UNIVERSIDADE ESTADUAL PAULISTA
"JÚLIO DE MESQUITA FILHO"
Campus de Presidente Prudente

Ingrid Marçal

Automatic Test Case Generation Guided by Ensemble Bug Prediction

Presidente Prudente
2024

Ingrid Marçal

Automatic Test Case Generation Guided by Ensemble Bug Prediction

Dissertation submitted to the Institute of Biosciences, Arts, and Exact Sciences (IBILCE - UNESP - São José do Rio Preto) as part of the requirements to obtain a Doctor of Philosophy in Computer Science degree.

Advisor: Prof. PhD. Rogério Eduardo Garcia

Presidente Prudente
2024

M313a

Marçal, Ingrid
Automatic Test Case Generation Guided by Ensemble Bug Prediction / Ingrid Marçal. -- Presidente Prudente, 2024
216 p.

Tese (doutorado) - Universidade Estadual Paulista (UNESP), Faculdade de Ciências e Tecnologia, Presidente Prudente
Orientador: Rogério Eduardo Garcia

1. Ciência da computação. 2. Engenharia de Software. 3. Inteligência artificial. I. Título.

Sistema de geração automática de fichas catalográficas da Unesp. Biblioteca da Universidade Estadual Paulista (UNESP), Faculdade de Ciências e Tecnologia, Presidente Prudente. Dados fornecidos pelo autor(a).

Essa ficha não pode ser modificada.

UNIVERSIDADE ESTADUAL PAULISTA

Câmpus de Presidente Prudente

CERTIFICADO DE APROVAÇÃO

TÍTULO DA TESE: Automatic Test Case Generation Guided by Ensemble Bug Prediction

AUTORA: INGRID MARÇAL
ORIENTADOR: ROGÉRIO EDUARDO GARCIA

Aprovada como parte das exigências para obtenção do Título de Doutora em Ciência da Computação, área: Computação Aplicada pela Comissão Examinadora:

Prof. Dr. ROGÉRIO EDUARDO GARCIA (Participaçao Virtual)
FCT / UNESP / Presidente Prudente (SP)

Prof. Dr. AURI MARCELO RIZZO VINCENZI (Participaçao Virtual)
Departamento de Computação / Universidade Federal de São Carlos

Prof. Dr. DANILO MEDEIROS ELER (Participaçao Virtual)
Departamento de Matematica e Computacao / Faculdade de Ciencias e Tecnologia de Presidente Prudente

Prof. Dr. MARCELO MEDEIROS ELER (Participaçao Virtual)
Escola de Artes, Ciências e Humanidades / Universidade de São Paulo

Prof. Dr. DANILLO ROBERTO PEREIRA (Participaçao Virtual)
Matemática e Computação / Unesp - Presidente Prudente - FCT

Presidente Prudente, 24 de maio de 2024

Acknowledgments

I am deeply grateful for the support and strength I have received during this challenging period. First and foremost, I thank God for giving resilience upon me throughout.

My wife, Nadia, has been my pillar of strength, standing by me during the toughest phases of my doctoral studies and reinvigorating my spirit whenever I faltered.

My family's constant encouragement and unconditional support have been invaluable to me. Additionally, I am thankful for Maila's friendship and support, which have been a great comfort to me. I thank Rogério for his mentorship and guidance over the past decade.

You all have played a crucial role in my journey, and I am profoundly thankful.

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES)

Resumo

As abordagens de Testes de Software Baseados em Busca (SBST) para geração automática de casos de teste têm melhorado a eficiência dos testes otimizando os tempos de desenvolvimento. No entanto, a capacidade dessas suítes de teste geradas automaticamente para identificar bugs ainda precisa ser aprimorada, uma vez que, na maioria das vezes, o processo de geração de casos de teste enfatiza a cobertura de áreas de código que são mais suscetíveis a bugs. Mesmo as tentativas de usar a previsão de defeitos para identificar seções problemáticas do código falham ao depender de abordagens de modelo único, que não conseguem abordar as complexidades únicas de diferentes projetos de software. A incorporação de modelos de previsão em conjunto no SBST poderia aumentar significativamente a capacidade das suítes de teste para detectar bugs. Apresentamos uma pesquisa empírica sobre o Algoritmo de Ordenação de Muitos Objetivos com Previsão em Conjunto (EPMOSA), uma abordagem SBST inovadora para gerar suítes de teste usando alocação de tempo dinâmica informada por modelos de previsão de bugs em conjunto â Alocação de Orçamento Previsivo em Conjunto para Testes de Software (EP-BAST). Os resultados experimentais em projetos do benchmark Defects4J indicam que o uso de modelos de previsão em conjunto melhora a eficaácia de detecção de bugs das suítes de teste geradas automaticamente.

Palavras-chave: Ciência da computação. Engenharia de Software. Inteligência artificial.

Abstract

Search-based Software Testing (SBST) approaches for automatic test case generation have improved testing efficiency by optimizing development times. Nevertheless, the capacity of these automatically generated test suites to identify bugs still needs to be improved since, most of the times, the test case generation process emphasizes coverage over code areas that are more susceptible to bugs. Even attempts to use defect prediction to identify problematic code sections fall short when relying on single-model approaches, which fail to address the unique complexities of different software projects. Incorporating ensemble prediction models into SBST could significantly enhance the ability of test suites to detect bugs. We present empirical research on Ensemble Predictive Many-Objective Sorting Algorithm (EPMOSA), a novel SBST approach to generate test suites using dynamic time budget allocation informed by ensemble bug prediction models – Ensemble Predictive Budget Allocation for Software Testing (EP-BAST). Experimental findings on projects from the Defects4J benchmark indicate that employing ensemble prediction models improves the bug detection efficacy of automatically generated test suites.

Keywords: Computer Science. Software Engineering. Artificial Intelligence.

List of Figures

List of Tables

Summary

CHAPTER

1

Introduction

In 1948, Frederic Calland Williams[1] wrote the first computer program (Croarken, 1993). This program, designed to calculate the highest factor of the integer 218, was stored in the memory of a digital computer, distinguishing it from earlier efforts where electronic devices had to be physically rewired to change their programming (Croarken, 1993). This advancement represented an important shift towards the software-driven approach that characterizes contemporary computing. After that, John Tukey introduced the term *software* in 1958 (Tukey, 1958).

The creation of the term *software* allowed us to differentiate between the physical components of a computer system (hardware) and the programmed instructions that direct the computer's operations (software), further solidifying the conceptual foundation of computer programming as a distinct discipline. This evolution from mechanical algorithms to software underscores the rapid advancements in technology and theoretical understanding that have shaped the field of computer science (Foster, 2001, Le Goues et al., 2010).

Since its creation, the software has been a transformative tool, enabling the achievement of previously unimaginable goals. Its impact is evident across healthcare, education, communication, and industry sectors. Software's ability to process, analyze, and manage data rapidly and at scale has introduced efficiencies and innovations that have significantly enhanced human capabilities, inspiring a new era of possibilities.

Despite technological progress, software development remains a fundamentally human task. Even with the advent of generative AI, the manual part of the development process continues to be crucial for addressing the nuanced and creative demands of software development. This recognition of the human element in software development is a testament to the unique skills and insights that humans bring to the table, underscoring their indispensable role in the process.

Grace Murray Hopper discovered the first computer bug on September 9th, 1947, when she found a moth inside the Harvard Mark II computer (O'Regan, 2013). The term *bug* in computer science refers to a failure in a program that causes an unexpected result or crash. Since Murray

[1] https://history.computer.org/pioneers/williams.html

first used it, bugs have been the cause of several significant incidents in history.

The Child Support System introduced by the UK government in 2004, known as CS2, was designed to improve the Child Support System. However, it faced numerous challenges and cost around £450 million. The system could have been better designed, tested, and implemented, causing over 1000 problems and 3000 IT incidents weekly. This resulted in a $153 million financial loss for Electronic Data Systems (EDS) and disrupted the management and distribution of child support services for the Child Support Agency (CSA) (Rohde, 2004). On June 4, 1996, the European Space Agency's Ariane 5 rocket, launched with a budget of $8 billion, experienced a catastrophic explosion just 40 seconds after liftoff. This failure was attributed to an integer overflow, a prevalent problem in computer programming. Specifically, the system attempted to fit a 64-bit numerical value into a 16-bit capacity, leading to the disastrous outcome (Dowson, 1997). These are only two examples of how software bugs can impact real life.

In the discipline of software engineering, two crucial practices ensure the quality and functionality of software products: *validation* and *verification* (Schmidt, 2013). Validation is the process that confirms whether the software meets its business requirements, addressing the question, *"Does the software fulfill its intended purpose?"*. It involves assessing the software's effectiveness in solving real-world problems. Verification, on the other hand, relates to the technical correctness of the software. It evaluates whether the software has been constructed correctly according to the specified design and technical requirements, asking, *"Is the software built right?"*. This process thoroughly examines the software's code, architecture, and design to ensure it adheres to predefined specifications and standards. Verification is fundamentally concerned with the internal workings of the software, ensuring that it operates without errors and behaves as expected under various conditions.

Validation and verification form the cornerstone of software testing, each addressing distinct aspects of software quality (Schmidt, 2013). Validation focuses on the software's external attributes and alignment with user expectations, while verification concentrates on its internal technical integrity. These practices are essential for achieving high-quality software that functions correctly and meets users' needs and requirements.

Dijkstra famously asserted that while testing can reveal the presence of bugs in software, it cannot conclusively prove their absence (Dijkstra, 1972). The foundations of Dijkstra's argument lie in the impracticality of exhaustive testing due to the vast or potentially infinite permutations of inputs and configurations a software system can encounter. Given this limitation, the primary objective of software testing shifts from attempting to prove software correctness to trying to uncover as many errors as possible.

Trying to uncover as many errors as possible aims to enhance the reliability and confidence in the software's functionality, acknowledging that absolute certainty of correctness is unattainable through testing alone (Rao et al., 2013). The essence of this perspective is not to diminish the value of testing but to highlight its role in error detection rather than as a definitive proof of software integrity. A clear comprehension of this concept is essential, as it influences the development of testing techniques and the expectations regarding the results of testing activities.

With the increasing importance of software systems in many industries, the consequences of software failures have gotten increasingly serious (Kizza, 2023). As a result, software testing has become a crucial activity in the development process (Leloudas, 2023). Nevertheless, this more significant focus on testing brings a substantial financial and temporal burden on the development process.

Testing operations use almost 50% of the resources allocated for software development, according to estimates (Kushwaha and Misra, 2008). The significant investment can be mainly attributed to two sources: (1) the challenge of verifying software correctness arises from the complex and multidimensional structure of software systems, and (2) ensuring that the software operates as intended under all possible scenarios is a daunting task (Kushwaha and Misra, 2008). Furthermore, the traditional approach to software testing mainly relies on manual procedures, where human testers are responsible for developing, executing, and evaluating test cases.

The need for manual intervention not only slows the speed of the test process but also increases the likelihood of human mistakes, resulting in incomplete or erroneous testing results (Bezbaruah et al., 2020). The use of manual testing approaches, combined with the inherent complexity of evaluating software performance, greatly contributes to the substantial expenses associated with software testing in terms of financial costs and time effort (Gupta and Bathla, 2022).

Search Based Software Testing (SBST) techniques automate the test case generation process, mostly focusing on maximizing code coverage (Zhou et al., 2023, Semujju et al., 2023, Neelofar et al., 2023). However, researchers found that code coverage does not necessarily correlate with effective bug detection (Kochhar et al., 2017, Andrade et al., 2020, Hossain et al., 2023, Barani et al., 2023). DynaMOSA, a SBST technique, could only identify an average of 22% of bugs in the Defects4j dataset within 30 seconds per class, highlighting a significant gap in bug detection capabilities (Just et al., 2014, Panichella et al., 2018).

Perera et al. (2023a) proposes enhancing Search-Based Software Testing (SBST) approaches by incorporating defect prediction data alongside traditional coverage metrics. The study simulates outputs from theoretical defect prediction models, demonstrating that integrating bug prediction information can enhance the bug detection capabilities of test suites generated through SBST. Despite that, there is no empirical evidence concerning the efficiency and efficacy of SBST guided by actual bug prediction models.

Therefore, we hypothesize that the selection of the prediction model significantly influences the outcomes of SBST, even when using models that exhibit acceptable performance (above 75% precision and recall, as suggested by Perera et al. (2023a)). We posit that the choice of prediction model affects the bug identification rate of the generated test suites, suggesting that not all models yield equivalent results in practical applications. This hypothesis underscores the importance of careful model selection in optimizing SBST for improved bug detection efficiency and effectiveness.

In this dissertations, we present empirical evidence demonstrating that ensemble bug prediction models significantly enhance the performance of Search-Based Software Testing (SBST)

compared to single models. Ensemble models were chosen because they combine the strengths of multiple algorithms, leading to more robust and accurate predictions than single models. Building on this evidence, we introduce the Ensemble Predictive Many-Objective Sorting Algorithm (EPMOSA), a novel SBST approach built upon the DynaMOSA algorithm, designed to dynamically adjust the allocation of time budgets based on the expected likelihood of bugs during execution. EPMOSA uses ensemble machine learning models to estimate the possibility of bugs in software components, strategically directing testing efforts towards areas that are less covered by existing tests and more likely to contain bugs. This strategic focus allows for more efficient use of testing resources and improves the chances of uncovering critical issues.

The empirical evaluation of EPMOSA provides valuable insights into its adaptability and effectiveness across different software projects. We conduct a series of experiments to compare EPMOSA with the state-of-the-art DynaMOSA approach (Panichella et al., 2018), applying it to four open-source applications from the Defects4J dataset: *commons-lang*, *commons-codec*, *joda-time*, and *commons-compress*. In Section 4.2.1, we provide the rationale for selecting each of the subject systems. The experiments were designed to assess the performance of EPMOSA under various testing scenarios and with different types of software.

The experimental setup includes the use of multiple ensemble prediction models:

- Bagging Meta-Estimator (BME)
- Gradient Boosted Trees (GBT)
- Random Forest (RF)
- Stacking Classifier (SC)
- Voting Classifier (VC)

Additionally, we selected single prediction models for performance comparison with ensemble models:

- Classification and Regression Trees (CART)
- k-Nearest Neighbors (KNN)
- Linear Discriminant Analysis (LDA)
- Logistic Regression (LR)
- Naive Bayes (NB)
- Support Vector Machines (SVM)

These models are rigorously tested to determine their effectiveness in enhancing SBST's bug detection capabilities, providing a comprehensive view of how different prediction strategies impact the testing process's efficacy.

1.1 Motivation

de Campos (2017) discusses a compelling study conducted using *Google Guava*, a popular open-source *Java* library on GitHub[2]. This library offers additional functionalities to *Java* programs, including new collection types like `multimap`[3], APIs for concurrency and string processing. In its version 20, a new package called `graph`[4] was introduced in *Guava*. This package was created to handle graph-structured data, which is especially important for modeling relational data structures, such as the representation of airports and the airline routes that connect them.

The fundamental structure provided by the `graph` package is composed of *nodes* (or vertices) and *edges*, where each edge serves as a link between two nodes. The `graph` package offers three types of graphs to accommodate diverse data modeling needs:

1. **`Graph`**: This type represents a basic graph form where edges are considered simple, undirected connections without any associated information or direction.

2. **`ValueGraph`**: In a `ValueGraph`, edges have values associated with them, enabling the representation of weighted or valued connections between nodes, thus offering details about relationships.

3. **`Network`**: The `Network` graph type uniquely identifies each edge as a different object, allowing for the representation of more complex relationships, including parallel edges and self-loops, thereby providing a comprehensive framework for modeling complex network structures.

On August 23rd, 2017, a problem was identified in the `ValueGraph`[5] interface, which is part of the `graph package`. This interface is designed to allow the manipulation of the graph structures. The issue was documented in a bug report[6]. The lifecycle of this issue, from its reporting to resolution, is visually represented in Figure 1.1.

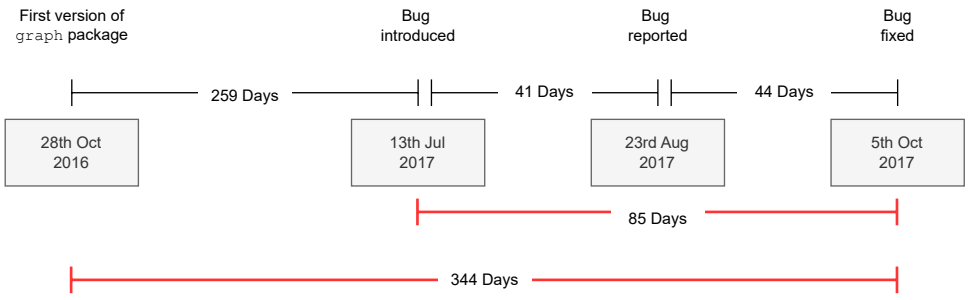

Figure 1.1: Lifecycle of Guava issue #2924 - *Adapted from de Campos (2017)*

[2]https://github.com/google/guava
[3]https://guava.dev/releases/19.0/api/docs/com/google/common/collect/Multimap.html
[4]https://guava.dev/releases/23.0/api/docs/com/google/common/graph/Graph.html
[5]https://guava.dev/releases/22.0/api/docs/com/google/common/graph/ValueGraph.html
[6]https://github.com/google/guava/issues/2924

The `edgeValueOrDefault` method within the `ConfigurableValueGraph` class was identified as faulty. According to the documentation[7], if there is an edge from `nodeU` to `nodeV`, the method should return the value associated with that edge. If no such edge exists, it should return a specified default value. For example, if there are two nodes, `A` and `B`, connected by an edge from `A` to `B` with a value of 5, `edgeValueOrDefault(A, B, 10)` correctly returns 5, the value of the edge. However, since there is no edge from `B` to `A`, `edgeValueOrDefault(B, A, 10)` should return the default value, 10.

On July 13th, 2017, the calls to the method `edgeValueOrDefault` started retuning `null` instead of the default value. This was due to a bug introduced during code refactoring. The bug went unnoticed for 41 days until it was identified, and it took 85 days from the introduction to fix.

This incident underscores software maintenance challenges and manual testing limitations, as the existing tests failed to catch the bug upon its introduction. When the package graph in *Guava* was released, a series of test cases were developed to validate the new functionalities, safeguarding the code against potential defects that could emerge in the future. This practice of accompanying new features with test cases is a standard procedure in software development to ensure the reliability and robustness of the software (Leloudas, 2023). However, despite the developer-written test cases exercising the package graph's functionalities, they failed to test the feature with a sufficiently diverse or appropriate set of inputs. This inadequacy in testing approaches led to an oversight where a specific bug, identified as bug #2924, was not detected earlier.

Manual test case development is labor-intensive and potentially incomplete; automated generation leverages computational methods to produce test cases systematically (Machado and Sampaio, 2010). This process has two primary steps: the *generation of test data* and the *creation of test oracles*. Test data comprises the inputs designed to exercise various aspects of the software, aiming to cover as much functionality as possible (Liu, 2021). Test oracles, or assertions, are criteria used to determine whether the execution of test data uncovers any faults within the software (Ibrahimzada et al., 2022).

The literature on automated test generation has introduced several approaches for generating test data. These include random testing, symbolic execution, and search-based testing. Random testing involves executing the software with inputs generated randomly, without any specific strategy, which can sometimes uncover unexpected errors (Huang et al., 2021, Chen et al., 2022a, Xia et al., 2023). Symbolic execution, on the other hand, systematically explores the control and data paths within the software, aiming to cover all possible execution paths and thereby identify potential faults (Vengadeswaran and Geetha, 2013, Duan et al., 2022, Qiu et al., 2023). Search-based testing employs meta-heuristic search algorithms to create test cases that mimic the characteristics of manually written tests (Gupta and Bathla, 2022). These algorithms optimize the generation of test cases by producing a minimal set of tests that exercise a maximal portion of the code under test. Each technique offers a unique approach to automating the test case

[7] https://guava.dev/releases/22.0/api/docs/com/google/common/graph/ValueGraph.html

generation process, contributing to creating more complete and effective test suites.

EvoSuite (Fraser and Arcuri, 2011) is a tool designed for automatic test case generation, developed explicitly for *Java* classes. It employs a search-based algorithm that iteratively searches through a vast space of possible solutions to find optimal or near-optimal solutions. The primary goal of the test suites generated with EvoSuite is to maximize code coverage, a metric used to measure the extent to which the generated tests cover the source code of the Java classes under test. Code coverage is crucial because higher coverage indicates that more potential bugs or issues have been tested, thereby increasing the software's reliability (Barani et al., 2023).

de Campos (2017) applied EVOSUITE to the *Guava* library, to the first version of the class `ConfigurableValueGraph`, just before the refactoring that introduced the bug discussed previously. In the experiment, EvoSuite was able to generate 39 test cases automatically without manual intervention. Among them, one involved creating a graph with a single node and evaluating the behavior of the `edgeValueOrDefault` function. According to the class's documentation, when querying the value of an edge that connects a node to itself (a scenario set by default with a `null` edge value), the function is expected to return a predefined object, referred to as `obj0`. However, the introduction of the bug #2924 altered the function's behavior, causing it to return `null` instead of `obj0`. This discrepancy between expected and actual outcomes signifies a failure in the test case, highlighting the presence of a bug.

In this scenario, a developer would need to examine the failing test case to determine if it exposes a bug or is no longer relevant, for example, if a requirement definition has been modified. If the test case had been examined, it would have effectively exposed the bug that took longer than 85 days to fix. This means that if a test generation tool had been used earlier, the bug could also have been identified earlier.

This example illustrates the importance of automatic test-generation approaches in modern software development. These methodologies streamline the testing process and enhance its accuracy and efficiency. By automating the generation of test cases, developers can cover more ground in less time, ensuring that more potential bugs are discovered before a product reaches production. This is particularly critical in environments where rapid deployment cycles are the norm, and the cost of a post-release bug can be substantial.

Automatic test generation tools can employ sophisticated algorithms that intelligently prioritize which parts of the software to test based on various criteria, such as bug prediction scores or code complexity. This targeted approach maximizes the use of resources. It reduces the testing burden on human developers, allowing them to focus on more complex testing scenarios and refining the software's functionality.

1.2 Research Problem

Test cases can be considered small programs involving a series of method calls to test the software system's functionalities systematically. The primary objective of testing is to replicate scenarios and behaviors associated with each functionality to verify the software's correct behavior under various settings. For instance, test cases may be developed to evaluate a specific System Under

Test (SUT) functionality or to expose possible defects by exercising edge cases or unexpected inputs (Zhao et al., 2023).

Search-based software Testing (SBST) is a technique for generating test cases in, for example, object-oriented, structural and functional programming environments. SBST usually employs evolutionary algorithms and heuristics to generate test cases that effectively detect bugs automatically. The automatic test case generation process explores the space of all possible inputs to determine those that allow exercising critical paths that could expose bugs.

Although search-based software testing has achieved notable accomplishments, it still faces several challenges (Sar, 2016, Xu et al., 2018, Neelofar et al., 2023, Perera et al., 2023a). An important challenge for SBST approaches is the need to improve their capacity to quickly generate test cases relevant to the SUT's functionality and effectively expose hidden bugs (Perera et al., 2023a).

Most SBST techniques prioritize code coverage as a testing target. Despite achieving high levels of code coverage, SBST techniques exhibit reduced effectiveness in bug detection (Cantoro et al., 2017, Zhu and Jiao, 2019, Perera et al., 2023a). This discrepancy arises because high code coverage does not guarantee the identification of bugs within the tested code.

The simple execution of all lines of code in a software system does not guarantee the identification of existing bugs. The RIP (Reachability, Infection, Propagation) model defines that to identify a bug. It must first be reachable (Reachability), cause a change in the program's state (Infection), and have this change observable at the program's output (Propagation) (Li and Offutt, 2017). This emphasizes the need for SBST approaches to focus on high coverage and creating ways to efficiently reveal bugs, especially semantic bugs and the absence of automated oracles.

Search-Based Software Testing (SBST) tools, such as EvoSuite, use genetic algorithms (GA) and other search approaches to generate test suites for a particular SUT automatically (Fraser and Arcuri, 2011). This type of tool is designed to meet specific testing objectives such as maximizing statement, branch, and method coverage, or a combination of these. A key element in this process is the *time budget*, which sets the duration for which the GA operates before stopping. This critical parameter is the algorithm's stopping criterion, influencing the search's thoroughness for potential test inputs. The assumption is that a more significant time budget allows the GA more time to explore the search space, thereby enhancing the chances of finding optimal test cases that effectively reveal bugs.

Previous studies suggest that allocating a more significant amount of time to SBST can improve its bug detection capabilities by enabling a more thorough exploration of possible test inputs (Sar, 2016, Xu et al., 2018, Neelofar et al., 2023). However, it may only sometimes be practical to increase the time budget. This happens because dedicating much time to testing activities may conflict with project deadlines and resource availability constraints. Therefore, while allocating more time could improve bug detection rates, its implications must be carefully weighed. It is important to balance the benefits of high bug detection rates with the overall software development demands.

Software projects often have thousands of classes, each needing to be tested carefully to

ensure their function reliably and without bugs. For instance, if an SBST tool is set to spend 10 minutes generating tests for each class in a project with 200 files, it would take approximately 33 hours to create a test suite for the entire project (Perera et al., 2023a). Although testing is generally more cost-effective than correcting or finding bugs in later stages of software development (Crosby, 1979), 33 hours to create a test suite can be impractical for most real-world development environments, particularly within Continuous Integration (CI) systems, which aim to integrate code changes rapidly and frequently into shared repositories. Developing efficient testing strategies that align with the limited resources and the need for rapid feedback cycles in software testing is crucial, especially in large-scale projects where the extensive code volume can make traditional SBST approaches too time-consuming.

SBST (Search-Based Software Testing) tools use fitness functions as a fundamental mechanism to evaluate the quality, or fitness, of generated tests. These functions are designed to align with specific testing objectives, such as covering branches, methods, lines, exceptions, and weak mutations within the code. These objectives aim to ensure that the generated tests can thoroughly explore the various paths of the System Under Test (SUT). The choice of a fitness function and its targeted testing goals can significantly affect the behavior captured by the test suite and, consequently, the effectiveness of the testing process in identifying bugs.

