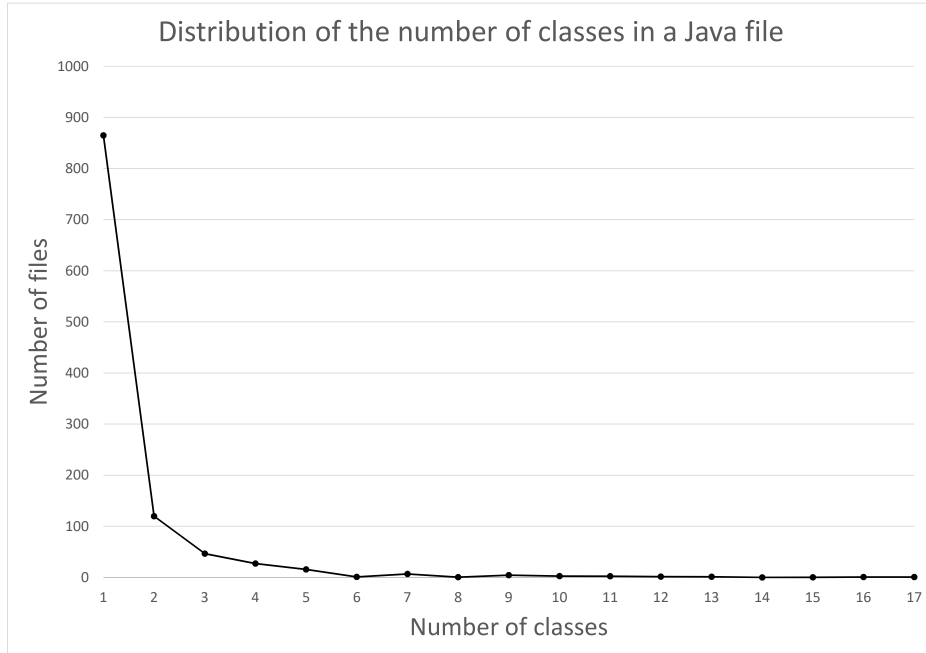


Figure 3.6: Distribution of the number of classes in a Java file

binary classification, we divided the instances into two classes. Instances with zero bugs were categorized as non-buggy, and those with one or more bugs were categorized as buggy.

Upon examining the learning sets, we observed that the dataset is imbalanced. The ratio of positive to negative entries is not equal, which could be misleading during model training. The imbalance is caused by the fact that fixing a bug may involve introducing or removing classes and/or methods from the source code, resulting in a difference in the number of affected entries before and after the fix. To address this issue, we employed random undersampling [62, 124] to obtain an equal number of elements in both categories. For example, if the final corpus at the method level contained 10 buggy methods and 50 non-buggy methods, we use random undersampling on the non-buggy set to reduce the number of samples and balance the ratio to 10-10. The training process, using this random undersampling, was repeated 10 times, and an average was calculated. As described in Section 2.4, we used 10-fold cross-validation to measure the accuracy of the models, and to compare the models, we used precision, recall, and F-measure metrics.

### 3.4.1 Filtering

As the data set should be suitable for studying the connection between different metrics and bug occurrences, it should serve as a practical input for different machine learning algorithms. It is possible, however, to have entries in the dataset that have the same metric values with different numbers of bugs assigned to them. For example, let us consider a buggy method $f$ with metric values $M_{f_1}$. After the bug fix, the metric values of $f$ are changed to $M_{f_2}$. Similarly, let us consider another buggy method $g$ with metric values $M_{g_1}$ and $M_{g_2}$, respectively. These two methods could contain two different bugs that are present in a system for distinct periods of time. In this case, the dataset would contain 4 entries: $M_{f_1}$, $M_{f_2}$, $M_{g_1}$, $M_{g_2}$, where $M_{f_1}$ and $M_{g_1}$ are buggy

and $M_{f_2}$ and $M_{g_2}$ are non-buggy entries. If any of these metric values are equal (e.g. $M_{f_1} = M_{g_2}$ or $M_{g_1} = M_{g_2}$), then redundancy occurs that can influence the accuracy of machine learning for bug prediction (overfitting, contradicting records).

To solve this issue, we used different approaches to filter the raw dataset and eliminate redundant entries. We tried various methods to reduce the noise in the learning set, whose entries are classified into either buggy or not buggy.

- **Removal**: keep the entries located in the class with the larger cardinality (e.g., for a 10:20 distribution, the result is 0:20)
- **Subtract**: reduce the number of entries in the class with the larger cardinality by removing as many entries as the cardinality of the smaller class (e.g., for a 10:20 distribution, the result is 0:10)
- **Single**: remove the entries of the class with the smaller cardinality and hold only one entry from the larger one (e.g., for a 10:20 distribution, the result is 0:1)
- **GCF**: divide the number of entries of both classes by their greatest common factor (or greatest common divisor) and retain only the resulting amounts of entries from the classes (e.g., for a 10:20 distribution, the result is 1:2)

Each selected approach can seriously modify the result set, thus we investigated all four options and, additionally, the basic case, where no filtering was applied. Tables 3.8, 3.10, and 3.12 present average F-measure values calculated for all of the machine learning algorithms we used for all of the projects. From these tables, we can see that the Single and GCF methods performed quite similarly but were less effective than Subtract or Removal.

Table 3.8: Filtering results at method level

| **Method** | **Precision** | **Recall** | **F-Measure** |
|---|---|---|---|
| No filter | 0.5553 | 0.5501 | 0.5317 |
| Removal | 0.6070 | 0.5963 | 0.5773 |
| Subtract | 0.5974 | 0.5893 | 0.5717 |
| Single | 0.5495 | 0.5448 | 0.5250 |
| GCF | 0.5445 | 0.5408 | 0.5218 |

We employed a statistical significance test, namely the Friedman test [52], with a threshold of $\alpha = 0.05$ to assess the significance of the differences between the averages, as it was done similarly in previous bug prediction studies [63, 53]. Our data do not follow a normal distribution; it consists of dependent samples and we have five paired groups, thus the Friedman test is the appropriate choice. The null hypothesis is that the multiple paired samples have the same distribution. The tests resulted in very low $p$ values ($p_{method}$ = 5.32e-80, $p_{class}$ = 2.03e-77, $p_{file}$ = 1.83e-40), therefore, we reject the null hypothesis, which means the distributions are not equal. Then, we applied the Nemenyi post-hoc test [93], which is usually used after a null hypothesis is rejected to gain more insight into the significance of the differences. Tables 3.9, 3.11, and 3.13 list the results of the tests.

Table 3.9: Significance test results for method level filtering

| | *No filter* | *Removal* | *Subtract* | *Single* | *GCF* |
|---|---|---|---|---|---|
| *No filter* | | **0.001** | **0.001** | 0.736 | **0.020** |
| *Removal* | **0.001** | | 0.210 | **0.001** | **0.001** |
| *Subtract* | **0.001** | 0.210 | | **0.001** | **0.001** |
| *Single* | 0.736 | **0.001** | **0.001** | | 0.343 |
| *GCF* | **0.020** | **0.001** | **0.001** | 0.343 | |

Let us consider the method level F-measure values in Table 3.8 where Removal has the highest average F-measure (0.5773) and Subtract is a close second (0.5717). In Table 3.9, the results of the significance tests for the method level show that the $p$ value of the test between Subtract and No filter is below the threshold ($p = 0.001 < \alpha = 0.05$), therefore, the difference is significant and with Subtract having a higher average F-measure (0.5717) than No filter (0.5317), we can state that it is significantly better. We can conclude the same when comparing Subtract with Single ($p = 0.001 < \alpha = 0.05$) or with GCF ($p = 0.001 < \alpha = 0.05$). The $p$ value between Subtract and Removal is $p = 0.210 > \alpha = 0.05$ which is not significant.

Table 3.10: Filtering results at the class level

| **Method** | **Precision** | **Recall** | **F-Measure** |
|---|---|---|---|
| No filter | 0.5265 | 0.5235 | 0.5128 |
| Removal | 0.5567 | 0.5528 | 0.5419 |
| Subtract | 0.5541 | 0.5499 | 0.5393 |
| Single | 0.5236 | 0.5206 | 0.5090 |
| GCF | 0.5221 | 0.5201 | 0.5077 |

Similar results can be concluded for class level and file level as well. We can state that the Removal and Subtract methods performed significantly better than the other methods in all three cases. The difference between the Removal and Subtract methods is not significant.

Table 3.11: Significance test results for class-level filtering

| | *No filter* | *Removal* | *Subtract* | *Single* | *GCF* |
|---|---|---|---|---|---|
| *No filter* | | **0.0010** | **0.0010** | 0.9000 | **0.0496** |
| *Removal* | **0.0010** | | 0.9000 | **0.0010** | **0.0010** |
| *Subtract* | **0.0010** | 0.9000 | | **0.0010** | **0.0010** |
| *Single* | 0.9000 | **0.0010** | **0.0010** | | 0.1828 |
| *GCF* | **0.0496** | **0.0010** | **0.0010** | 0.1828 | |

We speculate that a disadvantage to Single is that it drops the multiplicity of the records (i.e., the weight information). The problem with GCF, on the other hand, is

that it will only perform filtering when the greatest common factor is not one, and that it does not eliminate the noise completely (i.e., it will keep at least one entry from both classes). Removal removes the noise entirely, but it suffers from the fact that it ignores the minority.

Table 3.12: Filtering results at the file level

| **Method** | **Precision** | **Recall** | **F-Measure** |
|---|---|---|---|
| No filter | 0.5160 | 0.5117 | 0.4883 |
| Removal | 0.5451 | 0.5414 | 0.5194 |
| Subtract | 0.5407 | 0.5371 | 0.5147 |
| Single | 0.5187 | 0.5148 | 0.4910 |
| GCF | 0.5172 | 0.5129 | 0.4889 |

The Subtract method, however, neutralizes the positive and negative entries with identical feature vectors. This means that it removes the noise while also keeping the weight of the records, so this filtering method seems to be the best choice.

Table 3.13: Significance test results for file-level filtering

| | *No filter* | *Removal* | *Subtract* | *Single* | *GCF* |
|---|---|---|---|---|---|
| *No filter* | | **0.0010** | **0.0010** | 0.0682 | 0.9000 |
| *Removal* | **0.0010** | | 0.9000 | **0.0010** | **0.0010** |
| *Subtract* | **0.0010** | 0.9000 | | **0.0010** | **0.0010** |
| *Single* | 0.0682 | **0.0010** | **0.0010** | | 0.1262 |
| *GCF* | 0.9000 | **0.0010** | **0.0010** | 0.1262 | |

We have 3 levels of source code (method, class, file), 16 datasets (including the summarized dataset), 11 machine learning algorithms, and 5 filtering methods. Presenting all the results obtained would be too large for this format, thus we will only present the best-performing algorithms for the results achieved using the Subtract filtering method. Please note that the online appendix (see Section 3.1.8 for the Web link) contains all the analysis results in spreadsheet files.

### 3.4.2 Research Question 1

The first research question we will answer is the following:

**Research Question 1:** *Is the* BugHunter Dataset *usable for bug prediction purposes?*

To answer this question, we present the best results obtained by different machine learning algorithms at the method, class, and file levels. Similar to Section 3.4.1, we used the Friedman test to check whether the distributions of the samples are equal or not. We observed the same, very low $p$ values ($p_{method}$ = 1.21e-14, $p_{class}$ = 1.54e-07,

$p_{projected}$ = 3.77e-14, $p_{file}$ = 1.06e-05), thus the distributions are not equal. In Tables 3.14, 3.15, 3.16, and 3.17, we present the results of the Nemenyi post-hoc tests.

### Method level

We trained models to use method-level metrics to predict future failures at the method level. The results are shown in Table 3.18 containing the best five algorithms selected by F-measure values.

The fifth best algorithm with a 0.5983 F-measure value is DecisionTable. The SimpleLogistic algorithm resulted in a slightly higher F-measure (0.6031). The first three algorithms are all from the tree family. J48 is the second-best algorithm (0.6119). The third and the first algorithms also use trees to produce prediction models. RandomForest (0.6319) builds a forest from RandomTrees to get a slightly better result than RandomTree (0.6110). The results of the statistical tests in Table 3.14 show that the differences between the top five algorithms are not statistically significant ($p > \alpha = 0.05$), but the difference between the worst (NaiveBayes, NaiveBayesMultinomial, and VotedPerceptron) and the best-performing algorithms is significant.

Table 3.18: TOP 5 machine learning algorithms for the method level based on F-measure

| **Algorithm** | **Precision** | **Recall** | **F-Measure** |
|---|---|---|---|
| trees.RandomForest | 0.6335 | 0.6324 | 0.6319 |
| trees.J48 | 0.6147 | 0.6134 | 0.6119 |
| trees.RandomTree | 0.6115 | 0.6113 | 0.6110 |
| functions.SimpleLogistic | 0.6062 | 0.6043 | 0.6031 |
| rules.DecisionTable | 0.6138 | 0.6073 | 0.5983 |

At the method level, trees were performing the best and could result up to 0.6319 when considering F-measure values. We also investigated the results by projects and found that specific projects performed worse than others. Android Universal Image Loader and Eclipse Ceylon were the worst when considering precision, recall, or F-measure. Achieved F-measure values depend highly upon the project itself. A possible factor that plays a role in this is the size of the built corpus. These projects have a smaller training corpus and more inconsistencies in the feature vectors, consequently, it is harder to build a well-performing prediction model for them. This phenomenon does not only appear at the method level but at the class and file levels as well, since at these levels even fewer entries are created in the dataset. The best F-measure values (over 0.75 in one case) achieved on different projects are demonstrated in Table 3.19.

Table 3.14: Significance test results for the method level - Algorithms

| | *NaiveBayes* | *NaiveBayesMultinomial* | *Logistic* | *SGD* | *SimpleLogistic* | *VotedPerceptron* | *DecisionTable* | *OneR* | *J48* | *RandomForest* | *RandomTree* |
|---|---|---|---|---|---|---|---|---|---|---|---|
| *NaiveBayes* | | 0.900 | 0.129 | **0.045** | 0.053 | 0.900 | **0.045** | 0.302 | **0.023** | **0.001** | **0.019** |
| *NaiveBayesMultinomial* | 0.900 | | **0.007** | **0.002** | **0.002** | 0.900 | **0.002** | **0.027** | **0.001** | **0.001** | **0.001** |
| *Logistic* | 0.129 | **0.007** | | 0.900 | 0.900 | **0.001** | 0.900 | 0.900 | 0.900 | 0.302 | 0.900 |
| *SGD* | **0.045** | **0.002** | 0.900 | | 0.900 | **0.001** | 0.900 | 0.900 | 0.900 | 0.546 | 0.900 |
| *SimpleLogistic* | 0.053 | **0.002** | 0.900 | 0.900 | | **0.001** | 0.900 | 0.900 | 0.900 | 0.513 | 0.900 |
| *VotedPerceptron* | 0.900 | 0.900 | **0.001** | **0.001** | **0.001** | | **0.001** | **0.006** | **0.001** | **0.001** | **0.001** |
| *DecisionTable* | **0.045** | **0.002** | 0.900 | 0.900 | 0.900 | **0.001** | | 0.900 | 0.900 | 0.546 | 0.900 |
| *OneR* | 0.302 | **0.027** | 0.900 | 0.900 | 0.900 | **0.006** | 0.900 | | 0.900 | 0.129 | 0.900 |
| *J48* | **0.023** | **0.001** | 0.900 | 0.900 | 0.900 | **0.001** | 0.900 | 0.900 | | 0.679 | 0.900 |
| *RandomForest* | **0.001** | **0.001** | 0.302 | 0.546 | 0.513 | **0.001** | 0.546 | 0.129 | 0.679 | | 0.712 |
| *RandomTree* | **0.019** | **0.001** | 0.900 | 0.900 | 0.900 | **0.001** | 0.900 | 0.900 | 0.900 | 0.712 | |

Table 3.15: Significance test results for the class level - Algorithms

| | *NaiveBayes* | *NaiveBayesMultinomial* | *Logistic* | *SGD* | *SimpleLogistic* | *VotedPerceptron* | *DecisionTable* | *OneR* | *J48* | *RandomForest* | *RandomTree* |
|---|---|---|---|---|---|---|---|---|---|---|---|
| *NaiveBayes* | | 0.900 | 0.169 | 0.169 | **0.019** | 0.900 | **0.004** | 0.878 | 0.369 | 0.900 | 0.900 |
| *NaiveBayesMultinomial* | 0.900 | | 0.646 | 0.646 | 0.191 | 0.546 | 0.063 | 0.900 | 0.878 | 0.900 | 0.900 |
| *Logistic* | 0.169 | 0.646 | | 0.900 | 0.900 | **0.002** | 0.900 | 0.900 | 0.900 | 0.479 | 0.878 |
| *SGD* | 0.169 | 0.646 | 0.900 | | 0.900 | **0.002** | 0.900 | 0.900 | 0.900 | 0.479 | 0.878 |
| *SimpleLogistic* | **0.019** | 0.191 | 0.900 | 0.900 | | **0.001** | 0.900 | 0.679 | 0.900 | 0.098 | 0.406 |
| *VotedPerceptron* | 0.900 | 0.546 | **0.002** | **0.002** | **0.001** | | **0.001** | 0.113 | **0.009** | 0.712 | 0.302 |
| *DecisionTable* | **0.004** | 0.063 | 0.900 | 0.900 | 0.900 | **0.001** | | 0.406 | 0.900 | **0.027** | 0.169 |
| *OneR* | 0.878 | 0.900 | 0.900 | 0.900 | 0.679 | 0.113 | 0.406 | | 0.900 | 0.900 | 0.900 |
| *J48* | 0.369 | 0.878 | 0.900 | 0.900 | 0.900 | **0.009** | 0.900 | 0.900 | | 0.712 | 0.900 |
| *RandomForest* | 0.900 | 0.900 | 0.479 | 0.479 | 0.098 | 0.712 | **0.027** | 0.900 | 0.712 | | 0.900 |
| *RandomTree* | 0.900 | 0.900 | 0.878 | 0.878 | 0.406 | 0.302 | 0.169 | 0.900 | 0.900 | 0.900 | |

Table 3.16: Significance test results for projected - Algorithms

| | *NaiveBayes* | *NaiveBayesMultinomial* | *Logistic* | *SGD* | *SimpleLogistic* | *VotedPerceptron* | *DecisionTable* | *OneR* | *J48* | *RandomForest* | *RandomTree* |
|---|---|---|---|---|---|---|---|---|---|---|---|
| *NaiveBayes* | | 0.900 | **0.003** | **0.001** | **0.006** | 0.900 | **0.001** | **0.005** | **0.001** | **0.001** | **0.005** |
| *NaiveBayesMultinomial* | 0.900 | | **0.001** | **0.001** | **0.001** | 0.778 | **0.001** | **0.001** | **0.001** | **0.001** | **0.001** |
| *Logistic* | **0.003** | **0.001** | | 0.900 | 0.900 | 0.169 | 0.900 | 0.900 | 0.900 | 0.900 | 0.900 |
| *SGD* | **0.001** | **0.001** | 0.900 | | 0.900 | **0.011** | 0.900 | 0.900 | 0.900 | 0.900 | 0.900 |
| *SimpleLogistic* | **0.006** | **0.001** | 0.900 | 0.900 | | 0.271 | 0.878 | 0.900 | 0.900 | 0.900 | 0.900 |
| *VotedPerceptron* | 0.900 | 0.778 | 0.169 | **0.011** | 0.271 | | **0.002** | 0.242 | 0.113 | **0.003** | 0.242 |
| *DecisionTable* | **0.001** | **0.001** | 0.900 | 0.900 | 0.878 | **0.002** | | 0.900 | 0.900 | 0.900 | 0.900 |
| *OneR* | **0.005** | **0.001** | 0.900 | 0.900 | 0.900 | 0.242 | 0.900 | | 0.900 | 0.900 | 0.900 |
| *J48* | **0.001** | **0.001** | 0.900 | 0.900 | 0.900 | 0.113 | 0.900 | 0.900 | | 0.900 | 0.900 |
| *RandomForest* | **0.001** | **0.001** | 0.900 | 0.900 | 0.900 | **0.003** | 0.900 | 0.900 | 0.900 | | 0.900 |
| *RandomTree* | **0.005** | **0.001** | 0.900 | 0.900 | 0.900 | 0.242 | 0.900 | 0.900 | 0.900 | 0.900 | |

Table 3.17: Significance test results for the file level - Algorithms

| | *NaiveBayes* | *NaiveBayesMultinomial* | *Logistic* | *SGD* | *SimpleLogistic* | *VotedPerceptron* | *DecisionTable* | *OneR* | *J48* | *RandomForest* | *RandomTree* |
|---|---|---|---|---|---|---|---|---|---|---|---|
| *NaiveBayes* | | 0.900 | 0.129 | 0.900 | 0.148 | 0.900 | 0.443 | 0.148 | 0.513 | 0.098 | 0.098 |
| *NaiveBayesMultinomial* | 0.900 | | 0.900 | 0.900 | 0.900 | 0.513 | 0.900 | 0.900 | 0.900 | 0.845 | 0.845 |
| *Logistic* | 0.129 | 0.900 | | 0.098 | 0.900 | **0.009** | 0.900 | 0.900 | 0.900 | 0.900 | 0.900 |
| *SGD* | 0.900 | 0.900 | 0.098 | | 0.113 | 0.900 | 0.369 | 0.113 | 0.443 | 0.073 | 0.073 |
| *SimpleLogistic* | 0.148 | 0.900 | 0.900 | 0.113 | | **0.011** | 0.900 | 0.900 | 0.900 | 0.900 | 0.900 |
| *VotedPerceptron* | 0.900 | 0.513 | **0.009** | 0.900 | **0.011** | | 0.063 | **0.011** | 0.085 | **0.006** | **0.006** |
| *DecisionTable* | 0.443 | 0.900 | 0.900 | 0.369 | 0.900 | 0.063 | | 0.900 | 0.900 | 0.900 | 0.900 |
| *OneR* | 0.148 | 0.900 | 0.900 | 0.113 | 0.900 | **0.011** | 0.900 | | 0.900 | 0.900 | 0.900 |
| *J48* | 0.513 | 0.900 | 0.900 | 0.443 | 0.900 | 0.085 | 0.900 | 0.900 | | 0.900 | 0.900 |
| *RandomForest* | 0.098 | 0.845 | 0.900 | 0.073 | 0.900 | **0.006** | 0.900 | 0.900 | 0.900 | | 0.900 |
| *RandomTree* | 0.098 | 0.845 | 0.900 | 0.073 | 0.900 | **0.006** | 0.900 | 0.900 | 0.900 | 0.900 | |

Table 3.19: The best F-measure values by projects at the method level

| Project | F-measure | Algorithm |
|---|---|---|
| ANTLR v4 | 0.7573 | trees.RandomForest |
| Broadleaf Commerce | 0.7366 | trees.RandomForest |
| Hazelcast | 0.7170 | trees.RandomForest |
| Mission Control T. | 0.6876 | trees.RandomForest |
| Oryx | 0.6678 | trees.RandomForest |
| jUnit | 0.6638 | rules.DecisionTable |
| All* | 0.6622 | trees.RandomForest |
| Netty | 0.6412 | trees.RandomForest |
| Elasticsearch | 0.6411 | trees.RandomForest |
| OrientDB | 0.6236 | trees.RandomForest |
| Titan | 0.6216 | functions.SGD |
| Neo4j | 0.6086 | functions.Logistic |
| mcMMO | 0.5815 | trees.RandomForest |
| MapDB | 0.5610 | functions.SimpleLogistic |
| Android U. I. L. | 0.5569 | functions.SGD |
| Eclipse Ceylon | 0.5395 | trees.RandomTree |

### Class level

When considering the class level, we have quite a different set of algorithms in the top five than in the case of methods. Furthermore, the precision, recall, and F-measure values differ significantly from those we obtained at the method level. We suspect that the main reason behind this is the different set of metrics used to predict the possibility of bugs occurring in a class. At the class level, simple logistic, decision table, and SGD were the best. Function- and rule-based groups of machine learning algorithms can be emphasized as the best when considering class level. The best machine learning algorithms at the class level are shown in Table 3.20 with F-measure values around 0.56. In Table 3.15, the results of the significance tests show that the best algorithm, SimpleLogistic with a 0.5685 F-measure, achieved significantly better results than the worst two algorithms that are not in the top five (NaiveBayes $p = 0.019$ and VotedPerceptron $p = 0.001$). Between the top five algorithms, the differences are not significant ($p > \alpha = 0.05$).

Table 3.20: TOP 5 machine learning algorithms for class level based on F-measure

| Algorithm | Precision | Recall | F-Measure |
|---|---|---|---|
| functions.SimpleLogistic | 0.5760 | 0.5763 | 0.5685 |
| rules.DecisionTable | 0.5703 | 0.5705 | 0.5637 |
| functions.SGD | 0.5718 | 0.5676 | 0.5626 |
| functions.Logistic | 0.5561 | 0.5552 | 0.5537 |
| trees.J48 | 0.5531 | 0.5530 | 0.5520 |

The low F-measure values suggest that one cannot build efficient prediction models at the class level. However, we present the F-measure values of individual projects

in Table 3.21. Considering these F-measure values, we can see the same phenomenon as in the case of methods. mcMMO, Android Universal Image Loader, and Neo4j are in the worst 5, which supports the previous experience according to which different projects provide different amounts of "munition" for predicting faults. The best case, however, provides an F-measure of 0.74.

Table 3.21: The best F-measure values by projects at the class level

| **Project** | **F-measure** | **Algorithm** |
|---|---|---|
| Broadleaf Commerce | 0.7400 | trees.RandomForest |
| Oryx | 0.7095 | functions.SGD |
| jUnit | 0.6639 | rules.DecisionTable |
| Hazelcast | 0.6175 | trees.RandomTree |
| MapDB | 0.6138 | functions.SimpleLogistic |
| OrientDB | 0.6132 | functions.SimpleLogistic |
| Mission Control T. | 0.5825 | functions.SGD |
| Elasticsearch | 0.5817 | rules.DecisionTable |
| All* | 0.5803 | rules.DecisionTable |
| Eclipse Ceylon | 0.5789 | rules.DecisionTable |
| ANTLR v4 | 0.5685 | rules.OneR |
| mcMMO | 0.5670 | functions.SGD |
| Titan | 0.5614 | functions.SimpleLogistic |
| Netty | 0.5537 | functions.Logistic |
| Neo4j | 0.5413 | functions.SimpleLogistic |
| Android U. I. L. | 0.4713 | bayes.NaiveBayes |

### File level

In a Java context, a public class is almost equivalent to a file with a *'.java'* extension. However, despite the fact that we compute a different set of metrics for class and file level, the results are quite similar. Since we operate on a different set of metrics at class and file levels, this explains why different machine learning algorithms performed the best. The best algorithms for this level use tree-based approaches to predict bugs as it is shown in Table 3.22. Similarly to the class level, the differences between the top five algorithms are not considered significant ($p > \alpha = 0.05$), as can be seen in Table 3.17. The best-performing algorithm, achieving an F-measure value of 0.5476, is RandomTree. The top five algorithms achieved significantly better results compared to the worst algorithm (VotedPerceptron).

Table 3.22: TOP 5 machine learning algorithms for the file level based on F-measure

| Algorithm | Precision | Recall | F-Measure |
|---|---|---|---|
| trees.RandomTree | 0.5484 | 0.5484 | 0.5476 |
| trees.RandomForest | 0.5458 | 0.5461 | 0.5455 |
| functions.Logistic | 0.5528 | 0.5474 | 0.5367 |
| rules.OneR | 0.5358 | 0.5359 | 0.5348 |
| functions.SimpleLogistic | 0.5491 | 0.5474 | 0.5321 |

We also present the F-measure values obtained in projects in Table 3.23. The take-away remains the same, Android Universal Image Loader and Neo4j are located within the five worst projects again. On the other hand, Broadleaf Commerce, Oryx, and Hazelcast seem to be appropriate to use in model building. The best F-measure value is over 0.77.