Research has examined the effectiveness of Search-Based Software Testing (SBST) in detecting bugs, explicitly exploring how various fitness functions aligned with different testing goals perform in this capacity (Papadhopulli and Meçe, 2016, Sahoo and Ray, 2020, Avdeenko and Serdyukov, 2021). Among various fitness functions used in SBST approaches, those based on branch coverage have been identified as particularly effective. Branch coverage focuses on executing every possible decision point in the program's control flow, facilitating a comprehensive exploration of the program's structure. However, an empirical study using the Defects4J dataset – a compilation of real-world bugs from Java projects – shows that even branch coverage enabled SBST to detect only an average of 25.24% of bugs within a 10-minute time budget (Perera et al., 2023b). This highlights a significant challenge: although SBST can improve the testing process, its ability to detect bugs, even under ideal conditions (i.e., using an effective fitness function with a reasonable time budget), is still limited. *This limitation emphasizes the need for continued research and improvement to enhance the effectiveness of SBST in bug detection.*

Given that buggy code typically represents a small portion of code (Papadhopulli and Meçe, 2016), SBST's search across the whole code base targeting the maximum coverage can lead to inefficient allocation of testing resources (Perera et al., 2020). This inefficiency occurs due to the allocation of effort towards areas of code that are less likely to contain defects. As a result, less emphasis is put on more bug-prone code. Suppose we redirect the SBST efforts towards areas of code that are more likely to contain bugs. Enhancing the bug detection rate of automatically generated test suites is possible in that case.

Defect predictors are sophisticated tools designed to identify sections within the software that are more likely to contain bugs (Pandey et al., 2018). These tools categorize predictions based on the granularity of the analysis, which can range from broader categories, such as software

packages or specific files/classes, to more detailed analyses at the method level.

The granularity levels mentioned, such as package, file/class, and method levels, represent different scopes within the software's structure, with each level providing a varying degree of specificity in pointing to potential defects. The effectiveness of defect predictors relies on analyzing various features associated with the software's code (Kun et al., 2020, Shailza Kanwar and Shrivastava, 2023). These features include metrics like the size of the code, which could indicate potential complexity and thus a higher likelihood of bugs; the complexity of the code itself, which assesses how complex the code is to understand and maintain; the history of changes made to the code, suggesting areas that are frequently modified may be more error-prone; and organizational aspects, which might reflect on the coding practices and standards followed within the team or project (Kun et al., 2020, Shailza Kanwar and Shrivastava, 2023). By evaluating these metrics, defect predictors can efficiently forecast which parts of the software are defective.

The proven ability of defect predictors to locate bugs has made them invaluable in supporting developers during code reviews by highlighting areas requiring closer inspection and enabling a more focused allocation of testing resources on parts of the code likely to be buggy. This strategic application of defect predictors not only streamlines the testing process but also significantly improves the efficiency and effectiveness of identifying and resolving software bugs, instilling a sense of optimism about the potential of these tools in the audience.

Perera et al. (2023a) posits that by identifying areas within the code that are prone to defects (buggy code) through defect prediction techniques, SBST can be more effectively directed towards these areas. By informing SBST about predicted bug-prone regions, the test generation process can focus on generating test cases that cover these specific areas, thereby increasing the probability of detecting bugs.

This approach contrasts with traditional SBST methods that aim for broad code coverage without a specific focus on areas more likely to contain bugs. The idea is to leverage defect prediction to guide SBST towards likely defective regions of the code, thereby improving its efficiency and effectiveness in bug detection. This objective underscores a significant strategic shift in SBST methodology from a generalized search for bugs across the entire codebase to a targeted search in areas identified as high-risk through defect prediction, aiming to optimize testing resources and enhance bug detection outcomes. This shift in approach enlightens the audience about the evolving nature of their field and the potential for more efficient testing strategies.

However, SBST guided by defect prediction has its limitations. One of the primary constraints is the reliance on the accuracy of defect prediction models (Perera et al., 2024). The effectiveness of SBST in detecting bugs is significantly influenced by the recall of the defect predictor, which indicates the model's ability to identify actual defects. A high recall is crucial for minimizing false negatives, but the precision of the model, which suggests the rate of false alarms, needs to have a meaningful, practical significance. This suggests that while defect prediction can enhance the efficiency of SBST, the potential for missed bugs due to false negatives in the predictions remains a concern.

We hypothesize that while defect prediction models effectively identify bug-prone areas of code and can enhance search-based software testing (SBST) techniques, ensemble prediction models surpass their single-model counterparts in accuracy and reliability. We assert that ensemble models, by leveraging multiple learning algorithms, provide a robust and comprehensive framework for defect prediction. These models integrate various predictors, enabling the detection of complex patterns and anomalies that single models may overlook. This integration elevates the precision of defect predictions and mitigates the variance and bias often associated with individual models. The enhanced reliability of ensemble models is particularly vital in software defect prediction, where the financial and operational costs of undetected defects are significant. We aim to demonstrate that the strategic use of ensemble models in defect prediction will significantly reduce error rates and improve the effectiveness of SBST approaches, thereby optimizing software testing. Therefore, we formulate the main research objective of this dissertations as to *investigate the impact of different types of bug prediction models in guiding SBST techniques*.

1.3 Research Objective

As discussed previously, most SBST techniques focus on code coverage. However, high code coverage does not guarantee the detection of all potential bugs within the software. Concentrating the search efforts of SBST on areas of code that are more prone to bugs often identified through historical data, defect prediction models, or heuristic analysis significantly increases the likelihood of uncovering defects (Perera et al., 2023a). This approach is predicated on the rationale that not all code segments are equally likely to identify bugs. Due to complexity, frequent modifications, or other factors, some areas are more bug-prone. By directing the search towards these areas, SBST can use testing resources more efficiently, enhancing the probability of detecting bugs and optimizing the testing process. This strategic concentration on buggy code segments represents a refined application of SBST, where the focus is shifted, aligning the testing efforts with the areas where they are most needed and likely to yield impactful results in bug detection.

However, we argue that not all defect predictors can enhance SBST techniques as desirable. Ensemble prediction models, which can leverage multiple predictors and learning algorithms, prove more effective in this context than single prediction models. The strength of ensemble models lies in their capability to aggregate insights from diverse predictive algorithms, resulting in a more accurate and comprehensive assessment of which code areas are likely defect-prone. This refined accuracy is crucial for SBST, ensuring that the testing efforts are concentrated and directed at the software's most relevant and potentially problematic segments.

Further, ensemble models reduce the likelihood of overfitting – a common issue with single prediction models where the model is too closely fitted to the limited data points on which it was trained. This reduction in overfitting means that ensemble models are generally more robust and reliable when applied to different or unseen data sets, thus enhancing the adaptability and effectiveness of SBST techniques across various software applications (J. Harikiran, 2023).

In this dissertations, we aim to generate test suites with improved bug detection rates at class level in Java software projects. We use ensemble bug prediction models to identify patterns, derive defectiveness, and predict the system's most bug-prone code areas. We study how ensemble predictors can be used to improve the performance of SBST approaches to increase the bug detection rate of test suites.

To achieve our primary research objective, we define two research targets: (1) Develop EP-BAST, a new ensemble predictive time budget allocation strategy to classes for test generation based on its bug proneness. (2) Develop EPMOSA, an SBST approach built upon DynaMOSA algorithm that uses an ensemble predictive time budget allocation strategy to guide the search process to likely defective code areas.

1.4 Research Contributions

This section lists the contributions produced in this research to knowledge and practice. All artifacts used or generated for this dissertation is available in the EP-MOSA Experiments Repository.

1.4.1 Contributions to Knowledge

1. We demonstrate that ensemble bug prediction models can be used to guide SBST in the generation of test suites that are effective and efficient through time budget allocation in a resource-constrained environment. The proposed solution integrates class-level defect prediction and SBST by assigning higher time budgets to highly likely defective classes.

2. We demonstrate that not all bug prediction models perform well when guiding the SBST test generation process. Specifically, we show that ensemble models outperform single models across different subject projects.

1.4.2 Contributions to Practice

1. We developed EP-BAST (Ensemble Predictive Budget Allocation Strategy for Testing), a novel time budget allocation approach designed specifically for Search-Based Software Testing (SBST). EP-BAST harnesses the predictive power of ensemble machine learning models to dynamically allocate time budgets across various components of a software project. By integrating the probability of defect presence derived from ensemble models, EP-BAST intelligently directs testing resources towards areas most likely to contain bugs, ensuring that these critical sections receive adequate adequate testing time.

2. We developed EPMOSA, a novel SBST technique guided by the EP-BAST time budget allocation strategy. EPMOSA uses EP-BAST time budget information to dedicate most of its efforts to the classes with higher bug probability.

3. We developed a set of bug prediction models for the chosen subject systems (Section 4.2.1), that are used in the conducted experiments.

4. We developed a set of datasets specifically designed for training bug prediction models. These datasets encompass the subject system and include various features pertinent to effective bug prediction.

In Figure 1.2 we show an overview of the main contributions to practice we developed in this dissertation.

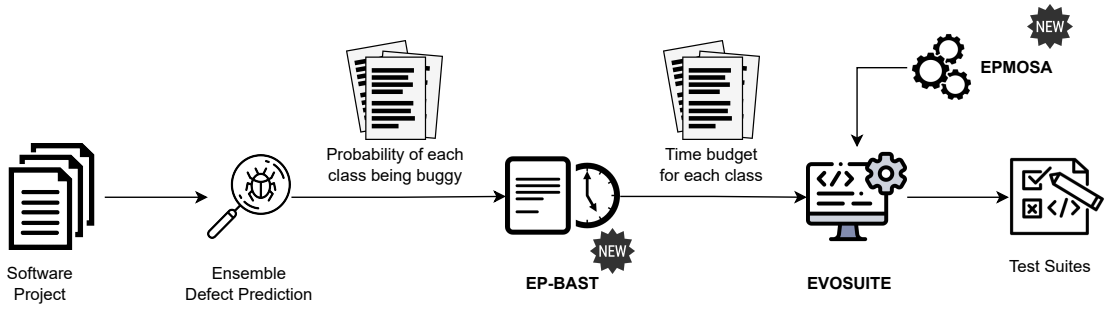

Figure 1.2: Contributions overview

1.5 Dissertations Structure

This dissertation is organized into a structure that guides the reader through the detailed investigation we have done, focusing on various testing and bug prediction strategies. In this chapter, *Introduction*, we described the foundational context of our research, highlighting the motivation behind this study, defining the research problem and objectives, and outlining the contributions to both knowledge and practice. The structure of the dissertation is also summarized here.

Chapter 2, *Literature Review*, systematically maps existing literature on bug research, presenting an in-depth review of methodologies for bug localization, analysis, patch generation, and prediction. It concludes with a discussion on the threats to validity and final considerations.

Chapter 3, *Background and Related Work*, delves deeper into Search-Based Software Testing (SBST) and software defect prediction, presenting different approaches and models, from single prediction models to ensemble prediction models.

In Chapter 4, *Methodology*, the research design and strategy are detailed, followed by a description of the experimental framework, data collection methods, and benchmarks used for comparison in test case generation with SBST approaches.

Chapter 5, *Time Budget Allocation*, introduces the formulation of the time budget optimization problem in software testing, specifically focusing on the Ensemble Predictive Budget Allocation for Software Testing (EP-BAST) and its use in EPMOSA.

Chapter 6, *Guiding Test Case Generation with Ensemble Predictive Models*, presents the empirical evaluation of the Ensemble Predictive Multi-Objective Sorting Algorithm (EPMOSA), comparing its efficiency and effectiveness against the chosen benchmark and discussing the associated threats to validity.

The final chapter, *Conclusions and Future Work* (Chapter 7), synthesizes the findings of this work and offers recommendations for future research in the field. This concluding chapter not only encapsulates the significant insights derived from the research but also sets the stage for further exploration and development in the area of bug research and software testing.

CHAPTER

2

Literature Review

The purpose of this chapter is to provide a comprehensive review of the existing literature relevant to the research on automatic test case generation and bug prediction in software engineering. This chapter systematically maps the current state of knowledge, methodologies, and tools that have been proposed to address software bugs. The literature review is structured to cover key areas such as bug localization, analysis of bugs and bug fixes, automatic patch generation, and bug prediction. By examining these topics, the chapter aims to identify the strengths and limitations of existing approaches, highlight significant research contributions, and uncover gaps that this dissertation seeks to address. The systematic mapping process employed in this review ensures a rigorous and thorough analysis of the literature, providing a solid foundation for the subsequent development and evaluation of new methodologies in automatic test case generation guided by ensemble bug prediction models.

2.1 Systematic Literature Mapping

The goal of software engineers is to write good software. Still, the software development process is complex and prone to errors, which can negatively impact stakeholders' satisfaction and signal low software quality even with great efforts (Saha et al., 2014). Thus, it is critical to understand the fundamentals of software bugs.

Researchers are believed to have extensive knowledge about introducing and removing bugs in software systems (Inozemtseva, 2015). However, this assumption might not be accurate. A study by Inozemtseva (2015) identified a significant need for more empirical data to support the understanding of causes and subsequent fixing of bugs. To fill this gap, an in-depth investigation was carried out to explore the current approaches used by the scientific community in dealing with software bugs. This mapping offers a thorough and quantitative evaluation of the growth in knowledge of software bugs over the past years.

According to our investigation, research on software bugs typically focuses on two topics: (1) the root causes of bugs and (2) strategies for bug fixing. Within these topics, they generally

cover a specific set of subjects. These include the identification, localization, and correction of software bugs. The human factors contributing to the root causes and the testing approaches to expose bugs are also studied. Other significant subjects are predicting bugs, allocating bugs to the right developers for resolution, and examining bug reports to identify the reasons behind code changes that caused bugs.

In the past years, the research community has shown a significant increase in interest in software bugs with the development of different approaches, methods, and concepts. Numerous studies have been published in this field. Nevertheless, systematic mappings frequently focus on particular subjects, such as automated fault localization or bug prediction, while neglecting more extensive investigations. More systematic studies are needed to summarize the significant aspects of software bugs comprehensively. This systematic mapping seeks to address this gap by thoroughly analyzing studies published between 2015 and 2024, focusing on the most relevant aspects of software bugs.

This study begins by defining a systematic mapping protocol, which includes a search strategy, specific inclusion and exclusion criteria, a selection process, and a data extraction and synthesis strategy. The systematic mapping presented in this chapter also examines the research productivity, demographics, and emerging trends within the domain of software bug research.

This systematic mapping is structured as follows: It begins with a description of the mapping protocol, which includes details about the research method, the definition of research questions, the study selection process, and the inclusion/exclusion criteria. Each subsequent section is organized according to a classification schema defined in Section 2.1.1.4, covering distinct subjects of software bug research. These subjects are (1) Automatic Patch Generation, (2) Bug Localization, (3) Bug Prediction, and (4) Analysis of Bugs and Bug Fixes with Various Approaches. The conclusions are presented in Section 2.4.

2.1.1 The Systematic Mapping Process

In this thesis, we conducted a systematic mapping study following the guidelines and procedures outlined by Kitchenham et al. (2009). The authors have provided a comprehensive framework for conducting systematic mappings in software engineering, which we have adopted to ensure rigor and thoroughness in our research. The provided guidelines involve several stages, each designed to enhance the reliability and validity of the findings. A visualization of the stages followed in our systematic mapping can be found in Figure 2.1. This figure illustrates each phase of the mapping process, offering insight into how data was collected, analyzed, and synthesized to support the conclusions drawn in this thesis.

Our systematic mapping consists of seven steps: (1) formulation of the research questions, (2) identification of relevant research, (3) paper screening, (4) assignment of keywords to abstracts, (5) development of a classification scheme, (6) identification of the most relevant literature based on the h-index of each classification category, and (7) extraction of data and mapping. Every step in the process produces a result; the outcome is the systematic mapping.

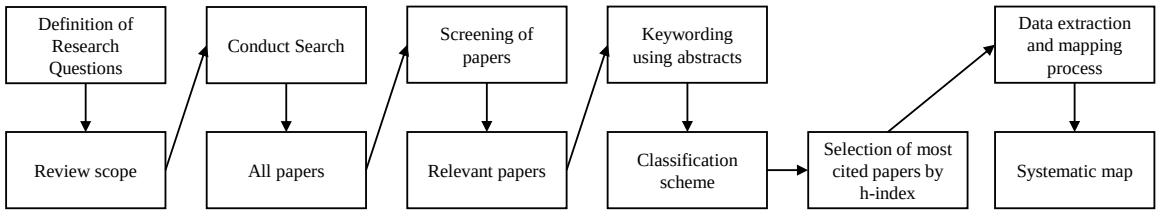

Figure 2.1: The systematic mapping process

2.1.1.1 Definition of Scope

The main objective of a systematic mapping is to offer an in-depth and comprehensive summary of a research field, highlighting the critical quantitative and qualitative characteristics of the research and its outcomes (Mujtaba et al., 2008). In our systematic mapping study (SMS), we provide an overview of the different approaches, strategies, and tools proposed for addressing software bugs[1]. Furthermore, we try to identify the main publication venues where research related to software bugs is published, evaluate the influence of citations, and analyze the geographical distribution and frequency of these publications. The objective is to identify patterns and trends in this field. The primary research question of this mapping study is formulated as

***RQ**: How are software bugs addressed in Software Engineering research?*

This question is broad enough to investigate two main aspects of bug research: (1) the most frequently studied topics and (2) the main characteristics of current publications. To thoroughly explore these aspects, we have defined three sub-research questions (SRQs 1, 2, and 3) focusing on the first aspect and an additional research question (SRQ4) dedicated to the second aspect. Table 2.1 presents these research questions and their respective objectives.

SRQ4 is addressed by examining the following: (1) Publication trends, which assesses the pattern of publications over the past years. (2) Publication venue, which identifies where the most significant studies have been published. (3) Citation impact evaluates the number of citations each selected study has received. (4) Geographical distribution, which determines the origin of the studies.

2.1.1.2 Searching for Primary Studies

We obtained the primary studies by applying search strings to the scientific databases in Table 2.3. We defined three electronic databases as the primary sources for relevant research. The search excluded *Web of Science* because it duplicates the indexing of both *IEEE Xplore* and *ACM*, which are already included in the search scope.

The search string was optimized to function effectively across multiple databases while preserving semantics. The structure of the search query for each database is specified in Table 2.4. These search strings were designed based on the concepts of Population, Intervention,

[1]The term bug is used as a synonym of a defect, failure, error, and fault

Table 2.1: Research questions

SRQ number	Sub-Research Question	Objective
SRQ1	What topics/issues are investigated?	Determine which topics or issues related to bugs in software systems are most frequently investigated and how research evolved.
SRQ2	What types of contributions have been made?	Identify the most common contributions in the field, such as frameworks, techniques/methods, tools, models, etc.).
SRQ3	What are the key findings and the research challenges identified, along with directions for future work?	Summarize the main findings and identify research challenges from the selected studies, providing guidance on potential topics for future research.
SRQ4	What are the demographic characteristics of the studies considered relevant?	To highlight the chronological and geographical distributions of studies within the software bug research domain, identify relevant publication venues, and pinpoint the most influential studies based on citation impact.

Comparison, and Outcome (PICO), as recommended by BA and Charters (2007), and were customized to match the research questions of this study. The search includes the specific terms in Table 2.2.

Table 2.2: PICO definitions

PICO aspect	Terms
Population	software, program, system
Intervention	localization, patch, fix, detection, prediction
Comparison	**As this work is a systematic mapping and we are not interested in analyzing and comparing specific methodologies/tools/technologies/procedures, we do not define the comparison terms*
Outcome	technique, approach, tool, framework

The context for the search strings was specified to include terms related to software bugs, such as *"faults"*, *"errors"*, *"failures"*, *"defects"*, *"flaws"*, and *"anomalies"*. The type of experimental design we want to cover in the SMS was defined to focus on empirical methods. The strategy to develop the final search string involved a five-step process: (1) deriving significant terms from the research questions, (2) identifying alternative spellings and synonyms for these significant terms, (3) checking for keywords in any relevant papers already acquired within the research domain, (4) using the boolean operator `OR` to include alternative spellings and synonyms, and (5) applying the boolean operator `AND` to connect the significant terms related to the population, intervention, and outcome.

2.1.1.3 Criteria and Process for Screening Papers: Inclusion and Exclusion of Relevant Studies

A set of inclusion and exclusion criteria, detailed in Table 2.5, was used to select papers pertinent to the research questions. The selection process involved three steps. (1) Initially, papers

Table 2.3: Electronic databases

Database name	Link
ACM	http://dl.acm.org/
IEEE Xplore	http://ieeexplore.ieee.org/
Scopus	http://scopus.com/

Table 2.4: Search strings for each database

Database	Search String
ACM	(("software" OR "program" OR "system") AND ("code" OR "coding" OR "programming") AND ("bug" OR "fault" OR "error" OR "failure" OR "defect" OR "flaw" OR "anomaly") AND ("identification" OR "localization" OR "detection" OR "prediction" OR "debugging" OR "debug") AND ("repair" OR "correction" OR "healing" OR "fix" OR "patch" OR "bug-fix") AND ("technique" OR "method" OR "approach" OR "tool" OR "framework" OR "procedure") AND ("quality" OR "reliability" OR "maintenance")) NOT ("survey" OR "systematic literature review" OR "position paper" OR "literature review" OR "opinion" OR "student" OR "professor" OR "a bibliography" OR "teaching"))
IEEE Xplore	(("software" OR "program" OR "system") AND ("code" OR "coding" OR "programming") AND ("bug" OR "fault" OR "error" OR "failure" OR "defect" OR "flaw" OR "anomaly") AND ("identification" OR "localization" OR "detection" OR "prediction" OR "debugging" OR "debug") AND ("repair" OR "correction" OR "healing" OR "fix" OR "patch" OR "bug-fix") AND ("technique" OR "method" OR "approach" OR "tool" OR "framework" OR "procedure") AND ("software quality" OR "reliability" OR "maintenance")) NOT ("survey" OR "systematic literature review" OR "position paper" OR "literature review" OR "opinion" OR "student" OR "professor" OR "a bibliography" OR "teaching"))
Scopus	(("software" OR "program" OR "system") AND ("code" OR "coding" OR "programming") AND ("bug" OR "fault" OR "error" OR "failure" OR "defect" OR "flaw" OR "anomaly") AND ("identification" OR "localization" OR "detection" OR "prediction" OR "debugging" OR "debug") AND ("repair" OR "correction" OR "healing" OR "fix" OR "patch" OR "bug-fix") AND ("technique" OR "method" OR "approach" OR "tool" OR "framework" OR "procedure") AND ("software quality" OR "reliability" OR "maintenance")) AND NOT ("survey" OR "systematic literature review" OR "position paper" OR "literature review" OR "opinion" OR "student" OR "professor" OR "a bibliography" OR "teaching"))

were retrieved from designated databases using the defined search string, yielding 2428 papers as indicated in Table 2.6. (2) Next, we reviewed titles and abstracts to filter to 444 relevant studies. (3) Finally, papers with a high h-index, based on citation count, were specifically targeted to ensure the inclusion of the most influential studies. We determined the h-index for a paper by the number of citations equal to or greater than its ranking position, with rankings ordered by decreasing citation count. This allowed categorizing highly cited papers according to the classification scheme outlined in Section 2.1.1.4. These papers are the primary focus of this mapping. We considered the remaining 407 papers in a quantitative analysis of the main characteristics of current publications.

Table 2.5: Inclusion and exclusion criteria

Criteria	Definition
Inclusion criteria	1) From the abstract, we can infer that the paper's focus contributes to research on software bugs. 2) Articles published within nine-year period (2015-2024)[1]
Exclusion criteria	1) The paper falls outside the scope of the software engineering domain, as it does not contribute to the study of software bugs. Terms related to software bugs are merely mentioned in the introductory sentences of the abstract and are not a focal point of the paper's contributions. 2) Studies that do not provide validation of their results 3) Studies written in languages other than English

[1] This period corresponds to the update of the systematic mapping conducted in 2020, covering five years.

Table 2.6: Number of selected papers

Database	Retrieved	Selected by title & abstract	Selected by h-index
IEEE Xplore	111	63	7
ACM	2000	212	18
Scopus	317	169	12
Total	**2428**	**444**	**37**

2.1.1.4 Developing a Classification Scheme: Keywords and Abstracts

The keywording process is essential in establishing the classification scheme to ensure its effectiveness in categorizing all relevant studies. The process consists of two main stages. (1) First, abstracts are carefully examined to determine the keywords and concepts that summarize the paper's contributions and research context. (2) Afterward, these keywords are compared across all papers to understand the different studies. This comparison facilitates the creation of a comprehensive set of categories that correctly reflects the content of the extracted papers. If abstracts do not provide significant keywords, additional analysis should be conducted on the introduction and conclusion sections.

After identifying all relevant keywords, they are categorized based on how they relate and used to define the mapping categories. The classification process, depicted in Figure 2.1, complies with the methodology described by Petersen et al. (2008).

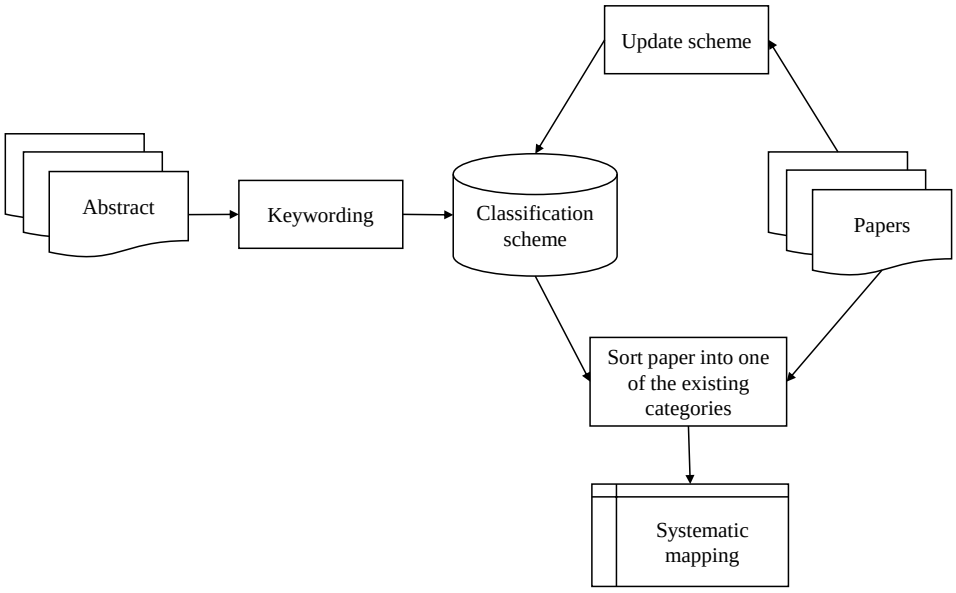

Figure 2.2: Building the classification scheme (Adapted from Petersen et al. (2008))

In our mapping study, five main categories were established: (1) Automatic Patch Generation, (2) Bug Localization, (3) Bug Prediction, (4) Search-Based Software Testing, and (5) Analysis of Bugs and Bug Fixes. The fifth category encompasses various papers that discuss topics related to bugs in software engineering but do not fit neatly into the primary categories (1 through 4). Each category is addressed in a dedicated section of the study, where all relevant research questions are explored and answered.

2.1.1.5 Data Extraction and Mapping of Studies (Systematic Map)

Once the classification scheme is established, the selected papers are sorted into the appropriate categories, initiating the data extraction process. The classification scheme is dynamic and evolves as the analysis of the selected papers progresses (see Figure 2.2). This adaptability allows for introducing new categories or modifying existing ones – such as merging or splitting – as a deeper understanding of the papers is developed.

The analysis starts with the abstracts, followed by the papers' introductions, conclusions, and results sections. The method sections are examined last. After all papers have been categorized, data extraction is performed for each category. The analysis within each category aims to address the research questions and provide a comprehensive overview of the category. The following sections are dedicated to an in-depth discussion of each established category.

2.2 Bug Research in Software Engineering: An Overview

2.2.1 Bug Localization

In this section, we first provide an overview of the current practices in bug localization as documented in the literature. Subsequently, we address the defined research questions specific to the bug localization category.

Bug localization is identified as one of the most resource-intensive activities in software engineering (Gong et al., 2015). Typically, developers perform bug localization manually, which can be tedious, challenging, time-consuming, and prone to errors. Over the past years, numerous models, tools, and techniques have been developed to streamline this process (refer to Table 2.7). These bug localization methods primarily focus on calculating the suspiciousness of program elements, which involves identifying elements potentially associated with a bug. The resulting list of suspicious elements is ranked in descending order of likelihood, allowing developers to prioritize their review according to the rankings. Effective bug localization approaches aim to place highly suspicious elements at the top of the list, enabling developers to pinpoint potential bugs quickly (Gong et al., 2015).

Various classifications for bug localization approaches[2] have been identified. Khatiwada et al. (2018) groups bug localization techniques into two main categories: (1) dynamic and (2) static. Dynamic approaches involve locating bugs by collecting and analyzing program data such as breakpoints or execution traces. These methods require running the program and analyzing the differences in control flow between passing and failing test cases. However, as Khatiwada et al. (2018) notes, executing the program is not always feasible, such as when an error affects multiple code modules, rendering execution impractical.

On the other hand, static approaches typically use information retrieval techniques to pinpoint the faulty code automatically. These methods leverage data from bug reports to identify buggy elements within the code. Advantages of static approaches include not requiring an executable version of the program, being language agnostic[3], and having a low computational cost (Khatiwada et al., 2018).

Gong et al. (2015) categorizes bug localization techniques into four categories: (1) Program slicing approaches focus on analyzing program slices to reduce the search space needed to identify buggy statements. (2) State-altering approaches involve intentionally modifying the program's state to produce incorrect results, thereby altering the control flow to facilitate bug identification and localization. (3) Model-based approaches aim to identify incompatibilities, unexpected interactions, and undesirable behaviors. These approaches infer object behavioral models from execution traces to detect behavior incompatibilities. (4) Statistical analysis approaches locate bug-related statements by comparing statistical differences in program elements between passed and failed test cases.

[2]Here, *approach* is used as an umbrella term for the various types of contributions identified in the analyzed studies: approaches, methods, techniques, tools, etc.

[3]Language-agnostic approaches can be applied to programs written in various languages.

Table 2.7: Works classified as bug localization

Paper/Contribution	Language	Datasets	Type of Contribution	Country	Institution
N/A (Khatiwada et al., 2018)	Lang. Agnostic	AspectJ, Eclipse, JodaTime, SWT, ZXing	Analysis	USA	Louisiana State Univ.
LDoctor (Song and Lu, 2017)	Lang. Agnostic	Apache, Tomcat, Ant, Chromium, GCC, Mozilla, MySQL, Toddler	Tool	USA	Univ. of Chicago, Fireeye Inc.
SDPM (Gong et al., 2015)	C	Siemens (tcas, replace, etc.), flex, grep, gzip	Model	China	Harbin Inst. of Tech.
DeepLocator (Xiao et al., 2017)	Java	AspectJ, Eclipse UI, JDT, SWT, Tomcat	Model	China	City Univ. of Hong Kong
JCHARMING (Nayrolles et al., 2015)	Java	Ant, ArgoUML, dnsjava, jfreechart, Log4j, MCT, pdfbox	Technique	Canada	Concordia Univ.
Gist (Kasikci et al., 2015)	C	Apache, SQLite, Memcached, CppCheck, Curl, Transmission, Pbzip2	Technique	Switzerland	EPFL
MUSEUM (Hong et al., 2015)	Lang. Agnostic	Azureus, SQLite, java-gnome, SWT	Technique	South Korea	KAIST

Xiao et al. (2017) categorizes existing bug localization methods into three groups: (1) Traditional approaches that use program analysis information, such as execution details from passing or failing test cases, to search for bugs. An example is spectrum-based fault[4] localization, which falls under this category. (2) Information retrieval-based approaches that search for and rank buggy files based on a given bug report. These methods employ textual similarities between bug reports and source code files to locate bugs. (3) Machine learning-based approaches, which use models trained to match the topics of bug reports with topics in source code files, aiding in the bug localization process.

Zakari et al. (2019) classifies bug localization approaches into ten distinct categories: (1) Spectrum-based, (2) Statistical-based, (3) Information retrieval-based, (4) Model-based, (5) Data mining-based, (6) Hybrid, (7) Slicing-based, (8) Machine learning-based, (9) Program state-based, and (10) Mutation-based. This classification incorporates the categories identified by Gong et al. (2015) and Xiao et al. (2017) and introduces additional ones, showcasing a comprehensive taxonomy of bug localization methods.

Spectrum-based fault localization (SBFL) is noted as particularly prevalent among these categories, according to Zakari et al. (2019). SBFL techniques determine potential bug locations by analyzing the likelihood of code elements being executed during an error occurrence (Hong et al., 2015). These methods use the outcomes of tests (i.e., pass or fail) and rely on the coverage data from these tests. SBFL is language-agnostic, making it versatile across different programming languages. However, the effectiveness of Spectrum-based techniques can be limited in larger software systems where their accuracy may decrease (Hong et al., 2015).

Statistical-based approaches have proven effective in identifying buggy locations with low computational complexity by comparing statistical differences between program elements executed in test cases (Gong et al., 2015). Interestingly, none of the highly cited papers selected for this study in the bug localization category were classified as Spectrum-based. This observation is attributed to the timeframe covered by the mapping, which only spans the last nine years (2015 to 2024), whereas Zakari et al. (2019) analyzed a broader period (2006 to 2017). Within the most cited papers of this mapping, diverse approaches were identified: one hybrid approach (Song and Lu, 2017), one information retrieval-based approach (Khatiwada et al., 2018), one machine learning-based approach (Xiao et al., 2017), one mutation-based approach (Hong et al., 2015), one statistical-based approach (Gong et al., 2015), and two slicing-based approaches (Kasikci et al., 2015, Nayrolles et al., 2015).

Additionally, bug reproduction is critical for locating buggy code. Bug reproduction techniques are broadly categorized into two types (Kasikci et al., 2015, Nayrolles et al., 2015): (1) *on-field record and in-house replay*, and (2) *in-house crash explanation*. The first category uses code instrumentation to capture program elements at runtime. When a bug manifests for the end-user, these elements are transmitted to developers for reproduction (Nayrolles et al., 2015). While on-field recording and in-house replay techniques are straightforward, they come with limitations. Firstly, code instrumentation can be computationally intensive. Secondly, capturing

[4]In this work, we generalize the term bug to incorporate other related terms, such as a fault, defect, and failure

data during end-user execution may include sensitive information and raise privacy concerns. The second category, in-house crash explanation, utilizes proprietary data to offer insights into potential bug causes, aiding in understanding the bugs. However, it does not facilitate their reproduction or localization (Nayrolles et al., 2015).

From the perspective of bug reproduction, Nayrolles et al. (2015) introduces JCHARMING, an approach that utilizes crash traces combined with model checking to reproduce bugs that have appeared to end-users. Nayrolles et al. (2015) emphasizes that JCHARMING does not require code instrumentation nor access to potentially sensitive information. Instead, it leverages a list of functions generated during an uncaught exception to pinpoint the code statements responsible for a failure, i.e., the buggy statements. Kasikci et al. (2015) also underscores the importance of reproducing bugs to accurately diagnose their root causes and identify the specific program elements involved. Reproducing bugs is deemed crucial for effectively locating and resolving them.

Xiao et al. (2017) propose DeepLocator, a deep learning-based model designed to enhance bug localization by fully leveraging semantic information. The model incorporates a Convolutional Neural Network (CNN) to extract local features and correlate bug reports with the corresponding buggy files. Performance bugs, as noted by Song and Lu (2017), are particularly time-consuming and critical, often not affecting software correctness directly but leading to redundant computations and performance slowdowns. To address this, Song and Lu (2017) developed LDoctor, a tool that identifies inefficient loops in the code, explaining their inefficiencies and guiding developers on optimizing them.

Static bug localization approaches using information retrieval techniques such as the Vector Space Model, Latent Semantic Indexing, and Latent Dirichlet Allocation (LDA) have shown poor performance (Khatiwada et al., 2018). Motivated by enhancing these static approaches, Khatiwada et al. (2018) investigated a new information retrieval method that capitalizes on the semantic attributes of the textual content within software artifacts. This study aims to improve the estimation of textual semantic similarity between source code elements and bug reports, enhancing the effectiveness of bug localization.

Research Questions

SRQ1 What topics/problems are investigated?

Some important topics and bug localization problems were observed in the mapping of the studies depicted in Table 2.7. They are summarized below:

- Many software systems are written in multiple programming languages. These are called multilingual programs (Hong et al., 2015). For example, high-level languages (e.g., Java, Python) provide standard libraries that use legacy code written in other languages (e.g., C language) to interface with the operating system (Hong et al., 2015). When a bug occurs in the interactions between code written in different languages, locating and understanding them is challenging. Current bug localization techniques do not help much in debugging

this bug. Generally, they cannot precisely locate multilingual bugs (Hong et al., 2015).

- Fixing bugs that cause crashes on the client side requires the developer's ability to reproduce them. Reproducing a bug allows us to locate its root causes and fix it. However, developers cannot reproduce some bugs due to the lack of infrastructure or time to observe the bug occurrence. Another problem related to bug reproduction is that the end-users data are usually scarce (Kasikci et al., 2015, Nayrolles et al., 2015).

- When analyzing large and complex systems to find bugs, information retrieval methods such as the Vector Space Model (VSM), Latent Semantic Indexing (LSI), or Latent Dirichlet Allocation (LDA) are usually sub-optimal. These methods are problematic because source code uses a very restricted vocabulary that lacks uniqueness and exhibits high repetition. Restricted vocabulary and repetition lead data-expensive methods such as LDA and LSI to perform poorly when used to locate bugs in source code (Khatiwada et al., 2018). It is possible to combine lexical methods that use text matching and more computationally expensive semantic methods to improve the accuracy of information retrieval methods, as shown by Khatiwada et al. (2018).

- Some specific bugs are more difficult to locate because they do not cause execution crashes or errors that can be directly perceived and traced to their root causes. Performance bugs cause unnecessary performance degradation in software and are common due to the high complexity of modern software. The localization of performance bugs differs from other types of bugs because their only symptom is execution slowness. Thus, to locate performance bugs, it is necessary to identify which code region is inefficient but not necessarily wrong and why (Song and Lu, 2017).

SRQ2 What types of contributions have been made?

Four different contributions were found: analysis, tool, model, and technique, as shown in Table 2.7. It was considered how the authors referred to their work to define types of contribution. In the third column of Table 2.7, it was observed that Khatiwada et al. (2018) is the only paper that contributes with an analysis. SDPM and DeepLocator (Gong et al., 2015, Xiao et al., 2017) are models, and JCHARMING, Gist, and MUSEUM are techniques (Nayrolles et al., 2015, Kasikci et al., 2015, Hong et al., 2015). LDoctor (Song and Lu, 2017) is the only contribution defined as a tool.

RQ3 What are the principal findings and identified research challenges with directions for future work?

- Most studies about bug localization are limited to the academic field. Therefore, most of the identified studies are, in fact, not used in industry because they are not considered sufficient and bias-free. Even though few studies try to fill the gap between academia and industry, more has to be done to apply bug localization techniques in the software industry.

- It identified the use of seven different evaluation metrics:

  1. T-Score (Gong et al., 2015),
  2. Precision (Gong et al., 2015, Xiao et al., 2017),
  3. Recall (Gong et al., 2015, Xiao et al., 2017),
  4. MAP (Gong et al., 2015, Xiao et al., 2017),
  5. MRR (Gong et al., 2015),
  6. Top-N (Gong et al., 2015),
  7. F-measure (Xiao et al., 2017).

  Song and Lu (2017), Nayrolles et al. (2015), Kasikci et al. (2015), Hong et al. (2015) do not clearly define an evaluation metric to analyze the proposed approaches.

- Bug localization techniques can be more accurate if a test suite covers more diverse execution paths. However, it is challenging for large real-world programs to build test cases that exercise several different paths since it is non-trivial to understand and control these types of software systems (Hong et al., 2015).

- It was identified the use of 30 different datasets (see Table 2.7) to validate bug localization techniques. The diversity of datasets can hinder the analysis and comparison of the existing bug localization techniques. However, the diversity of datasets can be used to create a standard bug dataset to analyze and create bug localization techniques.

- Information retrieval methods used in bug localization approaches are still far from achieving adequate accuracy levels for industry and practical applications. They usually achieve high recall with a large number of false positives. Low accuracy levels require developers to manually analyze the outcome of these methods (Khatiwada et al., 2018).

SRQ4 What are the demographic characteristics of the relevant studies?

To answer this SRQ, four aspects of the selected primary studies were examined: (1) publication trend, (2) publication venue (journals and proceedings), (3) citation impact, and (4) geographical distribution.

(1) Publication trend

From 2015 to 2018, 7 publications with many citations were extracted, following the defined methodology. Figure 2.3 shows the evolution of the publications about bug localization. The research activity was observed to be progressive and active; however, as only those papers with high citation impact were selected, only a few publications each year were considered in the mapping. Although our systematic mapping considered 2015-2024, papers with high citation impact were found only in 2015, 2017, and 2018. It is important to highlight that it is not possible

to make a firm and valid conclusion about why no papers published in 2016-2024 had a high citation impact. Generally, despite the number of publications with high citation impact, the research activity shows stable growth, as shown in Figure 2.4.

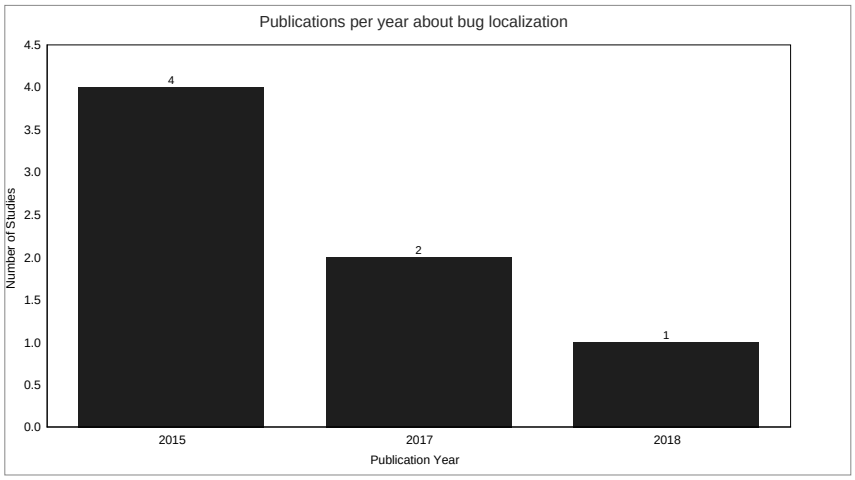

Figure 2.3: Publications per year about bug localization (High citation impact)

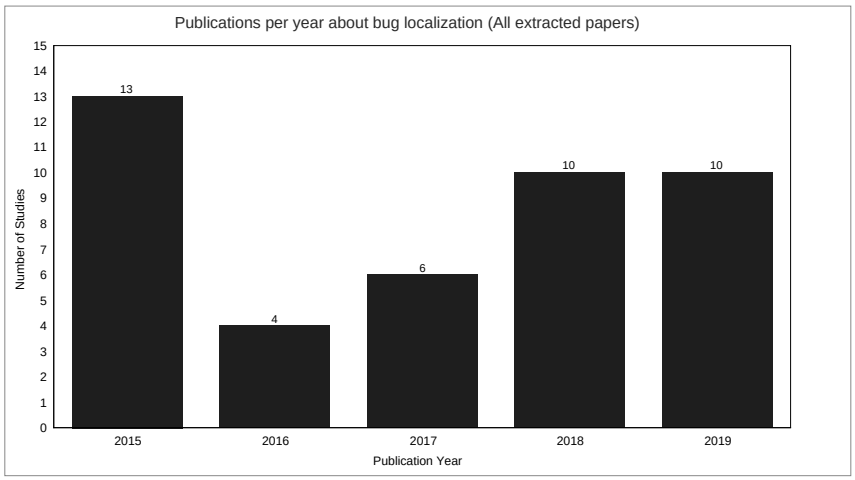

Figure 2.4: Publications per year about bug localization (All extracted papers)

(2) Publication venue

We identified 28 different publication venues that have featured papers on bug localization. However, the publications with the highest citation impact are concentrated within just six venues: five conference proceedings and one journal, as detailed in Table 2.8. Most high-impact publications were from conference proceedings, with only one journal article. Table 2.9 lists the most active journals and conferences, regardless of the citation impact of the publications. The conferences contributing most significantly include the International Conference on Software Engineering (ICSE), Asia-Pacific Software Engineering Conference (APSEC), Asia-Pacific Symposium on Internetware (Internetware), International Conference on Mining Software Repositories (MSR), International Conference on Software Engineering and Knowledge Engineering (SEKE), and International Conference on Artificial Intelligence and Computer Science (AICS). The Journal of Systems and Software stands out as the top contributor among the journals.

Table 2.8: Number of publications per venue and type of publication (High citation impact papers)

Publication Venue	**Type of publication**	**Total**
Information and Software Technology	Journal	2
Symposium on Operating Systems Principles (SOSP)	Proceedings	1
International Conference on Automated Software Engineering (ASE)	Proceedings	1
International Conference on Software Analysis, Evolution, and Reengineering (SANER)	Proceedings	1
Asia-Pacific Software Engineering Conference (APSEC)	Proceedings	1
International Conference on Software Engineering (ICSE)	Proceedings	1

(3) Citation impact

Figure 2.5 displays the citation counts for the most cited papers within the category of bug localization, focusing specifically on those papers that rank highly according to the h-index. In contrast, Figure 2.6 encompasses all bug localization papers that have received at least one citation, irrespective of their h-index. The citation data for each paper was sourced from various publishers and indexing services such as ACM, IEEE Xplore, and Scopus, and these figures are subject to change over time. Notably, seven papers have each reached more than eight citations, as listed in Table 2.7. The total number of citations for all papers with at least one citation, as shown in Figure 2.6, stands at 173, with an average of approximately five citations per paper.

(4) Geographical distribution

Table 2.10 showcases the countries most actively engaged in bug localization research. Analysis of all extracted papers from this domain revealed participation from 14 countries. China leads in activity with 15 published papers, making it the most prolific contributor, followed by

Table 2.9: Number of publications per venue and type of publication (All extracted papers)

Publication Venue (All extracted)	**Type of publication**	**Total**
International Conference on Software Engineering (ICSE)	Proceedings	4
Asia-Pacific Software Engineering Conference (APSEC)	Proceedings	3
Asia-Pacific Symposium on Internetware (Internetware)	Proceedings	3
International Conference on Mining Software Repositories (MSR)	Proceedings	2
International Conference on Software Engineering and Knowledge Engineering (SEKE)	Proceedings	2
Journal of Systems and Software	Journal	2
International Conference on Artificial Intelligence and Computer Science (AICS)	Proceedings	1
International Conference on Computer Science and Software Engineering (CASCON)	Proceedings	1
ACM International Conference on Information and Knowledge Management (CIKM)	Proceedings	1
International Conference on emerging Networking EXperiments and Technologies (CoNEXT)	Proceedings	1
International Workshop on Complex Faults and Failures in Large Software Systems (COUFLESS)	Proceedings	1
International Conference on Evaluation and Assessment in Software Engineering (EASE)	Proceedings	1
International Conference on Frontiers of Information Technology (FIT)	Proceedings	1
Genetic and Evolutionary Computation Conference (GECCO)	Proceedings	1
International Conference on Program Comprehension (ICPC)	Proceedings	1
International Conference on Computing for Sustainable Global Development (INDIACom)	Proceedings	1
India Software Engineering Conference (ISEC)	Proceedings	1
International Conference on Mobile Software Engineering and Systems (MOBILESoft)	Proceedings	1
Annual ACM Symposium on Applied Computing (SAC)	Proceedings	1
Brazilian Symposium on Software Engineering (SBES)	Proceedings	1
Symposium on Applied Computing (SIGAPP)	Proceedings	1
ACM Symposium on Operating Systems Principles (SOSP)	Proceedings	1
USENIX Annual Technical Conference (USENIX)	Proceedings	1
Information and Software Technology	Journal	1
Computing and Informatics	Journal	1
International Journal of Software Engineering and Knowledge Engineering	Journal	1
KSII Transactions on Internet and Information Systems	Journal	1
ACM Transactions on Design Automation of Electronic Systems	Journal	1

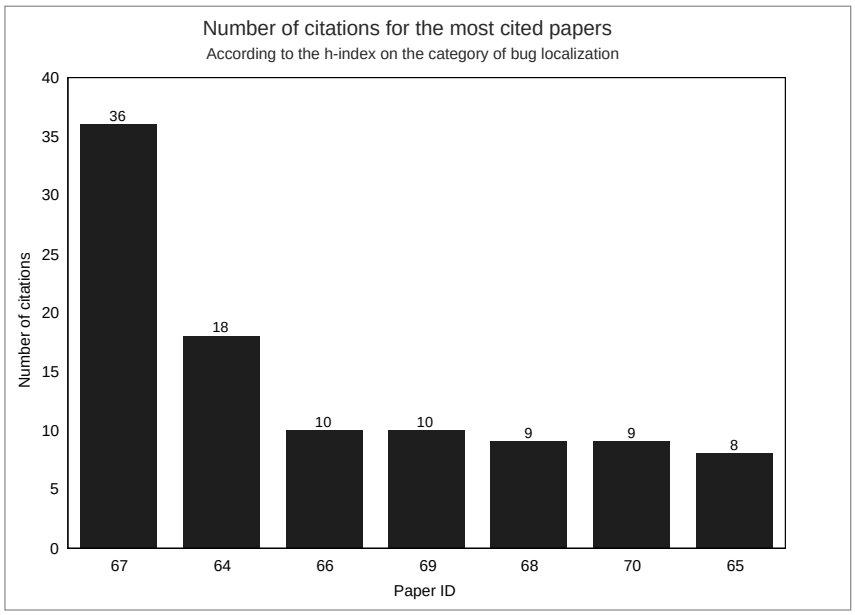

Figure 2.5: Number of citations for the most cited papers on bug localization

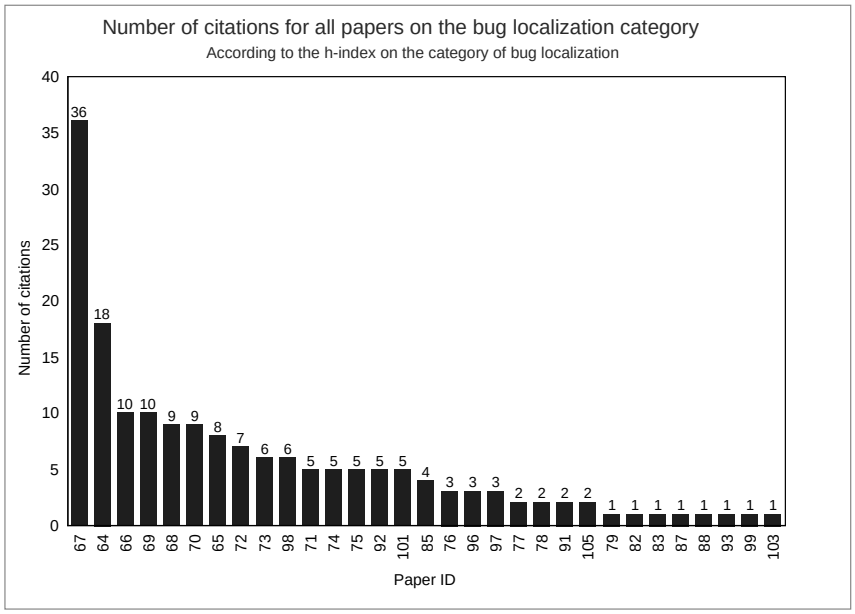

Figure 2.6: Number of citations for all cited papers on bug localization

the United States with six papers. South Korea ranks third, having published five papers. Notably, the selection of the most cited papers, based on the h-index, originates from only 5 of these 14 countries. The United States and China each have 2 of the most cited papers, while Canada, Switzerland, and South Korea each contribute one highly cited paper.

Table 2.10: Most active countries in bug localization research

Country	Number of published papers
China	15
United States	6
South Korea	5
India	3
Canada	2
United Kingdom	2
Brazil	2
Japan	1
Greece	1
Iran	1
Germany	1
Pakistan	1
Bangladesh	1
Belgium	1

2.2.2 Analysis of Bugs and Bug Fixes

This section discusses several topics and problems addressed by 10 of the selected papers (see Table 2.11). These papers were classified as Analysis of Bugs and Bug Fixes. In the following section, each study is briefly described, and then the findings are systematically mapped, answering the four research questions.

Table 2.11: Works classified as analysis and understanding of bugs and bug handling approaches

Papers/Contribution	Type of contribution	Country	Intitution
Zhou et al. (2015)	Analysis	USA	University of California Riverside
Zhong and Su (2015)	Tool	China	Shangai Jiao Tong University
Tantithamthavorn et al. (2015)	Analysis	Japan	Nara Institute of Science and Technology
Soto et al. (2016)	Analysis	Singapore	Singapore Management University
Jaafar et al. (2015)	Analysis	Canada	Queen's University
Ray et al. (2016)	Analysis	USA	University of Virginia
Leesatapornwongsa et al. (2016)	Taxonomy	USA	University of Chicago
Li and Paxson (2017)	Analysis	USA	University of California
Mondal et al. (2017)	Analysis	Canada	University of Saskatchewan
Zampetti et al. (2017)	Analysis	Italy	University of Sannio

Mobile applications are different from desktop applications, so it is their respective bugs. These devices differ on several levels: the mobile platform is newer than the desktop platform.