Table 3.23: The best F-measure values by projects at the file level

| Project | F-measure | Algorithm |
|---|---|---|
| Broadleaf Commerce | 0.7741 | trees.RandomForest |
| Oryx | 0.6458 | bayes.NaiveBayesMultinomial |
| Hazelcast | 0.6417 | trees.RandomTree |
| All* | 0.6234 | trees.RandomTree |
| OrientDB | 0.6200 | rules.DecisionTable |
| Elasticsearch | 0.6073 | trees.RandomTree |
| Eclipse Ceylon | 0.5857 | trees.J48 |
| Titan | 0.5793 | functions.SimpleLogistic |
| mcMMO | 0.5702 | trees.RandomForest |
| MapDB | 0.5525 | rules.OneR |
| jUnit | 0.5484 | rules.OneR |
| Netty | 0.5344 | trees.RandomTree |
| ANTLR v4 | 0.5212 | trees.RandomForest |
| Neo4j | 0.5138 | rules.DecisionTable |
| Android U. I. L. | 0.4781 | rules.DecisionTable |
| Mission Control T. | 0.4576 | functions.Logistic |

**Answering Research Question 1:** *Considering the results we obtained, we can state that creating bug prediction models at the method level is more successful than at file and class levels if we consider the full dataset. We also showed the diversion in F-measure values by projects, which strengthens our assumption that not all projects are capable of providing an appropriate training set. We can obtain F-measure values on separate projects up to 0.7573, 0.7400, and 0.7741 at method, class, and file levels, respectively, which is promising. To answer our next research question, we carry out an experiment and its results are even better. However, even without knowing that there is a better solution, we can answer this research question in a positive manner and say that the constructed dataset is usable for bug prediction.*

### 3.4.3 Research Question 2

The dataset contains the bug information on both method and class levels, and we also know the containing relationships between classes and methods. However, since classes have a different set of source code metrics than methods, a question arose: can we (and more importantly, should we) use method-level metrics to predict faulty classes? The second research question we will answer is the following:

**Research Question 2:** *Are the method-level metrics projected to the class level better predictors than the class-level metrics themselves?*

We carried out an experiment where we projected the results of the method-level learning to the class level. During the cross-validation of the method-level learning, we used the containing classes of the methods to calculate the confusion matrix from the number of classes classified as buggy and non-buggy. Classes containing at least one buggy method were considered buggy.

We compared this result with the result of the class-level prediction. The results in Table 3.24 show that the projection method performs much better than the prediction with class-level metrics.

We applied the Wilcoxon-signed-rank test [125] (a non-parametric paired test for dependent samples), with a threshold of $Z_{crit} = 1.96$ (for a two-tailed test with $\alpha = 0.05$) to check whether the difference is significant. We also calculated the effect size of these tests with the Pearson correlation coefficient (Pearson's r) from the formula $r = \frac{Z}{\sqrt{N}}$, where $N$ is the total number of samples and $Z$ is the z-score of the test [103]. According to Cohen [35], the effect size is considered small if $r \approx 0.1$, medium if $r \approx 0.3$, or large if $r \approx 0.5$.

After the test, we can confirm that the difference between the projection method and the prediction with class-level metrics is significant ($Z = 10.9 > Z_{crit} = 1.96$) and the effect size is considered large ($r = 0.58$).

We suspect that this is due to the generality of class-level metrics, which are therefore not powerful enough to effectively distinguish source code bugs. Although the bug information for methods does not include all bugs that affect the containing class (e.g. change of fields, interfaces, or superclasses), method-level metrics are more useful for bug prediction.

Table 3.24: The results of projected learning

| **Algorithm** | **Precision** | | **Recall** | | **F-Measure** | |
|---|---|---|---|---|---|---|
| | Projected | Class | Projected | Class | Projected | Class |
| trees.RandomForest | 0.7471 | 0.5336 | 0.7370 | 0.5336 | 0.7405 | 0.5334 |
| trees.RandomTree | 0.7421 | 0.5381 | 0.7273 | 0.5380 | 0.7330 | 0.5376 |
| functions.SGD | 0.7441 | 0.5718 | 0.7288 | 0.5676 | 0.7322 | 0.5626 |
| rules.DecisionTable | 0.7425 | 0.5703 | 0.7404 | 0.5705 | 0.7309 | 0.5637 |
| trees.J48 | 0.7390 | 0.5531 | 0.7250 | 0.5530 | 0.7290 | 0.5520 |

The results of the significance tests between the different machine learning algorithms are displayed in Table 3.16. The best performing algorithm is RandomForest

with a 0.7405 F-measure, which is significantly better than the worst three algorithms that are not displayed in Table 3.24 (NaiveBayes $p = 0.001$; NaiveBayesMultinomial $p = 0.001$; VotedPerceptron $p = 0.003$). The difference between the top five algorithms is not considered significant ($p > \alpha = 0.05$).

When using the projection approach to predict bugs in classes, the F-measure values reach 0.74. As an extension of the answer to the first research question, we can provide the above-described mechanism to locate class-level bugs with higher accuracy in a software system.

**Answering Research Question 2:** *Using method-level metrics for class-level bug prediction performed the best in our study. This fact also contributes to the answer given for Research Question 1. Furthermore, method-level metrics are better predictors when projected to the class level than class-level metrics by themselves.*

### 3.4.4 Research Question 3

The last research question we will answer is the following:

**Research Question 3:** *Is the* BugHunter Dataset *more powerful and expressive than the* GitHub Bug Dataset*?*

Comparing the expressive power of different datasets is a harsh task since, as the various datasets were created with different purposes, they often have only a few independent variables in common. The projects included in these datasets are different as well. Therefore, we provide an objective comparison between our traditional bug dataset, the *GitHub Bug Dataset*, and the *BugHunter Dataset*. The datasets include exactly the same 15 projects, and their sets of independent variables are common and also calculated in the same way with the same tool. We used the same machine learning algorithms to build prediction models. This way, it is quite straightforward to compare the expressiveness and compactness of these datasets.

Firstly, we compare the size of the datasets expressed with the number of entries located in the datasets. Table 3.25 shows the number of entries at method, class, and file levels. The number of entries contained in the traditional *GitHub Bug Dataset* is listed in the "Trad" column. The "BH" column represents the number of entries in the *BugHunter Dataset*, while "Rate" is calculated as follows:

$$Rate = \frac{\#\ of\ entries\ in\ the\ traditional\ dataset}{\#\ of\ entries\ in\ the\ BugHunter\ dataset}$$

The obtained rate is higher than 1.0 for most of the projects in the case of the method level, which shows that the new approach contains fewer entries at this level. A rate of *1.54* is achieved at the method level, *0.41* at the class level, and *0.33* at the file level. It is important to note that the traditional dataset only encompasses data for a six-month-long interval. On the other hand, the *BugHunter Dataset* contains information from the beginning of the project up to September 2017. One would expect that the new approach will contain fewer entries than the traditional one since the *BugHunter Dataset* only contains the entries that were affected by a closed bug. However, the traditional *GitHub Bug Dataset* only depends on the size (number of files, classes, and methods) of the projects included. In contrast, the *BugHunter Dataset* highly depends on the number of closed bugs in the system (a large project can have a small number

of reported bugs). Even if no feature development was performed on a project (the size of the project remains almost the same: in general, no new files and classes are added, only modified) the number of closed bugs implies more entries in the *BugHunter Dataset*, while the size is not affected in any way in the traditional dataset.

Table 3.25: Comparison of the size of the datasets

| **Project** | **Method** | | | **Class** | | | **File** | | |
|---|---|---|---|---|---|---|---|---|---|
| | Trad | BH | Rate | Trad | BH | Rate | Trad | BH | Rate |
| Android U.I.L. | 432 | 325 | 1.33 | 73 | 156 | 0.47 | 63 | 145 | 0.43 |
| ANTLR v4 | 3,640 | 840 | 4.33 | 479 | 314 | 1.53 | 411 | 347 | 1.18 |
| Broadleaf C. | 14,651 | 4,709 | 3.11 | 1,593 | 2,957 | 0.54 | 1,719 | 2,969 | 0.58 |
| Eclipse Ceylon | 8,787 | 2,087 | 4.21 | 1,611 | 1,275 | 1.26 | 700 | 946 | 0.74 |
| Elasticsearch | 34,324 | 35,862 | 0.96 | 5,908 | 24,994 | 0.24 | 3,035 | 17,724 | 0.17 |
| Hazelcast | 21,642 | 32,973 | 0.66 | 3,412 | 19,845 | 0.17 | 2,228 | 14,913 | 0.15 |
| jUnit | 2,441 | 462 | 5.28 | 731 | 316 | 2.31 | 309 | 177 | 1.75 |
| MapDB | 2,913 | 1,456 | 2.00 | 331 | 899 | 0.37 | 138 | 482 | 0.29 |
| mcMMO | 2,531 | 1,184 | 2.14 | 301 | 732 | 0.41 | 267 | 678 | 0.39 |
| Mission C. T. | 9,836 | 105 | 93.68 | 1,887 | 66 | 28.59 | 413 | 52 | 7.94 |
| Neo4j | 30,256 | 7,030 | 4.30 | 5,899 | 3,701 | 1.59 | 3,278 | 2,934 | 1.12 |
| Netty | 8,312 | 11,171 | 0.74 | 1,143 | 5,677 | 0.20 | 914 | 4,023 | 0.23 |
| OrientDB | 17,013 | 9,445 | 1.80 | 1,847 | 4,134 | 0.45 | 1,503 | 3,564 | 0.42 |
| Oryx | 2,506 | 810 | 3.09 | 533 | 598 | 0.89 | 281 | 536 | 0.52 |
| Titan | 8,424 | 785 | 10.73 | 1,468 | 428 | 3.43 | 976 | 378 | 2.58 |
| **Total** | 167,708 | 109,244 | **1.54** | 27,216 | 66,092 | **0.41** | 16,235 | 49,868 | **0.33** |

To sum up, we cannot clearly decide whether the novel dataset is more compact, however, it is clearly visible that BugHunter could compress the bug-related information at the method level. We achieved an F-measure value of 0.6319 at the method level (see Table 3.26) and the composed dataset contains 58,464 fewer entries than the traditional one. In both datasets, the number of entries is sufficient to build a predictive model from, however, we should investigate the predictive capabilities first to conclude our findings related to expressive power and compactness.

Next, we present tables that capture the differences in the prediction capabilities between the two datasets (using F-measures as before). Table 3.26 presents machine learning results for method, class, and file levels, and also the F-measure values for the projected method-level predictors, respectively. The complete tables are not presented here due to lack of space, however, average, standard deviation, min, and max values are calculated and included in the tables, which provide a general picture for the comparison. The `appendix.zip` file supplied as an online appendix (see Section 3.1.8) contains the complete tables with all F-measure values. For the sake of clarity, we describe how we obtained the averages presented here in detail. First, since the traditional dataset consists of multiple versions with bugs from six-month-long intervals, for each project we selected the version from the *GitHub Bug Dataset* that has the highest number of bugs assigned to it. After collecting the machine learning results of the selected versions, we calculated average F-measure values for each algorithm we used.

Then we ranked the algorithms based on these averages and selected the one with the highest average value. We used this average value for the traditional dataset in the comparison. From the *BugHunter Dataset*, we used the previously selected algorithm's average F-measure value, which was calculated on the results obtained after applying the Subtract filtering. We performed this process for method, class, file, and projected levels separately.

Table 3.26: Predictive capabilities

| **Dataset** | **Avg.** | **Std.dev.** | **Min** | **Max** |
|---|---|---|---|---|
| | | Method level | | |
| BugHunter | 0.6319 | 0.0836 | 0.3376 | 0.7573 |
| Traditional | 0.7348 | 0.0789 | 0.4019 | 0.8339 |
| | | Class level | | |
| BugHunter | 0.5685 | 0.0704 | 0.3572 | 0.7400 |
| Traditional | 0.7710 | 0.0869 | 0.3446 | 0.8331 |
| | | File level | | |
| BugHunter | 0.5147 | 0.0749 | 0.3328 | 0.7741 |
| Traditional | 0.6058 | 0.1076 | 0.2882 | 0.8247 |
| | | Projected | | |
| BugHunter | 0.7405 | 0.0914 | 0.3178 | 0.8386 |
| Traditional | 0.7831 | 0.0716 | 0.4399 | 0.8825 |

On the traditional *GitHub Bug Dataset*, the machine learning algorithms performed better, achieving higher F-measure values in every case. The two kinds of datasets differ fundamentally because they are constructed with two different methods. For the traditional dataset, we divided the history of the projects into six-month-long intervals by selecting release versions from the GitHub repository. We collected the reported bugs from these intervals and we assigned the buggy source code elements to these release versions based on the bug fixes. Then we used the state of the source code elements from these selected versions to assemble the bug dataset. This method was used in several previous studies [41, 57]. It is necessary because the bugs are usually reported after releasing a version, thus at the time of the release there are too few bugs to construct a bug dataset. For the bug assignment, we used a heuristic method - similar to other studies [57] - where we assigned each bug to the latest selected version before the bug was reported to the issue tracking system. This method leads to some uncertainty in the dataset because it could happen that the bug is not yet present in the assigned version. Table 3.27 shows some characteristics of this uncertainty.

The second column is the average number of days elapsed between the date of the release and the date of the bugs reported. The maximum that could occur is 180 days because we used intervals that were around 6 months long. We can see that these averages are quite high; the overall average is 85 days. The third column is the average number of commits contributed to the project between the release commit and the date of the bug report. These values vary for each project because it depends on

Table 3.27: Uncertainty in the traditional dataset

| **Project** | **Average days** | **Average commits before reported** | **Average commits before fixed** |
|---|---|---|---|
| Android U. I. L. | 78.78 | 179.04 | 22.82 |
| ANTLR v4 | 83.73 | 94.83 | 66.21 |
| Broadleaf Comm. | 96.40 | 524.88 | 116.74 |
| Eclipse Ceylon | 136.05 | 442.00 | 20.22 |
| Elasticsearch | 93.85 | 1,004.60 | 382.79 |
| Hazelcast | 84.61 | 1,905.88 | 143.54 |
| jUnit | 91.94 | 76.71 | 171.09 |
| MapDB | 102.09 | 150.47 | 25.06 |
| mcMMO | 108.71 | 289.83 | 41.72 |
| Mission Control T. | 64.00 | 203.00 | 55.93 |
| Neo4j | 39.53 | 535.77 | 189.30 |
| Netty | 83.65 | 411.60 | 48.96 |
| OrientDB | 99.21 | 568.76 | 179.30 |
| Oryx | 63.00 | 104.42 | 3.40 |
| Titan | 51.35 | 65.91 | 59.85 |

the developers' activity. For some projects (Elasticsearch, Hazelcast) it could mean thousands of modifications before the bug was reported. The more commits are performed, the higher the probability that the source code element became buggy after the release. The fourth column shows the average number of commits performed between the time when the bug was reported and when the fix was applied. These numbers are much smaller, which also demonstrates that bugs are fixed relatively fast. This fast corrective behavior causes before and after fix states to be less different for the BugHunter approach. Consequently, less difference in metric values makes building a precise prediction model more difficult.

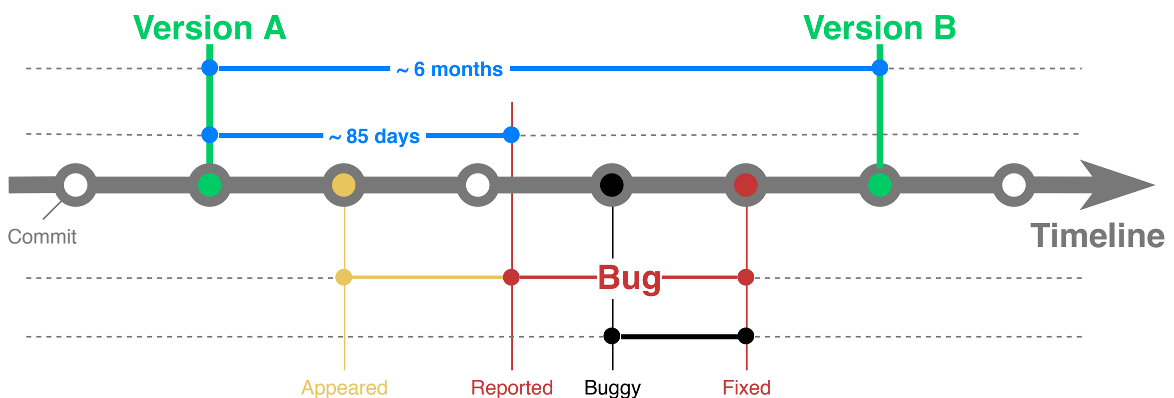


Figure 3.7: Uncertainty included in the traditional approach

The uncertainty is also visualized in Figure 3.7 for the sake of better comprehension. The timeline shows a case when the actual bug appeared in the code after a release, Version A, then the bug was reported at some point and finally fixed before the next release, Version B. The traditional dataset is created with the state captured at the time of Version A. However, the source code elements became buggy after Version A, thus the dataset incorrectly marked them as buggy at that point. This error comes from the methodology itself, which could be leveraged by narrowing the time window (which is traditionally 6 months long). On the other hand, this 6 month interval is not an unwitting choice. If we narrow down the time window, we will also have fewer bugs for an interval, which results in a more unbiased dataset, thus only less powerful predictive models can be built. It would be important to see how many source code elements were marked as buggy incorrectly, however, this cannot be easily measured since the exact time of the bug occurrence cannot be determined (we only know the time when a bug was reported).

The new BugHunter approach, however, is free from the uncertainty mentioned above because it only uses the buggy and the fully fixed states of the bug-related source code elements. This way, the produced bug dataset is more precise, hence it is more appropriate for machine learning. Therefore, we cannot clearly state that the traditional dataset is better, even despite the higher F-measure values. The difference between the values of the two datasets is around 0.10 at the method level, 0.21 at the class level, and 0.09 at the file level. Projecting method-level metrics to the class level achieved almost as high an F-measure value (0.7405) as in the traditional case (0.7831). The difference is only 0.04, yet it is on a much more precise dataset.

**Answering Research Question 3:** *Traditional datasets include a high risk when labeling source code elements as buggy since the elements may become buggy after the release version. This injects false labeling into the training set, which might end up causing deceptive machine learning results (as successfully predicting a bad label is not correct). Unfortunately, the number of incorrectly labeled source code elements cannot be determined since we only know the time when a bug was reported, but not the exact time when it was inserted into the system. These facts make it really hard to take one dataset and state that it is better for bug prediction.*

## 3.5 Threats to Validity

In this section, we briefly describe the threats to validity. Firstly, we present the possible threats to the construct validity, then to the internal and external validity.

### 3.5.1 Threats to Construct Validity

When constructing a dataset in an automatic way, there are always some possible threats to validity. We validated our matching algorithm on JUnit, which was fair in size. However, investigating the validity of the matching in other systems could have revealed additional findings.

As we have seen, commit mismatches can occur during this process, which can distort the final bug dataset. However, manually validating all bugs and the corresponding commits would have been an enormous task.

Deciding which source code elements are faulty and which are not can also cause a construct validity threat. We consider a source code element faulty before the corresponding bug is fixed (the source code element had to be modified in that fix) and after the corresponding bug report is present. The source code element can already be faulty before the report and can have multiple changes during that period. Moreover, it can still be faulty after the last fix, but we do not know the issue at that time, which can also distort the measurements. Unfortunately, these uncertainties cannot be solved, since there is no further data to rely on.

### 3.5.2 Threats to Internal Validity

It would be meaningful to use multiple static source code analyzers in order to decrease the threats to internal validity caused by measuring source code element characteristics with only one tool. However, it would mean much more work, and even then, additional manual validations would be needed to decide which tool measures a given metric more precisely, which often depends on interpreting the conceptual definitions.

### 3.5.3 Threats to External Validity

Currently, the constructed dataset consists of 15 projects, which may limit the capabilities of the bug prediction models. Selecting more projects to be included in the dataset would increase the generalizability of the built bug prediction models. Considering additional source hosting platforms (SourceForge, Bitbucket, GitLab) would also increase the external validity of the dataset.

Widely used and accepted programming constructs and structures can vary from programming language to programming language. Using different constructs may have a significant result on the calculated metric values. Selecting projects written in different programming languages, not only Java software systems, could further strengthen the generalizability of our method.

## 3.6 Summary

In this chapter, we presented a method that generates a bug dataset whose entries are source code elements altered by bug fixes mined from GitHub. The entries represent before and after states of source code elements on which bug fixes were applied. The presented approach allows the simultaneous processing of several publicly available projects located on GitHub, thereby resulting in the production of a large – and automatically expandable – dataset. In contrast, previous studies have only dealt with a few large-scale datasets, which were created under strict individual management. Additionally, our dataset contains new source code metrics compared to other datasets, allowing for the examination of the relationship between these metrics and software bugs. Furthermore, manual examinations showed the reliability of our approach, so the adaptation of project-specific labels to the presence of bugs remains the only non-automatic step. Our initial adaptation of 15 suitable Java projects led to the construction of the dataset, which is one of the main – publicly available – contributions of this research.

During empirical evaluations, we showed that the dataset can be used for further investigations, such as bug prediction. For this purpose, we used several machine

learning algorithms at three different granularity levels (method, class, file) from which the method-level prediction achieved the highest F-measure values. As a novel kind of experiment, we also investigated whether the method-level metrics projected to the class level are better predictors than the class-level metrics themselves, and found a significant improvement in the results.

As potential future work, we are planning to expand the dataset with additional projects and even additional data sources, such as SourceForge and Bitbucket. Supporting different external bug tracking systems is another option for extending our approach. We will also dedicate more attention to the concrete prediction models we generate, as this study focused solely on showing the conceptual usability of our dataset.

*"We are what we repeatedly do. Excellence, then, is not an act, but a habit."*

— Aristotle

# 4 Calculation of Process Metrics and their Bug Prediction Capabilities

In Chapter 3, we presented a method for characterizing software bugs with product metrics. It incorporates temporal information into the database by using the buggy source code elements before and after the bug fix occurred, but of course, more sophisticated temporal characteristics could be included.

Studies have shown that process metrics usually perform better in bug prediction than product metrics do. Rahman et al. [101] analyzed the properties of process metrics from the perspective of performance, stability, portability, and stasis. They found that product metrics have a higher stasis - which means they do not change much compared to the process metrics -, thus the same elements were predicted as defective over and over. Also, product metrics are less stable and less portable across projects.

In this chapter, we are going to present a method to compute software process metrics efficiently. After the calculation of the process metrics, we will evaluate their ability to predict bugs and compare them with the product metrics.

The main achievements of this study can be summarized as follows:

- **Research Artifact:** A freely available dataset containing process metrics of buggy Java source code elements (files, classes, methods).
- **Research Question 1:** Is the dataset containing process metrics usable for bug prediction purposes?
- **Research Question 2:** Which metrics are more effective for predicting software bugs: process metrics or product metrics?
- **Research Question 3:** What is the relation between product metrics and process metrics?

## 4.1 Difficulty of Computing Process Metrics

The difficulty of computing process metrics may arise from issues related to storing and processing historical information. A key requirement for computing process metrics is the availability of project history. Versioning systems such as Git or Subversion are commonly used in software development, and their APIs can provide access to project histories. Source code hosting services like GitHub or Bitbucket, which host open-source projects in various programming languages, are also increasingly popular sources of project history.

Another critical criterion for using process metrics effectively is ensuring that developers are using the versioning system correctly. If the system is not used correctly, the information gathered from it may be misleading or not useful. Thus, it is important to consider the accuracy and reliability of the data obtained from the versioning system when analyzing process metrics.

The primary technical challenge we encounter when dealing with process metrics is the sheer size of the dataset. Software projects can consist of hundreds of thousands of lines of source code, with thousands or even tens of thousands of commits. In order to compute process metrics for a particular software version, the entire project history needs to be processed, which necessitates an efficient method for handling such large datasets.

The granularity of process metrics is another aspect that presents challenges. Process metrics can be calculated at different levels, such as file, class, or method. At the file level, the calculations are relatively straightforward as versioning systems operate on files directly, requiring no additional analysis. However, at the class and method levels, a more in-depth source code analysis is necessary to extract the relevant source code elements and their positions. In order to obtain more accurate results, additional information, such as empty lines, comments, and other relevant details, may also need to be gathered during the analysis process. This adds complexity to the computation of process metrics, as it requires careful consideration of the appropriate level of granularity and the necessary source code analysis techniques for accurate results.

Another challenge is determining how to store the collected data in a readily accessible format. In recent years, graph databases have gained popularity due to advancements in their underlying technologies. Graph databases are capable of handling large amounts of data and are well-suited for storing loosely structured data. The historical data of a software package can be represented as a graph, which has led to research on the application of graph databases for assessing software quality [22], particularly in the calculation of software process metrics. Therefore, graph databases appear to be a viable choice for this task, as they provide a promising solution for efficiently storing and retrieving the data required for computing process metrics.

We decided to use Neo4j[1] for data storage. It was the most popular open-source graph database when conducting this study and it continues to hold this position in the present day. It also has a powerful query language called *cypher* that can be utilized to compute process metrics.

[1] `https://neo4j.com/`

## 4.2 Methodology

### 4.2.1 Database Schema

After analyzing the available data sets (project history, static source code analysis results), we designed a graph database schema that is shown in Figure 4.1. Our goal was to construct a graph with a structure that supports the computation of process metrics, so the change information would be easy to obtain. Actually, it contains seven types of nodes. The *Project* node represents the repository of a project. Since we used GitHub, it has two attributes: the GitHub user and repository identifiers. With this node, one database can be used for multiple projects. It may be useful if we are dealing with cross-repository issue referencing, which is also one of the GitHub features. At bigger companies and on GitHub, developers usually contribute to multiple repositories. If we put these repositories into this database, then the developers are connected to multiple projects, hence we can calculate with this feature as well. The *User* node simply represents a developer. It has one attribute, the *number of commits*, which represents the total number of commits made by the user in the system. This property is provided by the GitHub API. The *Issue* node represents a bug that was reported to the issue tracking system and was later fixed. It has two attributes, namely *opened* and *closed*. The former is the date of the bug report, while the latter is the date of closing the bug report or it is null if it is still open. It has an outgoing edge called *Prev* that points to the *Commit* node, which is the latest commit created before the issue was reported.

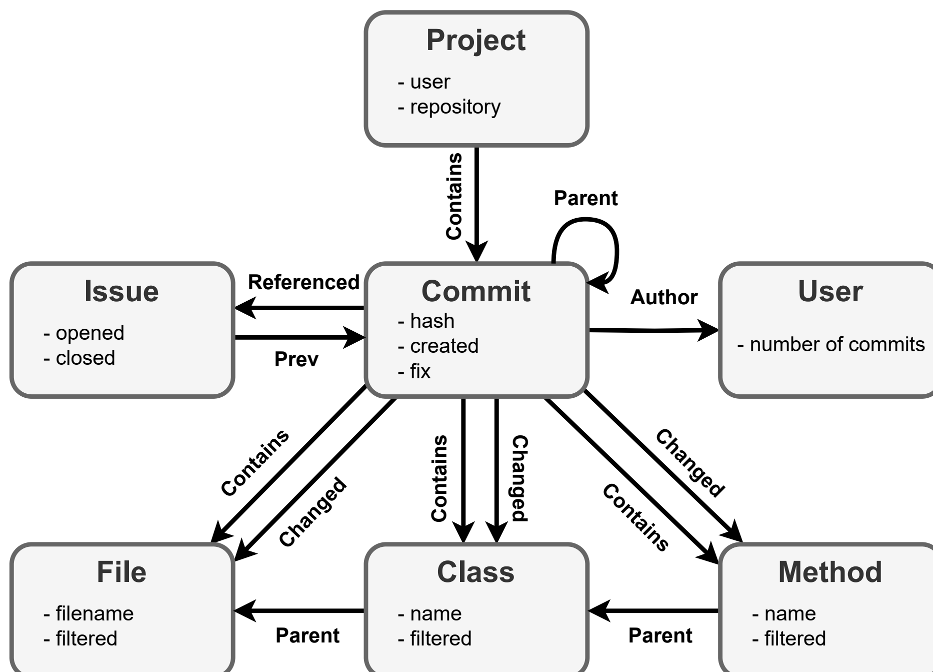


Figure 4.1: The graph database schema

The *Commit* node represents a software version. It has three attributes, these being *hash*, *created*, and *fix*. The first one is the unique hash of the commit. The second is the timestamp of the commit's creation. *Fix* is a Boolean property that tells us whether this commit is a bug fix or not. A commit is treated as a fix if the commit message references a bug report. This connection is provided by the GitHub API and in the

schema, it is represented as a *Referenced* edge between *Commit* and *Issue* nodes. A commit is made by one user, thus we connect the *Commit* nodes to the *User* nodes with an *Author* edge. Sometimes the commits do not have such an edge because the developer is removed from GitHub. The *Parent* edges of *Commit* nodes represent the relationship between consecutive commits. Two commits are connected if one of them is directly followed by the other. One commit may have multiple parents in the case of merge commits.

The bottom three nodes - *File*, *Class*, and *Method* - represent the source code elements. The *Parent* edges between them is the containment relationship. Since we focused on the Java programming language, other containment relations are not possible. *Method* and *Class* nodes have a *name* attribute, which is the fully qualified name of the elements. The *File* nodes always have a *filename* attribute. The *filename* contains the full path of the file. In Chapter 3, in order to avoid marking non-buggy test code as buggy, we filtered the test-related source code elements during the collection of bug information. This filtering is based on the file name and the qualified name. The *filtered* attribute of the *File*, *Class*, and *Method* nodes indicates whether a certain file, class, or method was filtered or not. The values of *name* and *filename* attributes are unique, and this means only one node is created for a given source element that lives across multiple software versions. This way, it is easy to get the changes of a file, class, or method. This is a crucial feature of the database in terms of creating an efficient method. The *Contains* edge between commits and source elements is responsible for showing whether a given commit actually contains the specific element.