Applications are built differently since mobile applications have to consider many sensor-, gesture-, and event-driven factors that are not concerns in desktop applications. Also, security and privacy concerns arise in mobile devices, which do not occur in desktop applications. Most research in bugs and bug-fixing activities is focused on traditional software. Zhou et al. (2015) analyze the similarities and differences in bug reports and bug-fixing processes between desktop and mobile platforms. Zhou et al. (2015) found out that the most frequent high-severity bugs are due to compilation and validation failures on desktop applications and concurrency in Android and crashes in iOS. Also, concurrency bugs are prevalent on the mobile platform.

The automatic program repair research domain has been gaining attention due to the necessity of reducing the effort of fixing bugs. A typical automatic repair approach first locates a program's bugs and then executes modifications on that location until the program passes all the test cases (Zhong and Su, 2015). Research in automatic program repair has already produced promising results. However, these positive results are constantly questioned (Zhong and Su, 2015) as empirical studies show that it requires much expertise and time to fix software bugs (Kim and Whitehead, 2006). According to Zhong and Su (2015), the criticism exposes the research community's lack of knowledge of bug fixes' nature. Zhong and Su (2015) propose a BugStat tool that automatically extracts, compares, and classifies bug fixes.

The generate-and-validate class of automatic program repair approaches works by generating many candidate patches using a pre-defined set of mutation operators. Then, these patches are validated using a test case suite (Soto et al., 2016). The mutation operators can be simple statements or entire change templates learned from a corpus of previous bug fixes done by humans (Soto et al., 2016). Soto et al. (2016) observe that a large proportion of works in automatic program repair target programs written in C language, which leads researchers targeting Java programs to compare their approaches with existing approaches implemented for the C language (DeMarco et al., 2014, Kim et al., 2013). However, as pointed out by Soto et al. (2016), Java and C are two very different languages, and the structure of code elements can also be different. Soto et al. (2016) study bug-fixing commits to Java programs to analyze these changes' characteristics, the applicability of previous Java repair templates, and the nature of additions and replacements of Java statements in bug-fixing commits.

Defect[5] models to identify bug-prone software elements. They can be used to predict program elements likely to be buggy and understand the impact of software metrics on software elements' defect-proneness. The accuracy of the results provided by defect models depends on the data quality from which the models are trained (Tantithamthavorn et al., 2015). Defect models are trained using datasets containing data about the relation between bug reports recorded in an Issue Tracking System (ITS) and software elements that must be modified to address the reported bug. The executed modifications are, in turn, recorded in a Version Control System (VCS). Thus, the quality of all the information recorded on ITS and VCS impacts the defect models. Bug reports describing bugs but not classified as such (or vice versa) also impact defect models' performance (Tantithamthavorn et al., 2015). This problem is known as *bug report mislabeling*

[5]here *defect* is used as a synonym of bug

Tantithamthavorn et al. (2015) investigated whether and how mislabeled bug reports can, in fact, impact prediction models.

Classes participating in design patterns and anti-patterns can have strong dependencies with other classes and propagate them. Jaafar et al. (2015) investigate the impact of such dependencies in object-oriented programs by analyzing the relations between the existence of static and co-change dependencies and the fault-proneness, the types of changes, and the types of faults that these classes exhibit. Jaafar et al. (2015) showed that having dependencies with anti-patterns is more fault-prone than others, which is not valid for classes with dependencies with design patterns.

Ray et al. (2016) observed that code in repositories is highly repetitive, and this repetition can be captured by language models. It means that language models can identify some properties of how code is supposed to be, which raises the question of what it means when language models consider a code fragment improbable. To investigate this question Ray et al. (2016) analyzed a large corpus of bug fix commits from several different projects to understand defective code's main characteristics.

Concurrence bugs manifest themselves in a non-determinist form, which makes them difficult to detect, diagnose, and fix (Leesatapornwongsa et al., 2016). Distributed systems are the most prone to concurrence bugs due to their nature. Leesatapornwongsa et al. (2016) offer a comprehensive study on real-world concurrence bugs.

Li and Paxson (2017) observed that bug fixes, in general, have mainly been studied by the software engineering community. However, there has been little effort to understand how to fix specific classes of bugs. Li and Paxson (2017) conducted a large-scale empirical study of security patches. The study analyzes the patch development lifecycle, including investigations about the code base life span of vulnerabilities, the timeliness of security fixes, and the degree to which developers can produce good patches. Also, Li and Paxson (2017) characterized the security fixes compared to other non-security bug patches to explore the complexity of patches for different types of bugs and their impact on the source code.

Code clones can be related to bugs and may propagate through code cloning (Mondal et al., 2017). According to Mondal et al. (2017), if a code fragment contains a temporarily hidden bug, and a programmer makes a clone of that fragment, the bug on the original code is propagated. To know the impacts of code clones on software evolution and maintenance, it is necessary to study bug propagation intensity for code clones. (Mondal et al., 2017) performed an empirical study on bug propagation in code clones; they define two patterns of bug propagation and propose an automatic mechanism to detect these two bug propagation patterns.

Research Questions

RQ1 What topics/problems are investigated?

Analysis of Bugs and Bug Fixes identified several papers, each giving a different perspective on software bugs and dealing with a specific topic/problem. The most interesting discussed

topics can be summarized as follows:

- The difference between bugs and bug-fixing processes in desktop and mobile applications (Zhou et al., 2015).
- The maturity of automatic program repair tools. It is indicated that most existing approaches are not ready to be applied in the industry (Zhong and Su, 2015).
- The differences between automatic program repair approaches according to the base language. Most approaches to automatic program repair were created based on the C language (it can be observed in the mapping study), making researchers propose automatic repair for new languages (e.g., Java) using C language approaches as the basis for comparison. Also, the C language approaches are usually directly translated to work with other languages. Soto et al. (2016) observe that this direct translation and comparison can hinder the automatic repair's effectiveness since different languages have different language constructs.
- Study and analysis of specific types of bugs in the context of patch generation and validation. Leesatapornwongsa et al. (2016) offer an in-depth analysis of concurrence bugs in distributed systems.
- The impact of design pattern and anti-pattern related bugs on object-oriented software Jaafar et al. (2015).
- Code clones can propagate bugs through the code base. To understand the impacts of code clones on software maintenance, it is also necessary to understand their intensity. A study in this direction is proposed by (Mondal et al., 2017).

RQ2 What types of contributions have been made?

Among the selected papers with high citation impact, it was found three different types of contributions: analysis, taxonomy, and tool, as shown in Table 2.11. It was considered how authors mention their contributions to define types of contributions. In the second column of Table 2.11, it is possible to observe one proposed tool (Tantithamthavorn et al., 2015) and one taxonomy (Leesatapornwongsa et al., 2016). Also, eight analyses of different topics related to software bugs were obtained.

RQ3 What are the principal findings and identified research challenges with directions for future work?

Many subjects related to software bugs were observed that can be explored to improve maintenance and software evolution activities. However, there is generally a lack of understanding of bug fixes' nature, which is fundamental to the automatic program repair research domain. Most empirical studies on bug-fix focus only on improving ways to explore the search space of fixing

bugs and do not answer important questions about how many bugs or what type of bugs could be fixed by automatic program repair (Zhong and Su, 2015). Besides, the existing automatic program repair tools may produce negative results when applied to real bugs (i.e., in the software industry). This happens because the existing tools are just immature research prototypes (Zhong and Su, 2015). More research must be done with industry parties to provide better automatic program repair solutions.

It is possible to notice the concern about understanding and fixing specific bugs (Jaafar et al., 2015, Leesatapornwongsa et al., 2016). Still, it is unclear how the current approaches to bug fixing, localization, and prediction (usually general in scope) behave according to each bug type. It would be important to study how each proposed approach behaves according to the different classes of bugs.

Another important observation is that the proposed repair, localization, and prediction approaches focus their attention on being general in specific programming languages. There is no research (that one knows of) studying the trade-offs or comparing different combinations of bug approaches: (1) language-specific approaches applied to specific bug types, (2) language agnostic approaches applied to specific bug types, (3) language-specific approaches applied to general bug types and (4) language agnostic approaches applied to general bug types.

Soto et al. (2016) suggested mutation-based program repair may need to consider field and method insertions to achieve human comparability. It indicated that the direct translation of mutation-based approaches from C to Java is not likely to succeed due to the different language constructs. Thus, it is important to put more effort into developing language-specific approaches for automatic program repair, not mixing approaches for different languages or directly translating them.

RQ4 What are the demographic characteristics of the relevant studies?

To answer this RQ, four aspects of the selected primary studies were examined: (1) publication trend, (2) publication venue (journals and proceedings), (3) citation impact, and (4) geographical distribution.

(1) Publication trend

From 2015-2020, ten publications with high citation impact were obtained, following our methodology. Figure 2.7 shows the evolutions in high citation impact publications over the last five years. It can be observed that the research activity about general topics in software bugs shows stability, as shown in Figure 2.8. However, only a few studies per year over 2015-2017 had a high citation impact.

(2) Publication venue

Thirty-one publication venues were identified that published papers on analyzing and understanding bugs and bug-handling approaches. However, the publications with high citation impact are published among only six: five conference proceedings and one journal article (Table 2.12).

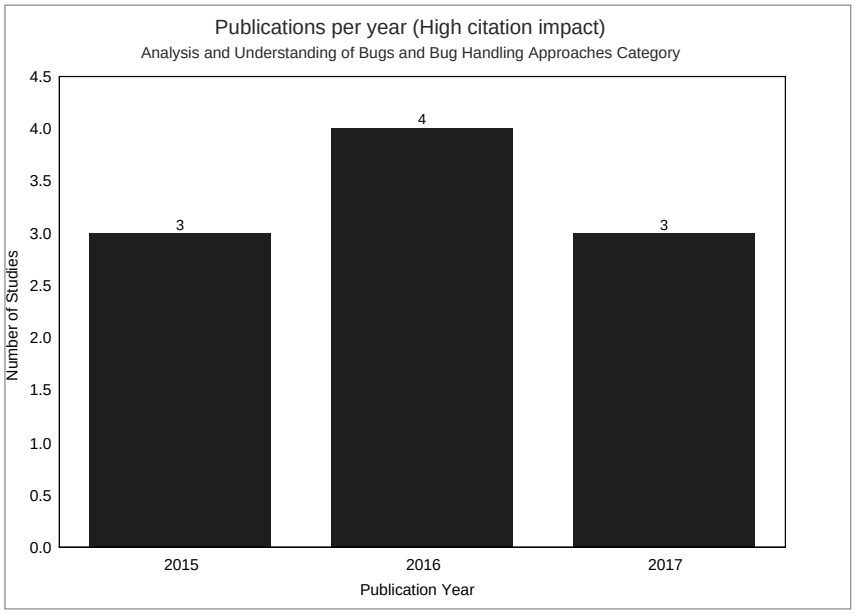

Figure 2.7: Publications per year about analysis and understanding of bugs and bug handling approaches (High citation impact)

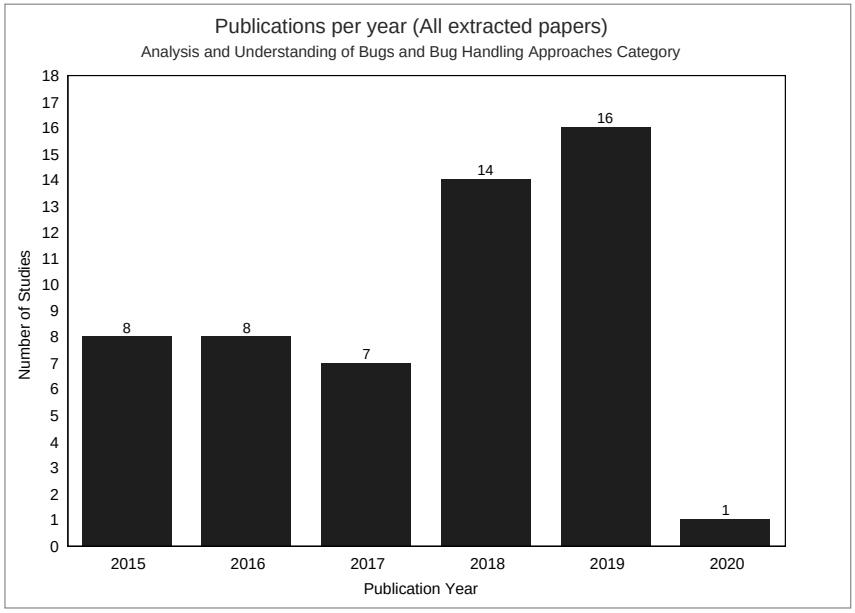

Figure 2.8: Publications per year about analysis and understanding of bugs and bug handling approaches (All extracted papers)

Table 2.13 shows the most active journals and conferences identified (despite the number of high citation impact publications). The International Conference on Software Engineering (ICSE) is the top conference contributor with eight published papers. The International Conference on Program Comprehension (ICPC), International Conference on Software Maintenance and Evolution (ICSME), and Empirical Software Engineering are the second-best contributors, with four published papers each. Among the journals, Empirical Software Engineering is the top contributor.

Table 2.12: Number of publications per venue and type of publication (High citation impact papers)

Publication Venue	Type of publication	Total
IEEE International Conference on Software Engineering (ICSE)	Proceedings	3
IEEE International Conference on Software Maintenance and Evolution (ICSME)	Proceedings	2
International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)	Proceedings	1
International Conference on Evaluation and Assessment in Software Engineering (EASE)	Proceedings	1
Conference on Computer and Communications Security (SIGSAC)	Proceedings	1
Empirical Software Engineering	Journal	1

Table 2.13: Number of publications per venue and type of publication (All extracted papers)

Publication Venue	Type of publication	Total
International Conference on Software Engineering (ICSE)	Proceedings	8
International Conference on Program Comprehension (ICPC)	Proceedings	4
IEEE International Conference on Software Maintenance and Evolution (ICSME)	Proceedings	4
Empirical Software Engineering	Journal	4
Asia-Pacific Software Engineering Conference (APSEC)	Proceedings	3
Annual International Conference on Computer Science and Software Engineering (CASCON)	Proceedings	3
ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)	Proceedings	3
International Conference on Mining Software Repositories (MSR)	Proceedings	3
Journal of Systems and Software	Journal	3
International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)	Proceedings	2
ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)	Proceedings	2
International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE)	Proceedings	2
International Conference on Software Analysis, Evolution and Reengineering (SANER)	Proceedings	2
Workshop on Emerging Trends in Software Metrics (WETSoM)	Proceedings	2
ACM Southeast Conference (ACM SE)	Proceedings	1
Amity International Conference on Artificial Intelligence (AICAI)	Proceedings	1
IEEE/ACM International Conference on Automated Software Engineering (ASE)	Proceedings	1
International Workshop on Complex Faults and Failures in Large Software Systems (COUFLESS)	Proceedings	1
Workshop on Hot Topics in Operating Systems (HotOS)	Proceedings	1
IEEE International Conference on Electronics Information and Emergency Communication (ICEIEC)	Proceedings	1
International Conference on Reliability, Infocom Technologies and Optimization (ICRITO)	Proceedings	1
International Conference on Computing for Sustainable Global Development (INDIACom)	Proceedings	1
Asia-Pacific Symposium on Internetware (Internetware)	Proceedings	1
International Conference on Mobile Software Engineering and Systems (MOBILESoft)	Proceedings	1
Workshop on Pre- and Post-Deployment Verification Techniques (PrePost)	Proceedings	1
International Conference on Software Engineering and Knowledge Engineering (SEKE)	Proceedings	1
Software Engineering Notes (SIGSOFT)	Proceedings	1
International Workshop on Software Qualities and Their Dependencies (SQUADE)	Proceedings	1
ACM Transactions on Architecture and Code Optimization (TACO)	Proceedings	1
International Conference on Evaluation and Assessment in Software Engineering (EASE)	Proceedings	1
Conference on Computer and Communications Security (SIGSAC)	Proceedings	1

(3) Citation impact

Figure 2.9 highlights the number of citations of the most cited papers among all papers classified as analysis and understanding of bugs and bug handling approaches. Figure 2.9 shows only the most cited papers based on the category's h-index. In contrast, Figure 2.10 shows all papers in the category with at least one citation, despite its h-index. The citation count of each paper was obtained from the respective publisher or indexing engine (e.g., ACM, IEEE Xplore, and Scopus), which is subject to change at any point in time. Ten papers with more than ten citations were identified each (listed in Table 2.11). The total number of citations for all papers

with at least one citation (Figura 2.10) is 345, and the average number of citations per paper is approximately 8.

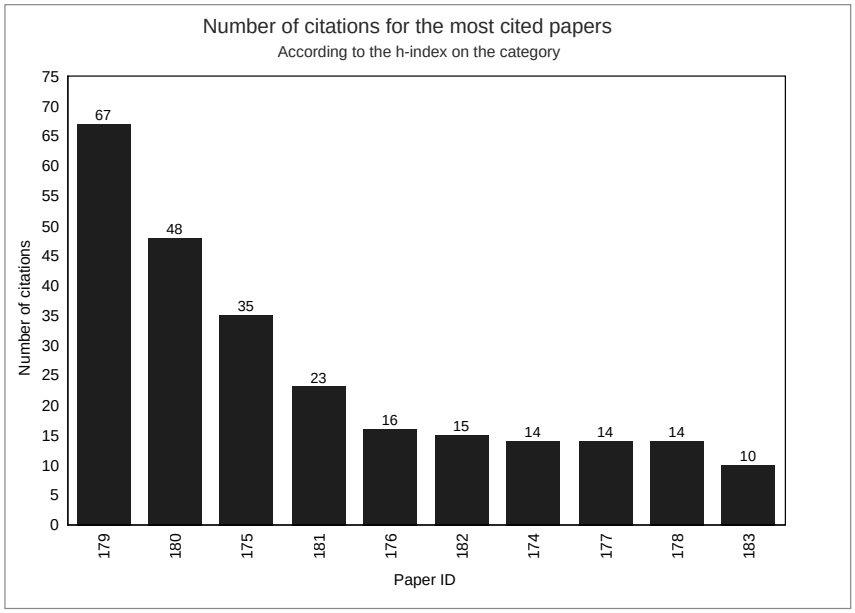

Figure 2.9: Number of citations for the most cited papers on analysis and understanding of bugs and bug handling approaches

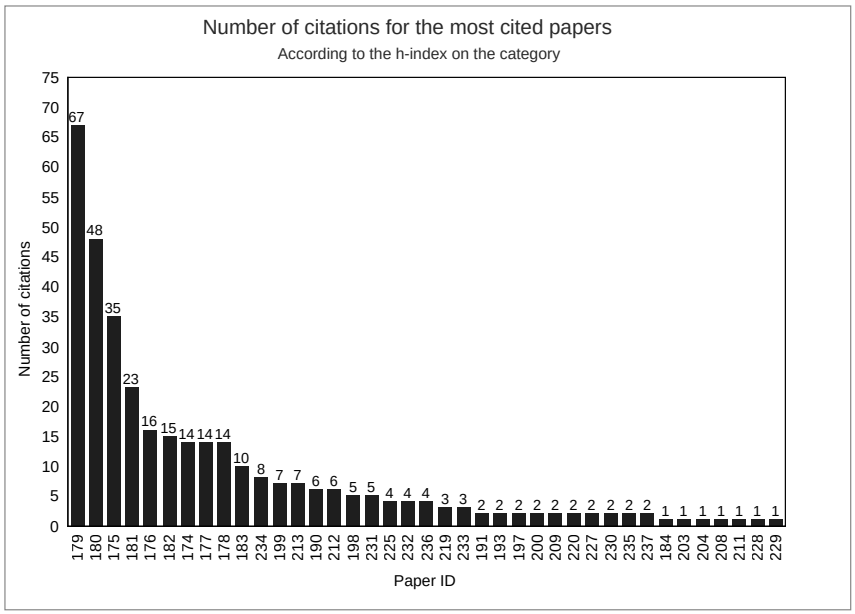

Figure 2.10: Number of citations for all cited papers on analysis and understanding of bugs and bug handling approaches

(4) Geographical distribution

Table 2.14 highlights the most active countries researching the analysis and understanding of bugs and bug handling approaches. It identified 14 countries from all the extracted relevant papers in the category. The United States of America, with 19 published papers, is the most active country, followed by Canada, with 15 published papers, and China, with ten published

papers. The selection of high citation impact papers according to the h-index comes from 6 of the 14 countries: the United States of America with 4 of the most cited papers, Canada with 2 of the most cited papers, China, Singapore, Japan, and Italy with 1 of the most cited papers each.

Table 2.14: Most active countries researching the analysis and understanding of bugs and bug handling approaches

Country	Number of published papers
United States	19
Canada	15
China	10
India	4
Italy	3
Brazil	3
Japan	2
Spain	2
South Korea	1
Thailand	1
Germany	1
Australia	1
Israel	1
Singapore	1

2.2.3 Automatic Patch Generation for Existing Bugs

As software applications grow in size and complexity, they also produce an increasing amount of bugs (Saha et al., 2017). A study from the University of Cambridge showed that debugging software can cost as much as $312 billion annually (Cambridge, 2013). Software bugs are so expensive because developers have to dedicate a considerable amount of manual effort (Ke et al., 2015). Although manual debugging is the current standard procedure in the industry, the approach is impracticable with the volume of bugs usually found in large-scale software systems (Ke et al., 2015).

Automated program repair can potentially reduce the negative impacts and costs of fixing bugs in software systems, consequently improving its quality (Ke et al., 2015). Several automated program repair tools, methods, techniques, and models have been proposed over the last five years, and it was identified the most prominent ones depicted in Table 2.15. Each of them adopts different approaches and addresses challenges related to the automation of bug-fixing activities.

It is possible to observe two significant types of automatic repair approach[6]: (1) generate-andvalidate and (2) correct-by-construction. Generate-and-validate approaches, also known as search-based approaches, use a set of transformations to generate several possible repairs (candidate patches) and validate each possible repair against a defined set of test cases (Hua et al., 2018). The set of candidate patches is usually huge. To explore it efficiently, generate-and-validate techniques use different approaches to find a patch capable of passing all the test cases.

[6]Here *approach* is used as an umbrella term for the several types of contributions identified in the analyzed studies: approaches, methods, techniques, tools, etc.

Table 2.15: Works classified as automatic repair or automatic patch generation

Paper/Contribution	Language	Datasets	Type of contribution	Country	Institution
Kali (Qi et al., 2015)	Java	Defects4J	System	USA	MIT
DirectFix (Mechtaev et al., 2015)	C	GNU Coreutils, Siemens: replace, Siemens: schedule, Siemens: schedule2, Siemens: tcas	Approach	Singapore	National University of Singapore
Elixir (Saha et al., 2017)	Java	Defects4J Bugs.jar	Technique	USA	University of Southern California
Angelix (Mechtaev et al., 2016)	C	CoREBench, PHP, Wireshark, gmp, gzip, libtiff	Tool	Singapore	National University of Singapore
NOPOL (Xuan et al., 2017)	Java	Apache Commons Lang, Apache Commons Math	Approach	China	Wuhan University
SearchRepair (Ke et al., 2015)	C	IntroClass	Tool	USA	Iowa State University
Relifix (Tan and Roychoudhury, 2015)	C	CoREBench	Approach	Singapore	National University of Singapore
QACrashFix (Gao et al., 2015)	Java	Misc.	Tool	China	Peking University
SketchFix (Hua et al., 2018)	Java	Defects4J	Technique	USA	The University of Texas at Austin
Janus Manager (Haraldsson et al., 2017)	Python	Janus Manager	System	Scotland	University of Stirling
LeakFix (Gao et al., 2015)	C	SPEC2000	Tool	China	Peking University
Repairnator (Urli et al., 2018)	Java	N/A	Bot	France	University of Lille
SemGraft (Mechtaev et al., 2018)	C	BusyBox, GNU Coreutils	Tool	Singapore	National University of Singapore
ssFix (Xin and Reiss, 2017)	Java	Defects4J	Technique	USA	Brown University

Generate-and-validate approaches require many candidates to provide a repair, especially for bugs at an acceptable granularity level (e.g., expression). Each candidate patch is applied and then compiled and tested. The costs of compilation and test execution are high (Hua et al., 2018). However, the number of candidates can be pruned substantially using runtime information and generating candidates only during test validation, as argued Hua et al. (2018).

Correct-by-construction repair approaches (also known as constraint-solving or semantic repair) rely on inferred or provided specifications to synthesize[7] candidate patches (Ke et al., 2015). This approach can dynamically collect path conditions and infer constraints based on passing and failing test executions (Hua et al., 2018). The constraints are translated to propositional satisfiability (SAT) formulas for SAT/SMT solvers. Then, solvers are responsible for synthesizing a repair that satisfies all inferred specifications (Hua et al., 2018). Correct-by-construction repair techniques can yield incomplete translation of constraints to SAT, and they can suffer from limitations of the symbolic execution engines to extract constraints (Hua et al., 2018).

Usually, an automated program repair technique has a buggy program and a test suite as input and produces a patched program that passes all the tests as output (Xin and Reiss, 2017). The patched program's validation process is entirely based on the provided test suite. Repair techniques using this validation approach show poor performance, meaning that most of the generated patches are incorrect (Xin and Reiss, 2017). One way to overcome this problem is by using information beyond the test suite to generate a set of candidate patches that are more likely to contain a correct patch. The current augment of open-source software systems has led to many large, publicly accessible code databases (Ke et al., 2015) such as GitHub, BitBucket, and the latter SourceForge. Ke et al. (2015) believed many programs include routines, data

[7] create from scratch

structures, and designs already implemented in several open-source software projects. So, if a software component has a bug, there is a high probability a similar but correct version of that component is available on a public software project that can be used to fix the bug. Ke et al. (2015) used semantic code search over existing open-source code to find correct implementations and automatically generate patches for software bugs.

The same principle is followed by Gao et al. (2015), who believes that bugs are recurrent, meaning that most bugs found in software projects often occur in different projects. The reason for the recurrence can be the crescent use of frameworks. In this case, problems occur when certain constraints in these frameworks are violated – for example, the requirement of calling specific methods. Programmers of different applications using the same framework may forget to call the specific method, leading to a recurring bug (Gao et al., 2015). The recurrence of bugs can be used by automatic repair techniques to fix them. Gao et al. (2015) proposed to infer fixes automatically via analyzing Q&A sites like StackOverflow, based on the assumption that recurring bugs have already been discussed over these types of sites.

When developing new approaches to automatic program repair, it is important to consider three major characteristics; (1) scalability, the approach has to work fine on large programs; (2) repairability, the approach has to repair most of the bugs and cover most of the classes of bugs, and (3) the approach has to produce high-quality patches which make the minimum number of changes to the program, deletes less functionality and are likely to be accepted by the maintainer (Mechtaev et al., 2016). According to Mechtaev et al. (2018), one of the reasons for the low quality of automatically generated patches is the lack of specifications of the intended behavior. Formal specifications are usually unavailable, so most program repair approaches rely on tests to validate patches. However, tests are incomplete specifications (Mechtaev et al., 2018), so there is a high probability that the generated patches merely overfit the tests. Mechtaev et al. (2018) tackled this problem by automatically inferring the missing specification for a buggy program from a correct reference program. A reference program is an alternative realization of the same functionality being fixed (Mechtaev et al., 2018). Thus, the generated patch should enforce the equivalence of the patched and the reference programs, which imposes two challenges (Mechtaev et al., 2018): (1) scalability and (2) applicability.

In the case of correct-by-construction techniques, low scalability is the main problem (Mechtaev et al., 2016), despite the promising results shown by some methods regarding high repairability (Nguyen et al., 2013). Mechtaev et al. (2016) proposed a semantic-based repair methodology that can also scale up to the same level as search-based repair tools (Le Goues et al., 2012, Long and Rinard, 2015). It is possible because the Mechtaev et al. (2016) define a lightweight repair constraint called *angelic forest*. This *angelic forest* can be automatically extracted via symbolic execution. According to Mechtaev et al. (2016), the angelic forest is simpler compared to the repair constraints used in other works (Nguyen et al., 2013, Mechtaev et al., 2015), and its size is independent of the size of the program under repair, which allows the method to scale.

According to an empirical study conducted by Saha et al. (2017) on Java projects, 57% of the statements on these projects contain one or more method invocations. Also, for the same

projects, 77% of one-line bug fixes involved a change or insertion of a method invocation. Thus, Saha et al. (2017) justified the need to study the repair and synthesis of method invocations for the automatic repair of object-oriented programs. Other techniques have already handled method invocations in automatic repair: Xuan et al. (2017) proposed NOPOL, a tool for automatic program repair capable of synthesizing (i.e., creating from scratch) method calls. However, a series of constraints are imposed; only method calls are manually specified, with no side effects and no parameters. Martinez and Monperrus (2015) justified the use of restrictions as a means to prevent the combinatorial explosion resulting from the expansion of the repair space (i.e., the search space for candidate patches) to include a larger scope of method call modifications and insertions (Saha et al., 2017). A repair approach cannot iterate through all possible repair candidates.

Despite the potential problems caused by large search spaces, Saha et al. (2017) proposed an aggressive use of method calls, local variables, fields, and constants to construct better repair expressions. Those repair expressions are then used to synthesize patches. Enlargement of the repair space is tackled by Saha et al. (2017) using a machine learning model to rank the most suitable repairs to fix the bug to the less suitable. The model relies on the repair context's characteristics (code surrounding the repair location).

Hua et al. (2018) believed that the enlargement of the repair space can be tackled by using runtime information and generating candidate patches *on-demand* during test validation. SketchFix (Hua et al., 2018) is a repair tool that translates buggy programs to *sketches*[8]. Each sketch is compiled once and represents several possible patch candidates due to the missing code fragments. The missing fragments of code can be replaced by any possible code, thus generating a concrete patch candidate. Using the runtime behavior of tests, the technique lazily initializes the candidate patches from the sketches while validating them against the tests. The concrete candidates are created only when a test executes the code.

As software evolves due to changes and bug fixes, regression bugs can be introduced (Tan and Roychoudhury, 2015). Regression occurs when previously passing test cases fail. As every change in a program can induce new bugs, the activity of fixing regression bugs can also introduce new bugs. This happens mainly due to low-quality patches and inadequate testing (Tan and Roychoudhury, 2015). Relifix (Tan and Roychoudhury, 2015) is an approach for the automated repair of software regression. Tan and Roychoudhury (2015) uses syntactic information between program versions and test execution history to repair changes that cause regressions. Relifix follows five criteria to fix regression bugs: (1) only minor changes can be introduced. Retaining as much of the new version's code as possible is important since major changes can introduce more regression bugs. (2) The code has to be readable. Developers must understand the patches to verify them easily. (3) The patched program must pass the progression tests. (4) The patched program has to pass the failed regression tests. (5) If changes cause other tests in the test suite to fail, the source code cannot be changed, and no patch is applied.

Urli et al. (2018) claimed that to use program repair in practice, it is necessary to execute

[8]Partial programs with *blank spaces* that can be filled with any fragment of code, generating several possible combinations

several activities before and after the execution of the repair algorithm (for example, failure detection, bug reproduction, etc.). Thus, Urli et al. (2018) proposed an end-to-end repair toolchain implemented in a program repair bot called Repairnator. The proposed program repair bot is an autonomous agent that continuously monitors failures reproduces bugs and runs program repair tools against each reproduced bug. When a patch is found by one of the repair tools, the bot reports it to the developers that can fix it (Urli et al., 2018).

Research Questions

RQ1 What topics/problems are investigated?

Based on the analysis of the selected primary papers, it was identified 14 prominent works classified as Automatic Repair or Automatic Patch Generation, which are depicted in Table 2.15. Several topics are addressed by different studies in automatic program repair and automatic patch generation. Among the most cited papers selected by the systematic mapping, five important topics were identified to some extent, omnipresent in all analyzed works. The topics are described below.

The first identified topic is ***patch complexity***. It is an important topic since the size and complexity of a patch can influence its probability of being accepted by the maintainer (Mechtaev et al., 2015). The second topic is the ***repair of specific types of bugs***. Most approaches to automatic program repair try to generalize the kind of bug they can resolve, but some authors argue that it is best to focus on specific types. Saha et al. (2017) discussed the repair of bugs related to method invocations in object-oriented programs. Saha et al. (2017) motivated this topic, showing that about 77% of one-line bug fixes in the studied Java projects involved method invocations, either stand-alone or as part of another language construct. However, Saha et al. (2017) showed that repairs that require the synthesis of substantially new method invocations are beyond the scope of current repair tools and are typically infeasible due to a combinatorial explosion in the number of candidate patches. Xuan et al. (2017) addressed the importance of automatically repairing IF conditional statements in bugs. According to Xuan et al. (2017) IF conditionals are among the most error-prone program elements in Java programs, and 12.5% of one-change commits simply update an IF condition.

The third topic concerns the ***search space size***. It is necessary to apply an appropriate change at the bug location to fix a bug. Allowing any arbitrarily complex change can result in infinite possible changes that can or cannot fix the bug. These changes can be called *candidate patches*. As it is impossible to iterate through an infinite set of candidate patches to find the correct fix, automatic program repair tools define restrictions on program transformations that can be used to generate a finite set of candidate patches.

The fourth topic is the ***scalability***. Semantic-based repair techniques are usually not efficient when applied to large-scale software systems, even though they show high repairability and high-quality repairs. They can be used to repair multiple locations (Mechtaev et al., 2016). Generate-and-validate techniques can scale to large software systems, but they are currently

restricted to fixing a single location on the buggy code (Mechtaev et al., 2016). Mechtaev et al. (2016) discuss how to scale a semantic-based repair technique.

RQ2 What types of contributions have been made?

It was found six different types of contributions: system, technique, tool, approach, and bot, as shown in Table 2.15. Defining these types of contributions was considered how the authors referred to their own work. On the third column of Table 2.15, it is possible to observe that Janus Manager and Kali are defined as Systems. Kali (Qi et al., 2015) is a generate-and-validate patch generation system that focuses on only removing functionality as a solution to bugs. The system generates patches that either delete a single line or block of code or replace an if condition with true or false to force the execution of the then or else branch of the conditional. Janus (Haraldsson et al., 2017) is a commercial software system with a self-improving capability. Janus has been in commercial use; during business hours, it is used by professionals, and after business hours and the log out of the last user, the system starts a self-analysis based on the day's recorded interactions. Janus uses Genetic Improvement to fix any recorded bug that has raised exceptions.

Elixir (Saha et al., 2017), SketchFix (Hua et al., 2018), and ssFix (Xin and Reiss, 2017) are defined as techniques by their authors. Angelix (Mechtaev et al., 2016), SearchRepair (Ke et al., 2015), QACrashFix (Gao et al., 2015), LeakFix (Gao et al., 2015) and SemGraft (Mechtaev et al., 2018) are defined as tools. DirectFix (Mechtaev et al., 2015) was defined as an approach by its authors and implemented as a prototype tool not available for use. The authors also state that Directfix is a method, so the proper definition is unclear. Other approaches identified are NOPOL (Xuan et al., 2017) and Relifix (Tan and Roychoudhury, 2015). Urli et al. (2018) is the only contribution defined as a bot.

RQ3 What are the principal findings and identified research challenges with directions for future work?

Several important observations can be made based on mapping the studies depicted in Table 2.15. These observations are summarized below:

- Machine learning algorithms can be used to enlarge the search space in semantic-repair techniques (Saha et al., 2017).
- Automatic repair techniques rely heavily on the bug localization approach. This means that the performance of an automatic program repair approach is attached to the bug localization approach's efficiency.
- Several automatic program repair techniques use Satisfiability Modulo Theories (SMT) to synthesize patches (Mechtaev et al., 2016, Xuan et al., 2017). However, as pointed out by Xuan et al. (2017), methods with parameters cannot be directly encoded in SMT for object-oriented languages. Thus, the solver suffers from a lack of information about method calls. This leads to a timeout, and a patch may not be provided (Xuan et al., 2017).

- Xuan et al. (2017) demonstrated the need to add, delete, or modify test cases to validate the generated patches. Also, Xuan et al. (2017) suggested a need for recommendation systems that tell when to add additional test cases for the sake of repair and what those test cases should specify.

- Use of several different datasets for patch validation. It identified the use of 21 different datasets among 14 works (Table 2.15). Some of them are well-known, such as Defects4J and PHP. However, some are datasets created by the paper's authors to validate their work and serve as one of their contributions, such as Bugs.jar. The diversity of datasets can hinder the comparison of automatic repair approaches and complicate the proposal of new ones. Maybe it would be an excellent contribution to creating a standard dataset for analyzing and creating automatic repair techniques.

- Most approaches to automatic program repair are test-driven. As tests are incomplete specifications, validating the generated patches can overfit the test-suite (Ke et al., 2015). Also, patches that pass on all tests in a specific test suite may not generalize to the complete program specification (Ke et al., 2015).

- New patches can generate regression. Thus, new bugs can be introduced when an automatic program repair technique tries to fix some other bugs. The regression problem is an interesting perspective that adds to the automatic program repair research and guarantees that patches do not insert other bugs in the code.

- It was observed that the granularity level of the studied techniques varies from expressions to whole fragments of code. Also, most techniques can modify only one bug location at a time. This means that more complex bugs that require changes in several locations may not be well handled by the current automatic repair techniques (Xuan et al., 2017).

RQ4 What are the demographic characteristics of the relevant studies?

To answer this RQ, four aspects of the selected primary studies were examined: (1) publication trend, (2) publication venue (journals and proceedings), (3) citation impact, and (4) geographical distribution.

(1) Publication trend

From 2015 to 2018, 14 publications with a high number of citations were extracted, following the defined methodology. Figure 2.11 shows the evolution of the publications about automatic program repair and automatic patch generation. The research activity was observed to be progressive and active; however, only those papers with high citation impact were selected, and only a few publications each year were considered. Although it was considered the year 2015-2020 in the systematic mapping, Figure 2.11 shows only 2015-2018. It happens due to the selected threshold by citation impact. It is important to highlight that it is not possible to make a firm and valid conclusion about why 2016 had a decrease in high citation impact publications.

Generally, despite the number of publications with high citation impact, the research activity continues to increase, and the research area shows stable growth, as shown in Figure 2.12.

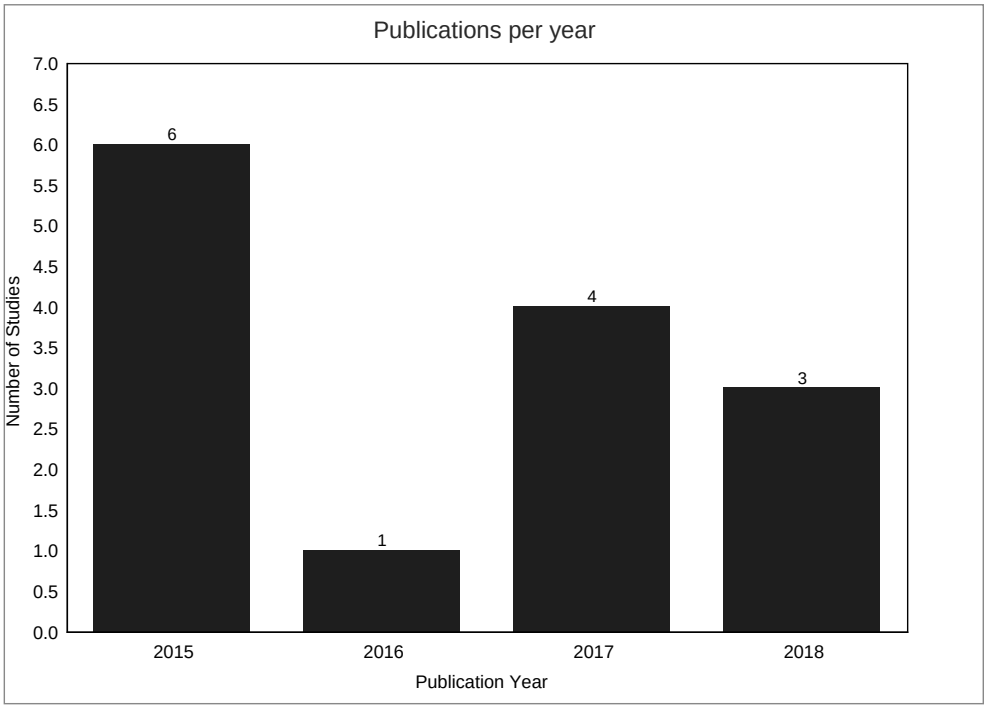

Figure 2.11: Publications per year about automatic program repair and automatic patch generation (High citation impact)

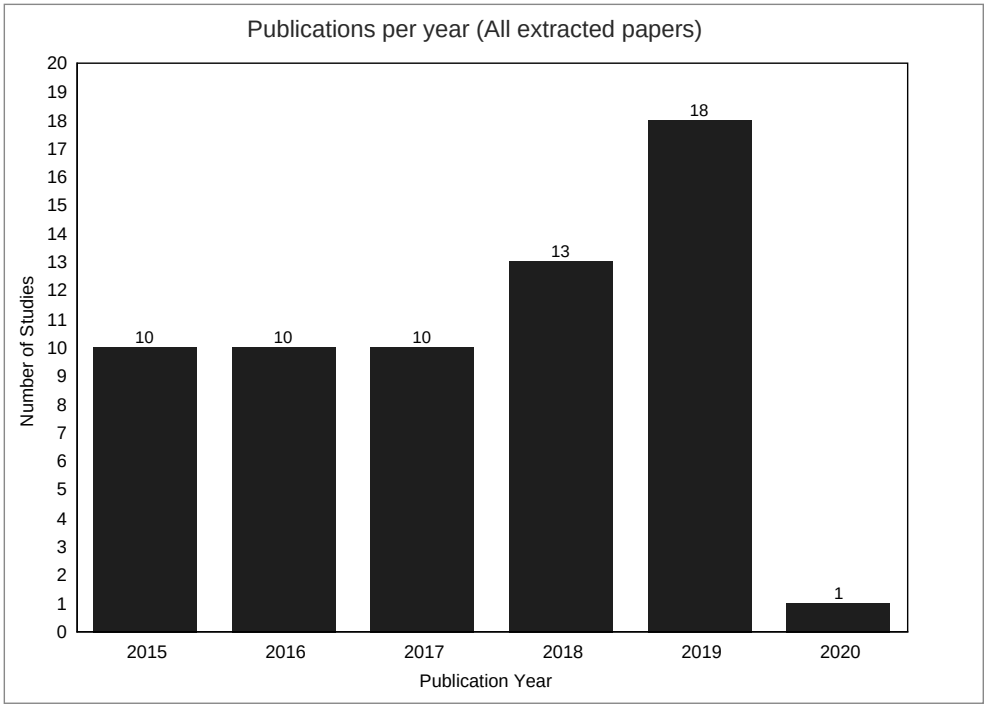

Figure 2.12: Publications per year about automatic program repair and automatic patch generation (All extracted papers)

(2) Publication venue

This mapping study identified 20 different publication venues, publishing papers related to automatic program repair and automatic patch generation. However, the publications with high citation impact are distributed among only 5: four conference proceedings and one journal (Table 2.16). It is possible to observe that most publications with high citation impact were conference proceedings, followed by one journal article. Concerning the publication venues in which automatic program repair papers are published, Table 2.17 shows the most active journals and conferences identified (despite the number of high citation impact publications). The International Conference on Software Engineering (ICSE) and International Conference on Automated Software Engineering (ASE) were the top conference contributors among all the conferences. Among journals, the Journal of Systems and Software was the top contributor.

Table 2.16: Number of publications per venue and type of publication (High citation impact papers)

Publication Venue	Type of publication	Total
International Conference on Automated Software Engineering (ASE)	Proceedings	4
Genetic and Evolutionary Computation Conference Companion (GECCO)	Proceedings	1
International Conference on Software Engineering (ICSE)	Proceedings	7
International Symposium on Software Testing and Analysis (ISSTA)	Proceedings	1
IEEE Transactions on Software Engineering	Journal	1

Table 2.17: Number of publications per venue and type of publication (All extracted papers)

Publication Venue (all extracted)	Type of publication	Total
International Conference on Software Engineering (ICSE)	Proceedings	16
International Conference on Automated Software Engineering (ASE)	Proceedings	10
Annual Computer Software and Applications Conference (COMPSAC)	Proceedings	3
Genetic and Evolutionary Computation Conference (GECCO)	Proceedings	3
Journal of Systems and Software	Journal	3
International Workshop on Genetic Improvement (GI)	Proceedings	2
IEEE International Conference on Software Maintenance and Evolution (ICSME)	Proceedings	2
IEEE International Workshop on Empirical Software Engineering in Practice (IWESEP)	Proceedings	2
IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft)	Proceedings	2
IEEE Transactions on Software Engineering	Journal	2
International Conference on Availability, Reliability, and Security (ARES)	Proceedings	1
International Workshop in Automation of Software Test (AST)	Proceedings	1
International Workshop on Bots in Software Engineering (BotSE)	Proceedings	1
International FME Workshop on Formal Methods in Software Engineering (FormaliSE)	Proceedings	1
International Workshop on Intelligent Bug Fixing (IBF)	Proceedings	1
Iranian Conference on Electrical Engineering (ICEE)	Proceedings	1
International Joint Conference on Neural Networks (IJCNN)	Proceedings	1
International Symposium on Software Reliability Engineering (ISSRE)	Proceedings	1
International Conference on Software Analysis, Evolution and Reengineering (SANER)	Proceedings	1
International Conference on Software Engineering and Knowledge Engineering (SEKE)	Proceedings	1
International Symposium on Information and Communication Technology (SoICT)	Proceedings	1
International Systems and Software Product Line Conference (SPLC)	Proceedings	1
Empirical Software Engineering	Journal	1
Applied Intelligence The International Journal of Research on Intelligent Systems for Real Life Complex Problems	Journal	1
Science China Information Sciences	Journal	1
International Journal of Control Theory and Applications	Journal	1

(3) Citation impact

Figure 2.13 shows the number of citations of the most cited papers among all papers classified as automatic program repair and automatic patch generation. Figure 2.13 shows only the most cited papers based on the h-index of the category. In contrast, Figure 2.14 shows all papers in the category with at least one citation despite its h-index. The citation count of each paper

was obtained from the respective publisher or indexing engine (e.g., ACM, IEEE Xplore, and Scopus), which is subject to change at any point in time. Overall, 14 papers with more than 15 citations were identified each (Table 2.15). The total number of citations for all cited papers (Figure 2.14) is 779, and the average number of citations per paper is 19.

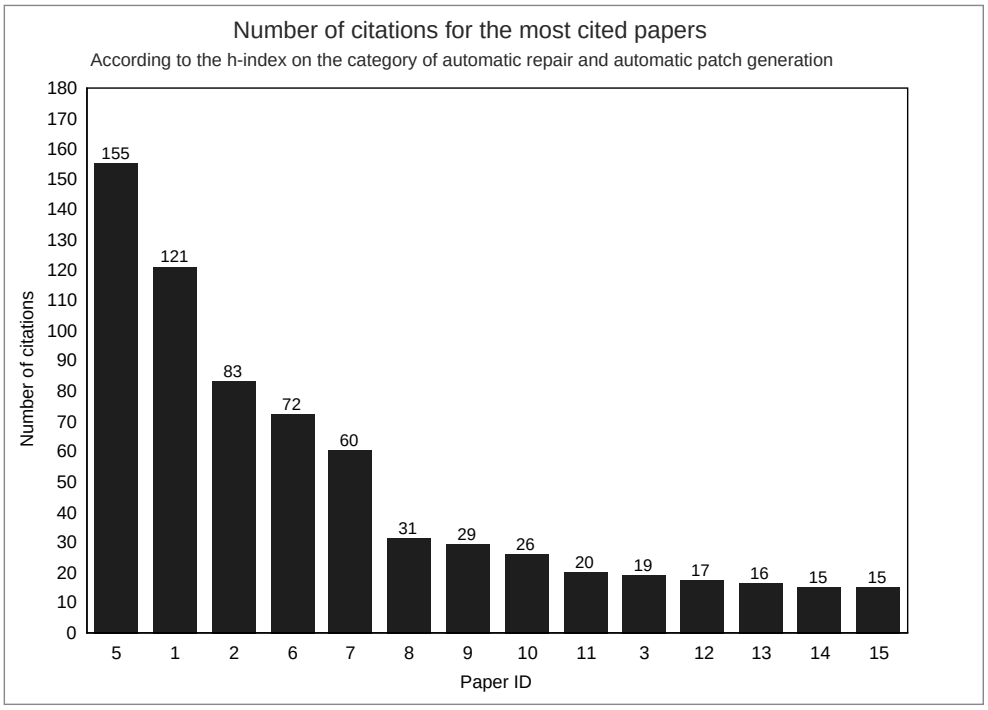

Figure 2.13: Number of citations for the most cited papers on Automatic Repair and Automatic Patch Generation

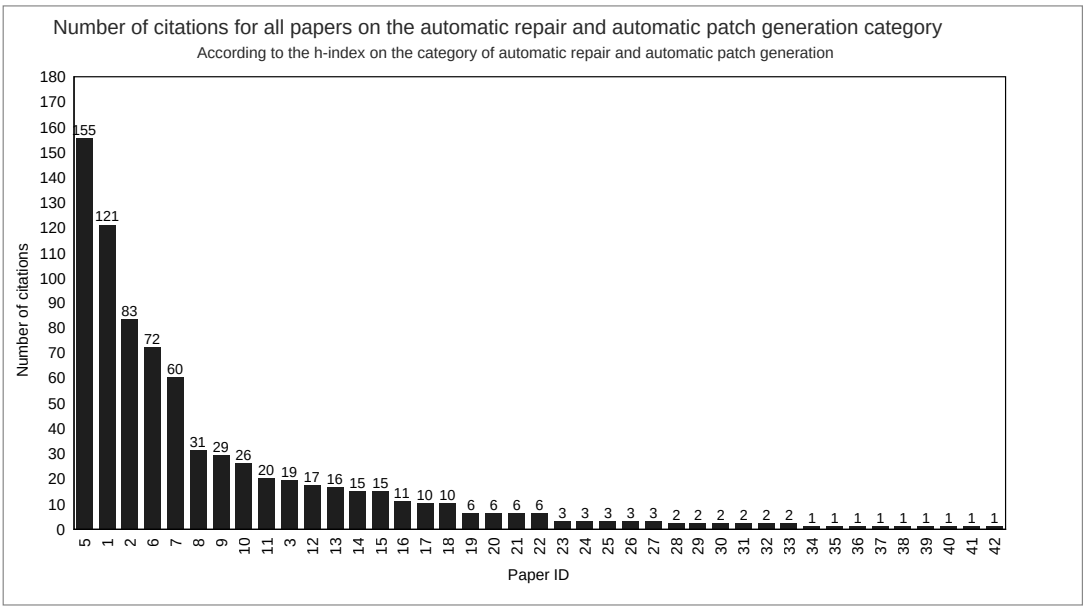

Figure 2.14: Number of citations for all cited papers on Automatic Repair and Automatic Patch Generation (At least one citation)

(4) Geographical distribution

Table 2.18 highlights the most active countries in the automatic repair and patch generation research area. It identified 14 unique countries from all the extracted papers in the research area. The United States of America, with 24 published papers, is currently the most active country, followed by China, with 11 published papers. Singapore is the third most active country, with six published papers. Our selection of most cited papers according to the h-index comes from only 5 of the 14 countries: the United States of America with five highly cited papers, followed by China with three highly cited papers, and Singapore with four highly cited papers. France and Scotland have one high-cited paper each.

Table 2.18: Most active countries in automatic program repair and automatic patch generation research

Country	Number of published papers
United States	24
China	11
Singapore	6
Japan	5
France	4
Iran	2
Brazil	2
Germany	2
India	1
Switzerland	1
Australia	1
Scotland	1
Tunisia	1
Vietnan	1

2.2.4 Bug Prediction

Bug prediction employs specific machine-learning techniques to anticipate the locations of bugs within a software system. By applying bug prediction models early in the development process, teams can focus their review and testing efforts on areas identified as bug-prone, thereby enabling more rigorous testing of these areas compared to those deemed less susceptible to bugs. According to Son et al. (2019), there are two primary methods for creating prediction models: (1) using measurable properties of the system under analysis, known as software metrics, and (2) leveraging data about bugs found in similar software projects. Once developed, these models can be applied to current software projects to identify components likely to contain bugs (Son et al., 2019).

This section provides an overview and responds to the specified research questions (SRQ) based on the most cited papers in bug prediction from recent years. A variety of models and approaches[9] have been proposed, each addressing different challenges associated with predicting buggy components. The most notable contributions are listed in Table 2.19[10], showcasing diverse

[9]Here *approach* is used as an umbrella term for the various types of contributions identified in the analyzed studies.