The *Changed* edges between commits and the source elements indicate whether an element is changed during a commit. These edges have the following attributes: *added* - number of added lines, *deleted* - number of deleted lines, and *modified* - number of modified lines. These values are computed from the commit patch file, which can also be obtained via the GitHub API. The patch file is based on files, hence additional mapping is required for classes and methods. For this task, we used the source position available from the static source code analysis. A patch file contains sections (deltas). A delta has a beginning line number and an end line number. The mapping is carried out by checking whether a delta intersects the position of a source code element. From the patch file, we can get the beginning and end line numbers of a change (delta) and from the output of the OpenStaticAnalyzer tool, we can get the source position of a source element in the form of row numbers. Now, let us consider a delta with line values 31-46 and a method with position 24-37. The first and last three lines of a delta are unchanged, so we can subtract them from the section. After this step, we get 34-43 as line values for the delta. The intersecting range is 34-37. This means that 4 lines of the method have changed. If we look at the original version of a delta, we can extract information about the type of change. If the size of the original is zero, then the change is an addition. Conversely, if the size of the new part is zero, then it is a deletion. Otherwise, it is a modification.

### 4.2.2 Calculating Process Metrics

Now that we have a schema definition, we can proceed to the metric calculation. Figure 4.2 shows a small part of the graph database created for the ANTLR v4 project (more details in Section 4.3). In this graph, we have at least 1 of each type of node and relationship. As an example, let us demonstrate how to compute a simple process

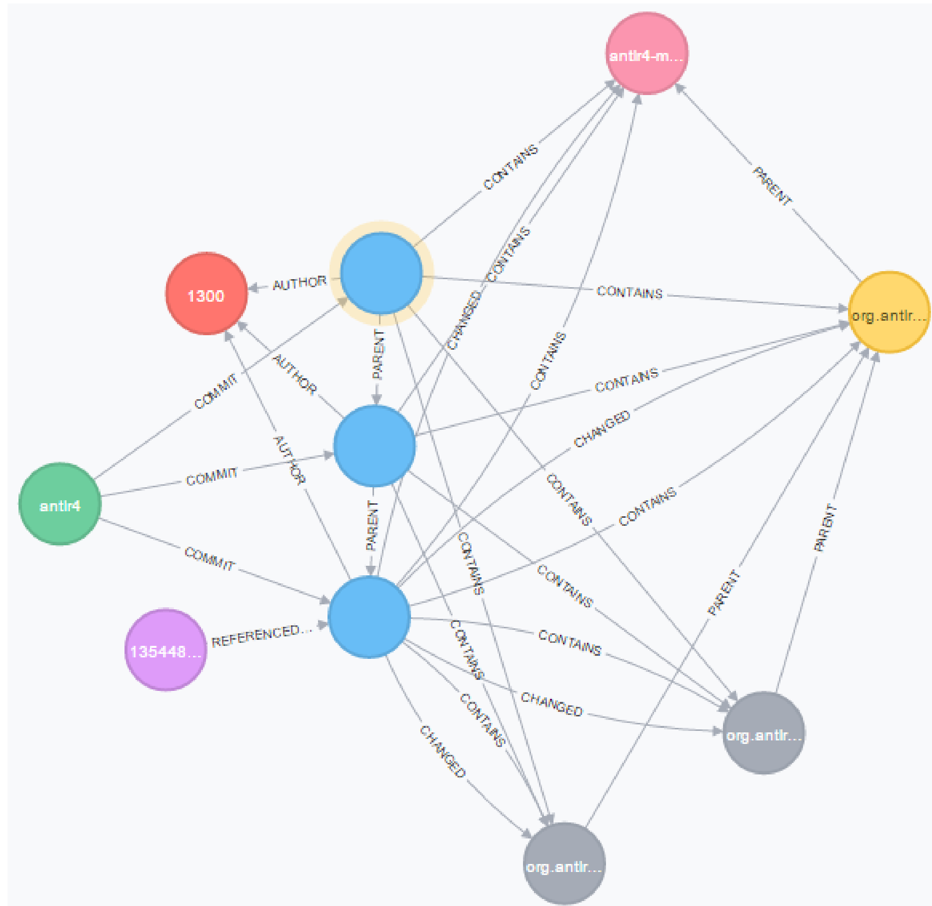


Figure 4.2: Example graph (green: project, purple: issue, red: user, blue: commit, pink: file, yellow: class, grey: method)

metric called the *Number of Modifications* in this graph. The basis of the calculation is the highlighted commit (uppermost) and the source element is the upper right method. We have to look for *Commit* nodes that are created before the subject commit and have a *Changed* relationship with the selected *Method* node. We can use the *Parent* edge between commits, or the *created* attribute for selecting the past commits. Adding the *Changed* edge to the match condition leads to the desired commits, which in this example is the lowermost *Commit* node. With a simple aggregation (counting) we get the value for the computed process metric. As we mentioned earlier, Neo4j has a query language called *cypher*. With this language, it is easy to formulate these process metrics. As an example, the query of the *Number of Modifications* metric is shown in Listing 4.1. We will not go into detail about the syntax of this query language. A

```
1 match
2   (n:METHOD{name:'...'})<-[:CONTAINS]-(c1:COMMIT{hash:'...'}),
3   (n)<-[:CHANGED]-(c2:COMMIT)
4 where
5   c1.created >= c2.created
6 return
7   n.name as name, count(c2) as `Number of Modifications`
```

Listing 4.1: Cypher query for calculating the *Number of Modifications* metric

detailed description is available on the official Neo4j website[2]. The other metrics can be formulated into a single query too, hence this is an easy way to compute them. Table 4.1 lists the implemented process metrics. Switching to the class level is simple, because all we need to do is change the node type in the query.

Table 4.1: The list of implemented process metrics

| **Process Metric** |
|---|
| Age |
| Average Number of Added Lines |
| Average Number of Deleted Lines |
| Average Number of Elements Modified Together |
| Average Time Between Changes |
| Last Contributor Commits |
| Maximum Number of Added Lines |
| Maximum Number of Deleted |
| Maximum Number of Elements Modified Together |
| Number of Additions |
| Number of Contributor Changes |
| Number of Contributors |
| Number of Deletions |
| Number of Fixes |
| Number of Fixed in the Last Six Months |
| Number of Modifications |
| Number of Modifications in the Last Six Months |
| Number of Versions |
| Sum of Added Lines |
| Sum of Deleted Lines |
| Time Passed Since the Last Change |
| Weighted Age |

### 4.2.3 Computing Bug Numbers

In order to calculate the number of bugs from the graph database and build a traditional bug database, first of all, we have to select a software version that we want to build the database from. As we discussed in Chapter 3, an often-used approach is to choose a release version. In this research, we process the whole history of a project, thus we handle more than one version. We assign each bug to a selected version that is the latest one before them (see Figure 3.5). These assignments are represented in the graph by the *prev* edges.

We formulated the calculation into a *cypher* query. The query to compute method-level bug numbers is shown in Listing 4.2. In this query, we match for any non-filtered method node $n$ that is present in the selected commit $c$. We also match for any commit $cc$ that changed the method $n$ and which references an issue $i$ that may or may not have a *prev* edge to the commit *assigned*. In the conditions part of the query, we specify

[2] `https://neo4j.com/developer/cypher/`

```
1 match
2   (n:METHOD{filtered:false})<-[:CONTAINS]-(c:COMMIT{hash:'...'}),
3   (n)<-[:CHANGED]-(cc:COMMIT)-[:REFERENCED]->
4       (i:ISSUE)-[:PREV*0..1]->(assigned:COMMIT)
5 where
6   c.created < cc.created and
7   (assigned = c or i.opened < c.created)
8 return
9   n.name as name, count(DISTINCT i) as 'Number of Bugs'
```

Listing 4.2: Cypher query for calculating method-level bug numbers

that the commit *cc* (the modification) had to come later than commit *c* (the selected version). We also specify that the bug *i* has to be assigned to commit *c* or that it was reported before commit *c*. In the latter case, it implies that issue *i* was closed after commit *c*, because there is a fix (commit *cc*) for this bug after the selected commit. At the end of this query, we return with the name of the source code elements and the associated number of bugs. If we want to compute these values for classes, we can just simply change the type of node *n* to *CLASS*. Hence it is a very general way of calculating bug numbers.

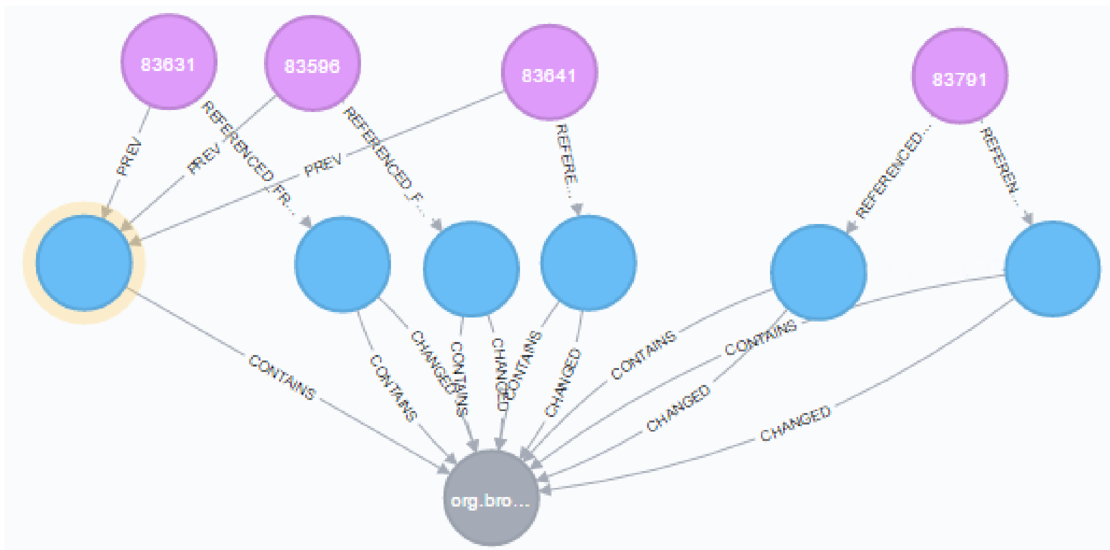


Figure 4.3: Example graph (purple: issue, blue: commit, grey: method)

Figure 4.3 shows a typical graph where we can see the matched nodes for a certain method (grey). The blue node on the left highlighted in yellow is the selected commit that contains the method node. The next five blue nodes are bug fixes that are committed later and changed the method. The first three of these commits reference three issues that are assigned to the selected commit. The last two commits reference an issue that is assigned to another commit, but which was opened before the selected commit. Here, we see that the Number of Bugs for this method in the given version is 4.

### 4.2.4 The Open-source Toolchain

We created a framework to automatically produce these metrics for a specific version of a project. Below, we will describe the overall picture of the framework.

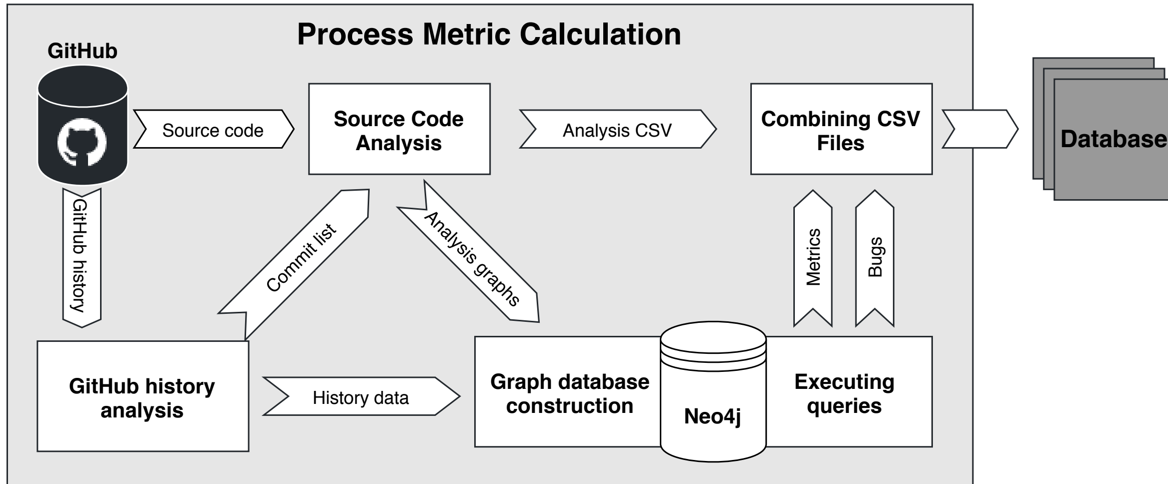


Figure 4.4: Overview of processes involved

An overview of the process is shown in Figure 4.4. The shape in the top left corner represents our data source, GitHub. The two connecting elements are the first two steps: GitHub history analysis and Source code analysis. In these two steps, we collect every bug report of the chosen project from GitHub's bug tracking system and we analyze all versions of the project. The source code analysis is carried out by the OpenStaticAnalyzer tool and the bug reports are exported from GitHub using the GitHub API[3]. Next, the graph nodes and edges - according to the previously presented schema - are exported into CSV files. These files can be directly imported into a Neo4j graph database. The next task is to produce the required process metrics listed in Table 4.1. The selection of the implemented metrics is based on the data available to us. The *cypher* queries are executed and the results are saved in separate CSV files for each metric. At this point, we also run the queries to calculate the number of bugs. The next and final step of the process is to merge the available CSV files so as to produce the desired database. OpenStaticAnalyzer also exports the source elements into CSV files with some essential properties, like qualified name, source position, and static source code metrics.

From these data sets, we create databases that contain the source code elements (files, classes, and methods) along with the static source code metrics, the process metrics, and the number of bugs. Here, these databases are in CSV form. The first line contains the header information - as in the previously published *GitHub Bug Dataset* - extended with the process metric names. The rest of the lines store the data of the source code elements according to the header.

The tools created are published as open-source projects in the following GitHub repository:
`https://github.com/sed-szeged/BugHunterToolchain`

[3] `https://docs.github.com/en/rest`

## 4.3 Experimental Set-up

To demonstrate our method, we processed 5 of the Java projects (see Table 4.2) we used in Chapter 3. During the process, we analyzed every single version of the systems to extract the change information. Although the analysis of individual versions was quick, the overall runtime was quite high. For the Broadleaf Commerce project, the analysis of nearly ten thousand versions took around 200 hours. From a process perspective, it was just an initial step, and only needed to be executed once.

The next step was to build the graph databases. The total runtime for all of the projects was around 6 hours. For a single version, this time is negligible, thus the database can be extended efficiently with the new version. Table 4.2 gives statistics on the size of the graph databases. The first column shows the name of the project. The next two columns are the number of nodes and the number of relationships (edges) in the graph, expressed in thousands. The next column contains the disk space occupied by the graph databases in Megabytes. The final column is the size of the results of source code analysis in Gigabytes. We can see that the graph is a compact way of storing information about project history and that process metrics can be efficiently derived from it.

Table 4.2: Statistics about the graph databases that we created

| **Project** | **kNodes** | **kEdges** | **Size of Graph (MB)** | **Size of Raw Data (GB)** |
|---|---|---|---|---|
| ANTLR v4 | 24 | 13,069 | 484 | 18 |
| Broadleaf C. | 91 | 145,966 | 4,828 | 220 |
| jUnit | 14 | 7,457 | 300 | 9 |
| MapDB | 13 | 4,629 | 208 | 7 |
| Titan | 219 | 21,471 | 804 | 30 |

After setting up the graph database, we computed the process metrics for 25 release versions of the systems (5 each). The release versions were selected just like those in Chapter 3, namely at 6-monthly intervals. Due to smaller inactive periods in project development, it may happen that the bug numbers are zero in a given release version. In such cases, we dropped this release version, therefore, the time interval between some of the versions was larger than six months.

Finally, using the framework, we created bug databases for the selected 5 projects' 25 release versions.

## 4.4 Evaluation

We applied machine learning algorithms to our bug database in order to check whether it was suitable for bug prediction. In the preliminary step, similar to Chapter 3, we grouped the source elements into two classes based on the number of bugs. Source elements with zero bugs formed the non-defective class, while the others formed the defective class. The structure of the learning tables was the following: it contained

a unique id for every instance; next, it contained the predictors (22 software process metrics); lastly, it contained the label of the class as a Boolean value (true - defective, false - non-defective). Separate learning tables were constructed for files, classes, and methods in each release version, hence we got 75 learning tables in total. The number of instances in the defective class was much smaller than the number of instances in the non-defective class. As in Section 3.4, we applied random undersampling to the databases to avoid any distortions in the results. This method helps balance the number of positive and negative instances in the training set. To achieve more reliable results, we applied this method ten times to the data sets and computed an average.

### 4.4.1 Research Question 1

The first research question we will answer is the following:

**Research Question 1:** *Is the dataset containing process metrics usable for bug prediction purposes?*

To answer the question above, we evaluated the 11 algorithms on all 25 release versions and only used process metrics as predictors.

Table 4.3: Average F-measure values at the class level

| **Project** | **#1** | **#2** | **#3** | **#4** | **#5** |
|---|---|---|---|---|---|
| ANTLR v4 | 0.7306 | 0.5380 | 0.6748* | 0.7080 | 0.7680 |
| Broadleaf C. | 0.6688 | 0.6715 | 0.6772* | 0.6732 | 0.6908 |
| MapDB | 0.5640 | 0.6531* | 0.7218 | 0.7259 | 0.7401 |
| jUnit | 0.6800 | 0.6417 | 0.5914 | 0.6143* | 0.8067 |
| Titan | 0.5713 | 0.6154 | 0.6574 | 0.6547 | 0.7386* |

Table 4.4: Average F-measure values at the file level

| **Project** | **#1** | **#2** | **#3** | **#4** | **#5** |
|---|---|---|---|---|---|
| ANTLR v4 | 0.7228 | 0.7487 | 0.6940* | 0.7311 | 0.6837 |
| Broadleaf C. | 0.6372 | 0.6730 | 0.6766* | 0.6951 | 0.6886 |
| MapDB | 0.5651 | 0.7860* | 0.6562 | 0.6783 | 0.8349 |
| jUnit | 0.7261 | 0.6089 | 0.7012 | 0.6805* | 0.5340 |
| Titan | 0.5971 | 0.7132 | 0.7041 | 0.7285 | 0.6968* |

Our first observation was that the F-measure values at the class level generally improved with time except for the jUnit project. Table 4.3 shows the average F-measure values at the class level for all 25 versions. The first column contains the project names, while the next columns are the average results for each version in chronological order. The first is the earliest in time, the following is the next, and so on. The value marked with an asterisk indicates the version with the highest number of bug entries. From this table, we can observe that the best results are achieved

Table 4.5: Average F-measure values at the method level

| Project | #1 | #2 | #3 | #4 | #5 |
|---|---|---|---|---|---|
| ANTLR v4 | 0.5456 | 0.3842 | 0.5142* | 0.5908 | 0.6461 |
| Broadleaf C. | 0.6186 | 0.6547 | 0.6116* | 0.6565 | 0.5643 |
| MapDB | 0.6607 | 0.7328* | 0.6633 | 0.7068 | 0.6713 |
| jUnit | 0.5761 | 0.5378 | 0.5332 | 0.5350* | 0.6413 |
| Titan | 0.4685 | 0.7023 | 0.6369 | 0.5125 | 0.7004* |

using the latest (#5) release versions. The best-performing project is jUnit with an average F-measure value of 0.8067. One possible explanation for this is the nature of the metrics used. Most of the process metrics are based on temporal characteristics, hence, these values may be more reliable with bigger time intervals. At the file and method levels, as shown in Tables 4.4 and 4.5, things are not so straightforward because the values do not follow this trend. The same release version for jUnit achieved the worst overall average F-measure value (0.5340) at the file level, however, relative to the project, it performed the best again at the method level. In Table 4.6, we list the number of bug entries in each release version for each source code level. Based on the results, the performances of the classification algorithms do not seem to correlate with the number of bug entries in a release version.

Table 4.6: Number of bug entries at the file (F), class (C) and method (M) levels

| Project | #1 | | | #2 | | | #3 | | | #4 | | | #5 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | F | C | M | F | C | M | F | C | M | F | C | M | F | C | M |
| ANTLR v4 | 28 | 13 | 21 | 18 | 6 | 5 | 41 | 21 | 32 | 21 | 6 | 8 | 10 | 7 | 10 |
| Broadleaf C. | 91 | 85 | 101 | 199 | 180 | 363 | 286 | 292 | 362 | 88 | 89 | 147 | 46 | 40 | 75 |
| MapDB | 14 | 26 | 64 | 22 | 40 | 89 | 7 | 9 | 12 | 5 | 5 | 9 | 8 | 19 | 24 |
| jUnit | 9 | 7 | 11 | 13 | 10 | 19 | 24 | 22 | 33 | 42 | 35 | 48 | 16 | 13 | 18 |
| Titan | 5 | 8 | 12 | 8 | 12 | 14 | 14 | 18 | 15 | 14 | 14 | 12 | 70 | 96 | 91 |

In Table 4.7, we list the F-measure values at the class level for the versions with the most bug entries. Although the process metrics did not perform the best in these versions, it is still reasonable to use them since we evaluated the *GitHub Bug Dataset* [7] using these versions. The first column contains the name of the machine learning algorithms that we used. The subsequent columns contain the resulting F-measure values for each project. The last column is the average F-measure value over projects. The table is ordered by the average value in decreasing order. Here, we notice that tree- and rule-based algorithms performed the best. The highest average F-measure value is 0.7430, while the lowest is 0.6117. The overall highest F-measure value in these versions is 0.7997 and it was achieved by the Titan project. The best-performing algorithm is the *RandomForest* method.

Table 4.8 shows the resulting F-measures at the file level for the versions with the most bug entries. The structure of the table is the same as that of Table 4.7. Similarly, in this case as well, the same set of algorithms, RandomForest and DecisionTable, perform the best, just like in the case of classes. The best results are over 0.8 (MapDB

Table 4.7: F-measure values at the class level

| **Algorithm** | ANTLR v4 | Broadleaf C. | MapDB | jUnit | Titan | **AVG** |
|---|---|---|---|---|---|---|
| RandomForest | 0.7328 | 0.7638 | 0.7483 | 0.6702 | 0.7997 | 0.7430 |
| DecisionTable | 0.6704 | 0.7147 | 0.6719 | 0.6712 | 0.7742 | 0.7005 |
| OneR | 0.7101 | 0.6982 | 0.6065 | 0.6770 | 0.7597 | 0.6903 |
| SimpleLogistic | 0.6970 | 0.7059 | 0.6163 | 0.6345 | 0.7878 | 0.6883 |
| J48 | 0.6971 | 0.6953 | 0.6618 | 0.6077 | 0.7567 | 0.6837 |
| SGD | 0.6196 | 0.7039 | 0.6838 | 0.6125 | 0.7832 | 0.6806 |
| RandomTree | 0.6530 | 0.6861 | 0.6547 | 0.6365 | 0.7540 | 0.6769 |
| Logistic | 0.5827 | 0.6885 | 0.6459 | 0.6006 | 0.7441 | 0.6523 |
| NaiveBayes | 0.7106 | 0.5846 | 0.6868 | 0.6457 | 0.5672 | 0.6390 |
| NaiveBayesMultinomial | 0.7065 | 0.5920 | 0.6042 | 0.5349 | 0.6689 | 0.6213 |
| VotedPerceptron | 0.6432 | 0.6159 | 0.6040 | 0.4663 | 0.7290 | 0.6117 |

Table 4.8: F-measure values at the file level

| **Algorithm** | ANTLR v4 | Broadleaf C. | MapDB | jUnit | Titan | **AVG** |
|---|---|---|---|---|---|---|
| RandomForest | 0.7804 | 0.7328 | 0.8180 | 0.6659 | 0.7271 | 0.7448 |
| DecisionTable | 0.7299 | 0.7152 | 0.8143 | 0.7061 | 0.7103 | 0.7352 |
| SimpleLogistic | 0.7234 | 0.6910 | 0.7869 | 0.7071 | 0.7094 | 0.7236 |
| SGD | 0.7033 | 0.7025 | 0.8036 | 0.6857 | 0.7159 | 0.7222 |
| NaiveBayesMultinomial | 0.6920 | 0.6342 | 0.8152 | 0.6732 | 0.7533 | 0.7136 |
| OneR | 0.6911 | 0.6642 | 0.7786 | 0.7371 | 0.6781 | 0.7098 |
| J48 | 0.6553 | 0.6994 | 0.7983 | 0.6614 | 0.7304 | 0.7090 |
| RandomTree | 0.6826 | 0.6740 | 0.7261 | 0.6933 | 0.6896 | 0.6931 |
| VotedPerceptron | 0.6267 | 0.6340 | 0.8019 | 0.6149 | 0.7406 | 0.6836 |
| Logistic | 0.6765 | 0.6950 | 0.6699 | 0.6649 | 0.6821 | 0.6777 |
| NaiveBayes | 0.6732 | 0.6008 | 0.8334 | 0.6757 | 0.5278 | 0.6622 |

project) and the worst result is 0.5278, however, on average, the F-measure values are around 0.7. The highest average F-measure value is 0.7448, while the lowest is 0.6622.

Table 4.9 displays the resulting F-measures at the method level for the versions with the highest number of bug entries. Overall, the F-measure values obtained are lower compared to those for classes and files. The RandomForest algorithm remains the best-performing algorithm in this context. It is worth noting that in some cases, very low F-measure values (<0.5) were observed, indicating a decrease in the predictive power of the process metrics at the method level. This suggests that the change metrics of a method do not directly correlate with the presence of a bug, unlike in the case of classes where it is more accurate. There could be several explanations for this observation. Firstly, changes in methods are often made in conjunction with other methods, and it is rare for only a single method to be independently modified. Consequently, if a method is not frequently edited, it is likely that other methods in the same class also undergo minimal changes, resulting in similar process metrics. Furthermore, it is more common to modify a single class alone, leading to greater variation in change patterns at the class level.

Table 4.9: F-measure values at the method level

| **Algorithm** | ANTLR v4 | Broadleaf C. | MapDB | jUnit | Titan | **AVG** |
|---|---|---|---|---|---|---|
| RandomForest | 0.5788 | 0.6617 | 0.8185 | 0.6390 | 0.7705 | 0.6937 |
| RandomTree | 0.5565 | 0.6286 | 0.7882 | 0.5804 | 0.7243 | 0.6556 |
| J48 | 0.5501 | 0.6180 | 0.7661 | 0.5655 | 0.7118 | 0.6423 |
| SimpleLogistic | 0.4595 | 0.6455 | 0.7890 | 0.4957 | 0.7416 | 0.6263 |
| SGD | 0.4584 | 0.6354 | 0.7823 | 0.4917 | 0.7519 | 0.6239 |
| Logistic | 0.4911 | 0.6289 | 0.7572 | 0.5319 | 0.7053 | 0.6229 |
| DecisionTable | 0.4618 | 0.6376 | 0.7515 | 0.5787 | 0.6773 | 0.6214 |
| OneR | 0.5233 | 0.5684 | 0.6935 | 0.5327 | 0.6690 | 0.5974 |
| VotedPerceptron | 0.4956 | 0.5654 | 0.6855 | 0.4655 | 0.7052 | 0.5834 |
| NaiveBayes | 0.5143 | 0.6010 | 0.6248 | 0.5127 | 0.6365 | 0.5779 |
| NaiveBayesMultinomial | 0.5670 | 0.5369 | 0.6046 | 0.4908 | 0.6107 | 0.5620 |

In summary, while change patterns at a smaller level of source code elements may not reliably indicate the presence of bugs, change patterns at a higher level, such as classes or files, could provide better indications of bug-proneness. Additionally, the Bayesian methods perform poorly overall, and their performance varies across different versions. Therefore, it can be concluded that using Bayesian algorithms is not the optimal approach for bug prediction.

**Answering Research Question 1:** *The results obtained suggest that databases with process metrics are suitable for bug prediction purposes, with the RandomForest and DecisionTable methods performing the best. Specifically, we achieved F-measure values of 0.7997 (Titan) at the class level, 0.8180 (MapDB) at the file level, and 0.8185 (MapDB) at the method level using the RandomForest algorithm. Furthermore, it is important to note that not all projects are equally capable of providing an appropriate training set for bug prediction. Additionally, on average, process metrics provide stronger indications of bug-proneness at higher levels of source code, such as classes or files.*

### 4.4.2 Research Question 2

The second research question we will answer is the following:

**Research Question 2:** *Which metrics are more effective for predicting software bugs: process metrics or product metrics?*

Table 4.10 allows for a comparison of the average F-measure values obtained from two different sets of predictors for the release versions with the highest number of bug entries. The higher value of the two is highlighted in bold. Analyzing these values, it can be observed that, at the class level, process metrics performed worse than product metrics in 4 out of 5 cases. However, at the file level, process metrics outperformed product metrics in 3 out of 5 cases. At the method level, product metrics performed better in 3 out of 5 cases. Despite the underperformance of process metrics in certain cases, the resulting F-measure values suggest that process metrics show

greater robustness compared to product metrics. Table 4.11 presents the statistical characteristics of the obtained F-measure values. Notably, the standard deviation for process metrics is generally lower, except for one instance at the class level for jUnit. The average difference in standard deviation is 0.0375, with a minimum difference of 0.0073, and a maximum difference of 0.0933. Additionally, the minimum F-measure values tend to be higher for process metrics in all cases, except for one instance at the method level for jUnit. The average difference in minimum values is 0.0809, with a maximum difference of 0.2685. The presence of 0 among the minimum F-measure values is due to an insufficient number of buggy entries in one of the release versions of the project. It is worth noting that the maximum F-measure values tend to be lower for process metrics in the majority of cases. The average difference in maximum values is 0.0577, with a maximum difference of 0.1305. Furthermore, the best-performing algorithms differ for each version when using product metrics, while the process metrics consistently yield a similar set of top-performing algorithms across all cases.