[10]*Language agnostic* approaches can be applied to multiple programming languages

methodologies employed to address the unique challenges of bug prediction.

Furthermore, the bug prediction process uses available information about the software being developed, such as code complexity, code authors' identities, and software development history, to predict potentially buggy areas in the source code. Existing bug prediction approaches aim to identify bugs in specific program components, such as files, methods, or changes (the code committed in a single file), thus helping teams prioritize their testing efforts effectively (Tan et al., 2015).

The concept of bug prediction at the change level, as discussed in (An and Khomh, 2015, Tan et al., 2015), allows for predicting buggy modifications at the time of commit, enabling developers to correct errors before the changes are finalized. This practice, known as just-in-time defect prediction, facilitates fine-grained predictions at the method level (An and Khomh, 2015). Tan et al. (2015) noted that since a change typically involves less content than an entire file, the effort required to inspect the buggy area and pinpoint the actual bug is relatively minimal. However, making accurate predictions at the time of commit presents more significant challenges (Tan et al., 2015).

Deep learning, a burgeoning field within machine learning, shows considerable promise for enhancing just-in-time defect prediction. Yang et al. (2015) introduced an approach termed Deeper, involving two crucial steps: (1) a feature selection step where expressive features are extracted from a basic set using Deep Belief Networks, and (2) a machine learning step where a classifier is built using these selected features.

Traditional bug prediction methods often rely on metrics like code churn or module size, which are only obtainable after code changes have been made and require additional data such as code history (Müller and Fritz, 2016). These methods typically overlook individual differences among developers, such as their level of expertise. In a novel approach, Müller and Fritz (2016) employed biometric sensing to detect code quality issues, including bugs, while the developer is still coding. Biometric measurements can indicate the mental effort exerted during coding; higher effort levels often correlate with a higher error rate and, consequently, more bugs. This predictive technique could enable preemptive code reviews, helping developers address potential errors before they are committed. The biometrics utilized include heart rate variability, respiratory rate, skin temperature, and electro-dermal activity. Despite its potential, applying biometric metrics in bug prediction is complex and prone to inaccuracies. These may stem from factors unrelated to coding complexity, such as the developer's personality traits or overall stress levels. Moreover, the current methods for capturing these metrics can be invasive and costly, potentially affecting the accuracy of the predictions.

Table 2.19: Works classified as bug prediction

Paper/Contribution	Language	Datasets	Type of Contribution	Country	Institution
Deeper (Yang et al., 2015)	Language Agnostic	Bugzilla, Columba, JDT, Platform, Mozilla, PostgreSQL	Approach	China	Zhejiang University
KPWE (Xu et al., 2019)	Language Agnostic	Promise, Nasa	Framework	China	Wuhan University
An and Khomh (2015)	Language Agnostic	Mozilla	Case study	Canada	Software Analytics and Technologies Lab
Tan et al. (2015)	Language Agnostic	Linux kernel, PostgreSQL, Xorg, Eclipse, Lucene, Jackrabbit	Case study	Canada	University of Waterloo
Müller and Fritz (2016)	Language Agnostic	N/A	Analysis	Switzerland	University of Zurich
Herzig et al. (2015)	Language Agnostic	N/A	Analysis	United Kingdom	Microsoft Research

Software crashes are unexpected interruptions of program execution in the user environment caused by bugs (An and Khomh, 2015). Usually, crashes are handled by crash reporting tools. Crash reporting tools collect information on the crash and send a detailed report to developers. A crash collecting system group crashes with the signature composed by the stack trace of the failing thread. Each crash group constitutes a crash type that has its impact analyzed. High-impact crash types are filed in bug-tracking systems like Bugzilla or Jira. Then, developers can focus their efforts on fixing the most critical bugs. Most crash capture approaches can only capture crashes after their occurrence, when they may have affected a large number of users. An and Khomh (2015) proposed a statistical model able to predict, at commit time, the crash-related bugs that lead to frequent crashes and impact many users. The prediction is made at the time of commit; this way, the software development team does not have to wait until crashes are collected, triaged, and filled into bug reports.

Several of the selected papers in this mapping reveal feature selection approaches. Feature selection is indispensable for a defect prediction approach (Song et al., 2011). Different feature selection methods have been used to select the best feature subset for defect prediction. There are two main types of feature selection methods: supervised and unsupervised. The supervised methods can be divided into wrapper and filter methods. The filter-based feature selection methods use statistical measures to calculate the correlation between input variables to choose the most relevant features. Wrappers feature selection methods create models with different subsets of input features and select the features that result in the best performance according to a performance metric (Kuhn and Johnson, 2013, Xu et al., 2019).

Filter-based selection methods select a subset of the original features without any transformation. Thus, the subset usually does not correctly represent the structure of the defect data. PCA has been widely used as a filter-based selection method for defect prediction (Xu et al., 2019). PCA transforms the original features into a low-dimensional space where the features are linear combinations of the original features. This method works well when separating the data and following a Gaussian distribution linearly is possible. However, real defect data may have complex structures that cannot be represented in a linear subspace. According to Xu et al. (2019), the features extracted by PCA methods are not plausible for defect prediction.

Xu et al. (2019) proposed a non-linear extension of PCA, called KPCA (Schölkopf et al., 1997), capable of projecting the original data into a latent high dimensional feature space. In the projected high dimensional space, the complex data structures of features can be adequately characterized, and it can be easier to do the linear separation of data. Studies have shown that KPCA performs better than PCA (Schölkopf et al., 1998).

Research Questions

RQ1 What topics/problems are investigated?

Some essential topics and bug prediction problems were investigated by the selected papers depicted in Table 2.19. They are summarized below:

- One important topic discussed is the use of deep learning algorithms to improve just-in-time defect prediction performance, which is still under-investigated in the literature (Yang et al., 2015).
- Xu et al. (2019) and Tan et al. (2015) discussed the complex structures and the imbalanced class distribution in software defect data, making it challenging to obtain suitable data features and build effective prediction models.
- Crashing-inducing commits are discussed by An and Khomh (2015). These are commits that would lead to crashes.
- Müller and Fritz (2016) investigates how biometrics can be used to predict software bugs. Biometrics are metrics obtained from the developers, such as heart rate.
- The handling of tangled code in commits in defect prediction models is discussed by Herzig et al. (2015).

RQ2 What types of contributions have been made?

Among the selected papers (most-cited) four different contributions were found: approach, framework, case study, and analysis as shown in Table 2.19. To define these types of contributions, it was considered how authors mention their contributions. In the third column of Table 2.19, it is possible to observe one proposed approach (Yang et al., 2015) and one framework (Xu et al., 2019). It was also obtained two case studies (An and Khomh, 2015, Tan et al., 2015), and two analysis (Müller and Fritz, 2016, Herzig et al., 2015).

RQ3 What are the principal findings and identified research challenges with directions for future work?

- Selecting the features that can reveal the defect data's intrinsic structure is crucial to accurately predicting them. When using traditional classification techniques and defect prediction models, the defect data's irrelevant and redundant features may degrade the performance of these models (Xu et al., 2019).
- Defect data has a high degree of class imbalance, meaning there are more non-defective modules than defective modules (Xu et al., 2019, Tan et al., 2015). It makes most classifiers classify the defective modules (minority samples) as the non-defective modules (majority samples). From the selected most cited papers, only Xu et al. (2019) address this issue but recognize that classifying imbalanced data is a known open challenge.
- Proprietary code usually has a lower buggy rate, which means fewer code areas are buggy. Thus, proprietary data tends to exhibit a high degree of imbalance, negatively impacting the results of bug prediction models. Tan et al. (2015) evaluated six open-source projects and observed that these systems' buggy rate is 14.7%-37,4%, whereas a proprietary software has a buggy rate below 14%.

- Most studies on bug prediction depend on the accuracy of the information they use (for example, commits). This accuracy is usually threatened by noise (Herzig et al., 2015). As pointed out by Herzig et al. (2015), one significant noise source is tangled changes, several unrelated code modifications done in a unique commit.

- New solutions are needed to convince developers to use bug prediction tools and their results (Tan et al., 2015). Also, the results from bug prediction tools have to be actionable (i.e., the development team has to be able to act on the results). The results from the prediction process need to be explainable to allow its use in the development process. Also, the precision of existing approaches to bug prediction has to improve since high-precision predictions must be applied. If developers find that most predicted bugs are false positives, they tend to ignore all prediction results Tan et al. (2015).

- Most defect prediction studies involve data obtained from open-source systems. Few studies use industrial datasets (Tan et al., 2015). It is essential to consider industrial data when building defect prediction models so the models can also be used in the industry.

RQ4 What are the demographic characteristics of the relevant studies?

To answer this RQ, we examined four aspects of the selected primary studies: (1) publication trend, (2) publication venue (journals and proceedings), (3) citation impact, and (4) geographical distribution.

(1) Publication trend

From 2015 to 2019, six publications with high citation impact were obtained, following the defined methodology methodology. Figure 2.15 shows the evolution of the publications about bug prediction. It was observed that the research activity had exhibited a low number of high-impact publications over the last five years. It is important to highlight that it is not possible to make a firm and valid conclusion about the reasons for that. Generally, despite the citation impact, bug prediction research shows some stability, as shown in Figure 2.16.

(2) Publication venue

It identified 24 different publication venues that published papers on bug localization. However, the publications with high citation impact are published among only 5: 3 conference proceedings and two journals (Table 2.20). It was observed that almost half of the high citation impact publications were published in conference proceedings, and two of them were published as a journal article. Concerning the publication venues in which bug prediction papers are published, Table 2.21 shows the most active journals and conferences identified (despite the number of high citation impact publications). The International Conference on Software Engineering (ICSE), International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE), and International Conference on Mining Software Repositories (MSR)

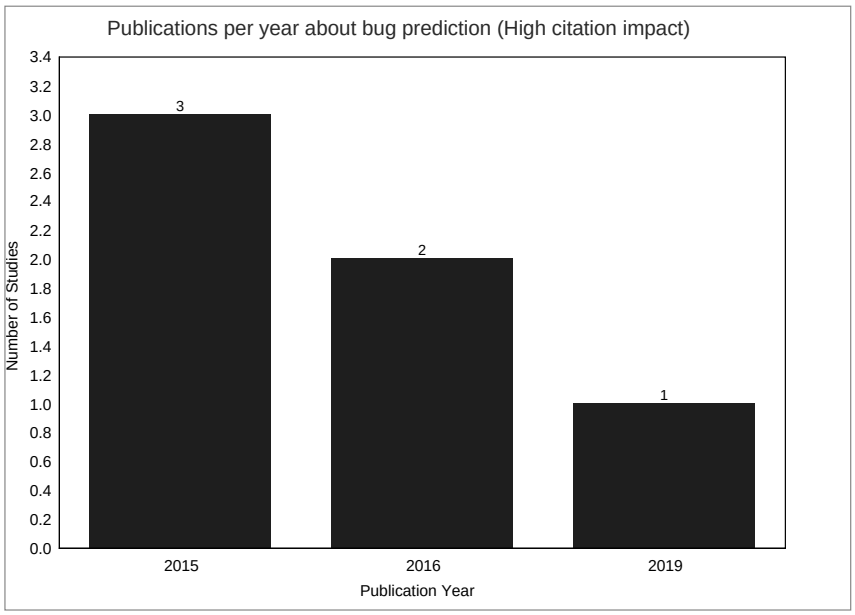

Figure 2.15: Publications per year about bug prediction (High citation impact)

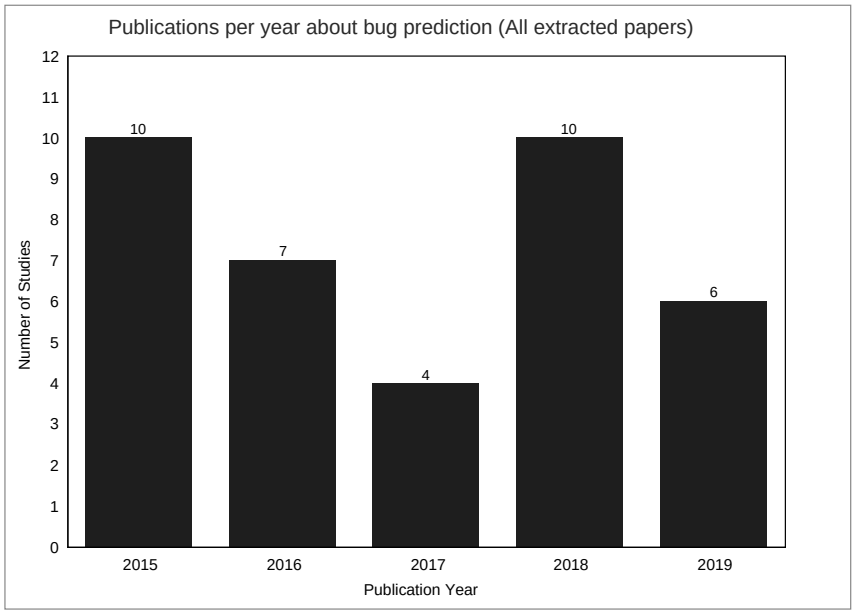

Figure 2.16: Publications per year about bug prediction (All extracted papers)

were the top three conference contributors among all the conferences. The Journal of Systems and Software was the top contributor among the journals.

Table 2.20: Number of publications per venue and type of publication (High citation impact papers)

Publication Venue	Type of publication	Total
International Conference on Software Engineering (ICSE)	Proceedings	2
International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE)	Proceedings	1
IEEE International Conference on Software Quality, Reliability and Security (QRS)	Proceedings	1
Empirical Software Engineering	Journal	1
Information and Software Technology	Journal	1

Table 2.21: Number of publications per venue and type of publication (All extracted papers)

Publication Venue	Type of publication	Total
International Conference on Software Engineering: New Ideas and Emerging Results (ICSE)	Proceedings	5
International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE)	Proceedings	4
International Conference on Mining Software Repositories (MSR)	Proceedings	3
ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)	Proceedings	2
International Conference on Program Comprehension (ICPC)	Proceedings	2
IEEE International Conference on Software Quality, Reliability and Security (QRS)	Proceedings	2
Journal of Systems and Software	Journal	2
International Conference on Evaluation and Assessment in Software Engineering (EASE)	Proceedings	1
International Conference on Software, Telecommunications and Computer Networks (SoftCOM)	Proceedings	1
International Conference on Software Engineering and Knowledge Engineering (SEKE)	Proceedings	1
Annual International Conference on Computer Science and Software Engineering (CASCON)	Proceedings	1
International Conference on Advances in Image Processing (ICAIP)	Proceedings	1
International Conference on Software and e-Business (ICSEB)	Proceedings	1
International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE)	Proceedings	1
International Arab Journal of Information Technology	Journal	1
International Conference for High-Performance Computing, Networking, Storage and Analysis (SC)	Proceedings	1
International Conference on Computing, Communication, and Automation (ICCCA)	Proceedings	1
International Conference on Intelligent Information Processing (ICIIP)	Proceedings	1
International Conference on Networks and Advances in Computational Technologies (NetACT)	Proceedings	1
International Journal of Open Source Software and Processes	Journal	1
International Workshop on Release Engineering (RELENG)	Proceedings	1
Brazilian Symposium on Software Engineering (SBES)	Proceedings	1
Empirical Software Engineering	Journal	1
Information and Software Technology	Journal	1

(3) Citation impact

Figure 2.17 highlights the most cited papers classified as bug prediction. Figure 2.17 shows only the most cited papers based on the category's h-index. In contrast, Figure 2.18 shows all papers on bug localization with at least one citation. The citation count of each paper was obtained from the respective publisher or indexing engine (e.g., ACM, IEEE Xplore, and Scopus), which is subject to change at any point in time. Overall, six papers with more than seven citations were identified each (listed in Table 2.19). The total number of citations for all papers with at least one citation (Figure 2.18) is 217, and the average number of citations per paper is approximately 8.

(4) Geographical distribution

Table 2.22 highlights the most active countries in the bug localization research domain. It identified 15 countries from all the extracted relevant papers in the bug prediction research domain. China is the most active country with eight published papers, followed by the United

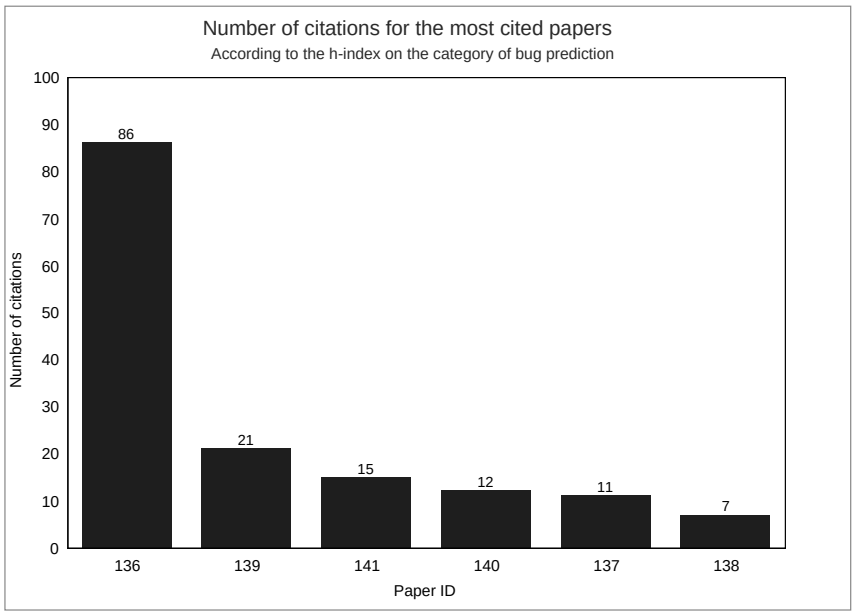

Figure 2.17: Number of citations for the most cited papers on bug prediction

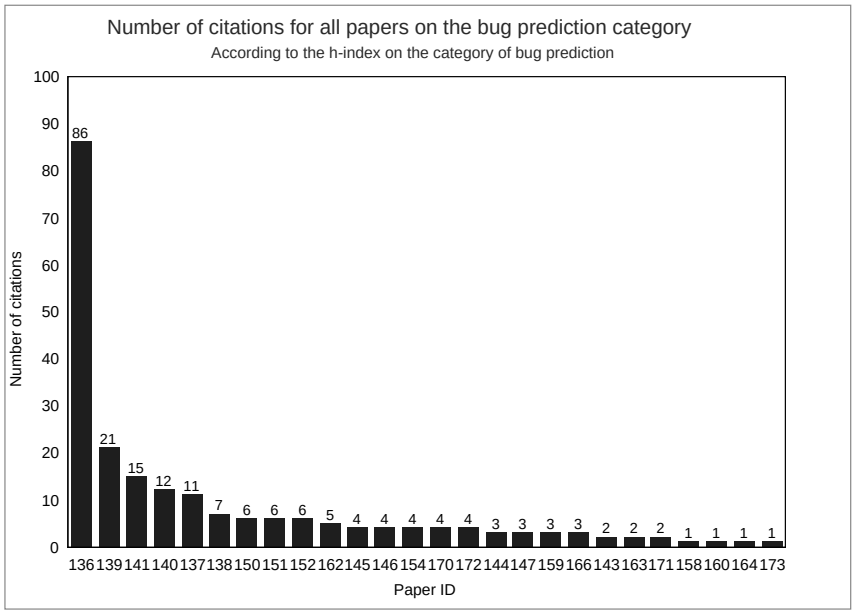

Figure 2.18: Number of citations for all cited papers on bug prediction

States of America and Canada with six published papers. India is the fourth most active country with three published papers. The selection of the most cited papers, according to the h-index, comes from only 4 of the 15 countries: Canada and China, with 2 of the most cited papers each, followed by Switzerland and the United Kingdom, with one high cited paper each.

Table 2.22: Most active countries in bug prediction research

Country	Number of published papers
China	8
United States	6
Canada	6
India	3
Singapore	2
United Kingdom	2
Italy	2
Austria	1
Japan	1
Greece	1
Brazil	1
Jordan	1
South Korea	1
Turkey	1
Switzerland	1

2.3 Threats to Validity

Secondary studies, as systematic mappings, are often subject to threats to validity due to the nature of research. The terms used in the search string were derived using the PICO criteria (population, intervention, comparison, outcome) to minimize the threats to construct validity of the presented systematic mapping. When defining the search strings, different terms that could represent the same aspect related to software bugs were considered and added to the string using the boolean operator OR. To maximise the external validity of the presented mapping a thorough protocol description was used, with details related to the venues and databases used for the search process, and a set o inclusion and exclusion criteria, which can be easily reconstructed and extended by other researchers. Two different researchers were involved in the selection process of each paper.

2.4 Final Considerations

This chapter presents a systematic literature mapping that resulted in a broad view of how software bugs are addressed in Software Engineering research. The works analyzed in the systematic mapping were classified into four general categories: (1) Bug Localization, (2) Bug Prediction, (3) Automatic Repair and Automatic Patch Generation, and (4) Analysis of Bugs and Bug Fixes.

Seven works were classified in the first category, Bug Localization. These works contribute

differently to the literature: analyses, tools, models, and techniques. Among the studies on bug localization, important topics for discussion were identified. One of the topics is the location of bugs in programs that involve more than one programming language. For example, Java has libraries that use legacy code written in C to interface with the operating system. When a bug occurs in the interaction between code written in different languages, it is more difficult to find and correct it. Another topic that is discussed is the reproduction of bugs to localize them. When a bug leads to a failure on the client side, away from the developers, and while the program is being used, the programmer must reproduce it to learn how to fix it. However, this reproduction is not easy.

Moreover, it is not easy because there is a lack of data about the failure generated by the bug, or there is not enough time to observe the occurrence of that failure. Some bugs are difficult to locate because they do not cause apparent failures. For example, bugs that decrease system performance. Bugs related to performance are common because of the complexity of current software systems. Still, they are difficult to locate as the only symptom of this type of bug is its performance drop, which can go unnoticed by the user.

It was observed that most studies related to bug localization are limited to the academic environment. It means that all the findings made in the research analyzed in the presented mapping are not being used in the software industry. Some studies try to narrow the gap between academia and industry, but none is significant enough to be adopted on a large scale. For bug localization approaches, the test suite is very important. It was observed that the larger the set of tests, the greater the coverage of execution paths, which increases the accuracy of identification approaches. However, it is tough for large programs to create a set of test cases that can execute most of the possible execution paths. The difficulty increases even more if the complexity of these systems is considered. Also, it was observed the use of several datasets for the validation of proposals for bug localization. The high number of datasets can hinder the comparison of approaches.

Six studies related to bug prediction were reviewed throughout the systematic mapping. Those works contributed to approaches, frameworks, case studies, and analyses. The most common topics in the literature on bug prediction are primarily related to algorithms to improve the prediction model's performance, such as the use of deep learning algorithms. Prediction performance is related to the quality of the prediction at the time of the commit, hence the term just-in-time prediction.

Another important topic that was observed is the creation of new prediction models. The creation of prediction models is complex because the data structures used in constructing these models are complex. Software bug data is generally not well structured or standardized. It makes the selection of characteristics more difficult, which hinders prediction. Interesting topics on bug prediction have been identified, such as the analysis of commits that induce system failures and the use of biological metrics in prediction.

It was observed that the accuracy of bug prediction models depends on the ability to select characteristics that represent the structure of the data about the bugs well. When redundant or

irrelevant characteristics are selected, the prediction is impaired and presents inaccurate results. Another critical point about defect data is the balancing of that data. Unbalanced means there are more software modules without defects than modules with defects in a system. Moreover, the classification result is inaccurate when this unbalanced data set is used in the prediction models. Classification refers to the automatic classification of code elements as defective or not. The imbalance problem is worse when considering proprietary software, generally larger systems. For example, the defect rate is usually low in a commercial system. It means that a few areas of code are buggy. Large systems have a more significant imbalance concerning bug data, and then the prediction models can give inaccurate results.

For the bug prediction to achieve good results, the data used by the models must be accurate. However, this data is usually noisy. For example, it is common for a program change to fix a bug to be committed along with code not related to the bug. Improving the accuracy of defect prediction is crucial because research efforts in this area will only be adopted when developers are convinced that it is worthwhile, which means high precision in the prediction results.

Fourteen works were classified as automatic repair. These works contributed to systems, techniques, tools, approaches, and bots for automatically repairing bugs in software systems. Automatic repair research is important as it aims to reduce the manual effort required to find and fix bugs. This effort is high, causing development costs to increase. Furthermore, if bugs are automatically fixed correctly and they are directly linked to software quality, this means that automatic repair has the potential to help improve software quality.

Papers classified as automatic repair in the presented systematic mapping discuss important topics such as patch complexity for fixing bugs, repairing specific types of bugs, the size of the search space for patch generation, and scalability. The concern with the complexity of patches is important because a developer analyzes every patch before being accepted for a bug's correction. So, the size and complexity of the patch can influence whether or not it will be accepted. Also, it has been observed that most of the proposed approaches for automatic software repair generalize the type of bug they try to fix. However, better results can be achieved if the approach focuses on a single bug type.

To fix a bug, the code must be changed. If an automatic repair approach considers every possible code change, the possibilities are endless. Considering an infinite set of possible changes, most will not be able to fix the bug. Since it is impossible to test an infinite set of bug-fix possibilities, automatic repair techniques often set restrictions on the type of modifications they can make to the code to fix a bug. Moreover, these restrictions are important to reduce the search space size for corrections and make automatic repair possible. The search space size problem can be solved by improving the quality of the approach used for bug localization. Mainly because to fix a bug automatically, it is necessary to locate it in the code. That is, the efficiency of the bug localization approach impacts the efficiency of the automatic repair techniques.

Scalability is also a topic widely discussed in the literature on automatic repair. Automatic repair techniques based on semantics are less efficient when applied to large systems. However, in small systems, they can repair high amounts of bugs. Repair techniques based on testing are

also effective when applied to large software, but they can only fix bugs in a single location in the code. When it comes to semantic-based techniques, this restriction ceases to exist, as semantic-based techniques can correct bugs that are scattered in various regions in the code.

Analyzing the studies classified as Analysis of Bugs and Bug Fixes, it was observed that there are several aspects related to bugs that can be exploited to help both in the maintenance and software evolution. However, there is no consensus on how bugs are fixed. Understanding how bugs are fixed is essential for research into automatic repair.

The presented systematic mapping does not show a clear trend regarding publications related to bugs in software systems. Few publications had a significant impact in terms of the number of citations, regardless of the category defined in the mapping. However, the number of articles published on automatic repair and analysis of bugs and bug fixes increased significantly between 2018 and 2019, while the number of publications on bug prediction and localization remained almost stable.

It was observed that high-impact articles (considering the h-index) are mostly published in a specific conference (ICSE). They are especially considering automatic repair papers. Papers on automatic repair had the highest number of citations. Bug localization articles are generally less often cited.

Demographically, China and the United States are ahead in the number of publications on software bugs. Despite this, several other countries have also published papers in this area of research. Some countries have published a considerable number of articles, averaging one or two publications annually over the past five years: Canada, South Korea, Japan, and Singapore.

In the systematic mapping presented, 444 articles were selected, and based on the h-index of each, 37 of them were considered in the final mapping (See Appendix A). Several research challenges were identified in software bugs, specifically, in each category considered. It was possible to trace the geographical distribution and the best publication places in the area and to have an idea of the impact of each work in terms of the number of citations. With that, it was possible to review the main relevant aspects of bugs.

> One of the main observations about the systematic mapping results is that test cases are important in most bug-related activities, for example, bug localization, automatic program repair, and automatic patch generation.

It was observed that software testing is an essential aspect for the vast majority of bug fixing activities, whether for automatic repair, bug localization, or prediction. The quality of the software and the quality of the approaches to solving bugs in software are directly linked to the test suite's quality. The next chapter presents the most relevant concepts of software testing.

CHAPTER

3

Background and Related Work

This chapter delves into the foundational concepts and prior research that underpin the study of automatic test case generation guided by ensemble bug prediction models. It begins with an overview of Search-Based Software Testing (SBST), exploring its principles, techniques, and the evolution of methods for automatic test case generation. The chapter then examines the field of software defect prediction, differentiating between single prediction models and ensemble prediction models. By reviewing related work in both SBST and defect prediction, this chapter aims to contextualize the proposed research within the broader landscape of software engineering. The discussions highlight the advancements, limitations, and ongoing challenges in these areas, establishing a clear rationale for the innovative approaches introduced in this dissertation. Through this comprehensive background, it is possible to gain a deeper understanding of the technical and theoretical underpinnings that inform the development and evaluation of the proposed methodologies.

3.1 Search-Based Software Testing (SBST)

This section presents the Search-Based Software Engineering (SBSE) concept, which combines meta-heuristic search algorithms with software engineering tasks to address complex problems. Harman and Jones (2001) introduced the term SBSE in 2021, officially establishing SBSE as a recognized field of research. Meta-heuristic search algorithms are powerful computational techniques specifically developed to discover optimal or nearly optimal solutions to problems that are too complex for conventional analytical methods (M. Almufti et al., 2023).

Applying optimization techniques in software engineering traces back to 1976 with the work of Miller and Spooner (1976). This work is significant because it represents one of the first attempts to apply numerical optimization techniques within software engineering. Miller and Spooner (1976) used optimization techniques to generate floating-point test data to cover different execution paths within a software program. This endeavor was groundbreaking because

it demonstrated the potential of optimization techniques to enhance the software testing process, laying the groundwork for what would later evolve into SBSE.

Significant advancements happened within the domain of search-based software engineering over the years (Ozkaya, 2021, Gupta et al., 2021, Mazzonetto et al., 2022, Afan et al., 2023). For example, generating test data to cover software paths and applying optimization techniques to this end marked a significant advancement in the field (Fan et al., 2021, Semujju et al., 2023). Another important advancement was the integration of meta-heuristic search algorithms in software testing, a subfield of SBSE now recognized as Search-Based Software *Testing* (SBST).

Meta-heuristic search algorithms are sophisticated computational strategies designed to solve complex optimization problems that are otherwise challenging to tackle using traditional deterministic algorithms (Ribagin and Lyubenova, 2021). These algorithms, including but not limited to genetic algorithms (Fan et al., 2021), simulated annealing (Henderson et al., 2003a), and tabu search (Nurmela and Ostergard, 1995, Cole Smith et al., 2011), that are characterized by their ability to efficiently explore vast search spaces to find near-optimal solutions (Ribagin and Lyubenova, 2021). SBST applies these algorithms to various testing tasks such as test case generation (Mishra et al., 2019), prioritization, and selection (Panwar et al., 2018, Di Nucci et al., 2020, Ali et al., 2021), aiming to optimize testing efforts in terms of effectiveness and efficiency.

The significance of SBST within the broader context of Search-Based Software Engineering (SBSE) is underscored by its success and popularity (McMinn, 2011, Harman et al., 2012, Sar, 2016). SBSE is an approach that applies search-based optimization techniques to address various software engineering problems, from requirements engineering to software maintenance and refactoring (McMinn, 2011). The particular focus on SBST within SBSE highlights the critical role of testing in software development and the substantial benefits that can be derived from automating and optimizing testing processes.

Structural testing is a fundamental aspect of software testing that emphasizes examining the internal mechanisms of a program rather than its external functions (Jones et al., 1996, Wegener et al., 2001, Tonella, 2004a, McMinn, 2004). In Search-Based Software Testing (SBST), structural testing focuses explicitly on creating methods to automatically generate test data, notably numeric inputs (Wegener et al., 2001). The initial focus of Search-Based Software Testing (SBST) research on structural testing was to create efficient algorithms for generating test input values (Jones et al., 1996). These inputs would explore different paths within the software's structure. The goal was to achieve extensive path coverage, covering all possible paths to thoroughly test the code for errors, bugs, or unexpected behaviors. This approach tries to identify areas of the code that are rarely or never executed under normal circumstances, potentially containing bugs.

Through the automation of test data generation, early research in SBST aimed to decrease the amount of manual work required by testers and improve the efficiency of the testing process (Jones et al., 1996, Wegener et al., 2001, Tonella, 2004a). Automation ensures software systems' robustness, reliability, and proper functioning in various conditions.

3.1.1 Automatic Test Case Generation

One of the pioneering works in Search-Based Software Testing (SBST) was introduced by Korel (1990), aimed at enhancing the efficiency and effectiveness of software testing. The authors instrument the program under test, a technique involving the insertion of additional code to monitor program execution. Instrumentation is crucial for collecting data on which parts of the code are executed during testing. After instrumentation, the software is tested using randomly generated inputs, which allows for evaluating the software's behavior under various conditions without any bias towards specific input values.

A key component of this approach is the software tester's selection of a target path, which represents a specific sequence of executions within the software that needs thorough testing. If the random inputs completely cover this target path, they are recorded for future reference. This optimal outcome verifies that the test has achieved all intended path targets per the tester's goals. However, it is common for executions to deviate from the target path at some point. When such deviation occurs, a branch distance metric is employed to quantify how far the execution has strayed from the target path. A local search algorithm is applied to adjust the random test inputs iteratively using this metric. These adjustments aim to redirect the test execution back towards the defined initial path and to maintain coverage of the software's functionalities up to the point of divergence. This methodical blend of random testing, branch distance analysis, and local search optimization delineates a strategic approach to achieving high path coverage in software testing.

Xanthakis et al. (1992) also presented one of the first works in software testing. Initially, their approach uses a random search strategy to traverse a selected path within the software's execution flow. This initial step is crucial as it allows for an unbiased and comprehensive exploration of the path, free of preconceived biases that might influence the selection process. All branch predicates encountered along the path are extracted during this exploration phase. Branch predicates are conditional statements that determine the execution flow in software, acting as decision points that can lead to different outcomes based on their evaluation (Taylor and Li, 2011).

Following extracting these predicates, Xanthakis et al. (1992) used a search algorithm to satisfy all extracted branch predicates simultaneously. This is an important step as satisfying branch predicates is synonymous with ensuring that the conditions for each decision point along the selected path are met, enabling the testing process to cover a wide range of scenarios and potential execution outcomes. The search algorithm is designed to systematically identify input values or conditions that will fulfill all the branch predicates, thus ensuring comprehensive testing of the software's execution paths. This methodological approach not only enhances the thoroughness of the software testing process but also contributes to identifying and rectifying potential errors or vulnerabilities within the software's code.

In software testing, the process of manually selecting program paths to identify bugs or ensure code quality can be labor-intensive, especially within the context of complex software systems (Rajagopal et al., 2023). Korel (1992) addressed this challenge by introducing a goal-oriented technique, a method that significantly streamlines the testing process. The essence of Korel

technique lies in its shift of focus from the manual selection of program paths to the definition of a target goal by the developer, such as a specific statement within the code that needs to be tested. Upon establishing a target goal, the technique leverages the program's control flow graph, which represents all possible paths through the program, delineated by nodes (statements or decisions) and edges (the flow between these nodes). It systematically filters out branches of the control flow irrelevant to achieving the target goal. These non-relevant branches are those that do not need to be traversed or satisfied to reach the specified goal, thereby narrowing down the focus to only those paths that are of interest.

Following the filtration process, a search algorithm is applied. This algorithm is designed to generate a set of inputs that will guide the program's execution along the relevant branches only, ensuring that the target goal is met. This approach not only optimizes the testing process by reducing the manual effort involved but also enhances the efficiency and effectiveness of testing by focusing on specific, goal-oriented outcomes. Through this technique, Korel (1992) contributed to the advancement of software testing methodologies, offering a practical solution to the challenges posed by the complexity of modern software systems.

Pargas et al. (1999) introduced a method focused on enhancing software testing by targeting specific structural components within the code, such as individual statements. The core idea revolves around control dependency, which refers to the relationship between different parts of a program where the execution of one segment (a node) directly influences whether another segment is executed. In simpler terms, a structural point in the code, like a specific statement, depends on certain conditions being met, which are determined by the execution of other control-dependent nodes. The proposed technique aims to increase testing effectiveness by selecting test inputs that maximize the execution of these control-dependent nodes. The underlying hypothesis is that a test input that triggers more of these nodes will likely be more effective in reaching and, thus, testing the targeted structural point. This approach is beneficial in identifying test cases that are more likely to expose bugs related to the program's control flow.

However, as described by Pargas et al. (1999), the technique does not offer a mechanism to quantify how close a test input is to covering a node. This means that while it can guide the selection of test inputs towards those that execute more control-dependent nodes, it does not provide feedback on the proximity of these inputs to the structural point in question. This gap highlights a potential area for further refinement in the methodology, where developing metrics or indicators of "closeness" could significantly enhance its utility in practical testing scenarios.

In the Object-oriented paradigm (OOP), features such as inheritance and polymorphism introduce a level of complexity not found in procedural programming (Babu and Singh, 2022). These features allow objects to inherit properties and behaviors from parent objects and present different implementation methods based on the object's type. Consequently, the state of an object (i.e., the values of its attributes) becomes crucial in testing because specific methods may only be executable or relevant when an object is in a particular state (Huang et al., 2021). This requirement significantly complicates the testing process compared to procedural programs, where the focus is typically on testing functions with a set of input values (Scalabrino et al.,

2016). In OOP software, a test case transcends the simplicity of a value set. It evolves into a sequence of method calls, each potentially altering the object's state and specifying appropriate parameters for those methods (Zivkovic and Zivkovic, 2020). This sequence must be carefully designed to ensure that the object reaches the necessary states to cover the target goals of the test, such as executing a particular method or achieving a specific outcome. Thus, the complexity of testing OOP software is higher, necessitating a more sophisticated approach to test case generation that accounts for the intricate interplay of object states, method calls, and the effects of inheritance and polymorphism (Chen et al., 2022a,b, Panigrahi and Jena, 2023).

Tonella (2004b) was the first to apply meta-heuristic search algorithms, specifically genetic algorithms, to test Object-Oriented (OO) software. Tonella's method uses genetic algorithms to generate test cases to achieve specific testing goals, such as covering different code branches. These algorithms simulate the process of natural selection by applying evolutionary operators – crossover and mutation – to evolve a population of test cases over generations. Each test case is an *individual* in this population, and the goal is to develop these individuals towards achieving higher coverage of the target goals. The process culminates in the union of all the evolved test cases into a test suite. However, this approach has scalability issues for software with several target goals due to the potential exponential growth in the required test cases.

In contrast, Wappler and Lammermann (2005) initially adopted standard evolutionary operators in their approach, which led to the generation of infeasible test cases that could adversely affect the evaluation of test case quality or *fitness*. Recognizing this problem, they proposed using genetic programming in a subsequent work (Wappler and Wegener, 2006). Genetic programming, an evolutionary algorithm, enhanced the test case generation process, ensuring the individual's (test cases) feasibility and thereby improving the test suite generation process (Wappler and Wegener, 2006). This evolution in methodology highlights the ongoing refinement in applying evolutionary algorithms to software testing, aiming to optimize test case generation while ensuring the practicality and applicability of the generated test cases.

Tonella (2004b) approach, as discussed previously, involves an iterative method where individual test cases are generated and applied one after another, each targeting a specific coverage goal. This implies prioritizing or ordering coverage goals, which can be complex and time-consuming, as it requires determining the most effective sequence to achieve comprehensive testing.

In contrast, Fraser and Arcuri (2011, 2013) introduced the Whole Test Suite (WTS) approach, which improved the test generation process by evolving entire test suites simultaneously rather than individual test cases. In this context, a *test suite* refers to a collection of test cases, and the *individuals of a population* represent these test suites in the evolutionary algorithm. Each test case within these suites is a sequence of calls designed to test the software. The primary innovation of WTS is its concurrent targeting of all testing goals, thereby eliminating the necessity to prioritize or sequence coverage goals. This methodological shift was significant because it addressed the inefficiencies associated with the iterative generation and application of test cases by adopting a more global perspective on test suite evolution.

The empirical evidence provided by Fraser and Arcuri (2013) supports the superiority of the WTS approach over the iterative generation of individual test cases, demonstrating its effectiveness in achieving test coverage more efficiently. This underscores the advancements in automated testing strategies, highlighting the importance of evolving methodologies to enhance the effectiveness and efficiency of software testing processes.

In the domain of Search-Based Software Testing (SBST), genetic algorithms are prevalent. Inspired by natural selection and genetics principles, these algorithms can explore complex search spaces to identify optimal or near-optimal solutions for software testing problems, such as generating test cases that maximize code coverage or uncover faults efficiently (Zhan, 2022).

There is a diversity of algorithmic strategies beyond genetic nature-inspired algorithms for SBST. Nature-inspired algorithms encompass a broad array of computational techniques that mimic biological processes, geological patterns, or physical phenomena to solve optimization problems (Mohammed Aarif et al., 2022). This category includes, but is not limited to, algorithms based on the behavior of ants – Ant Colony Optimization (Dorigo and Di Caro, 1999), the flocking of birds – Particle Swarm Optimization) (Bansal, 2018), or even the annealing process of metals – Simulated Annealing (Henderson et al., 2003b). Including random-search algorithms (Zabinsky, 2011), which rely on purely stochastic methods rather than mimicking natural processes, indicates the breadth of approaches considered in the field of SBST. The notion that no single algorithm can universally outperform others across all possible domains is a fundamental principle in computational optimization, known as the *No Free Lunch* theorem (Adam et al., 2019). This principle posits that an algorithm's effectiveness is contingent upon the specific characteristics of the problem domain it is applied to, thereby necessitating a diverse toolkit of algorithms to tackle various challenges in SBST effectively.

3.1.2 Approaches to Test Case Generation in SBST

3.1.2.1 Random Search

Random search is a fundamental concept within evolutionary algorithms, which are computational methods inspired by the principles of natural evolution (Zabinsky, 2011). Unlike more sophisticated evolutionary strategies that employ crossover, mutation, and selection to iteratively refine a population of solutions towards an optimum, random search adopts a more straightforward approach. Crossover refers to the combination of genetic material from two parent solutions to produce offspring, mutation involves the random alteration of solution components to introduce variability, and selection is choosing superior solutions based on a fitness criterion for further breeding. These mechanisms collectively drive the evolutionary process towards convergence on high-quality solutions by exploiting and exploring the solution space.

In contrast, random search avoids these mechanisms in favor of a straightforward replacement strategy. This strategy involves generating solutions randomly within the search space and evaluating their quality based on a predefined objective function (Zabinsky, 2011). If a newly generated random solution performs better than an existing solution, the former replaces the latter. This process is repeated iteratively. The absence of crossover, mutation, and selection

means that random search does not explicitly leverage information from the current population to guide the search process. Instead, it relies on the stochastic sampling of the search space, making it a baseline strategy against which the performance of more complex evolutionary algorithms can be benchmarked (Zabinsky, 2011). Despite its simplicity, random search can be surprisingly effective for specific optimization problems, particularly those where the solution space does not favor structured exploration mechanisms of more sophisticated algorithms.

3.1.2.2 Local Search Algorithms

3.1.2.2.1 Hill Climbing

Hill Climbing is a heuristic search algorithm used in mathematical optimization problems. This algorithm makes incremental changes to a solution, aiming to find a better solution by evaluating the *fitness* or value of neighboring solutions (Hernando et al., 2018). The fitness function is a crucial component, as it quantifies the quality of solutions, guiding the search towards the optimum (Basseur and Goeffon, 2015, Henaux et al., 2020). In a simple *1-dimensional* scenario, the algorithm evaluates adjacent solutions (neighbors) to the current solution, one on each side (right and left), to determine if either neighbor represents an improvement in the fitness value. If an improvement is found, the current solution is updated to this better neighbor (Basseur and Goeffon, 2015).

A significant limitation of the Hill Climbing algorithm is its *myopic* nature. It lacks a global perspective of the problem's landscape, representing solutions and their fitness values (Duvivier et al., 1996). The algorithm does not make assumptions about this landscape's overall shape or features, such as multiple peaks (local optima) or valleys. Consequently, it is prone to becoming trapped in a local optimum – a solution that is better than its immediate neighbors but not the best possible solution across the entire landscape as in Figure 3.1 (McMinn, 2011, Zhang et al., 2021). This limitation stems from the algorithm's strategy of only accepting new solutions that offer an immediate improvement, without mechanisms to escape or bypass local optima in search of the global optimum (Zhang et al., 2021).

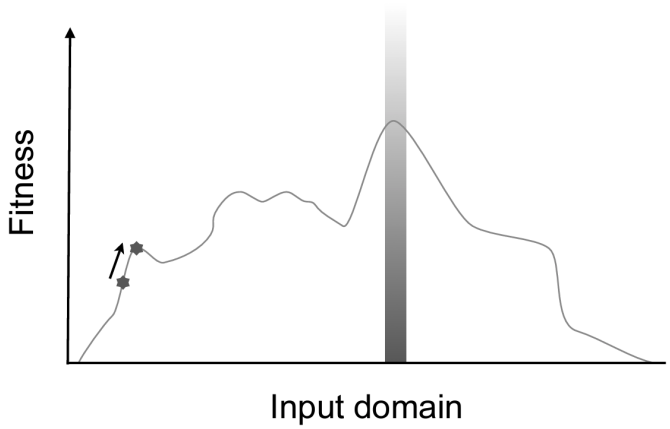

Figure 3.1: Hill Climbing: Climbing to a local optimum – From McMinn (2011)

3.1.2.2.2 Tabu-search

Tabu-search is an advanced local search algorithm distinguished by its use of adaptive memory and responsive exploration mechanisms to navigate the solution space of optimization problems (Tsai and Chiang, 2023). It begins with a randomly generated solution and is then evaluated based on a predefined criterion. Subsequently, the algorithm iterates by developing and assessing multiple new solutions. If it identifies a solution superior to the current one, it updates it with this alternative. This process of generation and evaluation continues iteratively. However, like the Hill Climbing algorithm, Tabu-search is susceptible to the pitfall of converging on local optima – situations where the algorithm might oscillate between two or more optimal solutions within a local scope but not necessarily the best overall solution.

To mitigate its limitations, Tabu-search employs a strategic mechanism: it maintains a *tabu list*, a form of adaptive memory that records previously visited solutions. This list acts as a constraint, preventing the reselection of solutions that have already been explored. By doing so, the algorithm effectively broadens its search space, thereby enhancing the probability of escaping local optima and moving towards the global optimum solution (Tsai and Chiang, 2023).

The feature of maintaining a tabu list to avoid cyclic exploration distinguishes Tabu-search from more straightforward local search strategies. It contributes to its efficacy in solving complex optimization problems by ensuring a more exhaustive and less redundant search through the solution space (Tsai and Chiang, 2023).

3.1.2.2.3 Simulated Annealing

Simulated Annealing is an optimization technique inspired by the annealing process in metallurgy (Henderson et al., 2003b, Bo Zhang and Chen Wang, 2011, Arcuri, 2013). This meta-heuristic algorithm is designed to find a good approximation to the global optimum of a given function in a large search space.

Simulated Annealing introduces a mechanism to escape potentially suboptimal solutions, unlike the Hill Climbing algorithm, which incrementally adjusts solutions towards a local optimum and may get stuck in it. This mechanism is controlled by a parameter known as *temperature*, representing the algorithm's willingness to accept worse solutions than the current one (Henderson et al., 2003b).

Initially, the temperature is set high, allowing the algorithm to accept worse solutions with relatively high probability. This feature enables the exploration of a broader portion of the search space, potentially avoiding local optima and discovering better solutions.

As the algorithm progresses, the temperature gradually decreases according to a *cooling schedule*. This reduction in temperature reduces the likelihood of accepting worse solutions, thereby focusing progressively on areas of the search space with higher-quality solutions. When the temperature reaches zero, the algorithm essentially behaves like the Hill Climbing algorithm, accepting only solutions that improve upon the current one.

This dynamic adjustment of solution acceptance criteria based on the temperature parameter allows Simulated Annealing to balance exploration and exploitation throughout its execution, enhancing its ability to identify near-optimal solutions in complex optimization problems (Henderson et al., 2003b).

3.1.2.3 Global Search Algorithms

3.1.2.3.1 Genetic Algorithm (GA)

Genetic Algorithms (GA) are prominent examples of evolutionary algorithms (EAs), favored for their simplicity and effectiveness across various fields (Garai, 2022). The process begins with generating an initial, random population of potential solutions, each represented by a set of parameters or *genes*, with the population size.

The selection of individuals for reproduction is based on some well-defined strategy, such as rank-based selection, elitism, or tournament selection, which determines the fitness or quality of solutions (Goldberg and Deb, 1991). Following selection, a crossover operation is performed on pairs of selected individuals to produce offspring $(o1, o2)$, incorporating genetic material from both parents. This crossover occurs with a certain probability, allowing for the introduction of new genetic combinations into the population.

Mutation, another key mechanism, introduces random changes to the offspring's genes at a probability (mp), typically set to the inverse of the number of genes $\left(\frac{1}{n}\right)$, to maintain genetic diversity and prevent premature convergence on suboptimal solutions (Stoltzfus, 2021).

The mutated offspring are then added to the population for the next iteration. The algorithm iteratively evaluates each individual's fitness in the population to optimize a specific set of objectives. This iterative selection, crossover, mutation, and evaluation process continues until a termination condition is met, such as a maximum number of generations or a satisfactory fitness level, ultimately leading to optimized solutions.

3.1.2.3.2 Monotonic GA

Monotonic GA enhances the Standard Genetic Algorithm (GA) 's effectiveness for solving optimization and search problems (Lengler, 2020). The Standard GA operates through a selection, crossover, mutation, and evaluation cycle to evolve solutions towards optimality (Section 3.1.2.3.1). However, its efficiency can be hindered by including both offspring in the subsequent generation, regardless of their fitness levels (Alam et al., 2020). This can lead to suboptimal solutions propagating, slowing the convergence towards the best solution.

To address this issue, a variant known as the *monotonic* version of the Standard GA has been proposed (Lengler, 2020). This variant introduces a critical modification in the population update phase. After the mutation and evaluation of each offspring, a decision is made between including either the best offspring or the best parent in the next generation. This departs from the traditional approach, where both offspring are automatically included.

The rationale behind this modification is to ensure that the population does not regress in terms of fitness. By selectively including only the individuals that represent an improvement (or at least, no decline in fitness), the algorithm aims to maintain or enhance the overall quality of the population. This methodical curation of the population is designed to accelerate the convergence towards optimal solutions by reducing the likelihood of weakening the gene pool with less fit individuals.

3.1.2.3.3 Many-Objective Sorting Algorithm (MOSA)

The Many-Objective Sorting Algorithm (MOSA) diverges from the approach of generating entire test suites as proposed by Fraser and Arcuri (2013) by treating each coverage goal as a distinct optimization objective (Panichella et al., 2015).

MOSA provides a significant shift because it allows for a more granular focus on achieving coverage goals rather than optimizing a collective suite of tests. MOSA, an adaptation of the Non-dominated Sorting Genetic Algorithm II (NSGA-II) (Mkaouer and Kessentini, 2014), employs a preference sorting criterion. This criterion is pivotal as it prioritizes the selection of the best individual tests that achieve coverage of previously uncovered targets without considering the dominance relationships these tests may have with other tests in the population. This is different from traditional methods, where the dominance relation often influences the selection process.

A notable feature of MOSA is its use of an *archive* to store tests that successfully cover new targets. This archival system helps retain the optimal tests discovered during each algorithm iteration. By maintaining a record of these tests, MOSA ensures that the progress toward achieving the target coverage is preserved between iterations. This approach facilitates a systematic and iterative enhancement of the test suite, with each cycle aiming to extend coverage to new targets while preserving the achievements of previous iterations.

MOSA's strategy of treating each coverage goal independently, combined with its archival mechanism, represents a nuanced approach to optimizing test suites. This approach allows for targeted improvements in test coverage and ensures the preservation of optimal solutions throughout the optimization process.

3.1.2.3.4 Dynamic Many-Objective Sorting Algorithm (DynaMOSA)

The Dynamic Many-Objective Sorting Algorithm (DynaMOSA) is an advanced many-objective solver explicitly designed for the test case generation problem, focusing on coverage testing. It extends the Many-Objective Sorting Algorithm (MOSA) (Panichella et al., 2015) by incorporating a dynamic selection of coverage targets based on the control dependency hierarchy, significantly enhancing its effectiveness and efficiency, especially under limited search budgets (Panichella et al., 2018).

Unlike MOSA, which treats all coverage targets as independent objectives from the start, DynaMOSA dynamically reduces the number of objectives optimized at each generation, considering structural dependencies among targets. This approach allows DynaMOSA to focus only on coverage targets with minimum approach levels, thereby reducing the computational load compared to MOSA, which computes fitness scores for all uncovered targets.

DynaMOSA outperforms MOSA in several aspects. Empirical studies done by Panichella et al. (2018) show that DynaMOSA achieves higher code coverage across various criteria, including statement, branch, and mutation coverage, demonstrating improvements in effectiveness over MOSA.