Table 4.10: F-measure values for the release versions with the highest number of bug entries

| | **File** | | **Class** | | **Method** | |
|---|---|---|---|---|---|---|
| | *Product* | *Process* | *Product* | *Process* | *Product* | *Process* |
| ANTLR4 | **0.7035** | 0.6940 | **0.7129** | 0.6748 | **0.7198** | 0.5142 |
| Broadleaf C. | **0.6926** | 0.6766 | **0.7470** | 0.6772 | **0.7830** | 0.6116 |
| MapDB | 0.6420 | **0.7860** | **0.6939** | 0.6531 | 0.6709 | **0.7328** |
| jUnit | 0.5647 | **0.6805** | **0.7164** | 0.6143 | **0.5824** | 0.5350 |
| Titan | 0.5759 | **0.6968** | 0.6971 | **0.7386** | 0.6252 | **0.7004** |

Table 4.11: Statistical characteristics of the F-measure values

| | | **ANTLR4** | | **Broadleaf C.** | | **MapDB** | | **jUnit** | | **Titan** | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | *Product* | *Process* | *Product* | *Process* | *Product* | *Process* | *Product* | *Process* | *Product* | *Process* |
| **File** | *Std. dev.* | 0.1419 | 0.0486 | 0.1133 | 0.0423 | 0.1744 | 0.1472 | 0.1092 | 0.0952 | 0.1946 | 0.1241 |
| | *Min* | 0.3333 | 0.6018 | 0.3333 | 0.5459 | 0.0000 | 0.0000 | 0.3333 | 0.4066 | 0.0000 | 0.0000 |
| | *Max* | 0.8796 | 0.8305 | 0.7937 | 0.7423 | 0.8999 | 0.8895 | 0.7810 | 0.8162 | 0.9021 | 0.8493 |
| | *Avg.* | 0.7061 | 0.7161 | 0.6533 | 0.6741 | 0.6846 | 0.7041 | 0.5967 | 0.6501 | 0.5607 | 0.6879 |
| **Class** | *Std. dev.* | 0.1349 | 0.1110 | 0.1040 | 0.0436 | 0.1696 | 0.1369 | 0.0880 | 0.1016 | 0.1284 | 0.0846 |
| | *Min* | 0.2473 | 0.3312 | 0.3555 | 0.5802 | 0.0000 | 0.0000 | 0.3442 | 0.4663 | 0.3483 | 0.4239 |
| | *Max* | 0.8903 | 0.8711 | 0.8242 | 0.7638 | 0.9630 | 0.9598 | 0.7838 | 0.8842 | 0.8391 | 0.7997 |
| | *Avg.* | 0.6679 | 0.6839 | 0.6768 | 0.6763 | 0.7084 | 0.6810 | 0.6880 | 0.6668 | 0.6382 | 0.6475 |
| **Method** | *Std. dev.* | 0.1505 | 0.1395 | 0.0766 | 0.0601 | 0.1042 | 0.0969 | 0.1150 | 0.0855 | 0.1678 | 0.1196 |
| | *Min* | 0.0000 | 0.0000 | 0.4857 | 0.4924 | 0.4053 | 0.4056 | 0.4414 | 0.4090 | 0.2570 | 0.3706 |
| | *Max* | 0.8568 | 0.7430 | 0.8012 | 0.7266 | 0.8582 | 0.8661 | 0.8842 | 0.7678 | 0.9037 | 0.7732 |
| | *Avg.* | 0.6219 | 0.5362 | 0.6723 | 0.6211 | 0.6752 | 0.6870 | 0.6669 | 0.5647 | 0.6242 | 0.6041 |

Upon examining the average F-measure values, it is observable that process metrics surpassed product metrics at the file level in all cases. At the class level, the results were

mixed, with product metrics performing better in two cases, process metrics performing better in two cases, and the average F-measure values being similar in one case. At the method level, product metrics achieved higher average values in 4 out of 5 cases. These findings indicate that process metrics are more effective predictors for higher levels of source code when compared to product metrics.

**Answering Research Question 2:** *Based on the obtained results, it can be concluded that creating bug prediction models based on process metrics at the file level generally yields better performance compared to those using product metrics. However, at lower levels of source code, such as the class or method level, product metrics often outperform process metrics. An interesting observation is that the standard deviation of the resulting F-measure values is consistently lower for process metrics in almost all cases, with an average difference of 0.0375. Furthermore, the overall lowest achieved F-measure values are higher for process metrics in almost all cases, with an average difference of 0.0809. These findings suggest that bug prediction results obtained from process metrics are more robust and reliable, indicating their suitability for predicting bugs in source code.*

### 4.4.3 Research Question 3

The last research question we will answer is the following:

**Research Question 3:** *What is the relation between product metrics and process metrics?*

We computed the Pearson correlation coefficient values between product and process metrics for class-level and method-level metrics in our databases. As the results are similar across all versions, we will only present the correlations for one version to avoid lengthy listings. Instead of including the correlation matrices that have over a hundred rows and columns, we illustrate these with colored tables (Figures 4.5, and 4.6). The black cells denote the low absolute value of the correlation (close to zero), while the white cells denote the high absolute value of the correlation (near one or minus one).

Figure 4.5 shows the correlation of metrics at the method level. The product metrics (including rule violation metrics) are separated from the process metrics by a red line. From this image, we can see that the process metrics (bottom right quarter) correlate more with each other than with product metrics. There are some higher correlation values around 0.4-0.5 between a few size-based product metrics (e.g. Lines of Code, Number of Statements) and process metrics (e.g. Number of Added Lines, Number of Modifications), but in general, there are no dominant correlation values between product and process metrics.

If we look at the correlation results between any two class-level metrics in Figure 4.6, we notice that the relation between process and product metrics is a little stronger. Nonetheless, the correlation between metrics of the same type remains notably higher. At the file level, we cannot draw any conclusions from the database, as it only contains a few product metrics.

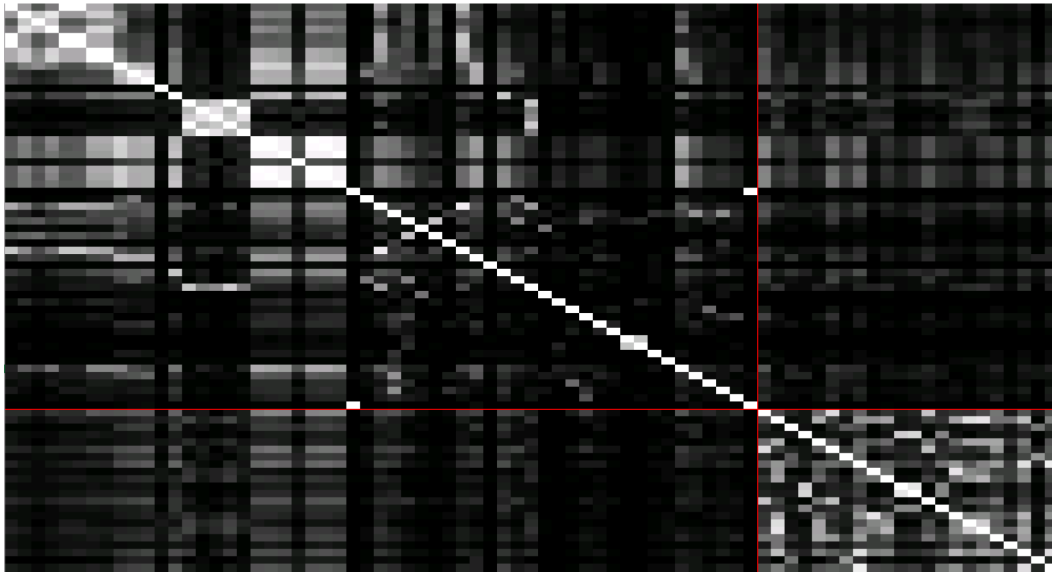

Figure 4.5: Correlation of method-level metrics

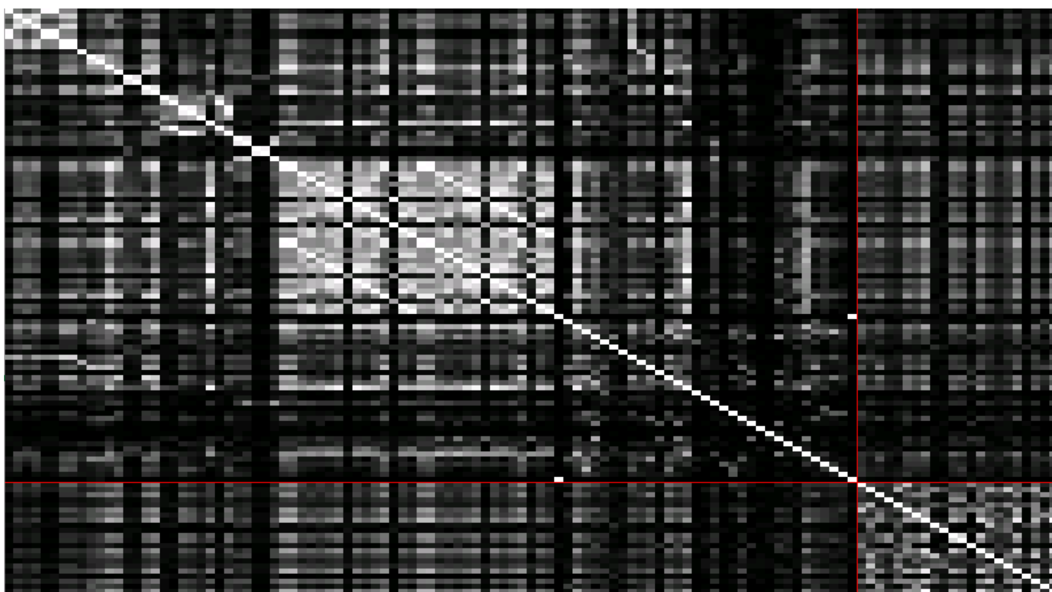

Figure 4.6: Correlation of class-level metrics

**Answering Research Question 3:** *From the correlation results presented above, it can be observed that process metrics and product metrics are of a different nature. Process metrics provide a distinct perspective in characterizing the source code elements compared to product metrics, as there are no strong correlations between the two types of metrics.*

Due to the size of the learning tables and correlation matrices, we have not included all the evaluation tables here. Please refer to the online appendix, which contains all the analysis results in spreadsheet files:
`https://www.inf.u-szeged.hu/~pgyimesi/papers/ActaCybernetica2016/`

## 4.5 Summary

In this chapter, we presented a method that efficiently computes software process metrics in a graph database. Also, we made an implementation available on GitHub as an open-source project that is capable of computing 22 process metrics. We selected 5 Java projects and produced databases for 25 release versions. The databases created contain the implemented process metrics for files, classes, and methods along with the number of bugs. Afterward, we applied 11 machine learning algorithms to them to investigate whether the database is suitable for bug prediction purposes. Based on the F-measure values, we found that tree- and rule-based methods perform the best and, in particular, the *RandomForest* method performed well in every case.

In the future, we intend to implement more process metrics and experiment with new ones. We also plan to extend the list of processed systems.

In addition to the study, our method was also implemented in the QualityGate[4] service, although the implementation differs slightly. QualityGate is a service that continuously measures and monitors the quality of the source code, as well as the performance of the development team. Source code is analyzed on-site without ever leaving the local network. In such an environment, our method had to be integrated directly into the analysis pipeline. The implementation had to be highly resource-efficient, which meant that process metrics had to be calculated quickly and using very little memory. Due to the continuous measurements, we had to adapt our method to calculate the metrics incrementally from version to version. As a result of this integration, process metrics are calculated continuously for more than 450 open-source and closed-source projects. The metrics of the open-source projects are freely accessible through a REST API. This makes it the largest dataset available that contains process metrics for various source code elements, such as classes, methods, annotations, and so on. QualityGate currently does not process bug tracking information, thus the number of bugs could not be calculated. In the future, we plan to use this huge dataset to further investigate the properties of the process metrics.

[4] `https://www.quality-gate.com/`

*“Knowing there is a structure, hidden or felt, to the random gives pleasure.”*

— Cecil Balmond

# 5 A Public Dataset of JavaScript Bugs

Despite all the JavaScript-related research, to date, a well-organized repository of labeled JavaScript bugs is still missing. The plethora of different JavaScript implementations available (*e.g.*, V8, JavaScriptCore, Rhino) further makes devising a cohesive bugs benchmark nontrivial. To fill this gap, we present BugsJS, a benchmark of JavaScript-related bugs from 10 open-source JavaScript projects, based on Node.js and the Mocha testing framework.

To emphasize the research artifact contributions and the research questions, we list them here:

- **Research Artifact 1:** BugsJS, a benchmark of 453 manually selected and validated JavaScript bugs from 10 JavaScript Node.js programs pertaining to the Mocha testing framework.
- **Research Artifact 2:** A Docker-based framework to download, analyze, and run test cases exposing each bug in BugsJS and the corresponding real fixes implemented by developers. The infrastructure includes a set of precomputed data from the subjects and tests as well.
- **Research Artifact 3:** A bug taxonomy of server-side JavaScript bugs in BugsJS, which, to our knowledge, is the first of its kind.
- **Research Question 1:** Do the bug-fixing patterns for JavaScript bugs in BugsJS match existing classification schemes?
- **Research Question 2:** How do the bug-fixing patterns in BugsJS relate to our taxonomy of bugs?

## 5.1 Methodology

To construct a benchmark of real JavaScript bugs, we identify existing bugs from the programs’ version control histories and collect the real fixes provided by developers.

Developers often manually label the revisions of the programs in which reported bugs are fixed (*bug-fixing commits*, or patches). As such, we refer to the revision preceding the bug-fixing commit as the *buggy commit*. This allowed us to extract detailed bug reports and descriptions, along with the buggy and bug-fixing commits they refer to. Particularly, each bug and fix should adhere to the following properties:

- **Reproducibility:** One or more *test cases* are available in a *buggy commit* to demonstrate the bug. The bug must be reproducible under reasonable constraints. We excluded non-deterministic features and flaky tests from our study, since replicating them in a controlled environment would be excessively challenging.
- **Isolation:** The bug-fixing commit applies to JavaScript source code files only; changes to other artifacts, such as documentation or configuration files, are not considered. The source code of each commit must be *cleaned* from irrelevant changes (*e.g.*, feature implementations, refactorings, changes to non-JavaScript files). The *isolation* property is particularly important in research areas where the presence of noise in the data has detrimental impacts on the techniques (*e.g.*, automated program repair, or fault localization approaches).

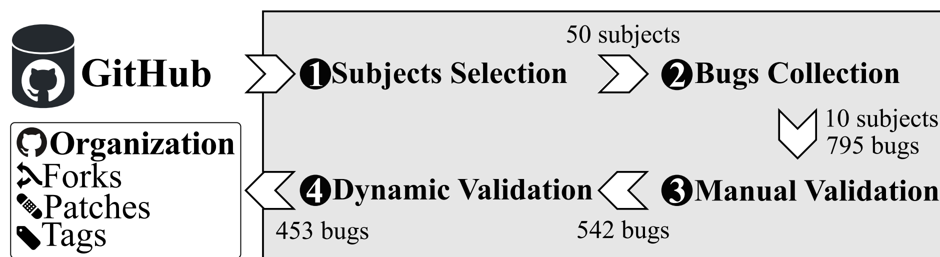


Figure 5.1: Overview of the bug selection and inclusion process

Figure 5.1 depicts the main steps of the process we performed to construct our benchmark. First, we adopted a systematic procedure to select the JavaScript subjects to extract the bug information from ❶. Then, we collected bug candidates from the selected projects ❷ and manually validated each bug for inclusion by means of multiple criteria ❸. Next, we performed a dynamic sanity check to make sure that the tests introduced in a bug-fixing commit can detect the bug in the absence of its fix ❹. Finally, the retained bugs were cleaned from irrelevant patches (*e.g.*, whitespaces).

### 5.1.1 Subject Systems Selection

To select relevant programs to include in BugsJS, we focused on popular and trending JavaScript projects on GitHub. Such projects often engage large communities of developers, and therefore, are more likely to follow software development best practices, including bug reporting and tracking. Moreover, GitHub's *issue IDs* allow conveniently connecting bug reports to bug-fixing commits.

Popularity was measured using the projects' *Stargazers count* (*i.e.*, the number of stars owned by the subject's GitHub repository). We selected server-side Node.js applications that are popular (Stargazers count $\geq 100$) and mature (number of commits $> 200$), and have been actively maintained (year of the latest commit $\geq 2017$). We

currently focus on Node.js because it is emerging as one of the most pervasive technologies to enable using JavaScript on the server side, leading to the so-called full-stack web applications [13]. Limiting the subject systems to server-side applications and specific testing frameworks is due to technological constraints, as running tests for browser-based programs would require managing many complex and time-consuming configurations. We discuss the potential implications of this constraint in Section 5.4.

We examined the GitHub repository of each retrieved subject system to ensure that bugs were properly tracked and labeled. Particularly, we only selected projects in which bug reports had a dedicated *issue label* on GitHub's *Issues* page, which allows filtering irrelevant issues (pertaining to, *e.g.*, feature requests, build problems, or documentation), so that only *actual bugs* are included. Our initial list of subjects included 50 Node.js programs, from which we filtered out projects based on the number of candidate bugs found and the adopted testing frameworks.

### 5.1.2 Bugs Collection

#### Collecting bugs and bug-fixing commits

For each subject system, we first queried GitHub for *closed issues* assigned with a specific bug label using the official GitHub API.[1] For each closed bug, we exploit the *links* existing between issues and commits to identify the corresponding bug-fixing commit. GitHub automatically detects these links when there is a specific keyword (belonging to a predefined list[2]), followed by an issue ID (*e.g.*, `Fixes #14`).

Each issue can be linked to zero, one, or more source code commits. A closed bug without a bug-fixing commit could mean that the bug was rejected (*e.g.*, it cannot be replicated), or that developers did not associate that issue with any commit. We discarded such bugs from our benchmark, as we require each bug to be identifiable by its bug-fixing commit. At last, similarly to existing benchmarks [69], we discarded bugs linked to more than one bug-fixing commit, as this might imply that they were fixed in multiple steps, or that the first attempt for fixing them was unsuccessful.

#### Including corresponding tests

We require each fixed bug to have *unit tests* that demonstrate the absence of the bug. To meet this requirement, we examined the bug-fixing patches to ensure they also contain changes or additions in the test files. For this filtering, we manually examined each patch to determine whether test files were involved. The result of this step is the list of *bug candidates* for the benchmark. From the initial list of 50 subject systems, we considered the projects having at least 10 bug candidates.

#### Testing frameworks

There are several testing frameworks available for JavaScript applications. We collected statistics about the testing frameworks used by the 50 considered JavaScript projects. Our results show that there is no single predominant testing framework for JavaScript (as compared to, for instance, JUnit which is used by most Java developers). We found

[1] `https://docs.github.com/en/rest`

[2] `https://docs.github.com/en/issues/tracking-your-work-with-issues/linking-a-pull-request-to-an-issue#linking-a-pull-request-to-an-issue-using-a-keyword`

Table 5.1: Bug-fixing commit inclusion criteria

| Rule Name | Description |
|---|---|
| Isolation | The bug-fixing changes must fix only one (1) bug (i.e., must close exactly one (1) issue) |
| Complexity | The bug-fixing changes should involve a limited number of files ($\leq$ 3), lines of code ($\leq$ 50) and be understandable within a reasonable amount of time (max 5 minutes) |
| Dependency | If a fix involves introducing a new dependency (e.g., a library), there must also exist production code changes and new test cases added in the same commit |
| Relevant Changes | The bug-fixing changes must involve only changes in the production code that aim at fixing the bug (whitespace and comments are allowed) |
| Refactoring | The bug-fixing changes must not involve the refactoring of the production code |

that the majority of tests in our pool were developed using Mocha[3] (52%), Jasmine[4] (10%), and QUnit[5] (8%). Consequently, the initial version of BugsJS only includes projects that use Mocha, whose prevalence as a JavaScript testing framework is also supported by a recent large-scale empirical study [48].

**Final Selection**

Table 5.2 reports the names and descriptive statistics of the 10 applications we ultimately retained. Notice that all these applications have at least 1000 LOC (frameworks excluded), thus being representative of modern web applications (Ocariza et al. [94] report an average of 1,689 LOC for AngularJS web applications on GitHub with at least 50 stars).

Table 5.3 lists the number of tests of the subject projects, and the source code coverage achieved by running the test suite. A test is considered *Passing* if it runs without an issue, and it is considered *Failing* if it throws any error. A *Pending* test means that it was skipped by the testing framework due to some criteria. The coverage is measured at *Statement*, *Branch*, *Function*, and *Line* levels.

The subjects represent a wide range of domains. Bower is a front-end package management tool that exposes the package dependency model through an API. Express is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications. Hessian.js is a JavaScript binary web service protocol that makes web services usable without requiring a large framework, and without learning a new set of protocols. Hexo is a blog framework powered by Node.js. Karma is a popular framework agnostic test runner tool for JavaScript. Mongoose is a MongoDB object modeling tool for Node.js. Node-redis is a Node.js

[3] `https://mochajs.org/`
[4] `https://jasmine.github.io/`
[5] `https://qunitjs.com/`

Table 5.2: Subjects included in BugsJS

| Program | | Stats | | | |
|---|---|---|---|---|---|
| **Name** | **Description** | **kLOC (JS)** | **Stars** | **Commits** | **Forks** |
| Bower[6] | package manager | 16 | 15,290 | 2,706 | 1,995 |
| Eslint[7] | linting tool | 240 | 12,434 | 6,615 | 2,141 |
| Express[8] | web framework | 11 | 40,407 | 5,500 | 7,055 |
| Hessian.js[9] | serialization service | 6 | 104 | 217 | 23 |
| Hexo[10] | blog framework | 17 | 23,748 | 2,545 | 3,277 |
| Karma[11] | test runner | 12 | 10,210 | 2,485 | 1,531 |
| Mongoose[12] | ODM | 65 | 17,036 | 9,770 | 2,457 |
| Node-redis[13] | database client | 11 | 10,349 | 1,242 | 1,245 |
| Pencilblue[14] | CMS | 46 | 1,596 | 3,675 | 276 |
| Shields[15] | badge service | 20 | 6,319 | 2,036 | 1,432 |

client for the Redis database. Pencilblue is a CMS and blogging platform, powered by Node.js. Shields is a web service for badges in SVG and raster format.

### 5.1.3 Manual Patch Validation

In this study, two participants manually investigated each bug and its corresponding bug-fixing commit and labeled them based on a well-defined set of *inclusion criteria* (Table 5.1). The bugs meeting all the criteria were initially marked as "Candidate Bugs" for further consideration.

To ensure relatedness to the fixed bug, the participants investigated the code of the commit simultaneously. However, some bug-fixing commits were too complex for investigators to comprehend due to domain knowledge requirements or a large number of modified files or lines of code. In such cases, the bug-fixing commits were labeled as "Too complex" and discarded from BugsJS. This was done to maintain the size of the patches within reasonable thresholds, ensuring a high-quality corpus of bugs that can be easily processed and analyzed through both manual inspection and automated techniques. A commit was deemed too complex if it involved changes in more than three files or more than 50 LOC, or if understanding the fix required more than 5 minutes. If the participants unanimously agreed that a fix was too complex, the case was ignored.

Another reason for exclusion was refactoring operations in the analyzed code. The participants aimed to preserve the original behavior of the code as written by developers. Therefore, only modifications that did not affect the program's behavior, such as whitespaces, were restored. It is often challenging to decouple refactoring from bug fixing due to the requirement for in-depth domain knowledge, especially in JavaScript, which is a dynamic language. Refactoring can also affect multiple parts of a project, affecting metrics such as code coverage and making it more challenging to restore the

Table 5.3: Statistics about the projects included in BugsJS

| Program | Tests (#) | | | | Coverage (%) | | | |
|---|---|---|---|---|---|---|---|---|
| **Name** | **All** | **Passing** | **Pending** | **Failing** | **Satements** | **Branches** | **Functions** | **Lines** |
| Bower | 455 | 103 | 19 | 36 | 81.11 | 66.91 | 80.62 | 81.11 |
| Eslint | 18,528 | 18,474 | 0 | 54 | 99.21 | 98.19 | 99.72 | 99.21 |
| Express | 855 | 855 | 0 | 0 | 98.71 | 94.32 | 100 | 99.95 |
| Hessian.js | 225 | 223 | 2 | 0 | 96.42 | 91.27 | 98.99 | 96.42 |
| Hexo | 875 | 868 | 7 | 0 | 96.20 | 90.51 | 98.54 | 97.27 |
| Karma | 331 | 331 | 0 | 0 | 54.61 | 34.03 | 43.98 | 54.76 |
| Mongoose | 2,107 | 2,071 | 36 | 0 | 90.97 | 85.95 | 89.65 | 91.04 |
| Node-redis | 966 | 965 | 0 | 1 | 99.06 | 98.19 | 97.99 | 99.06 |
| Pencilblue | 807 | 802 | 0 | 5 | 35.21 | 19.09 | 22.91 | 35.22 |
| Shields | 482 | 469 | 13 | 0 | 75.98 | 65.60 | 83.26 | 75.97 |

original code changes.

We manually validated a total of 795 commits, out of which 542 (68.18 %) met the criteria. The result of this step for each application and across all applications is shown in Table 5.4 (Manual). Excluding bugs due to fixes being deemed too complex was the most common reason (136). Other common scenarios were bug-fixing commits addressing more than one bug (32), fixes not involving production code (29), and containing refactoring operations (39). We also came across four instances in which the patch did not involve the actual test's source code, but rather comments or configuration files.

### 5.1.4 Dynamic Validation

To ensure that the test cases introduced in a bug-fixing commit were actually intended to test the buggy feature, we adopted a systematic and automatic approach. Let $V_{bug}$ be the version of the source code that contains a bug $b$, and let $V_{fix}$ be the version in which $b$ is fixed. The existing test cases in $V_{bug}$ do not fail due to $b$. However, at least one test of $V_{fix}$ should fail when executed on $V_{bug}$. This allows us to identify the test in $V_{fix}$ used to demonstrate $b$ (*isolation*) and to discard cases in which tests immaterial to the considered buggy feature were introduced.

To run the tests, we obtained the dependencies and set up the environment for each specific revision of the source code. Over time, however, developers made major changes

[6] `https://github.com/bower/bower`
[7] `https://github.com/eslint/eslint`
[8] `https://github.com/expressjs/express`
[9] `https://github.com/node-modules/hessian.js`
[10] `https://github.com/hexojs/hexo`
[11] `https://github.com/karma-runner/karma`
[12] `https://github.com/Automattic/mongoose`
[13] `https://github.com/redis/node-redis`
[14] `https://github.com/pencilblue/pencilblue`
[15] `https://github.com/badges/shields`

Table 5.4: Manual and dynamic validation statistics per application for all considered commits

| | | Bower | Eslint | Express | Hessian.js | Hexo | Karma | Mongoose | Node-redis | Pencilblue | Shields | **Total** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | *Initial number of bugs* | 10 | 559 | 39 | 17 | 24 | 37 | 56 | 25 | 18 | 10 | **795** |
| Manual | ✘ Fixes multiple issues | 0 | 18 | 1 | 0 | 1 | 5 | 2 | 5 | 0 | 0 | 32 |
| | ✘ Too complex | 0 | 94 | 0 | 4 | 8 | 4 | 8 | 7 | 9 | 2 | 136 |
| | ✘ Only dependency | 1 | 9 | 0 | 0 | 1 | 0 | 2 | 0 | 0 | 0 | 13 |
| | ✘ No production code | 0 | 20 | 4 | 0 | 1 | 1 | 2 | 0 | 0 | 1 | 29 |
| | ✘ No tests changed | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 4 |
| | ✘ Refactoring | 0 | 36 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 39 |
| | *After manual validation* | 8 | 382 | 33 | 13 | 13 | 26 | 41 | 11 | 8 | 7 | **542** |
| Dynamic | ✘ Test does not fail at $V_{bug}$ | 1 | 11 | 6 | 4 | 1 | 2 | 8 | 3 | 1 | 3 | 40 |
| | ✘ Dependency missing | 3 | 17 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 22 |
| | ✘ Error in tests | 1 | 7 | 0 | 0 | 0 | 0 | 3 | 1 | 0 | 0 | 12 |
| | ✘ Not Mocha | 0 | 14 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 15 |
| | ✔ ***Final Number Of Bugs*** | 3 | 333 | 27 | 9 | 12 | 22 | 29 | 7 | 7 | 4 | **453** |

to some of the projects' structure and environment, making test replication infeasible. These cases occurred, for instance, when older versions of required dependencies were no longer available, or when developers migrated to a different testing framework (*e.g.*, from QUnit to Mocha).

For the projects that used scripts (*e.g.*, `grunt`, `bash`, `Makefile`) to run their tests, we extracted them, so as to isolate each test's execution and avoid possible undesirable side effects caused by running the complete test suite.

After the dynamic analysis, 453 bug candidates were ultimately retained for inclusion in BugsJS (84% of the 542 bug candidates from the previous step).

Table 5.4 (Dynamic) reports the results of the dynamic validation phase. In 22 cases, we were unable to run the tests because dependencies were removed from the repositories. In 15 cases, the project at revision $V_{bug}$ did not use Mocha for testing $b$. In 12 cases, tests were failing during the execution, whereas in 40 cases no tests failed when executed on $V_{bug}$. We excluded all such bug candidates from the benchmark.

### 5.1.5 Patch Creation

We performed *manual cleaning* on the bug-fixing patches, to make sure they only include changes related to bug fixes. In particular, we removed *irrelevant files* (*e.g.*, `*.md`, `.gitignore`, `LICENSE`), and *irrelevant changes* (*i.e.*, source code comments, when only comments changed, and comments unrelated to bug-fixing code changes, as well as changes solely pertaining to whitespaces, tabs, or newlines). Furthermore, for easier analysis, we separated the patches into two separate files, the first one including the modifications to the tests, and the second one pertaining to the production code fixes.

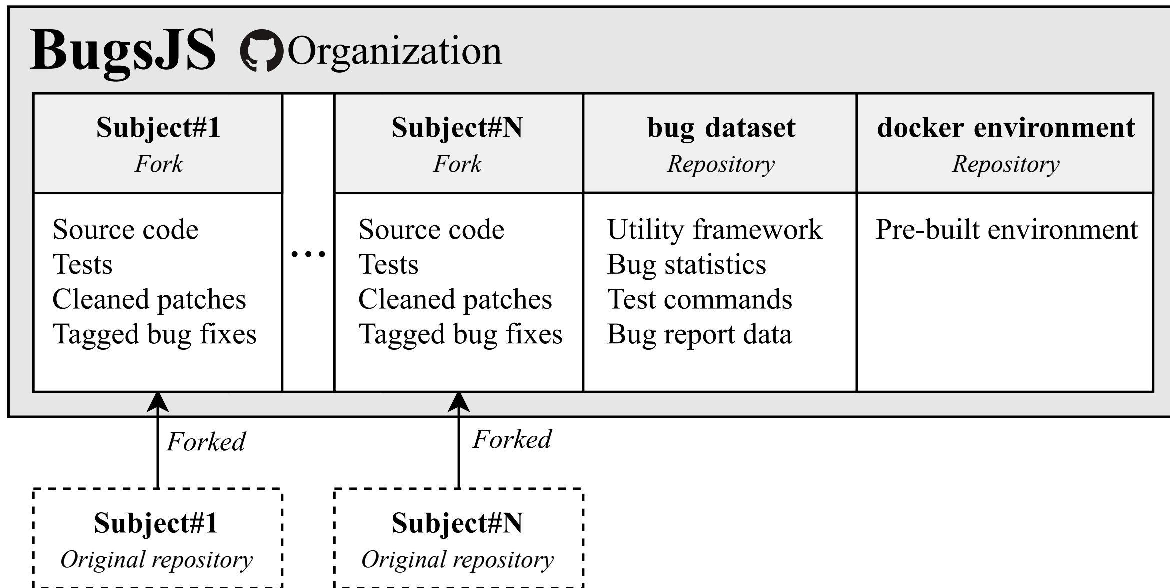


Figure 5.2: Overview of BugsJS architecture

### 5.1.6 Benchmark Infrastructure and Implementation

#### Infrastructure

Figure 5.2 illustrates the overall architecture of BugsJS, which supports common activities related to the benchmark, such as running the tests at each revision, or checking out specific commits. The framework's command-line interface includes the following commands:

- `info`: Prints out information about a given bug.
- `checkout`: Checks-out the source code for a given bug.
- `test`: Runs all tests for a given bug and measures the test coverage.
- `per-test`: Runs each test individually and measures the per-test coverage for a given bug.

For the `checkout`, `test`, and `per-test` commands, the user can specify the desired code revision: *buggy*, *buggy with the test modifications applied*, or the *fixed version*. BugsJS is equipped with a pre-built environment that includes the necessary configurations for each project to execute correctly. This environment is available as a Docker image along with a detailed step-by-step tutorial. The interested reader can find more information on BugsJS and access the benchmark on our website:
`https://bugsjs.github.io/`

#### Source code commits and tests

We used GitHub's *fork* functionality to make a full copy of the *git* history of the subject systems. The unique identifier of each commit (*i.e.*, the commit `SHA1` hashes) remains intact when forking. In this way, we were able to synchronize the copied fork with the original repository and keep it up-to-date. Importantly, our benchmark will not be lost if the original repositories get deleted.

The fork is a separate *git* repository, therefore, we can push commits to it. Taking advantage of this possibility, we have extended the repositories with additional commits, to separate the bug-fixing commits and their corresponding tests. To make such commits easily identifiable, we tagged them using the following notation (`X` denotes a sequential bug identifier):

- `Bug-X`: The parent commit of the revision in which the bug was fixed (*i.e.*, the buggy revision);
- `Bug-X-original:` A revision with the original bug-fixing changes (including the production code and the newly added tests);
- `Bug-X-test:` A revision containing only the tests introduced in the bug-fixing commit, applied to the buggy revision;
- `Bug-X-fix:` A revision containing only the production code changes introduced to fix the bug, applied to the buggy revision;
- `Bug-X-full:` A revision containing both the cleaned fix and the newly added tests, applied to the buggy revision.

### Test runner commands

For each project, we have included the necessary test runner commands in a CSV file. Each row of the file corresponds to a bug in the benchmark and specifies:

1. A sequential bug identifier;
2. The test runner command required to run the tests;
3. The test runner command required to produce the test coverage results;
4. The Node.js version required for the project at the specific revision where the bug was fixed, so that the tests can execute properly;
5. The preparatory test runner commands (*e.g.*, to initialize the environment to run the tests, which we call `pre-commands`);
6. The cleaning test runner commands (*e.g.*, the teardown commands, which we call `post-commands`) to restore the application's state.

### Bug report data

Forking repositories does not maintain the issue data associated with the original repository. Thus, the links appearing in the commit messages of the forked repository still refer to the original issues. In order to preserve the bug reports, we obtained them via GitHub's API and stored them in Google's Protocol Buffers[16] format. Particularly, for each bug report, we store the original issue identifier paired with our sequential bug identifier, the text of the bug description, the issue open and close dates, the `SHA1` of the original bug-fixing commit along with the commit date and commit author identifiers. Lastly, we save the comments from the issues' discussions.

[16] `https://developers.google.com/protocol-buffers/`

In the following paragraphs, we list several artifacts that we added to BugsJS in the form of *precomputed data*.[17] These can be reproduced by performing suitable static and dynamic analyses on the benchmark, however, we supply this information to better facilitate further bug-related research, including bug prediction, fault localization, automatic repair, etc.

#### Test coverage data

We included pre-computed information in BugsJS. We used the tool Istanbul[18] to compute per-test coverage data for `Bug-X` and `Bug-X-test` versions of each bug, and the results are available in JSON format. Particularly, for each project, we included information about the tests of the `Bug-X` versions in a separate CSV file. Each row in such file contains the following information:

1. A sequential bug identifier;
2. Total LOC in the source code, as well as LOC covered by the tests;
3. The number of functions in the source code, as well as the number of functions covered by the tests;
4. The number of branches in the source code, as well as the number of branches covered by the tests;
5. The total number of tests in the test suite, along with the number of passing, failing, and pending tests (*i.e.*, the tests that were skipped due to execution problems).

#### Static source code metrics

Furthermore, to support studies based on source code metrics, we run static analysis on `Bug-X-full` and `Bug-X` versions of each bug. For the static analysis, we used the tool OpenStaticAnalyzer, which calculates 41 static source code metrics for JavaScript. The results are available in a zip file named `metrics`.

### 5.1.7 Extending BugsJS

BugsJS was designed and implemented in a way that is easy to extend with new JavaScript projects, however, there are some restrictions. The current version of the framework only supports projects that are in a *git* repository and use the Mocha testing framework. If the project is hosted on GitHub, it can be also forked under BugsJS's GitHub Organization to preserve the state of the repository.

Mining an appropriate bug to add to BugsJS takes four steps as described at the beginning of Section 5.1. It is mainly manual work, but some of it could be done programmatically. In our case, we used GitHub as the source of bug reports, which has a public API, thus we could automate the bug collection step. Validating the bug-fixing patches requires manual work, but it can be partially supported by automation, e.g., for filtering patches that modify both the production code and the

[17] `https://github.com/BugsJS/bug-dataset`
[18] `https://istanbul.js.org/`

tests. However, in our experience during the development of BugsJS, the location of test files varies across different projects, and sometimes across versions as well. Thus, it is still challenging to automatically determine it for an arbitrary set of JavaScript projects. Dynamic validation also requires some manual effort. Despite the fact that we limited the support only to the most common testing framework (Mocha), running the tests programmatically is not so trivial because the command that runs the test suite is, in some cases, assembled at run-time (e.g., with `grunt` or `Makefile`) and can change over time. Extracting it programmatically is only possible for standard cases, e.g., when it is located in the default `package.json` file. Due to the great variety of JavaScript projects, this process can hardly be automated, as compared to other languages like Java, where project build systems are more homogeneous.

After a suitable bug is found, some preparatory steps are required before a bug can be added to BugsJS. If necessary, irrelevant white space and comment modifications can be removed from the bug-fix patch, which is re-added to the repository as a new commit. Next, the production code modifications should be separated from the test modifications by committing the changes separately on top of the buggy version. Then, the commits should be tagged according to the notation described in Section 5.1.6. Finally, the new bug can be submitted to BugsJS using a GitHub *pull request*. The pull request has to contain the modified or added CSV files that contain the repository URL, the test runner command, and any additional commands (pre and post) if any. Adding precomputed data is not mandatory, but beneficial.

## 5.2 Taxonomy of Bugs in BugsJS

In this section, we present a detailed overview of the root causes behind the bugs in our benchmark. Hanam et al. [59] identified 13 cross-project bug patterns in JavaScript projects, and we tried to assign all 453 bugs from BugsJS to one of these categories. Our analysis revealed that there were 42 occurrences of the categories proposed by Hanam et al., with the majority falling under the category of *Dereferenced non-values*. This demonstrates that these patterns do exist in the bugs present in BugsJS, although they only cover a small subset. The majority of the remaining bugs are logical errors made by developers during implementation and do not necessarily fall under recurring patterns. Therefore, BugsJS includes bugs that are diverse in nature, making it an ideal platform to evaluate a wide range of analysis and testing techniques. Our taxonomy proved to be more appropriate for categorizing such logical errors in BugsJS. However, this came at a cost, as our categories are more high-level and independent of the language and the domain of the subject systems.

We constructed our taxonomy using faceted classification [115], i.e., we created the categories/subcategories of our taxonomy in a bottom-up fashion, by analyzing different sources of information about the bugs.

### 5.2.1 Manual Labeling of Bugs

Each bug and associated information (i.e., bug report and issue description) was manually analyzed by four authors (referred to as “taggers” hereafter) following an open coding procedure [108]. Four taggers specified a descriptive label for each bug assigned to them. The labeling task was performed independently, and the disagreements were

discussed and resolved through dedicated meetings. Unclear cases were also discussed and resolved during such meetings.

First, we performed a pilot study, in which all taggers reviewed and labeled a sample of 10 bugs. Bugs for the pilot were selected randomly from all projects in BugsJS. The consensus on the procedure and the final labels was high, therefore, for the subsequent rounds, the four taggers were split into two pairs, that were shuffled after each round of tagging.

The labels were collected in separate spreadsheets; the agreement on the final labels was found by discussion. During the tagging, the taggers could reuse existing labels previously created, should an existing label apply to the bug under analysis. This choice was meant to limit introducing nearly-similar labels for the same bug and help taggers use consistent naming conventions.

When inspecting the bugs, we looked at several sources of information, namely (1) the bug-fixing commit on GitHub's web interface containing the commit title, the description as well as at the code changes, and (2) the entire issue and pull request discussions.

In order to achieve internal validation in the labeling task, we performed cross-validation. Specifically, we created an initial version of the taxonomy labeling around 80% of the bugs (353). Then, to validate the initial taxonomy, the remaining 20% (100) were simply assigned to the closest category in the initial taxonomy, or a new category was created, when appropriate. Bugs for the initial taxonomy were selected at random, but they were uniformly selected among all subjects, to avoid over-fitting the taxonomy towards a specific project. Analogously, the validation set was retained so as to make sure all projects were represented. Internal validation of the initial taxonomy is achieved if few or no more categories (i.e., labels) were needed for categorizing the validation bugs. The labeling process involved four rounds: the first round (the pilot study) involved labeling 10 bugs, the second round 43 bugs, and 150 bugs were analyzed in both the third and fourth rounds.

### 5.2.2 Taxonomy Construction

After enumerating all causes of bugs in BugsJS, we began the process of creating a taxonomy, following a systematic process. During a physical meeting, for each bug instance, all taggers reviewed the bugs and identified candidate equivalence classes to which descriptive labels were assigned. By following a bottom-up approach, we first clustered tags that correspond to similar notions into categories. Then, we created parent categories in such a way that categories and their subcategories follow specialization relationships.

### 5.2.3 Taxonomy Internal Validation

We performed the validation phase in a physical meeting. Each of the four taggers classified one-fourth of the validation set (25 bugs) independently, assigning each of them to the most appropriate category. After this task, all taggers reviewed and discussed the unclear cases to reach a full consensus. All 100 validation bugs were assigned to existing categories, and no further categories were needed.

### 5.2.4 The Final Taxonomy

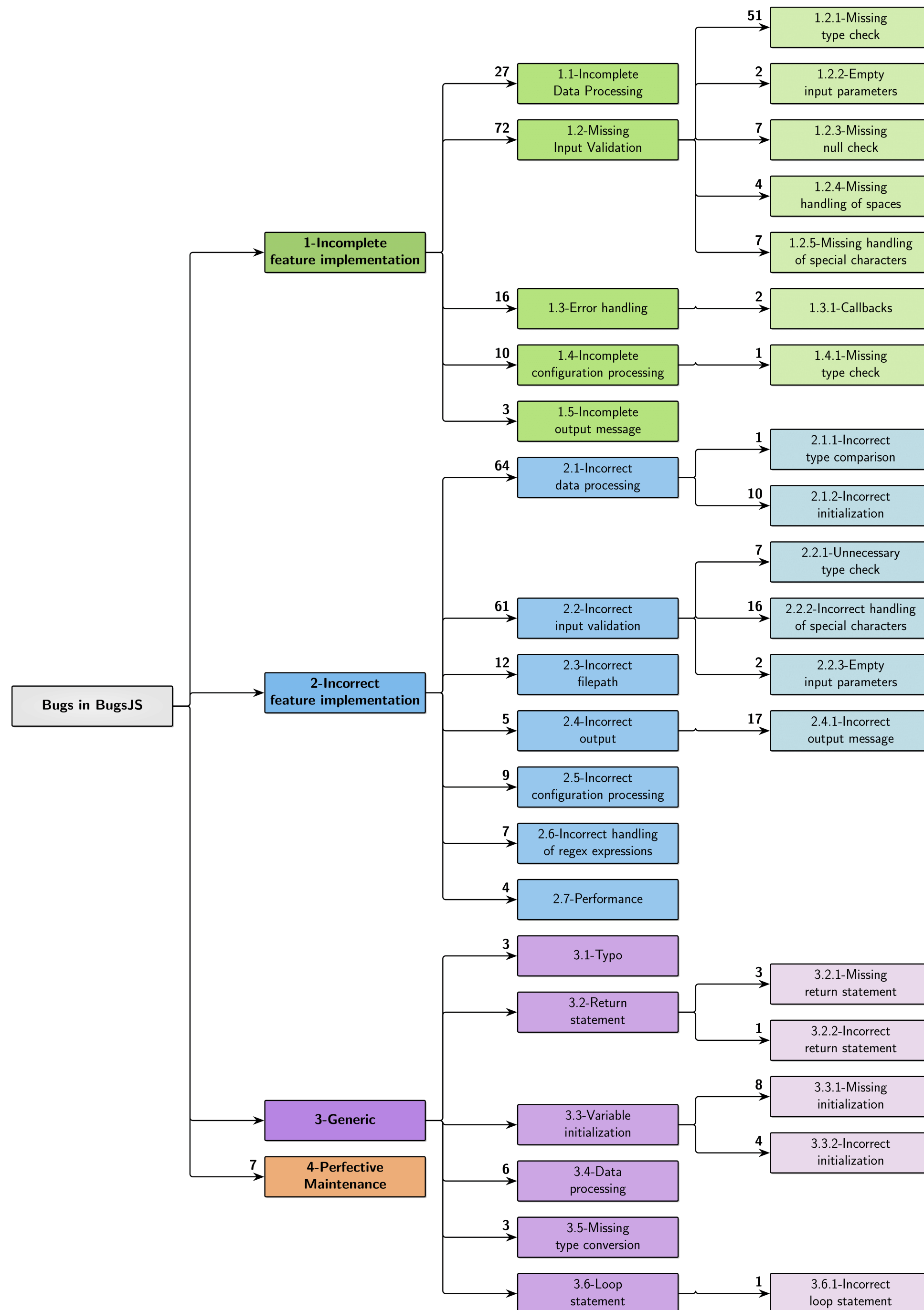


Figure 5.3: Taxonomy of bugs in the benchmark of JavaScript programs of BugsJS.

Figure 5.3 presents a graphical view of our taxonomy of bugs in the JavaScript benchmark. Nodes represent classes and subclasses of bugs, and edges represent specialization relationships. Specializations are not complete, disjoint relationships. Even though during labeling we tried assigning the most specific category, we found out during taxonomy creation that we had to group together many app-specific corner cases. Thus, some bugs pertaining to inner nodes were not further specialized to avoid creating an excessive number of leaf nodes with only a few corner cases.

At the highest level, we identified four distinct categories for the causes of bugs, as

follows:

1. Causes related to ***incomplete feature implementation***. These bugs are related either to an incomplete understanding of the main functionalities of the considered application, or a refinement in the requirements. In these cases, the functionalities have already been implemented by developers according to their best knowledge, but over time, users or other developers found out that they do not consider all aspects of the corresponding requirements. More precisely, given a requirement $r$, the developer implemented a program feature $f'$ which corresponds to only a subset $r' \subset r$ of the intended functionality. Thus, the developer has to adapt the existing functionality $f' \subset f$ to $f$, in order to satisfy the requirement in $r$. Typical instances of this bug category are related to one or more specific corner cases that were unpredictable at the time in which that feature was initially created, or when the requirements for the main functionalities are changed or extended to some extent.

2. Causes related to ***incorrect feature implementation***. These bugs are also related to the mainstream functionalities of the application. Differently from the previous category, the bugs in this category are related to wrong implementation by the developers, for instance, due to an incorrect interpretation of the requirements. More precisely, suppose that given a requirement $r$, the developer implemented a program feature $f'$, to the best of her knowledge. Over time, other developers found out by the usage of the program that the behavior of $f'$ does not reflect the intended behavior described in $r$, and opened a dedicated issue in the GitHub repository (and, eventually, a pull request with a first fix attempt).

3. Causes related to ***generic*** programming errors. Bugs belonging to this category are typically not related to an incomplete/incorrect understanding of the requirements by developers, but rather to common coding errors, which are also important from the point of view of a taxonomy of bugs.

4. Causes related to ***perfective maintenance***. Perfective maintenance involves making functional enhancements to the program in addition to the activities to increase its performance even when the changes have not been suggested by bugs. These can include completely new requirements to the functionalities or improvements to other internal or external quality attributes not affecting existing functionalities. When composing BugsJS, we aimed at excluding such cases from the candidate bugs (see Section 5.1), however, the bugs that we classified in the taxonomy with this category were labeled as bugs by the original developers, so we decided to retain them in the benchmark.

We now discuss each of these categories in turn, in each case considering the subcategories beneath them.

## 1 - Incomplete feature implementation

This category contains 45% of the bugs overall and has five subcategories, which we describe in the following subsection.

### 1.1 - Incomplete data processing

The bugs in this category are related to an incomplete implementation of a feature's logic, i.e., the way in which the input is consumed and transformed into output.

Overall, 27 bugs were found to be of this type. An example is `Bug#7` of Hexo,[19] in which an HTML anchor is undefined, unless correct escaping of markdown characters is used.

```
1 − var text = $(this).html();
2 + var text = _.escape($(this).text());
```

### 1.2 - Missing input validation

The bugs in this category are related to incomplete input validation, i.e., the way in which the program checks whether a given input is valid and can be processed further.

Overall, 16% of the bugs were found to be of this type, and a further 16% in more specialized instances. This prevalence was mostly due to the nature of some of our programs. For instance, ESLint provides linting utilities for JavaScript code, and it is the most represented project in BugsJS (73%). Therefore, being its main scope to actually validate code, we found many cases related to invalid inputs being unmanaged by the library, even though we found instances of these bugs also in other projects. For instance, in `Bug#4` of Karma,[20] a file parsing operation should not be triggered on URLs having no line number. As such, in the bug-fixing commit, the proposed fix adds one more condition.

```
1 − if (file && file.sourceMap) {
2 + if (file && file.sourceMap && line) {
```

Another prevalent category is due to missing type checks on inputs (11%), whereas less frequent categories were missing checks of null inputs, empty parameters, and missing handling of spaces or other special characters (e.g., in URLs).

### 1.3 - Error handling

The bugs in this category are related to incomplete handling of errors, i.e., the way in which the program manages erroneous cases, i.e., exception handling.

Overall, 3% of the bugs were found to be of this type. For instance, in `Bug#14` of Karma,[21] the program does not throw an error when using a plugin for a browser that is not installed, which is a corner case missed in the initial implementation. Additionally, we found two cases specific to callbacks.

### 1.4 - Incomplete configuration processing

The bugs in this category are related to an incomplete configuration, i.e., the values of parameters accepted by the program.

Overall, 2% of the bugs were found to be of this type. For instance, in `Bug#10` of ESLint,[22] an invalid configuration is used when applying extensions to the default configuration object. The bug fix updates the default configuration object's constructor

[19] https://github.com/BugsJS/hexo/releases/tag/Bug-7-original
[20] https://github.com/BugsJS/karma/releases/tag/Bug-4-original
[21] https://github.com/BugsJS/karma/releases/tag/Bug-14-original
[22] https://github.com/BugsJS/eslint/releases/tag/Bug-10-original

to use the correct context, and to make sure the config cache exists when the default configuration is evaluated.

## 1.5 - Incomplete output message

The last subcategory pertains to bugs related to incomplete output messages by the program.

Only three bugs were found to be of this type. For instance, in `Bug#8` of Hessian.js,[23] the program casts the values exceeding `Number.MAX_SAFE_INTEGER` as string, to allow safe readings of large floating point values.

## 2 - Incorrect feature implementation

This category contains 48% of the bugs overall and has seven subcategories, which we describe in the following subsections.

## 2.1 - Incorrect data processing

The bugs in this category are related to the wrong implementation of a feature's logic, i.e., the way in which the input is consumed and transformed into output.

Overall, 75 bugs were found to be of this type, with two subcategories due to a wrong type comparison (1 bug), or an incorrect initialization (10 bugs). An example of this latter category is `Bug#238` of ESLint,[24] in which developers remove the default parser from CLIEngine options to fix a parsing error.

```
1 - parser: DEFAULT_PARSER
2 + parser: ""
```

## 2.2 - Incorrect input validation

The bugs in this category are related to the wrong input validation, i.e., the way in which the program checks whether a given input is valid, and can be processed further.

Overall, 19% of the bugs were found to be of this type, with three subcategories due to unnecessary type checks (7 bugs), incorrect handling of special characters (16 bugs), or empty input parameters given to the program (2 bugs). As an example of this latter category, in `Bug#171` of ESLint,[25] the `arrow-spacing` rule did not check for all spaces between the arrow character (`=>`) within a given code. Therefore, it is updated as follows:

```
1 - while (t.type !== "Punctuator" || t.value !== "=>") {
2 + while (arrow.value !== "=>") {
```

## 2.3 - Incorrect filepath

The bugs in this category are related to wrong paths to external resources necessary to the program, such as files. For instance, in `Bug#6` of ESLint,[26] developers failed to

[23] https://github.com/BugsJS/hessian.js/releases/tag/Bug-8-original
[24] https://github.com/BugsJS/eslint/releases/tag/Bug-238-original
[25] https://github.com/BugsJS/eslint/releases/tag/Bug-171-original
[26] https://github.com/BugsJS/eslint/releases/tag/Bug-6-original

check for configuration files within sub-directories. Therefore, the code was updated as follows:

```
1 - if (!directory)
2 + if (directory) directory = path.resolve(this.cwd, directory);
```

## 2.4 - Incorrect output

The bugs in this category are related to incorrect output by the program. For instance, in Bug#7 of Karma,[27] the exit code is wrongly replaced by null characters (\0x00), which results in squares (□□□□□□) being displayed in the standard output.

```
1 - return exitCode
2 + return {exitCode: exitCode, buffer: buffer.slice(0, tailPos)}
```

## 2.5 - Incorrect configuration processing

The bugs in this category are related to an incorrect configuration of the program, i.e., the values of parameters accepted.

Nine bugs were found to be of this type. For instance, in Bug#145 of ESLint,[28] a regression was accidentally introduced where parsers would get passed additional unwanted default options even when the user did not specify them. The fix updates the default parser options to prevent any unexpected options from getting passed to parsers.

```
1 - let parserOptions = Object.assign({}, defaultConfig.parserOptions);
2 + let parserOptions = {};
```

## 2.6 - Incorrect handling of regex expressions

The bugs in this category are related to an incorrect use of regular expressions.

Seven bugs were found to be of this type. For instance, in Bug#244 of ESLint,[29] a regular expression is wrongly used to check that the function name starts with `setTimeout`.

## 2.7 - Performance

The bugs in this category caused the program to use an excessive amount of resources (e.g., memory). Only four bugs were found to be of this type. For instance, in Bug#85 of ESLint,[30] a regular expression susceptible to catastrophic backtracking was used. The match takes quadratic time in the length of the last line of the file, causing Node.js to hang when the last line of the file contains more than 30,000 characters. Another representative example is Bug#1 of Node-Redis,[31] in which parsing big JSON files takes substantial time due to an inefficient caching mechanism that makes the parsing time grow exponentially with the size of the file.

[27] https://github.com/BugsJS/karma/releases/tag/Bug-7-original
[28] https://github.com/BugsJS/eslint/releases/tag/Bug-145-original
[29] https://github.com/BugsJS/eslint/releases/tag/Bug-244-original
[30] https://github.com/BugsJS/eslint/releases/tag/Bug-85-original
[31] https://github.com/BugsJS/node_redis/releases/tag/Bug-1-original

## 3 - Generic

This category contains 6% of the bugs overall and has six subcategories, which we describe next.

### 3.1 - Typo

This category refers to typographical errors by the developers.

We found three such bugs in our benchmark. For instance, in Bug#321 of ESLint,[32] a rule is intended to compare the *start* line of a statement with the end line of the previous token. Due to a typo, it was comparing the *end* line of the statement instead, which caused false positives for multiline statements.

### 3.2 - Return statement

The bugs in this category are related to either missing return statements (3 bugs), or incorrect usage of return statements (1 bug). For instance, in Bug#8 of Mongoose,[33] the fix involves adding an explicit return statement.

```
1 - this.constructor.update.apply(this.constructor, args);
2 + return this.constructor.update.apply(this.constructor, args);
```

### 3.3 - Variable initialization

The bugs in this category are related to either missing initialization of variables statements (8 bugs), or to the incorrect initialization of variables (4 bugs). For instance, in Bug#9 of Express,[34] the fix involves correcting a wrongly initialized variable.

```
1 -  mount_app.mountpath = path;
2 +  mount_app.mountpath = mount_path;
```

### 3.4 - Data processing

The bugs in this category are related to the incorrect processing of information.

Six bugs were found to be of this type. For instance, in Bug#184 of ESLint,[35] developers fixed the possibility of passing negative values to the string.slice function.

```
1 - currentText.slice(node.range[0] - (beforeCount || 0)
2 + currentText.slice(Math.max(node.range[0] - (beforeCount || 0), 0)
```

### 3.5 - Missing type conversion

The bugs in this category are related to missing type conversions.

Three bugs were found to be of this type. For instance, in Bug#4 of Shields,[36] developers forgot to convert labels to strings prior to applying the uppercase transformation.

```
1 - data.text[0] = data.text[0].toUpperCase();
2 + data.text[0] = ('' + data.text[0]).toUpperCase();
```

[32] https://github.com/BugsJS/eslint/releases/tag/Bug-321-original
[33] https://github.com/BugsJS/mongoose/releases/tag/Bug-8-original
[34] https://github.com/BugsJS/express/releases/tag/Bug-9-original
[35] https://github.com/BugsJS/eslint/releases/tag/Bug-184-original
[36] https://github.com/BugsJS/shields/releases/tag/Bug-4-original

### 3.6 - Loop statement

We found only one bug of this type—Bug#304 of SHIELDS,[37]—related to the incorrect usage of loop statements.

```
1  - while ((currentAncestor = currentAncestor.parent))
2  -  if (isConditionalTestExpression(currentAncestor)) {
3  -    return currentAncestor.parent;
4  -  }
5  - }
6
7  + do {
8  +  if (isConditionalTestExpression(currentAncestor)) {
9  +    return currentAncestor.parent;
10 +  }
11 + } while ((currentAncestor = currentAncestor.parent));
```

### 4 - Perfective maintenance

This category contains only 1% of the bugs. For instance, in Bug#209 of ESLINT,[38] developers fix JUnit parsing errors which treat no test cases having an empty output message as a failure.

## 5.3 Evaluation

To gain a better understanding of the characteristics of bug-fixes of bugs included in BUGSJS, we have performed two analyses to quantitatively and qualitatively assess the representativeness of our benchmark. This serves as an addition to the taxonomy presented in Section 5.2 which, by connecting the bug types to the bug-fix types, can support applications, such as automated fault localization and automated bug repair.

### 5.3.1 Research Question 1

The first research question we will answer is the following:

> **Research Question 1:** *Do the bug-fixing patterns for JavaScript bugs in BUGSJS match existing classification schemes?*

To observe the occurrence of recurring low-level bug-fixing patterns in BUGSJS, we conducted further analysis. Similar studies [96, 131, 29] in the past have explored patterns in bug-fixing changes within Java programs, suggesting that the existence of such patterns reveals certain code constructs (such as if conditionals) that could indicate weak points in the source code, where developers are consistently more likely to introduce bugs [96].

All 453 bug-fixing commits in BUGSJS were manually investigated by four participants of this study, who attempted to assign the bug-fixing changes to the predefined categories proposed in previous studies. The categories suggested by Pan et al. [96] were used for this purpose, but they were originally related to Java bug fixes. The aim of this study is to assess whether these categories generalize to JavaScript or whether

[37] https://github.com/BugsJS/eslint/releases/tag/Bug-304-original
[38] https://github.com/BugsJS/eslint/releases/tag/Bug-209-original

Table 5.5: Bug-fixing change types found in BugsJS

| | Category | Example | # |
|---|---|---|---|
| Existing | `if`-related | Changing `if` conditions | 291 |
| | Assignments | Modifying the RHS of an assignment | 166 |
| | Function calls | Adding or modifying an argument | 152 |
| | Class fields | Adding/removing class fields | 22 |
| | Function declarations | Modifying a function's signature | 94 |
| | Sequences | Adding a function call to a sequence of calls, all with the same receiver | 42 |
| | Loops | Changing a loop's predicate | 5 |
| | `switch` blocks | Adding/removing a switch branch | 6 |
| | `try` blocks | Introducing a new `try-catch` block | 1 |
| New | `return` statements | Changing a `return` statement | 40 |
| | Variable declaration | Declaring an existing variable | 2 |
| | Initialization | Initializing a variable with empty object literal/array | 3 |

there are specific bug-fix patterns that emerge in JavaScript. Any disagreements regarding classification or the potential need for new categories were resolved through further discussion among the participants. In order to identify the occurrences of patterns, a manual analysis was conducted to ensure the coverage of potential new patterns and to add an additional layer of validation against possible misclassifications.

It should be noted that since the categories suggested by Pan et al. were originally developed for Java programs, we needed to ensure that we correctly applied them to JavaScript code. Specifically, prior to ECMAScript 2015, JavaScript did not have built-in syntactical support for classes. Instead, classes were simulated using functions as constructors, with methods and fields added to the prototype [113, 104, 47]. Additionally, object literals could be used to represent classes, with a comma-separated list of name-value pairs enclosed in curly braces defining the class fields and methods. To prevent any misclassifications, we carefully considered all of these aspects during the classification process.

The bug-fixing categories and the number of occurrences of each are presented in Table 5.5. In cases where a bug fix spanned multiple lines, it was possible for us to assign more than one category to a single bug-fixing commit. As a result, the overall number of occurrences listed in Table 5.5 is greater than the number of bugs analyzed.

The most common fix patterns involved modifying an `if` statement, either by changing its condition or adding a precondition, modifying assignment statements, and changing function call arguments (Table 5.5). We found that the same three categories were also found to be the most frequent in Java code. Furthermore, we observed that changes to class fields were also prevalent in JavaScript bug fixes.

### JavaScript-related bug-fixing patterns

In our benchmark, we identified three recurring patterns that had not been previously reported:

- `return` **statements:** We identified a frequent pattern in bug fixes that involved modifying the return statement of a function. Developers changed the expression used in the return statement to address issues with the function's behavior or output. This pattern was the most common JavaScript-related pattern.
- **Variable declaration:** In JavaScript, it is permissible to use a variable without declaring it explicitly, which can cause subtle bugs. When a variable is utilized inside a function without prior declaration, it gets "hoisted" to the top of the global scope. As a result, it becomes visible to all functions, even *outside its original lexical scope*, leading to potential naming conflicts. The recurring pattern we observed involves fixing this issue by declaring a variable that was previously in use.
- **Initialization:** This bug-fixing pattern aims to fix uninitialized variables. This pattern provides an alternative to using an `if` statement by utilizing a logical "or" operator. For example, `a = a || {}` assigns the value of `a` to itself if it already has a "non-falsey" value. If `a` has a "falsey" value, then it will be initialized with an empty object. This pattern also corresponds to Hanam et al.'s [59] first bug pattern.

> **Answering Research Question 1:** *Our analysis of bugs in BugsJS revealed that 88% of them had fixes falling into one of the proposed categories. The most common fix patterns involved modifying an* `if` *statement. Interestingly, the same three categories were also found to be the most frequent in Java code. Furthermore, we identified three JavaScript-specific recurring patterns.*

## 5.3.2 Research Question 2

The second research question we will answer is the following:

> **Research Question 2:** *How do the bug-fixing patterns in BugsJS relate to our taxonomy of bugs?*

We compared the taxonomy with the bug-fixing patterns used to fix the bugs. Figures 5.4, 5.5, and 5.6 present Sankey diagrams showing the relationship of the assigned bug categories with the bug-fixing patterns of Pan et al. [96] and the JavaScript-related bug-fixing patterns described in Section 5.3.1. We focused our analysis on the first three main bug categories of our taxonomy. For presentational clarity, each figure shows a diagram for one of such main categories. The left side of the diagrams shows one of the main bug categories, unfolded into its sub-categories. The right side depicts the associated bug-fixing patterns. The *None* node indicates that no patterns were applicable. In the middle, each bug category is connected to each bug-fixing pattern that is used to fix the bugs belonging to the bug category. The thickness of the curved lines between the nodes indicates the cardinality of the association.

Table 5.6: Bug-fixing patterns

| Category | Pattern name |
|---|---|
| | *Pan* [96] |
| Assignment (AS) | Change of assignment expression (AS-CE) |
| Class Field (CF) | Addition of a class field (CF-ADD)<br>Change of class field declaration (CF-CHG)<br>Removal of a class field (CF-RMV) |
| If-related (IF) | Addition of an else branch (IF-ABR)<br>Addition of precondition check (IF-APC)<br>Addition of precondition check with jump (IF-APCJ)<br>Addition of post-condition check (IF-APTC)<br>Change of if condition expression (IF-CC)<br>Removal of an else branch (IF-RBR)<br>Removal of an if predicate (IF-RMV) |
| Loop (LP) | Change of loop condition (LP-CC)<br>Change of the expression that modifies the loop variable (LP-CE) |
| Method Call (MC) | Method call with different actual parameter values (MC-DAP)<br>Different method call to a class instance (MC-DM)<br>Method call with different number or types of parameters (MC-DNP) |
| Method Declaration (MD) | Change of method declaration (MD-CHG)<br>Addition of a method declaration (MD-ADD)<br>Removal of a method declaration (MD-RMV) |
| Sequence (SQ) | Addition of operations in an operation sequence of field settings (SQ-AFO)<br>Addition of operations in an operation sequence of method calls to an object (SQ-AMO)<br>Addition or removal method call operations in a short construct body (SQ-AROB)<br>Removal of operations from an operation sequence of field settings (SQ-RFO)<br>Removal of operations from an operation sequence of method calls to an object (SQ-RMO) |
| Switch (SW) | Addition/removal of switch branch (SW-ARSB) |
| Try (TY) | Addition/removal of a catch block (TY-ARCB)<br>Addition/removal of a try statement (TY-ARTC) |
| | *BugsJS* |
| JavaScript (JS) | Changing a return statement (JS-Return)<br>Initializing a variable with empty object literal/array (JS-Initialization)<br>Declaring an existing variable (JS-Declaration) |

For example, in Figure 5.4, the node *incomplete feature implementation* is connected to the subcategory *missing input validation* with a thick line, due to the majority of the bugs in this main category belonging to that subcategory. Then, the node of that subcategory is connected to the node of the *IF-CC* bug-fixing pattern with a

relatively thick line, because a considerable amount of bug fixes are classified into that category. The nodes of the bug categories can be wider than the total width of the lines connected from the right side because bug fixes can be assigned with multiple bug patterns, whereas each bug is assigned with exactly one bug category. Also, there are bugs that are only assigned to a main category, e.g., *Missing input validation*, but not to any subcategory, which resulted in the direct lines between the pattern and main category nodes.

Table 5.6 gives a description of the bug-fixing pattern abbreviations. We now discuss each of the bug categories in detail.

### Incomplete feature implementation

Figure 5.4 illustrates the connection between the bugs under the Incomplete feature implementation category and the related bug-fixing patterns.

**Missing input validation.** Table 5.7 shows that the majority of the bugs assigned to this category are fixed by `if`-related (53), assignment related (29), and method declaration related (22) changes. The most common are changing a condition in an `if` statement (*IF-CC*), changing an assignment expression (*AS-CE*), or adding a method declaration (*MD-ADD*). The most numerous subcategory is the Missing type check and the related bugs are mainly fixed by `if`-related changes (39), the top two are *IF-CC* and *IF-APCJ* and by changing an assignment expression (18 *AS-CE*). The fixes of bugs in the Missing handling of special characters subcategory often contain changes in an assignment expression (6 *AS-CE*) and in an `if` statement (5), mainly adding post-condition checks (*IF-APTC*). Bugs belonging to the Missing null check and the Empty input parameters sub-categories are mainly fixed by changing an `if` condition (5). The Missing handling of spaces subcategory is connected to various bug-fixing patterns and none of them is dominant.

**Incomplete configuration processing.** The majority of the bug fixes of this category contain `if`-related patterns (9), mainly *IF-APC*, adding a precondition check. The Missing type check subcategory of Incomplete configuration processing is too small to draw meaningful conclusions.

**Error handling.** Bugs of this category are typically fixed with `if`-related fixes (12), namely *IF-APCJ*, *IF-APC*, and *IF-CC*. The second most common fix patterns are *MC-DNP*, changing the number of parameters (3) and *MD-ADD*, adding a method declaration (4). The Callbacks subcategory of Error handling is related to a variety of bug-fixing patterns (4 `if`-related, 1 assignment related, 1 method call related, and 1 sequence related).

**Incomplete output message.** Bugs assigned to this category are usually fixed by changing the parameters of function calls (2 *MC-DAP*, 1 *MC-DNP*).

**Incomplete data processing.** Bugs belonging to this category were fixed in a variety of ways. The most common pattern is method call related (11), but there is no dominant bug-fixing pattern.

### Incorrect feature implementation

Figure 5.5 illustrates the connection between the bugs under the Incorrect feature implementation category and the related bug-fixing patterns.

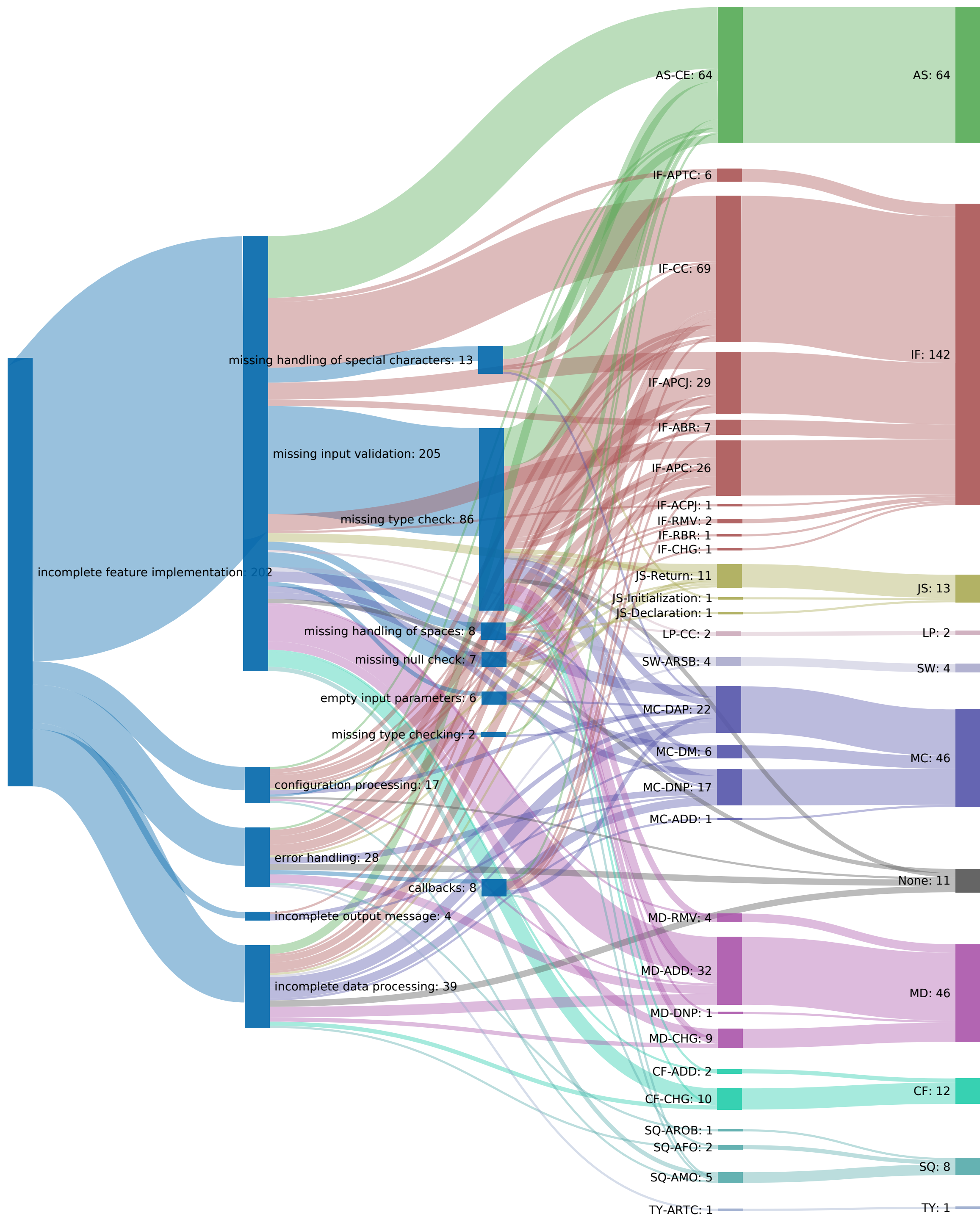


Figure 5.4: Bug-fixing patterns used in the Incomplete feature implementation category

**Incorrect input validation.** The most dominant bug-fixing patterns of this category are `if`-related (56), method-call-related (27), and assignment-related (25). The most numerous `if`-related pattern is *IF-CC* and the most numerous method-call-related pattern is *MC-DAP*. Furthermore, `return`-statement-related (12 *JS-Return*) and method-declaration-related (10 *MD-ADD* and 5 *MD-CHG*) patterns are also quite common. In the Unnecessary type check subcategory, the most common pattern is `if`-related (6 *IF-CC* and 1 *IF-RMV*) and the second most common pattern is the assignment-related

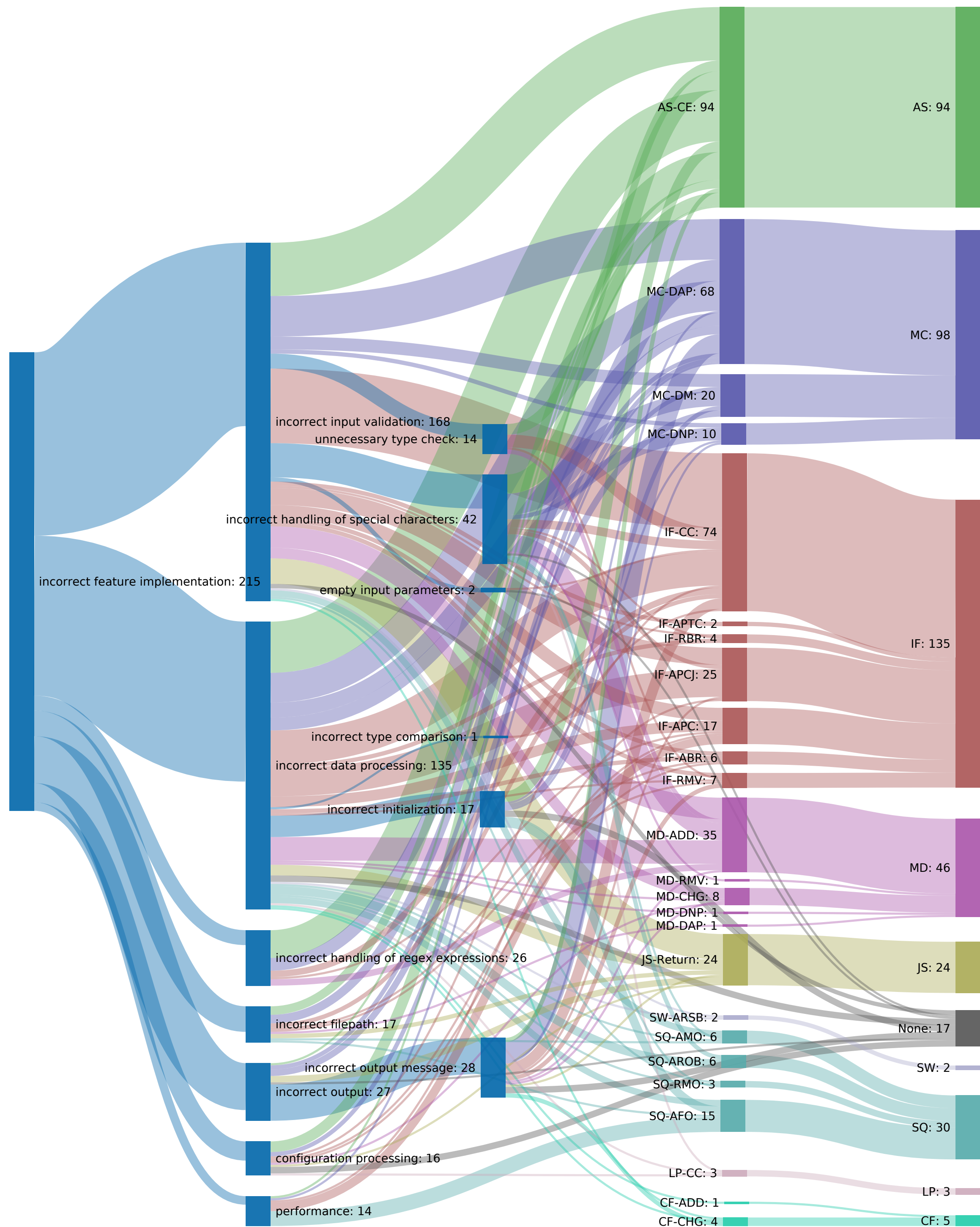


Figure 5.5: Bug-fixing patterns used in the Incorrect feature implementation category

pattern (5 *AS-CE*). Bugs belonging to the biggest input-validation-related subcategory, the Incorrect handling of special characters, are usually fixed with method-call-related changes (10 *MC-DAP* and 2 *MC-DM*), with assignment-related changes (9 *AS-CE*) and with method-declaration-related changes (9 *MD-ADD* and 1 *MD-RMV*). The Empty input parameters subcategory of Incorrect input validation is too small to draw meaningful conclusions.

**Incorrect data processing.** Similarly to the Incomplete data processing category,

bugs belonging to this category were fixed in a variety of ways. There is no dominant bug-fixing pattern. The most numerous patterns are `if`-related (39), method-call-related (27), and assignment-related (24). The Incorrect initialization subcategory is interestingly not connected to `if`-related patterns at all. The bug fixes of this category are connected to only assignment-related (5), sequence-related (5), and method-call-related (4) patterns. The Incorrect type comparison subcategory of Incorrect data processing is too small to draw meaningful conclusions.

**Incorrect handling of regex expressions.** Bugs assigned to this category are mostly fixed with assignment-related (13 *AS-CE*) and method-call-related (6 *MC-DAP*) changes.

**Incorrect filepath.** Here, the most dominant bug-fixing patterns are assignment-related (4), method-call-related (4), and `if`-related (4). The variety of connected patterns is large.

**Incorrect output.** Bugs of this category are usually fixed by changing method calls (3*MC-DM* and 2 *MC-DAP*) and by changing `return` statements (3 *JS-Return*). The Incorrect output message subcategory contains bugs with fixes that mostly involve changes to the parameters of method calls (9 *MC-DAP* and 1 *MC-DNP*) and changes to condition expressions in `if` statements (5 *IF-CC*, 2 *IF-RMV*, and 1 *IF-APCJ*).

**Incorrect configuration processing.** The most dominant bug-fixing pattern for this category is assignment-related (5 *AS-CE*), but a variety of other patterns occur as well (3 `if`-related, 2 method-call-related, 1 JavaScript-related, 1 loop-related, and 1 method-declaration-related).

**Performance.** The majority of the bug-fixing patterns used for fixing bugs of this category are *SQ-AFO*, adding operations in an operation sequence of field settings (7). The second most common patterns are `if`-related (5 *IF-APC* and 1 *IF-APCJ*).

### Generic

Figure 5.6 illustrates the connection between the bugs under the Generic category and the related bug-fixing patterns.

**Variable initialization.** The bug fixes of the Missing variable initialization subcategory contain five types of bug-fixing patterns. The most dominants are the JavaScript-related patterns (2 *JS-Return*, 2 *JS-Initialization*, and 1 *JS-Declaration*) and the class-field-related patterns (2 *CF-CHG* and 2 *CF-ADD*). In the other subcategory, Incorrect variable initialization, changing an assignment expression (3 *AS-CE*) is the most dominant.

**Data processing.** Similarly to the other two cases of data processing bugs, the associated bug fixes contain a variety of patterns (2 JavaScript related, 2 method-call-related, 1 `if`-related, 1 assignment-related, and 1 sequence-related) and there is no dominant bug-fixing pattern.

**Missing type conversion.** The bug fixes of this category mostly involve changes in an assignment expression (2 *AS-CE*), but adding a precondition check to an `if` statement (1 *IF-APC*) and removing an operation from an operation sequence of method calls (1 *SQ-RMO*) also appears.

**Typo.** For this category, there is no dominant bug-fixing pattern (2 `if`-related, 2 method-call-related, 1 assignment-related, and 1 method-declaration-related), which essentially means typos can occur at any place in the code.

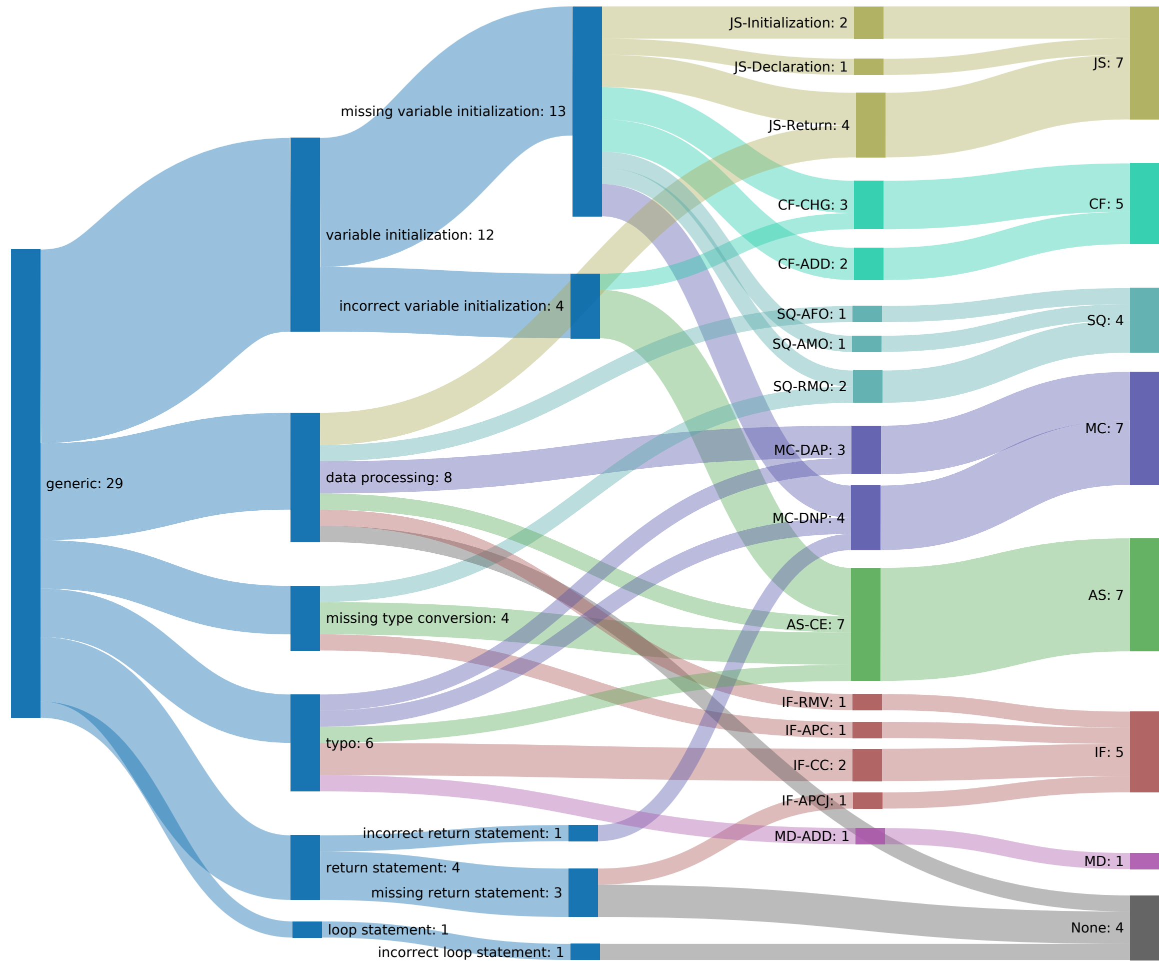


Figure 5.6: Bug-fixing patterns used in the Generic category

**Return statement.** This category contains too few bug-fixing patterns to draw meaningful conclusions. Surprisingly, the bug fixes do not fall into the *JS-Return* category.

**Loop statement.** There is only one bug in this category and its fix does not contain any bug-fixing patterns, therefore, we cannot draw meaningful conclusions here.

### Highlights

Table 5.7 provides statistics about the occurrences of each bug-fix type corresponding to the bug types. The table can serve to analyze the emergence of correlations between bug-fix types and bug types.

**Answering Research Question 2:** *Overall, the most common bug-fixes in BugsJS are `if`-related (291), the second most common are assignment-related (166), and the third most common are method-call-related (152) bug-fixes. These bug-fix types are mostly related to the most prominent bug categories, namely missing input validation, incorrect input validation, and incorrect data processing. Another correlation is that assignment-related fixes are also the preferred way to fix regexes. These are perhaps the only correlations between bug-fix types and bug types that are observable in our benchmark.*

Table 5.7: Taxonomy and bug-fixing types

| | AS | CF | IF | JS | LP | MC | MD | None | SQ | SW | TY |
|---|---|---|---|---|---|---|---|---|---|---|---|
| INCOMPLETE FEATURE IMPLEMENTATION | | | | | | | | | | | |
| configuration processing | 1 | 0 | 9 | 1 | 0 | 2 | 1 | 1 | 1 | 0 | 0 |
| missing type check | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| error handling | 1 | 0 | 12 | 1 | 0 | 3 | 4 | 3 | 1 | 0 | 1 |
| callbacks | 1 | 0 | 4 | 0 | 0 | 2 | 0 | 0 | 1 | 0 | 0 |
| incomplete data processing | 4 | 2 | 9 | 1 | 0 | 11 | 7 | 3 | 1 | 1 | 0 |
| incomplete output message | 0 | 0 | 1 | 0 | 0 | 3 | 0 | 0 | 0 | 0 | 0 |
| missing input validation | 29 | 8 | 53 | 4 | 1 | 11 | 22 | 2 | 2 | 2 | 0 |
| empty input parameters | 1 | 0 | 3 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |
| missing handling of spaces | 2 | 0 | 2 | 1 | 0 | 1 | 2 | 0 | 0 | 0 | 0 |
| missing handling of special characters | 6 | 0 | 5 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| missing null check | 0 | 0 | 5 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| missing type check | 18 | 2 | 39 | 2 | 1 | 10 | 10 | 2 | 1 | 1 | 0 |
| INCORRECT FEATURE IMPLEMENTATION | | | | | | | | | | | |
| configuration processing | 5 | 0 | 3 | 1 | 1 | 2 | 1 | 3 | 0 | 0 | 0 |
| incorrect data processing | 24 | 2 | 39 | 5 | 1 | 27 | 13 | 3 | 9 | 1 | 0 |
| incorrect initialization | 5 | 0 | 0 | 0 | 0 | 4 | 0 | 3 | 5 | 0 | 0 |
| incorrect type comparison | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| incorrect filepath | 4 | 0 | 4 | 2 | 0 | 4 | 1 | 0 | 2 | 0 | 0 |
| incorrect handling of regex expressions | 13 | 0 | 4 | 0 | 0 | 6 | 3 | 0 | 0 | 0 | 0 |
| incorrect input validation | 25 | 1 | 56 | 12 | 0 | 27 | 15 | 2 | 4 | 1 | 0 |
| empty input parameters | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| incorrect handling of special characters | 9 | 0 | 7 | 0 | 1 | 12 | 9 | 1 | 3 | 0 | 0 |
| unnecessary type check | 5 | 0 | 7 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 |
| incorrect output | 1 | 0 | 0 | 3 | 0 | 5 | 0 | 1 | 0 | 0 | 0 |
| incorrect output message | 2 | 2 | 8 | 1 | 0 | 10 | 2 | 3 | 0 | 0 | 0 |
| performance | 1 | 0 | 5 | 0 | 0 | 1 | 0 | 0 | 7 | 0 | 0 |
| GENERIC | | | | | | | | | | | |
| data processing | 1 | 0 | 1 | 2 | 0 | 2 | 0 | 1 | 1 | 0 | 0 |
| loop statement | | | | | | | | | | | |
| incorrect loop statement | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| missing type conversion | 2 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| return statement | | | | | | | | | | | |
| incorrect return statement | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| missing return statement | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 |
| typo | 1 | 0 | 2 | 0 | 0 | 2 | 1 | 0 | 0 | 0 | 0 |
| variable initialization | | | | | | | | | | | |
| incorrect variable initialization | 3 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| missing variable initialization | 0 | 4 | 0 | 5 | 0 | 2 | 0 | 0 | 2 | 0 | 0 |
| PERFECTIVE MAINTENANCE | 1 | 0 | 9 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |

## 5.4 Discussion

Our results might drive devising novel software analysis and repair techniques for JavaScript, with BugsJS being a suitable real-world bug benchmark for their evaluation, as well as inform developers of the most error-prone constructs.

In the rest of this section, we discuss some of the potential uses of our taxonomy,

together with possible use cases of our benchmark in supporting empirical studies in software analysis and testing, as well as its limitations, and the threats to validity of our study.

### 5.4.1 Directing Developer Efforts

#### Improving manual repair processes

In the absence of automated program repair techniques, knowledge about the causes of bugs could help developers manually fix programs, both by increasing their awareness of bug causes, and helping them prioritize the inspection of possible causes based on the relative importance of such causes in our taxonomy.

#### Avoiding bugs

Bug avoidance pertains to cases in which programmers are provided with information to assist them in avoiding bugs, and it is up to them to choose what information to utilize and how. One way to promote bug avoidance involves educating developers and maintainers of JavaScript applications as to the causes and probabilities of bugs. Such education could be supported by the information present in our taxonomy and the data that underlie it. First, consider code change activities. We observed that many of the bugs involving missing type checks resulted from simple code changes, i.e., missing an IF-condition in a statement. Second, consider code creation tasks. When creating new code, programmers can enforce the practice of adding input validation as a must-do of their daily activities. Finally, consider test case creation. Testers can use our taxonomy to focus their test case creation to avoid the most bug-prone categories.

#### Preventing bugs

Bug prevention, in contrast to bug avoidance, involves the use of automated approaches for ensuring that bugs do not occur, even though humans might be included in the feedback loop. For example, programmers may take advantage of tools like checkers and linting tools that enable static and dynamic analysis automatically [74], and our taxonomy could help improve on certain constructs that are particularly challenging for developers.

#### IDE enhancements

Another class of bug prevention approaches involves improvements in web programming and testing IDEs. Analysis techniques that operate concurrently with program development and maintenance may be quite effective, and such techniques could also be guided by our taxonomy. Modern IDEs for code development typically employ such approaches: as programmers edit, they point out problems or provide useful information based on the programmers' actions. JavaScript application development IDEs employ such approaches also, but to our knowledge, no such IDEs also build in assistance related to testing efforts. Such IDEs could aid in bug avoidance by alerting developers to possible effects or bad practices and letting them choose whether or not to act on them. They could aid in prevention by prohibiting certain actions or by recommending the creation of constructs. Let us take as an example the return-statement-related issues. Our taxonomy highlighted that developers often fix bugs by changing the return

statement. Thus, IDEs can be improved with new data flow testing techniques that check that JavaScript objects' states are preserved during the execution before they are returned, or that inform a change-impact analysis technique to show how the change to an object affects the final output.

### 5.4.2 Limitations

In the JavaScript ecosystem, the presence of numerous implementations, standards, and testing frameworks poses significant technical obstacles in creating a uniform bug infrastructure. Similarly, choosing Mocha as a reference testing framework presented its own challenges. BugsJS only comprises server-side JavaScript applications created using the Node.js framework, and it does not currently support evaluating client-side programs. This limitation and bias for experiments using BugsJS is worth noting, and we are considering expanding the benchmark in the future to accommodate other environments. However, due to the unique features of the JavaScript ecosystem, we do not anticipate fully covering the numerous execution environments. Nevertheless, all the subjects included in BugsJS have previously been employed in at least one study on bugs.

### 5.4.3 Threats to Validity

The main threat to the *internal validity* of this work is the possibility of introducing bias when selecting and classifying the surveyed papers and the bugs included in the benchmark.

Our paper selection was driven by the keywords related to software analysis and testing for JavaScript. We may have missed relevant studies that are not captured by our list of terms. We mitigate this threat by performing an issue-by-issue, manual search in the major software engineering conference proceedings and journals, followed by a snowballing process. We, however, cannot claim that our survey captures all relevant literature; yet, we are confident that the included papers cover the major related studies.

Concerning the bugs, we manually classified all candidate bugs into different categories (Section 5.1.3), and the retained bugs into categories pertaining to existing bug and fix taxonomies (Section 5.3.1). To minimize classification errors, multiple authors simultaneously analyzed the source code and performed the classifications individually, and disagreements were resolved by further discussions among the authors. Concerning the bug classification for taxonomy construction, the first four authors classified the bugs manually. This task, however, requires reasoning that cannot be automated, so it is difficult to envision less threat-prone approaches. To reduce the subjectivity involved in the task, the authors followed a systematic and structured procedure, with multiple interactions.

Threats to *external validity* concern the generalization of our findings. We, by no means, claim that our benchmark represents all relevant web apps. We selected only 10 applications and our bugs may not generalize to different projects. Also, other relevant classes of bugs might be unrepresented or underrepresented within our benchmark, which is to date quite overfitted towards ESLint, i.e., the most represented project. Nevertheless, we tried to mitigate this threat by selecting applications with different sizes and pertaining to different domains. We hope that our framework will

provide an entry point and a reference for future improvements as other subject systems are necessary to fully confirm the generalizability of our results, and corroborate our findings.

Another generalization threat concerns our taxonomy. Taxonomies are conceptual maps derived from empirical observations; as such, they typically evolve as additional observations of the world are made. We expect the same to be true of our taxonomy, and thus, it is not necessarily the case that any attempt to apply the taxonomy to subsequent programs will allow every type of bug in those applications to be categorized. In such cases, the taxonomy will require adjustments. This work attempts to reduce this threat by applying a validation phase to our initial taxonomy.

With respect to the *reproducibility* of our results, all classifications, subjects, and experimental data are available online, making the analysis reproducible.

## 5.5 Summary

In this chapter, we presented BugsJS, a benchmark of 453 real, manually-verified JavaScript bugs from 10 popular programs. Our investigation included both quantitative and qualitative analyses, with a particular focus on grouping the bugs into a dedicated taxonomy. Our research showed a wide range of bugs that BugsJS encompasses, and it can serve as a reliable benchmark for conducting reproducible research in software analysis and testing. It can be utilized in various areas of research, including regression testing, bug prediction, and fault localization. Additionally, a flexible framework has been implemented in BugsJS, enabling researchers to automate the process of examining specific revisions of the source code, executing tests, and producing test coverage reports.

We intend to broaden the dataset by incorporating other JavaScript testing frameworks, and our long-term objective is to encompass client-side JavaScript web applications in BugsJS as well.

*"Science can amuse and fascinate us all, but it is engineering that changes the world."*

— Isaac Asimov

# 6 Conclusions

In this thesis, we discussed the studies we conducted in the area of bug databases.

We showed that previous datasets created using traditional approaches contain uncertainties. Therefore, we developed a novel method to capture the before-fix and after-fix states of buggy source code elements, resulting in a dataset with reduced uncertainties. We carried out empirical evaluations and found that this dataset can be useful for bug prediction. We also conducted an experiment to compare the bug prediction capabilities of method-level metrics projected to the class level with those of class-level metrics. Our findings indicate that projecting method-level metrics to the class level enhances their predictive power for bug prediction.

Despite previous studies showing that process metrics outperform product metrics in bug prediction, the use of process metrics is not widespread, and there is a scarcity of research on process metrics. To address this, we developed a method to efficiently compute process metrics for files, classes, and methods using a graph database. We confirmed that bug databases with process metrics are suitable for bug prediction. We also compared process and product metrics and found that the use of process metrics in bug prediction yields more stable results. Moreover, process metrics provide a different perspective on characterizing source code elements compared to product metrics.

Lastly, we created a benchmark of real, manually-verified JavaScript bugs, along with a framework to automate research processes. We analyzed the bugs included in the benchmark and found that most have fixes that fall into existing bug-fixing patterns. We also created a bug taxonomy and investigated whether this categorization relates to bug-fixing patterns. This benchmark serves as a reliable source for conducting reproducible research in software analysis and testing.

## Future Work

In addition to the future work discussed in the summaries of each chapter, we plan to combine these studies. Given the unique perspective with which process metrics characterize source code elements, utilizing them as predictors for bug prediction on a database created through our novel method would make for an interesting study.

Furthermore, employing bug prediction techniques using our JavaScript benchmark would present an outstanding opportunity for experimentation, although a challenging one due to the dynamic nature of the JavaScript language.

## Epilogue

This research journey has been immensely enriching for me. Throughout this process, I have acquired valuable skills that will accompany me throughout my life. From conducting comprehensive reviews of existing literature to meticulously documenting the research process and presenting the results, each step has contributed to my growth. Obtaining a PhD is not the ultimate goal, but rather a stepping stone towards a lifelong pursuit of knowledge and opportunities. Whether I am engaged in academic research, working at an IT company, or venturing into new business endeavors, I will be committed to giving my best.

While I have always been attentive to the details when writing code, my experience in researching software bugs has increased my awareness of their impact. As a result, I can confidently say that I will never look at programming the same way again.

# Appendices

# Summary in English

Despite the assistance provided by various integrated development environments, programmers often make mistakes, which can lead to bugs. Therefore, it is crucial to acquire more advanced tools that facilitate the automatic detection of errors. The research conducted for this thesis focuses on enhancing the precision of software bug prediction. This involves gaining a better understanding of the bug occurrence patterns and effectively characterizing them.

The research studies are organized into three thesis points. In the first thesis point, we discuss a novel method for constructing a bug database. This method aims to provide a more precise repository of bugs. In the second thesis point, we compare the predictive power of product metrics with that of process metrics. The objective is to determine which set of metrics holds greater potential for accurately predicting software bugs. In the third thesis point, we present a benchmark of JavaScript bugs and conduct detailed analyses of these bugs. This analysis contributes to the overall understanding of bug patterns and assists in the development of more effective bug detection techniques.

Overall, these thesis points form the core of our research, enabling us to advance the field of software bug prediction and mitigation.

**I. A Novel Bug Prediction Dataset and its Validation**

The contributions of this thesis point are discussed in Chapter 3.

Previously published datasets have followed a conventional approach to creating benchmark datasets for testing bug prediction techniques. These datasets typically include all code elements, both faulty and non-faulty, from one or more versions of the analyzed system. In this thesis point, we introduce a new approach that focuses on collecting snapshots of source code elements affected by bugs, along with their characteristics, before and after the bugs were fixed. This approach excludes code elements that were not impacted by bugs. By utilizing this kind of dataset, we can effectively capture the changes in software product metrics during bug fixes. This enables us to examine the differences in source code metrics between faulty and non-faulty code elements and gain valuable insights. To conduct our analysis, we selected 15 open-source projects from GitHub and

considered all reported bugs from their bug tracking systems. We constructed databases at three source code levels: file, class, and method. Using 11 machine learning algorithms, we built prediction models and demonstrated the dataset's potential for bug prediction and further investigations. Our findings indicate that creating bug prediction models at the method level yields better results compared to the file and class levels when considering the complete dataset. We also observed variations in F-measure values across different projects, supporting our hypothesis that not all projects provide suitable training sets. However, we achieved promising F-measure values for individual projects, reaching up to 0.7573 at the method level, 0.7400 at the class level, and 0.7741 at the file level. Furthermore, we conducted a novel experiment comparing the predictive capabilities of method-level metrics when projected to the class level against the class-level metrics themselves. The results revealed a significant improvement in the prediction accuracy when using the method-level metrics projected to the class level. These findings contribute to the advancement of bug prediction techniques and demonstrate the potential of our dataset as a valuable resource for future research in this field.

***The Author's Contributions***

The author has made several significant contributions to this research. Firstly, he designed and implemented the novel method presented for constructing bug databases. Additionally, he actively participated in the literature review and the process of defining criteria for the inclusion of projects in the BugHunter dataset. The author was responsible for executing the method, constructing the dataset, and gathering all the statistics related to the projects. Lastly, the author took a leading role in producing and analyzing the results obtained from the machine learning techniques utilized in this study. The publications related to this thesis point are:

- **Péter Gyimesi**, Gábor Gyimesi, Zoltán Tóth, and Rudolf Ferenc. Characterization of Source Code Defects by Data Mining Conducted on GitHub. In *15th International Conference on Computational Science and Its Applications (ICCSA 2015), Banff, AB, Canada, June 22–25*, pages 47–62, LNCS, Volume 9159. Springer International Publishing, 2015.
- Zoltán Tóth, **Péter Gyimesi**, and Rudolf Ferenc. A Public Bug Database of GitHub Projects and its Application in Bug Prediction. In *16th International Conference on Computational Science and Its Applications (ICCSA 2016), Beijing, China, July 4–7*, pages 625–638, LNCS, Volume 9789. Springer International Publishing, 2016.
- Rudolf Ferenc, **Péter Gyimesi**, Gábor Gyimesi, Zoltán Tóth, and Tibor Gyimóthy. An Automatically Created Novel Bug Dataset and its Validation in Bug Prediction. Journal of Systems and Software, 2020, 169: 110691.

**II. Calculation of Process Metrics and their Bug Prediction Capabilities**

The contributions of this thesis point are discussed in Chapter 4.

Studies have shown that process metrics outperform product metrics in bug prediction. However, the use of process metrics remains limited, and there is a scarcity of research in this area. Therefore, in this thesis point, we aim to

address these gaps by presenting an effective method for computing a variety of software process metrics. To accomplish this, we leverage graph technologies, specifically utilizing the widely-used Neo4j graph database. We implemented the calculation of 22 process metrics for files, classes, and methods, along with the corresponding bug counts. To evaluate the effectiveness of process metrics, we selected five open-source Java projects from GitHub and generated databases for 5 release versions of each project. Subsequently, we applied 11 machine learning algorithms to construct prediction models, confirming that a bug database incorporating process metrics is indeed suitable for bug prediction. Notably, based on the F-measure values, we observed that tree-based methods consistently outperformed others, with the RandomForest method displaying the best performance across all cases. Additionally, we conducted a comparative analysis with product metrics and observed that the utilization of process metrics in bug prediction yielded more robust results. Moreover, process metrics offer a different perspective on characterizing source code elements compared to product metrics, further enhancing their value in this context.

***The Author's Contributions***

This study is the author's independent work. He reviewed the literature, designed the methodology, and implemented the necessary tools. Subsequently, he generated the bug databases containing process metrics, conducted the evaluations, and drew conclusions. The publications related to this thesis point are:

- **Péter Gyimesi**. Automatic Calculation of Process Metrics and their Bug Prediction Capabilities. In *Acta Cybernetica*, pages 537–559, Volume 23, No 2, 2017.
- **Péter Gyimesi**. An open-source solution for automatic bug database creation. In *Proceedings of the 10$^{th}$ International Conference on Applied Informatics (ICAI 2017), Eger, Hungary, January 30–February 1*, pages 111–119, 2017.

## III. A Public Dataset of JavaScript Bugs

The contributions of this thesis point are discussed in Chapter 5.

Despite the extensive research on JavaScript (JS), a well-organized repository of labeled JS bugs was still missing. The presence of numerous JS implementations further complicated the task of creating a cohesive bugs benchmark. To address this gap, in this thesis point, we introduced a benchmark comprising a total of 453 manually selected and validated JS-related bugs from 10 open-source JS projects. Additionally, we have developed a framework to automate research processes utilizing our benchmark. We conducted an investigation to determine whether the bug-fixing patterns for JS bugs align with existing classification schemes. Interestingly, we found that a majority of the bugs in our dataset fit into existing bug-fixing patterns. Notably, we discovered that the same three categories were also prevalent in Java code. Furthermore, we identified three JS-specific recurring patterns. By conducting both quantitative and qualitative analyses on the bugs, we constructed a taxonomy of JS bugs present in the benchmark. We also explored the relationship between this categorization and bug-fixing patterns. Our analysis revealed that the most common bug-fix types in the benchmark

are if-related fixes (291), followed by assignment-related fixes (166), and method call-related fixes (152). In the taxonomy, the most prominent bug categories are identified as missing input validation, incorrect input validation, and incorrect data processing. Another notable finding is that assignment-related fixes are frequently used to fix regexes. This comprehensive analysis demonstrates that the dataset encompasses a wide range of bugs and can serve as a reliable benchmark for conducting reproducible research in software analysis and testing.

***The Author's Contributions***

During this research, the author actively participated in designing the research plan and implementing the framework, which included the benchmark. He was responsible for collecting and analyzing JavaScript projects, selecting suitable bugs, and extracting relevant bug fixes. Additionally, he took part in manually validating the bugs. Throughout the analysis of the bugs, the author actively contributed to examining bug-fixing patterns, as well as creating and validating the bug taxonomy. Finally, the author took a leading role in analyzing the correlation between the bug taxonomy and the bug-fixing patterns. The publications related to this thesis point are:

- **Péter Gyimesi**, Béla Vancsics, Andrea Stocco, Davood Mazinanian, Arpád Beszédes, Rudolf Ferenc and Ali Mesbah. BugsJS: A Benchmark of JavaScript Bugs. In *12th IEEE Conference on Software Testing, Validation and Verification (IEEE ICST 2019), Xi'an, China, April 22–27*, pages 90–101, IEEE, Volume 1. IEEE Computer Society Press, 2019.
- **Péter Gyimesi**, Béla Vancsics, Andrea Stocco, Davood Mazinanian, Arpád Beszédes, Rudolf Ferenc and Ali Mesbah. BugsJS: A Benchmark and Taxonomy of JavaScript Bugs. Journal of Software Testing, Verification and Reliability (STVR 2021), John Wiley & Sons Publishing. 38 pages.
- Béla Vancsics, **Péter Gyimesi**, Andrea Stocco, Davood Mazinanian, Arpád Beszédes, Rudolf Ferenc and Ali Mesbah. Poster: Supporting JavaScript Experimentation with BugsJS. In *12th IEEE Conference on Software Testing, Validation and Verification (IEEE ICST 2019), Poster Track, Xi'an, China, April 22–27*, pages 375–378, IEEE, Volume 1. IEEE Computer Society Press, 2019.

Table A.1 summarizes the main publications and how they relate to our thesis points.

| **№** | [4] | [7] | [1] | [2] | [3] | [5] | [6] | [8] |
|---|---|---|---|---|---|---|---|---|
| I. | ♦ | ♦ | ♦ | | | | | |
| II. | | | | ♦ | ♦ | | | |
| III. | | | | | | ♦ | ♦ | ♦ |

Table A.1: Thesis contributions and supporting publications

# B
# Magyar nyelvű összefoglaló

Annak ellenére, hogy a különböző integrált fejlesztői környezetek számos támogatást nyújtanak, a programozók gyakran hibáznak, amely szoftverhibákhoz vezethet. Ebből kifolyólag lényeges, hogy a hibák automatikus felismerését megkönnyítő, minél jobb eszközök álljanak rendelkezésre. A jelen disszertációt megalapozó kutatás a szoftverhibák pontosabb előrejelzésének javítását hivatott elősegíteni a hibák előfordulási mintázatait és a hibás kódrészek jellemzőit vizsgálva.

A kutatási eredményeinket három tézispontba szervezve taglaljuk. Az első tézispontban bemutatunk egy új elméleti módszert hibaadatbázisok létrehozására, melynek célja, hogy pontosabb, kevesebb bizonytalanságot tartalmazó adatbázist kapjunk. A második tézispontban a hibás kódrészek jellemzőit vizsgáljuk úgy, hogy összehasonlítjuk a termékmetrikák hiba-előrejelző képességét a folyamatmetrikákéval abból a célból, hogy meghatározzuk, melyik metrikahalmaz rendelkezik nagyobb potenciállal a szoftverhibák pontosabb előrejelzésére. A harmadik tézispontban bemutatunk egy JavaScript hibákat tartalmazó adathalmazt, és részletes elemzéseket végzünk a benne található hibákról. Ezzel hozzájárulunk a gyakori JavaScript hibamintázatok feltárásához, elősegítve a hatékonyabb hibafelismerő technikák kidolgozását.

Ezek a tézispontok képezik a kutatásunk középpontját, lehetővé téve a szoftverhibák előrejelzésének területén történő előrelépést.

## I. Újszerű predikciós hibaadatbázis és kiértékelése

Az ide tartozó kutatási eredményeket a 3. fejezet tárgyalja.

Az eddig publikált hiba adatbázisok hagyományos megközelítést alkalmazva kerültek létrehozásra. Ezeket az adatbázisokat a különféle hiba-előrejelzési technikák tesztelése céljából hozták létre. Egy hagyományos módon előállított adathalmaz tartalmazza az összes kódelemet, hibásat és hibamenteset egyaránt, az elemzett rendszer egy vagy több verziójából. Ebben a tézispontban egy új módszert mutatunk be hiba adatbázisok előállítására, amely a hibával érintett forráskódelemek pillanatképeinek gyűjtésére összpontosít. Ezen pillanatképek a hibák javítása előtti és utáni állapotát reprezentálják. A módszer nem veszi figyelembe azokat a kódelemeket, amelyeket nem érintett hiba. Az így előállított újszerű

adathalmazzal hatékonyan vizsgálhatjuk a forráskódmetrikák változásait a hibajavítások során. Az újszerű adatbázis kiértékeléséhez 15 nyílt forráskódú projektet választottunk ki a GitHub-ról, és feldolgoztuk az összes bejelentett hibát. Három forráskódszinten hoztunk létre adatbázisokat: fájl, osztály és metódus. Ezután 11 gépi tanulási algoritmust felhasználva előrejelző modelleket építettünk, és bemutattuk az adathalmaz potenciálját a hibaelőrejelzés terén. Eredményeink azt mutatják, hogy a metódusszintű hibaelőrejelzési modellek jobb eredményeket hoznak a fájl és az osztály szintekhez képest. Továbbá a különböző projekteken elért F-measure értékek közötti eltérések alátámasztják azon hipotézisünket, miszerint nem minden projekt nyújt megfelelő tanulási halmazt. Ígéretes F-measure értékeket értünk el bizonyos projektek esetén, elérve akár 0,7573-at metódusszinten, 0,7400-t osztályszinten és 0,7741-et fájlszinten. Emellett egy újszerű kísérletet is végrehajtottunk, ahol összehasonlítottuk a metódusszintű metrikák osztályszintre vetített hiba-előrejelző képességét az osztályszintű metrikák hiba-előrejelző képességével. Azt tapasztaltuk, hogy ezt a technikát alkalmazva jelentős javulás érhető el a predikció pontosságában. Összességében, az elért eredmények tekintetében kimondható, hogy kutatásunkkal hozzájárultunk a hiba-előrejelzési technikák fejlődéséhez, illetve bemutattuk az adathalmazunk potenciálját a jövőbeli kutatások számára ezen a területen.

***A szerző hozzájárulása***

A szerző számos hozzájárulást tett a kutatás során. Először is, ő tervezte meg az elvi alapjait a bemutatott újszerű hiba adatbázis építő módszernek. Továbbá a módszer implementációjában is jelentős szerepet vállalt. Ezen felül aktívan részt vett a szakirodalom áttekintésében és a projektkiválasztási kritériumok meghatározásában. A szerző volt felelős a módszer futtatásáért, az adathalmazok létrehozásáért és a projektekkel kapcsolatos összes statisztika előállításáért. A szerző ugyancsak jelentős szerepet vállalt a gépi tanulási technikákkal elért eredmények előállításában és elemzésében. A tézispont a következő publikációkra épül:

- **Péter Gyimesi**, Gábor Gyimesi, Zoltán Tóth, and Rudolf Ferenc. Characterization of Source Code Defects by Data Mining Conducted on GitHub. In *15th International Conference on Computational Science and Its Applications (ICCSA 2015), Banff, AB, Canada, June 22–25*, pages 47–62, LNCS, Volume 9159. Springer International Publishing, 2015.
- Zoltán Tóth, **Péter Gyimesi**, and Rudolf Ferenc. A Public Bug Database of GitHub Projects and its Application in Bug Prediction. In *16th International Conference on Computational Science and Its Applications (ICCSA 2016), Beijing, China, July 4–7*, pages 625–638, LNCS, Volume 9789. Springer International Publishing, 2016.
- Rudolf Ferenc, **Péter Gyimesi**, Gábor Gyimesi, Zoltán Tóth, and Tibor Gyimóthy. An Automatically Created Novel Bug Dataset and its Validation in Bug Prediction. Journal of Systems and Software, 2020, 169: 110691.

### II. Folyamatmetrikák számítása és hiba-előrejelző képességük

Az ide tartozó kutatási eredményeket a 4. fejezet tárgyalja.

Korábbi kutatások kimutatták, hogy a folyamatmetrikákkal jobb eredmény érhető el a hiba-előrejelzésben, mint a termékmetrikákkal. Azonban a folyamat-

metrikák használata továbbra sem terjedt el széles körben, és viszonylag kevés kutatás történt ezen a területen. A jelen tézispontban ezt a hiányosságot céloztuk meg orvosolni azáltal, hogy egy hatékony módszert mutatunk be különböző folyamatmetrikák számítására. Ennek elérésére gráf technológiákat hívunk segítségül, egészen pontosan a széles körben elterjedt Neo4j gráf adatbázist használjuk. Megvalósítottuk a kalkulációját 22 folyamatmetrikának, illetve ugyanezt a módszert alkalmazva, a hibás forráskódelemek meghatározását. Ezen számításokat három forráskódszinten végezzük: fájl, osztály és metódus. A folyamatmetrikák hiba-előrejelző hatékonyságának vizsgálatához 5 nyílt forráskódú Java projektet választottunk ki a GitHub-ról, és hiba adatbázisokat hoztunk létre mindegyik projekt 5 verziójához. Ezután 11 gépi tanulási algoritmust alkalmaztunk predikciós modellek létrehozásához, amelyek megerősítették, hogy egy folyamatmetrikákat tartalmazó hiba adatbázis valóban alkalmas a hiba-előrejelzésre. A kapott F-measure értékek alapján azt tapasztaltuk, hogy a fa alapú gépi tanulási algoritmusok következetesen jobb eredményeket értek el a többihez képest, a RandomForest módszer pedig a legjobb teljesítményt hozta minden esetben. Emellett összehasonlító elemzést végeztünk a termékmetrikákkal, és azt az eredményt kaptuk, hogy bár a folyamatmetrikák használata a hiba-előrejelzésben nem minden esetben hoz javulást, de a kapott eredmények robusztusabbak, stabilabbak. Ezenkívül tovább növeli a folyamatmetrikák értékét az, hogy alacsony a korrelációjuk a termékmetrikákkal, tehát más nézőpontot kínálnak a forráskódelemek jellemzésére a termékmetrikákhoz képest.

***A szerző hozzájárulása***

Ez a tanulmány a szerző önálló munkája. Áttekintette a szakirodalmat, megtervezte a módszert és implementálta a szükséges eszközöket. Ezt követően előállította a folyamatmetrikákat tartalmazó hiba adatbázisokat, majd elvégezte a kiértékeléseket és levonta a következtetéseket. A tézispont a következő publikációkra épül:

- **Péter Gyimesi**. Automatic Calculation of Process Metrics and their Bug Prediction Capabilities. In *Acta Cybernetica*, pages 537–559, Volume 23, No 2, 2017.
- **Péter Gyimesi**. An open-source solution for automatic bug database creation. In *Proceedings of the 10$^{th}$ International Conference on Applied Informatics (ICAI 2017), Eger, Hungary, January 30–February 1*, pages 111–119, 2017.

## III. Publikus JavaScript hibaadatbázis

Az ide tartozó kutatási eredményeket az 5. fejezet tárgyalja.

Annak ellenére, hogy rengeteg kutatás történt a JavaScript (JS) területén, még mindig hiányzik egy címkézett JS hibákat tartalmazó adathalmaz. A sok JS implementáció tovább nehezíti egy egységes, jól szervezett adatbázis létrehozását. Ezen hiányosságokat orvosolva ebben a tézispontban bemutatunk egy új adathalmazt, amelybe összesen 453 manuálisan validált JS hibát választottunk be 10 nyílt forráskódú JS projektből. Emellett létrehoztunk egy keretrendszert az adathalmazunkon alapuló kutatási folyamatok automatizálására. Ezután vizsgálatot végeztünk annak megállapítására, hogy a JS hibák javítási mintái egyeznek-e

a meglévő osztályozási rendszerekkel. Érdekes módon arra az eredményre jutottunk, hogy az adathalmazban található hibák nagy része illeszkedik a már meglévő javítási mintákhoz. Nevezetesen, ugyanaz a három kategória a leggyakoribb a Java kódokban is. Ezenkívül azonosítottunk három JS-specifikus ismétlődő mintát. A hibákon végzett kvantitatív és kvalitatív elemzések segítségével létrehoztunk egy JS hibataxonómiát, amely szintén része az adathalmaznak. Ezenkívül vizsgáltuk a kapcsolatot ezen kategorizáció és a hibajavítási minták között. Elemzésünk azt mutatta, hogy az adathalmazban a leggyakoribb hibajavítási típusok az `if` feltételekkel kapcsolatos (291), az értékadással kapcsolatos (166) és a metódushívással kapcsolatos (152) minták. A taxonómiában a legjelentősebb hibakategóriák az input validáció hiánya, a helytelen input validáció és a helytelen adatfeldolgozás. Egy másik jelentős megállapítás, hogy az értékadással kapcsolatos hibajavítási minták gyakran használatosak a reguláris kifejezések javítására. Ez az átfogó elemzés bizonyítja, hogy az adatbázisunk változatos hibákat tartalmaz, és megbízható forrása lehet szoftverelemzéssel és -teszteléssel kapcsolatos, reprodukálható kutatásoknak.

***A szerző hozzájárulása***

A kutatás során a szerző aktívan részt vett a kutatási terv kialakításában és a keretrendszer implementálásában, amely magában foglalta az adathalmazt. Felelős volt a JavaScript projektek gyűjtéséért és elemzéséért, megfelelő hibák kiválasztásáért, valamint releváns hibajavítások kinyeréséért. Emellett részt vett a hibák kézi validálásában. A hibák elemzése során a szerző aktívan hozzájárult a hibajavítási minták vizsgálatához, valamint a hibataxonómia létrehozásához és validálásához. Végül a szerző vezető szerepet vállalt a hibataxonómia és a hibajavítási minták közötti összefüggés elemzésében. A tézispont a következő publikációkra épül:

- **Péter Gyimesi**, Béla Vancsics, Andrea Stocco, Davood Mazinanian, Árpád Beszédes, Rudolf Ferenc and Ali Mesbah. BugsJS: A Benchmark of JavaScript Bugs. In *12$^{th}$ IEEE Conference on Software Testing, Validation and Verification (IEEE ICST 2019), Xi'an, China, April 22–27*, pages 90–101, IEEE, Volume 1. IEEE Computer Society Press, 2019.
- **Péter Gyimesi**, Béla Vancsics, Andrea Stocco, Davood Mazinanian, Árpád Beszédes, Rudolf Ferenc and Ali Mesbah. BugsJS: A Benchmark and Taxonomy of JavaScript Bugs. Journal of Software Testing, Verification and Reliability (STVR 2021), John Wiley & Sons Publishing. 38 pages.
- Béla Vancsics, **Péter Gyimesi**, Andrea Stocco, Davood Mazinanian, Arpád Beszédes, Rudolf Ferenc and Ali Mesbah. Poster: Supporting JavaScript Experimentation with BugsJS. In *12$^{th}$ IEEE Conference on Software Testing, Validation and Verification (IEEE ICST 2019), Poster Track, Xi'an, China, April 22–27*, pages 375–378, IEEE, Volume 1. IEEE Computer Society Press, 2019.

A tézispontokhoz tartozó publikációkat a B.1. táblázat foglalja össze.

| **№** | [4] | [7] | [1] | [2] | [3] | [5] | [6] | [8] |
|---|---|---|---|---|---|---|---|---|
| I. | ♦ | ♦ | ♦ | | | | | |
| II. | | | | ♦ | ♦ | | | |
| III. | | | | | | ♦ | ♦ | ♦ |

B.1. táblázat. A tézispontokhoz kapcsolódó publikációk

# Bibliography

## Corresponding Publications of the Author

[1] Rudolf Ferenc, Péter Gyimesi, Gábor Gyimesi, Zoltán Tóth, and Tibor Gyimóthy. An automatically created novel bug dataset and its validation in bug prediction. *Journal of Systems and Software*, 169:110691, 2020.

[2] Péter Gyimesi. Automatic calculation of process metrics and their bug prediction capabilities. *Acta Cybernetica*, 23(2):537–559, 2017.

[3] Péter Gyimesi. An open-source solution for automatic bug database creation. In *Proceedings of the 10th International Conference on Applied Informatics*, page 111–119, 2017.

[4] Péter Gyimesi, Gábor Gyimesi, Zoltán Tóth, and Rudolf Ferenc. Characterization of source code defects by data mining conducted on github. In *International Conference on Computational Science and Its Applications*, pages 47–62. Springer, 2015.

[5] Péter Gyimesi, Béla Vancsics, Andrea Stocco, Davood Mazinanian, Arpád Beszédes, Rudolf Ferenc, and Ali Mesbah. Bugsjs: A benchmark of javascript bugs. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 90–101. IEEE, 2019.

[6] Péter Gyimesi, Béla Vancsics, Andrea Stocco, Davood Mazinanian, Árpád Beszédes, Rudolf Ferenc, and Ali Mesbah. Bugsjs: a benchmark and taxonomy of javascript bugs. *Software Testing, Verification And Reliability*, 31(4):e1751, 2021.

[7] Zoltán Tóth, Péter Gyimesi, and Rudolf Ferenc. A public bug database of github projects and its application in bug prediction. In *International Conference on Computational Science and Its Applications*, pages 625–638. Springer, 2016.

[8] Béla Vancsics, Péter Gyimesi, Andrea Stocco, Davood Mazinanian, Arpád Beszédes, Rudolf Ferenc, and Ali Mesbah. Poster: Supporting javascript experimentation with bugsjs. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 375–378. IEEE, 2019.

## Other Publications

[9] IEEE standard classification for software anomalies. IEEE Std 1044-2009, 2009.

[10] JHawk. `http://www.virtualmachinery.com/jhawkprod.htm`, 2018. Accessed: 2018-01-25.

[11] Christoffer Quist Adamsen, Anders Møller, Rezwana Karim, Manu Sridharan, Frank Tip, and Koushik Sen. Repairing event race errors by controlling nondeterminism. In *Proc. of 39th International Conference on Software Engineering (ICSE)*, 2017.

[12] Saba Alimadadi, Ali Mesbah, and Karthik Pattabiraman. Hybrid DOM-sensitive change impact analysis for JavaScript. In *Proc. of European Conference on Object-Oriented Programming (ECOOP)*, 2015.

[13] Saba Alimadadi, Ali Mesbah, and Karthik Pattabiraman. Understanding asynchronous interactions in full-stack JavaScript. In *Proc. of 38th International Conference on Software Engineering (ICSE)*, 2016.

[14] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proc. of International Conference on Software Engineering*, 2005.

[15] T. Bakota, P. Hegedus, P. Kortvelyesi, R. Ferenc, and T. Gyimothy. A probabilistic software quality model. In *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, pages 243 –252, sept. 2011.

[16] Dénes Bán and Rudolf Ferenc. Recognizing antipatterns and analyzing their effects on software maintainability. In *International Conference on Computational Science and Its Applications*, pages 337–352. Springer, 2014.

[17] Dénes Bán, Rudolf Ferenc, István Siket, Ákos Kiss, and Tibor Gyimóthy. Prediction models for performance, power, and energy efficiency of software executed on heterogeneous hardware. *The Journal of Supercomputing*, pages 1–25, 2018.

[18] P. Bangcharoensap, A. Ihara, Y. Kamei, and K. Matsumoto. Locating source code to be fixed based on initial bug reports - a case study on the eclipse project. In *Empirical Software Engineering in Practice (IWESEP), 2012 Fourth International Workshop on*, pages 10–15, Oct 2012.

[19] Victor R Basili, Lionel C Briand, and Walcélio L Melo. A validation of object-oriented design metrics as quality indicators. *Software Engineering, IEEE Transactions on*, 22(10):751–761, 1996.

[20] Stefan Bellon, Rainer Koschke, Giuliano Antoniol, Jens Krinke, and Ettore Merlo. Comparison and evaluation of clone detection tools. *Software Engineering, IEEE Transactions on*, 33(9):577–591, 2007.

[21] Abraham Bernstein, Jayalath Ekanayake, and Martin 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*, pages 11–18. ACM, 2007.

[22] Pamela Bhattacharya, Marios Iliofotou, Iulian Neamtiu, and Michalis Faloutsos. Graph-based analysis and prediction for software evolution. In *Proceedings of the 34th International Conference on Software Engineering*, pages 419–429. IEEE Press, 2012.

[23] Marina Billes, Anders Møller, and Michael Pradel. Systematic black-box analysis of collaborative web applications. In *Proc. of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2017.

[24] C. Bird, P.C. Rigby, E.T. Barr, D.J. Hamilton, D.M. German, and P. Devanbu. The promises and perils of mining git. In *Mining Software Repositories, 2009. MSR '09. 6th IEEE International Working Conference on*, pages 1–10, May 2009.

[25] Barry Boehm, Chris Abts, and Sunita Chulani. Software development cost estimation approaches — a survey. *Annals of Software Engineering*, 10(1):177–205, Nov 2000.

[26] Cathal Boogerd and Leon Moonen. Assessing the value of coding standards: An empirical study. In *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, pages 277–286. IEEE, 2008.

[27] David Bowes, Tracy Hall, Mark Harman, Yue Jia, Federica Sarro, and Fan Wu. Mutation-aware fault prediction. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 330–341. ACM, 2016.

[28] Magiel Bruntink and Arie Van Deursen. Predicting class testability using object-oriented metrics. In *Source Code Analysis and Manipulation, 2004. Fourth IEEE International Workshop on*, pages 136–145. IEEE, 2004.

[29] E. C. Campos and M. d. A. Maia. Common bug-fix patterns: A large-scale observational study. In *Proc. of ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2017.

[30] Cagatay Catal. Software fault prediction: A literature review and current trends. *Expert systems with applications*, 38(4):4626–4636, 2011.

[31] Cagatay Catal and Banu Diri. A systematic review of software fault prediction studies. *Expert systems with applications*, 36(4):7346–7354, 2009.

[32] Gemma Catolino, Fabio Palomba, Andy Zaidman, and Filomena Ferrucci. Not all bugs are the same: Understanding, characterizing, and classifying bug types. *Journal of Systems and Software*, 152:165–181, 2019.

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

[34] Ram Chillarege, Inderpal S. Bhandari, Jarir K. Chaar, Michael J. Halliday, Diane S. Moebus, Bonnie K. Ray, and Man-Yuen Wong. Orthogonal defect classification - A concept for in-process measurements. *IEEE Trans. Software Eng.*, 18(11):943–956, 1992.

[35] Jacob Cohen. A power primer. *Psychological bulletin*, 112(1):155, 1992.

[36] H. Coles. PITest Mutation Framework. http://pitest.org/, 2018. Accessed: 2018-01-25.

[37] C. Couto, C. Silva, M.T. Valente, R. Bigonha, and N. Anquetil. Uncovering causal relationships between software metrics and bugs. In *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*, pages 223–232, March 2012.

[38] Jan Salomon Cramer. The origins of logistic regression. 2002.

[39] Valentin Dallmeier and Thomas Zimmermann. Automatic extraction of bug localization benchmarks from history. Technical report, Universitat des Saarlandes and Saarbrücken and Germany, 2007.

[40] Valentin Dallmeier and Thomas Zimmermann. Extraction of bug localization benchmarks from history. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 433–436. ACM, 2007.

[41] Marco D'Ambros, Michele Lanza, and Romain Robbes. An extensive comparison of bug prediction approaches. *Proceedings of MSR 2010 (7th IEEE Working Conference on Mining Software Repositories)*, pages 31 – 41, 2010.

[42] Steven Davies, Marc Roper, and Murray Wood. Using bug report similarity to enhance bug localisation. In *Reverse Engineering (WCRE), 2012 19th Working Conference on*, pages 125–134. IEEE, 2012.

[43] Naji Dmeiri, David A. Tomassi, Yichen Wang, Antara Bhowmick, Yen-Chuan Liu, Premkumar Devanbu, Bogdan Vasilescu, and Cindy Rubio-Gonzalez. BugSwarm: Mining and Continuously Growing a Dataset of Reproducible Failures and Fixes. In *Proc. of 41st International Conference on Software Engineering (ICSE)*, 2019.

[44] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Softw. Engg.*, 10(4):405–435, October 2005.

[45] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 422–431. IEEE Press, 2013.

[46] Markus Ermuth and Michael Pradel. Monkey see, monkey do: Effective generation of GUI tests with inferred macro events. In *Proc. of 25th International Symposium on Software Testing and Analysis (ISSTA)*, 2016.

[47] Laleh Eshkevari, Davood Mazinanian, Shahriar Rostami, and Nikolaos Tsantalis. JSDeodorant: Class-awareness for JavaScript Programs. In *Proceedings of the 39th International Conference on Software Engineering Companion*, 2017.

[48] Amin Milani Fard and Ali Mesbah. JavaScript: The (un)covered parts. In *Proc. of IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2017.

[49] Michael Fischer, Martin Pinzger, and Harald Gall. Populating a release history database from version control and bug tracking systems. In *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, pages 23–32. IEEE, 2003.

[50] G. Fraser and A. Arcuri. Sound empirical evidence in software testing. In *Proc. of 34th International Conference on Software Engineering (ICSE)*, 2012.

[51] Y. Freund and R. E. Schapire. Large margin classification using the perceptron algorithm. In *11th Annual Conference on Computational Learning Theory*, pages 209–217, New York, NY, 1998. ACM Press.

[52] Milton Friedman. A comparison of alternative tests of significance for the problem of m rankings. *The Annals of Mathematical Statistics*, 11(1):86–92, 1940.

[53] Baljinder Ghotra, Shane McIntosh, and Ahmed 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.

[54] Antonios Gkortzis, Dimitris Mitropoulos, and Diomidis Spinellis. VulinOSS: A dataset of security vulnerabilities in open-source systems. In *Proc. of 15th International Conference on Mining Software Repositories*, 2018.

[55] R. Gopinath, C. Jensen, and A. Groce. Mutations: How close are they to real faults? In *Proc. of International Symposium on Software Reliability Engineering*, 2014.

[56] Todd L Graves, Alan F Karr, James S Marron, and Harvey Siy. Predicting fault incidence using software change history. *IEEE Transactions on software engineering*, 26(7):653–661, 2000.

[57] Tibor Gyimothy, Rudolf Ferenc, and Istvan Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *Software Engineering, IEEE Transactions on*, 31(10):897–910, 2005.

[58] Tracy Hall, Min Zhang, David Bowes, and Yi Sun. Some code smells have a significant but small effect on faults. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(4):33, 2014.

[59] Quinn Hanam, Fernando S. de M. Brito, and Ali Mesbah. Discovering bug patterns in JavaScript. In *Proc. of 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2016.

[60] Mark Harman, Syed Islam, Yue Jia, Leandro L Minku, Federica Sarro, and Komsan Srivisut. Less is more: Temporal fault predictive performance over multiple hadoop releases. In *International Symposium on Search Based Software Engineering*, pages 240–246. Springer, 2014.

[61] Ahmed E Hassan. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering*, pages 78–88. IEEE Computer Society, 2009.

[62] Haibo He, Edwardo Garcia, et al. Learning from imbalanced data. *Knowledge and Data Engineering, IEEE Transactions on*, 21(9):1263–1284, 2009.

[63] S. Herbold, A. Trautsch, and J. Grabowski. A comparative study to benchmark cross-project defect prediction approaches. *IEEE Transactions on Software Engineering*, 44(9):811–833, Sep. 2018.

[64] Tin Kam Ho. Random decision forests. In *Proceedings of 3rd international conference on document analysis and recognition*, volume 1, pages 278–282. IEEE, 1995.

[65] Robert C Holte. Very simple classification rules perform well on most commonly used datasets. *Machine learning*, 11:63–90, 1993.

[66] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *Transactions on Software Engineering*, 37(5), 2011.

[67] Bryant Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 672–681. IEEE, 2013.

[68] Marian Jureczko and Lech Madeyski. Towards identifying software project clusters with regard to defect prediction. In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, page 9. ACM, 2010.

[69] René Just, Darioush Jalali, and Michael D. Ernst. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proc. of 2014 International Symposium on Software Testing and Analysis*, 2014.

[70] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. Are mutants a valid substitute for real faults in software testing? In *Proc. of ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2014.

[71] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian. The promises and perils of mining github. *MSR 2014 Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 92–101, 2014.

[72] Sunghun Kim, Thomas Zimmermann, Kai Pan, and E James Whitehead Jr. Automatic identification of bug-introducing changes. In *Automated Software Engineering, 2006. ASE'06. 21st IEEE/ACM International Conference on*, pages 81–90. IEEE, 2006.

[73] Ron Kohavi. The power of decision tables. In *8th European Conference on Machine Learning*, pages 174–189. Springer, 1995.

[74] D. R. Krishna Murthy and M. Pradel. Change-aware dynamic program analysis for javascript. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 127–137, September 2018.

[75] Sandeep Krishnan, Chris Strasburg, Robyn R Lutz, and Katerina 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.

[76] Niels Landwehr, Mark Hall, and Eibe Frank. Logistic model trees. 95(1-2):161–205, 2005.

[77] Claire Le Goues, Neal Holtschulte, Edward K. Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. The ManyBugs and IntroClass benchmarks for automated repair of C programs. *IEEE Transactions on Software Engineering (TSE)*, 41(12):1236–1256, December 2015.

[78] Z. Li, X. Jing, and X. Zhu. Progress on approaches to software defect prediction. *IET Software*, 12(3):161–175, 2018.

[79] Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. Have things changed now?: an empirical study of bug characteristics in modern open source software. In *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, pages 25–33. ACM, 2006.

[80] Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. QuixBugs: A multi-lingual program repair benchmark set based on the Quixey Challenge. In *Proc. of International Conference on Systems, Programming, Languages, and Applications: Software for Humanity: Companion.*

[81] Yan Ma, Lan Guo, and Bojan Cukic. A statistical framework for the prediction of fault-proneness. *Advances in Machine Learning Application in Software Engineering, Idea Group Inc*, pages 237–265, 2006.

[82] Lech Madeyski and Marian Jureczko. Which process metrics can significantly improve defect prediction models? An empirical study. *Software Quality Journal*, 23(3):393–422, 2015.

[83] Magnus Madsen, Frank Tip, Esben Andreasen, Koushik Sen, and Anders Møller. Feedback-directed instrumentation for deployed JavaScript applications. In *Proc. of 38th International Conference on Software Engineering (ICSE)*, 2016.

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

[85] Tim Menzies, Zach Milton, Burak Turhan, Bojan Cukic, Yue Jiang, and Ayşe Bener. Defect prediction from static code features: current results, limitations, new approaches. *Automated Software Engineering*, 17(4):375–407, 2010.

[86] Pryor D. Menzies T., Krishna R. The Promise Repository of Empirical Software Engineering Data, 2015. North Carolina State University, Department of Computer Science.

[87] Audris Mockus and Lawrence G Votta. Identifying reasons for software changes using historic databases. In *Software Maintenance, 2000. Proceedings. International Conference on*, pages 120–130. IEEE, 2000.

[88] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. Analysis of the reliability of a subset of change metrics for defect prediction. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 309–311. ACM, 2008.

[89] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*, pages 181–190. IEEE, 2008.

[90] John C Munson and Sebastian G Elbaum. Code churn: A measure for estimating the impact of code change. In *Software Maintenance, 1998. Proceedings., International Conference on*, pages 24–31. IEEE, 1998.

[91] Nachiappan Nagappan and Thomas Ball. Use of relative code churn measures to predict system defect density. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, pages 284–292. IEEE, 2005.

[92] N. K. Nagwani, S. Verma, and K. K. Mehta. Generating taxonomic terms for software bug classification by utilizing topic models based on latent dirichlet allocation. In *2013 Eleventh International Conference on ICT and Knowledge Engineering*, pages 1–5, November 2013.

[93] P. Nemenyi. *Distribution-free multiple comparison.* PhD dissertation, Princeton University, 1963.

[94] F. S. Ocariza, K. Bajaj, K. Pattabiraman, and A. Mesbah. A Study of Causes and Consequences of Client-Side JavaScript Bugs. *IEEE Transactions on Software Engineering*, 43(2):128–144, February 2017.

[95] Thomas J Ostrand, Elaine J Weyuker, and Robert M Bell. Predicting the location and number of faults in large software systems. *Software Engineering, IEEE Transactions on*, 31(4):340–355, 2005.

[96] Kai Pan, Sunghun Kim, and E. James Whitehead. Toward an understanding of bug fix patterns. *Empirical Software Engineering*, 14(3):286–315, June 2009.

[97] Jean Petrić, David Bowes, Tracy Hall, Bruce Christianson, and Nathan Baddoo. The jinx on the nasa software defect data sets. In *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*, pages 1–5, 2016.

[98] Adam Porter, Richard W Selby, et al. Empirically guided software development using metric-based classification trees. *Software, IEEE*, 7(2):46–54, 1990.

[99] Ross Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufmann Publishers, San Mateo, CA, 1993.

[100] Danijel Radjenović, Marjan Heričko, Richard Torkar, and Aleš Živkovič. Software fault prediction metrics: A systematic literature review. *Information and Software Technology*, 55(8):1397–1418, 2013.

[101] Foyzur Rahman and Premkumar Devanbu. How, and why, process metrics are better. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 432–441. IEEE Press, 2013.

[102] Jacek Ratzinger, Martin Pinzger, and Harald Gall. EQ-Mine: Predicting short-term defects for software evolution. In *International Conference on Fundamental Approaches to Software Engineering*, pages 12–26. Springer, 2007.

[103] Robert Rosenthal, H Cooper, and L Hedges. Parametric measures of effect size. *The handbook of research synthesis*, 621:231–244, 1994.

[104] S. Rostami, L. Eshkevari, D. Mazinanian, and N. Tsantalis. Detecting function constructors in JavaScript. In *Proc. of IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2016.

[105] Chanchal K Roy, James R Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470–495, 2009.

[106] David Saad. Online algorithms and stochastic approximations. *Online Learning*, 5(3):6, 1998.

[107] Ripon K Saha, Matthew Lease, Sarfraz Khurshid, and Dewayne E Perry. Improving bug localization using structured information retrieval. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 345–355. IEEE, 2013.

[108] Carolyn B. Seaman. Qualitative methods in empirical studies of software engineering. *IEEE Trans. Softw. Eng.*, 25(4):557–572, July 1999.

[109] Martin Shepperd, Qinbao Song, Zhongbin Sun, and Carolyn Mair. Data quality: Some comments on the nasa software defect datasets. *IEEE Transactions on Software Engineering*, 39(9):1208–1215, 2013.

[110] Emad Shihab, Zhen Ming Jiang, Walid M Ibrahim, Bram Adams, and Ahmed E Hassan. Understanding the impact of code and process metrics on post-release defects: a case study on the Eclipse project. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, page 4. ACM, 2010.

[111] Thomas Shippey, Tracy Hall, Steve Counsell, and David Bowes. So you need more method level datasets for your software defect prediction?: Voilà! In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '16, pages 12:1–12:6, New York, NY, USA, 2016. ACM.

[112] J Sayyad Shirabad and Tim J Menzies. The promise repository of software engineering databases. *School of Information Technology and Engineering, University of Ottawa, Canada*, 24, 2005.

[113] Leonardo Humberto Silva, Marco Tulio Valente, and Alexandre Bergel. Refactoring legacy JavaScript code to use classes: The good, the bad and the ugly. In *Mastering Scale and Complexity in Software Reuse*, 2017.

[114] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? In *ACM sigsoft software engineering notes*, volume 30, pages 1–5. ACM, 2005.

[115] Elaine Svenonius. *The Intellectual Foundation of Information Organization*. MIT Press, Cambridge, MA, USA, 2000.

[116] Lin Tan, Chen Liu, Zhenmin Li, Xuanhui Wang, Yuanyuan Zhou, and Chengxiang Zhai. Bug characteristics in open source software. *Empirical Software Engineering*, 19(6):1665–1705, 2014.

[117] Ferdian Thung, Xuan-Bach D Le, and David Lo. Active semi-supervised defect categorization. In *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*, pages 60–70. IEEE Press, 2015.

[118] Ferdian Thung, David Lo, and Lingxiao Jiang. Automatic defect categorization. In *2012 19th Working Conference on Reverse Engineering*, pages 205–214. IEEE, 2012.

[119] Christopher Timperley, Susan Stepney, and Claire Le Goues. Poster: BugZoo – A Platform for Studying Software Bugs. In *International Conference on Software Engineering*, ICSE '18, 2018. To appear.

[120] Deqing Wang, Mengxiang Lin, Hui Zhang, and Hongping Hu. Detect related bugs from source code using bug information. *Computer Software and Applications Conference (COMPSAC)*, 2010.

[121] J. Wang, W. Dou, C. Gao, Y. Gao, and J. Wei. Context-based event trace reduction in client-side JavaScript applications. In *Proc. of International Conference on Software Testing, Verification and Validation (ICST)*, 2018.

[122] J. Wang, W. Dou, Y. Gao, C. Gao, F. Qin, K. Yin, and J. Wei. A comprehensive study on real world concurrency bugs in Node.js. In *Proc. of International Conference on Automated Software Engineering*, 2017.

[123] Shaowei Wang and David Lo. Version history, similar report, and structure: Putting them together for improved bug localization. In *Proceedings of the 22nd International Conference on Program Comprehension*, pages 53–63. ACM, 2014.

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

[125] F Wilcoxon. Individual comparisons by ranking methods. biom bull 1 (6): 80–83, 1945.

[126] Chadd Williams and Jaime Spacco. Szz revisited: verifying when changes induce fixes. In *Proceedings of the 2008 workshop on Defects in large software systems*, pages 32–36. ACM, 2008.

[127] Rongxin Wu, Hongyu Zhang, Sunghun Kim, and Shing-Chi Cheung. Relink: recovering links between bugs and changes. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 15–25. ACM, 2011.

[128] Toth Z., Novak G., Ferenc R., and Siket I. Using version control history to follow the changes of source code elements. *Software Maintenance and Reengineering (CSMR)*, 2013.

[129] Harry Zhang. Exploring conditions for the optimality of naive bayes. *International Journal of Pattern Recognition and Artificial Intelligence*, 19(02):183–198, 2005.

[130] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. An empirical study on tensorflow program bugs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 129–140. ACM, 2018.

[131] Hao Zhong and Zhendong Su. An empirical study on real bug fixes. In *Proc. of 37th International Conference on Software Engineering (ICSE)*, pages 913–923, 2015.

[132] Jian Zhou, Hongyu Zhang, and David Lo. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. *Software Engineering (ICSE), 2012 34th International Conference on*, 2012.

[133] Yuming Zhou and Hareton Leung. Empirical analysis of object-oriented design metrics for predicting high and low severity faults. *Software Engineering, IEEE Transactions on*, 32(10):771–789, 2006.

[134] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. Predicting defects for eclipse. In *Predictor Models in Software Engineering, 2007. PROMISE'07: ICSE Workshops 2007. International Workshop on*, pages 9–9. IEEE, 2007.