For instance, DynaMOSA leads to statistically significantly higher statement coverage in 12% of the classes compared to MOSA, with enhancements ranging between 0.10 and 88%. Moreover, DynaMOSA's ability to dynamically select fewer mutants as objectives in the initial generations, based on control dependencies, allows for more efficient use of computational

resources and higher effectiveness in killing mutants than MOSA.

DynaMOSA's dynamic selection mechanism, which leverages the control dependencies between test targets, is a key differentiator from MOSA. It enables DynaMOSA to prioritize and optimize the most relevant objectives at each generation.

3.2 Software Defect Prediction

Software Defect Prediction models (a.k.a bug prediction models) can be used in software engineering to enhance the testing phase of software development (Perera et al., 2023a). These models employ machine learning techniques to analyze historical data related to software development processes, including code complexity, change history, and previous defects. By scrutinizing this data, prediction models can identify patterns and predict which components or sections of the software are more likely to contain defects or bugs (Mohanty and Ravi, 2017, Ihsan Aquil, 2020).

The primary utility of defect predictors lies in their ability to guide test practitioners in prioritizing efforts. In software testing, resources such as time are often limited (Paterson et al., 2019). Traditional testing approaches that aim to cover all parts of the software equally could be more efficient and lead to the oversight of critical defects in more complex or change-prone areas. By highlighting the parts of the software that are most likely to be defect-prone, bug prediction models enable testers to allocate their resources more effectively (Ren, 2023). This targeted approach ensures that the software's most critical and vulnerable areas receive the highest testing priority.

It is possible to use different modeling approaches to predict bugs, each serving its unique purpose (Kuhn and Johnson, 2013). (1) Classification models can be designed to categorize software components into two classes: defective or non-defective. This binary classification is foundational in software defect prediction, enabling the identification of areas of code that are likely to contain bugs. (2) Regression models aim to quantify the number of defects in a given instance. Unlike classification models that offer a binary outcome, regression models provide a numerical estimate of defectiveness, offering a more granular perspective on the expected defect count within software components. As detailed by (Weyuker et al., 2010), this approach allows for a more nuanced understanding of software quality, enabling precise resource allocation for defect remediation. (3) Ranking models prioritize software components based on their susceptibility to defects. This approach does not merely identify or count defects but orders instances by their likelihood of being defective. Yang et al. (2015) highlight the utility of ranking models in prioritizing testing and quality assurance efforts, ensuring that resources are allocated to the most defect-prone components first.

These three modeling approaches offer complementary tools for addressing the multifaceted challenge of software defect prediction. Each contributes uniquely to enhancing software quality through informed decision-making and strategic resource allocation. In the following sections, we overview single and ensemble prediction models, focusing specifically on the models we used in our experiments.

3.2.1 Single Prediction Models

The goal shared by machine learning, pattern recognition, and data mining fields is the development of effective models based on datasets (Mendonça et al., 2024, Zhou, 2012). This objective is critical because the essence of these disciplines lies in their ability to learn from data, discern patterns, and make predictions or decisions without being explicitly programmed for specific tasks (Zhou, 2012). Machine learning involves algorithms that can improve their performance at some task with experience or data. Pattern recognition focuses on identifying patterns and regularities in data, which can be used for classification or categorization (Razzaghi et al., 2023). On the other hand, data mining is concerned with extracting useful information or knowledge from large volumes of data, which can then be used for decision-making, strategic planning, or other purposes (Duque et al., 2023).

Constructing *"good models"* means developing strategies that are not only accurate in their predictions or classifications but are also efficient, scalable, and robust against variations in data (Bognár and Fauszt, 2022). These models are built through a process of *training*, where the algorithm learns from a given dataset, and *validation*, where the model's performance is evaluated using a separate dataset. The ultimate goal is to generalize from the training data to new, unseen data, thereby providing reliable insights or predictions that can be applied in real-world scenarios (Zhou, 2012).

A *dataset* is a collection of data points used for training, testing, and validating models. These data points are organized into what are known as *feature vectors* (Gong et al., 2023). A feature vector is essentially a numerical representation of an object's characteristics or attributes, referred to as features. Each feature within a vector corresponds to a particular aspect of the object being described, and the value of each feature reflects the characteristic's quantitative or qualitative measure. For instance, in a data set comprising images of animals, each image (object) could be described by a feature vector where features might include aspects like color, size, or shape (Zhao et al., 2020).

A collection of feature vectors forms the dataset, the main source for prediction algorithms to learn patterns, make predictions, or classify objects. The concept of feature vectors is crucial because it translates real-world observations into a mathematical form that algorithms can process, automating decision-making and recognition tasks based on empirical data (Zhou, 2012). This transformation into a quantifiable format allows machine learning models to discern patterns or differences among the objects described, leading to insights or predictions relevant to the task.

A dataset's number of features or attributes is called *dimensionality* (Erokhin et al., 2022). For instance, if a dataset is composed of information on individuals' age, income, and education level, the dimensionality of this data set would be three, as it contains three distinct features. These features, interchangeably known as attributes, serve as the variables based on which analyses are conducted or models are built. Thus, as stated previously, the *feature vector* is a term used to describe the array or the list of values corresponding to the features of a single observation or instance within the dataset (Zhou, 2012).

This terminology emphasizes the vectorized nature of data representation in machine learning,

where each instance is considered a point in a multidimensional space, with each dimension corresponding to a feature. The term *instance* is synonymous with a data point or observation, encapsulating the values for each feature for a single entity. Lastly, using *sample* as an alternative to *dataset* highlights the statistical perspective, where a sample represents a subset of observations drawn from a larger population. This nuanced vocabulary is crucial for clear communication and understanding of prediction models, where concepts from statistics, computer science, and domain-specific knowledge converge (Zhou, 2012).

The term *model* refers to a computational representation of reality. These models are algorithms that, once trained on datasets, can make predictions or decisions without being explicitly programmed to perform the task. For example, a decision tree is a model that represents decisions and their possible consequences, including chance event outcomes, resource costs, and utility. It's a way to display an algorithm that only contains conditional control statements. A support vector machine (SVM) is another model used for classification and regression tasks. It works by finding the hyperplane that best divides a dataset into classes in the feature space. Each of these models, whether a decision tree or a support vector machine, serves the purpose of understanding the structure of the data or making predictions about future observations. The choice of model depends on the specific characteristics of the data and the problem at hand, with each model having its own set of assumptions, strengths, and weaknesses (Zhou, 2012).

In the following sections, we will briefly introduce each model used in the experiments conducted for this dissertation.

3.2.1.1 Classification and Regression Trees - CART

Classification and Regression Trees (CART) is a method used in machine learning to construct models that can predict outcomes based on input variables. This technique is distinguished by its non-parametric nature, meaning it does not assume a specific form for the underlying data distribution. Instead, CART analyzes the data at hand (training data) to build a model tailored to the particular characteristics of that data set. The training data comprises instances characterized by predictor variables and a known outcome, which the model uses to learn how to make predictions for new, unseen data (Trendowicz and Jeffery, 2013).

CART is part of a broader category of machine learning algorithms focused on creating decision trees. Decision trees are graphical representations that illustrate the decision-making process, where each node represents a decision based on a specific condition, and each branch represents the outcome of that decision leading to further decisions or outcomes (Trendowicz and Jeffery, 2013).

Among the various decision tree methodologies, C4.5 is a notable example. Developed by Quinlan (1986), C4.5 is a refinement of the basic decision tree approach that enhances the model's ability to handle discrete and continuous data, deal with missing values, and prune trees to avoid overfitting. While C4.5 is specifically a classification tree method, CART encompasses both classification and regression tasks (Trendowicz and Jeffery, 2013).

Traditionally, decision tree methods have been limited to regression-type or classification-type problem categories based on the nature of the problem they aim to solve: regression-type

problems and classification-type problems. In the context of regression-type issues, the objective is to predict the value of a continuous dependent variable. This prediction is based on analyzing one or more predictor variables, which can be continuous, representing a range of values, or categorical, representing distinct categories or groups. In this scenario, the decision tree structures its branches and nodes to model the relationship between these predictor variables and the continuous outcome, facilitating predicting future or unknown values of the dependent variable (Trendowicz and Jeffery, 2013).

Conversely, in classification-type problems, the focus shifts to predicting the category or class of a categorical dependent variable. Like regression-type problems, this prediction relies on one or more predictor variables that can be continuous or categorical. However, the end goal here is not to predict a continuous quantity but to classify the observations into predefined categories based on the values of the predictor variables. The decision tree method for classification problems organizes the data into a tree-like structure. Each path from the root to a leaf represents a rule that assigns a category based on the input variables (Trendowicz and Jeffery, 2013).

Both these approaches leverage the inherent structure of decision trees to model complex relationships between variables, enabling precise predictions and classifications based on the input data.

Proposed by Breiman et al. (1984), the Classification and Regression Trees (CART) model represents a significant advancement in statistical analysis techniques, particularly in addressing both regression and classification problems. This dual capability signifies that CART can handle datasets with a diverse range of variable types – both continuous and categorical.

Continuous variables can take on infinite values within a given range, such as age or income, allowing for fine-grained analysis. Categorical variables, on the other hand, represent distinct categories or groups, such as gender or marital status, which are essential for classification tasks (Trendowicz and Jeffery, 2013).

By overcoming the limitation of methods that are specialized to only one type of problem (either regression or classification), CART provides a comprehensive framework for building predictive models. This framework is particularly valuable in fields such as machine learning, where accurately predicting outcomes based on various data types can significantly enhance decision-making processes and predictive accuracy.

The Classification and Regression Trees (CART) process is done in two phases that enable the construction of models that effectively handle diverse data types. In the initial phase, CART constructs a tree model. This involves the recursive partitioning of the dataset into subsets based on the values of the input variables. The objective is to split the data in a manner that groups similar outcomes or responses, thereby increasing the homogeneity of the resultant subsets. For classification tasks, this means creating partitions that maximize the purity of the class within each subset.

In contrast, for regression tasks, the aim is to minimize the variance within each group. A top-down, greedy approach characterizes this phase. The best split at each node is chosen without revisiting previous decisions, leading to a fully grown, potentially overfit tree (Trendowicz and

Jeffery, 2013).

The second phase involves pruning the fully grown tree to mitigate overfitting. Pruning is achieved by systematically removing branches that contribute least to the model's predictive power based on a cost-complexity criterion. This process simplifies the model, enhancing its predictive accuracy on unseen data by retaining only the most informative splits. The outcome is a model that balances complexity with predictive performance, tailored to either classification or regression objectives (Trendowicz and Jeffery, 2013). In Algorithm 1, we show the CART process, followed by its explanation.

Algorithm 1 CART Algorithm – Adapted from Altay and Cinar (2016)

Input: Training data
Calculate $GINI(t)$ for all nodes
Compute $GINI(S, A_j) = \min_{i \in S} GINI(S, A_i)$
if attribute A_j has two nodes **then**
    Branch for attribute A_j
else
    Compute $GINI(t) = \min_{t \in A_j} IG(t)$
    Create two branches as $(t_1, \ldots, t_l)$ and $(t_{l+1}, \ldots, t_n)$ where $t_1, \ldots, t_n \in A_j$
end if
if GINI of the branch is 0 **then**
    Terminate this branch
end if
if all branches terminated **then**
    Terminate the algorithm
else
    Update S
    Go to Step 1
end if
Output: Decision tree

The CART algorithm constructs a decision tree by recursively splitting the training data based on attributes that lead to the highest Gini impurity decrease. Initially, the algorithm calculates the Gini impurity for all nodes. For each attribute A_j, the split that results in the minimum Gini impurity is selected. If an attribute can split into exactly two nodes, it branches on that attribute. Otherwise, it computes the Gini impurity for the potential splits within the attribute and branches into two segments based on the best-split point. Each branch is created by partitioning the training instances into two subsets. If a branch has zero Gini impurity, indicating perfect classification, that branch is terminated. The algorithm continues until all possible branches have been terminated, at which point the construction of the decision tree is complete (Altay and Cinar, 2016).

3.2.1.2 k-Nearest Neighbor - KNN

The k-nearest neighbor (KNN) algorithm is a non-parametric method used for classification and regression. The core principle underlying this algorithm is the assumption that objects similar in the input space tend to exhibit similarity in the output space as well (Kramer, 2013). This

means that the algorithm predicts the output for a new data point based on the outputs of its k closest neighbors in the training dataset. The *input space* refers to the feature space or the set of variables that describe the data points. For example, in a two-dimensional input space, these could be the *x* and *y* coordinates of each point. The *output space*, on the other hand, refers to the labels or values that we are trying to predict. In classification tasks, this could be categories like *spam* or *not spam*, while in regression tasks, it could be continuous values like house prices (Zhou, 2012).

The similarity between objects is typically measured using distance metrics such as *Euclidean*, *Manhattan*, or *Hamming* distance, depending on the nature of the input data (Putro Dirgantoro et al., 2021, Lubis et al., 2021, Li et al., 2021, Gui et al., 2021, Kalra et al., 2022, Prasetyo et al., 2023). The assumption of similarity implies that by examining the k nearest data points to a given query point (where k is a user-defined parameter), one can infer the output for the query point by aggregating the outputs of these neighbors. In classification, this might mean taking a majority vote among the neighbors, while in regression, it could involve calculating the average or median of the neighbors' values. This principle leverages the intuitive idea that similar inputs often lead to similar outputs in many real-world scenarios, making KNN a versatile and widely used algorithm in machine learning tasks (Zhou, 2012). The pseudo-code of the classical k-nearest neighbor algorithm is given in Algorithm 2.

Algorithm 2 The pseudocode of classical KNN – Adapted from Almomany et al. (2022)

Input: X: training data, Y: class labels of X, K: number of nearest neighbors
Output: Class of a test sample x.
procedure CLASSIFY(X, Y, x)
    for each sample x_k **in** X **do**
        Calculate the distance: $d(x, x_k) = \sqrt{\sum_{i=1}^{n}(x_{ki} - x_i)^2}$
    end for
    Classify x in the majority class:
    $C(x) = \arg\max_y \sum_{x_k \in \text{NN}(x)} I(y_k = y)$
end procedure

The classical KNN algorithm begins with a given set of training data X and their corresponding class labels Y, along with a specified number K of nearest neighbors to consider. The algorithm mainly aims to classify a new test sample x based on these inputs.

The process starts with the calculation of the Euclidean distance between the test sample x and each sample x_k in the training dataset X. This distance is computed using the formula $d(x, x_k) = \sqrt{\sum_{i=1}^{n}(x_{ki} - x_i)^2}$, where n is the number of features in the dataset.

After calculating the distances, the algorithm identifies the K nearest neighbors to the test sample x. A majority vote among these neighbors determines the classification of x. Specifically, the class $C(x)$ assigned to x is the one most frequent among the K nearest neighbors. This is formalized as $C(x) = \arg\max_y \sum_{x_k \in \text{NN}(x)} I(y_k = y)$, where $\text{NN}(x)$ denotes the set of nearest neighbors and I is an indicator function that is 1 if the condition $y_k = y$ is true, and 0 otherwise.

This method effectively assigns the class label based on the predominant class among the nearest neighbors, leveraging the assumption that similar samples reside in close proximity in

the feature space.

3.2.1.3 Linear Discriminant Analysis - LDA

A linear classifier operates through a mechanism that involves a weight vector $\mathbf{w}$ and a bias term b, which are fundamental in determining the classification of an instance $\mathbf{x}$ into a specific class label y. The process is mathematically represented by the equation $y = \text{sign}(\mathbf{w}^\mathsf{T}\mathbf{x} + b)$, where the sign function outputs the class label based on the sign of the argument (positive or negative) (Zhou, 2012).

The previous equation encapsulates the essence of linear classification, where the weight vector $\mathbf{w}$ and the bias b collectively define a decision boundary in the feature space. The first step in the classification process involves the transformation of the multidimensional instance space into a one-dimensional space, effectively projecting the instances onto a line. This is achieved through the dot product of the weight vector $\mathbf{w}$ and the instance vector $\mathbf{x}$, which geometrically can be interpreted as mapping the instances onto a new axis defined by the weight vector. The bias term b then shifts this line away from the origin, allowing for flexibility in the placement of the decision boundary (Zhou, 2012).

The second step involves identifying a specific point along this line that acts as a threshold to separate instances into positive or negative classes. Instances on one side of this threshold are classified into one class, while those on the opposite side are classified into the other. This two-step process simplifies the complex multidimensional classification problem into a more manageable one-dimensional decision-making problem, leveraging the geometric properties of linear algebra to classify instances (Zhou, 2012) efficiently.

The primary goal of LDA is to project the data points onto a line (in the simplest case) so that the separation between different classes is maximized while minimizing the variance within each class. This is achieved by maximizing a function that represents the ratio of the between-class variance to the within-class variance (Zhou, 2012).

The between-class variance quantifies how separated the different class means are when projected onto the line defined by $\mathbf{w}$, while the within-class variance measures how spread out the individual data points are around their respective class means (Zhou, 2012).

The optimal $\mathbf{w}$ maximizes this ratio, indicating a projection that best separates the classes. The bias term b adjusts the position of the decision boundary along the direction of $\mathbf{w}$. In the essence, the LDA algorithm seeks to find the projection that best discriminates between classes, making it a powerful tool for dimensionality reduction and classification in a linearly separable feature space (Zhou, 2012).

3.2.1.4 Logistic Regression - LR

Logistic regression is a statistical method predominantly used for binary classification problems, where the outcome variable can take one of two possible values, often coded as 0 and 1 (Bertsimas and King, 2017). The model operates on a dataset comprising a response vector $\mathbf{y}_{n\times 1}$, which contains the binary outcomes for n observations, and a model matrix $\mathbf{X} = [\mathbf{X}'_1, \ldots, \mathbf{X}'_n] \in \mathbb{R}^{n\times p}$, which includes p predictors (or features) for each observation. The regression coefficients,

denoted as $\boldsymbol{\beta} \in \mathbb{R}^{p\times 1}$, quantify the relationship between each predictor and the log odds of the outcome being one rather than 0. Specifically, the logistic regression model posits that the log odds of the event $y_i = 1$ given the predictors $\mathbf{x}_i$ is linearly related to the predictors through the equation:

$$\log\left(\frac{P(y_i = 1 \mid \mathbf{x}_i)}{P(y_i = 0 \mid \mathbf{x}_i)}\right) = \boldsymbol{\beta}'\mathbf{x}_i. \tag{3.1}$$

Logistic regression aims to find the set of coefficients $\boldsymbol{\beta}$ that best fits the data. This is achieved by minimizing the negative log-likelihood function, denoted as $f(\boldsymbol{\beta})$, which quantifies the difference between the observed outcomes and the outcomes predicted by the model. The function

$$f(\boldsymbol{\beta}) = \sum_{i=1}^{n} -y_i(\boldsymbol{\beta}'\mathbf{x}_i) + \log(1 + \exp(\boldsymbol{\beta}'\mathbf{x}_i))$$

aggregates the discrepancies across all observations, penalizing deviations from the observed data. Minimizing this function, as shown in equation 3.1

$$\min_{\boldsymbol{\beta}} f(\boldsymbol{\beta}),$$

allows for estimating the regression coefficients that make the observed data most probable under the logistic model. This optimization process is central to fitting a logistic regression model to a given dataset, enabling the prediction of binary outcomes based on a set of predictors (Bertsimas and King, 2017). The Algorithm 3 shows the pseudo-code for logistic regression.

Algorithm 3 Logistic Regression

Input: Data matrix $\mathbf{X} \in \mathbb{R}^{n\times p}$, Response vector $\mathbf{y}_{n\times 1}$
Output: Regression coefficients $\boldsymbol{\beta} \in \mathbb{R}^{p\times 1}$
Initialize $\boldsymbol{\beta}$ (e.g., to zeros or small random values)
repeat
    Calculate the predicted probabilities:
    $\hat{\mathbf{y}}_i = \frac{1}{1+\exp(-\mathbf{X}_i\boldsymbol{\beta})}$ for all i
    Compute the gradient of the negative log-likelihood:
    $\nabla f(\boldsymbol{\beta}) = \mathbf{X}^\top(\hat{\mathbf{y}} - \mathbf{y})$
    Update $\boldsymbol{\beta}$ using a gradient descent step:
    $\boldsymbol{\beta} \leftarrow \boldsymbol{\beta} - \alpha\nabla f(\boldsymbol{\beta})$
until convergence
return $\boldsymbol{\beta}$

The logistic regression algorithm begins with a dataset consisting of a model matrix $\mathbf{X} \in \mathbb{R}^{n\times p}$, representing p predictors for each of the n observations, and a response vector $\mathbf{y}_{n\times 1}$, which holds the binary outcomes for the observations. The algorithm's goal is to estimate the regression coefficients $\boldsymbol{\beta} \in \mathbb{R}^{p\times 1}$ that best fit the data by minimizing the negative log-likelihood function.

The coefficients $\boldsymbol{\beta}$ are initially set to zero or small random values, and the algorithm proceeds iteratively. In each iteration, the predicted probability of the outcome being 1 for each observation is computed using the logistic function. This function applies a linear transformation of the

predictors through $\boldsymbol{\beta}$ and passes it through a non-linear transformation. Next, the gradient of the negative log-likelihood function is computed based on the coefficients. This gradient provides the direction in which the function increases the most steeply, indicating how to update $\boldsymbol{\beta}$ to minimize the function.

Using gradient descent, the coefficients $\boldsymbol{\beta}$ are updated by moving in the opposite direction of the gradient by a step size determined by the learning rate α. This iterative process continues until convergence, typically defined as a point where the changes in $\boldsymbol{\beta}$ between iterations fall below a predetermined threshold, indicating that the coefficients have stabilized. The final result is a set of regression coefficients that make the observed data most probable under the logistic model (Bertsimas and King, 2017).

3.2.1.5 Naïve Bayes - NB

The Bayes' Theorem, expressed as

$$P(y \mid x) = \frac{P(x \mid y)P(y)}{P(x)},$$

allows us to compute the posterior probability $P(y \mid x)$, which is the probability of a class y given an observation x. In the context of classifier design, this posterior probability helps decide the class of x based on the observed data. The term $P(y)$ represents the prior probability of class y, which can be estimated by the frequency of y in the training set. The term $P(x \mid y)$ is the likelihood of observing x in class y, and $P(x)$ is the evidence, a normalizing factor ensuring probabilities sum up to one (Zhou, 2012).

In practice, $P(x)$ is often disregarded when comparing different classes for the same x, as it remains constant across these comparisons. The focus then shifts to accurately estimating $P(x \mid y)$, the likelihood, which is crucial for achieving the theoretical best classifier, known as the Bayes optimal classifier. This classifier achieves the Bayes error rate, which is the theoretically lowest possible given the data. However, estimating $P(x \mid y)$ poses significant challenges due to the need to compute an exponentially large number of joint probabilities among features, necessitating simplifying assumptions to make the problem tractable. These assumptions aim to reduce the complexity of the model while striving to maintain its predictive accuracy (Zhou, 2012).

The Naïve Bayes classifier is a probabilistic model that simplifies the computation of probabilities for classification tasks by assuming that all features (or attributes) are independent, given the class label. This assumption, known as conditional independence, allows the model to calculate the probability of observing a set of features given a class label, $P(x \mid y)$, as the product of the individual probabilities of each feature given the class, $P(x_i \mid y)$, for $i = 1, \ldots, n$, where n is the number of features. This approach significantly reduces the complexity of the model by avoiding the calculation of joint probabilities of the feature set, which would be computationally expensive and require a large amount of data to estimate accurately (Zhou, 2012).

During the training phase, the Naïve Bayes classifier estimates the prior probability of each class, $P(y)$, and the conditional probability of each feature given a class, $P(x_i \mid y)$, using the

frequency of occurrences in the training data. In the prediction phase, for a given test instance x, the classifier calculates the posterior probability of each class, $P(y \mid x)$, using Bayes' theorem. The class with the highest posterior probability, which is proportional to the product of the prior probability of the class and the product of the conditional probabilities of the observed features given the class, is selected as the predicted class. This method allows for efficient and effective classification, even with many features (Zhou, 2012).

3.2.1.6 Support Vector Machine - SVM

Conceptualized initially for binary classification problems, SVMs operate on the principle of large-margin classification. This method distinguishes between two classes by identifying a hyperplane in the feature space that maximizes the margin between the classes. In this context, the margin is the smallest distance between the hyperplane and the nearest instances (support vectors) from each class (Zhou, 2012).

The objective of an SVM is to construct a hyperplane in a high-dimensional space (which could be infinite-dimensional, depending on the kernel used) that separates instances of different classes with the maximum possible margin. This approach is grounded in structural risk minimization, aiming to enhance the generalization ability of the classifier (Zhou, 2012).

By maximizing the margin, SVMs inherently implement a form of regularization, which helps reduce the risk of overfitting the training data. This characteristic makes SVMs particularly robust and effective for a wide range of classification tasks, including those where the dimensionality of the data is high relative to the number of training samples (Zhou, 2012).

3.2.2 Ensemble Prediction Models

Model ensembling is a machine learning approach where the core strategy diverges from the development of traditional single prediction models. Unlike modeling approaches that develop a single model from the training data to solve a specific problem, ensemble approaches deploy multiple models (Zhou, 2012). Each model is trained in the same dataset, potentially using different perspectives or subsets of data. The main hypothesis is that an ensemble M of prediction models can outperform any single model $m_1 \in M$. The better performance of ensemble models is due to the diverse aspects of the data captured by different learners, reducing the likelihood of different models within the ensemble making the same mistakes simultaneously (Zhou, 2012, Kozlov and Myasnikov, 2023). As a result, when one model makes an error in prediction due to possible limitations in its training, for example, another model in the ensemble might not share the same weaknesses and thus might not make the same error. When the outputs of the individual models in an ensemble are combined, the errors made by single models can be canceled out by the correct prediction of others (Zhou, 2012, Kozlov and Myasnikov, 2023).

Combining models in an ensemble can be done in various ways, such as by voting, averaging, or more complex methods like stacking, where the outputs of individual models serve as input to a final model responsible for making the final prediction. This collective approach to problem-solving in machine learning is also called committee-based learning or learning multiple classifier systems, highlighting the collaborative effort of various models working together as a committee

to achieve a common goal (Zhou, 2012).

Figure 3.2 illustrates a typical ensemble architecture. At the core of this architecture are the base learners, individual learning models trained to solve the same problem. These base learners are not chosen arbitrarily. They are generated through a systematic process that involves applying a base learning algorithm to the training data. The choice of the base learning algorithm is pivotal and can vary widely depending on the specific requirements of the task at hand. Common examples of base learning algorithms include decision trees, known for their interpretability and ease of use, and neural networks, favored for their ability to model complex, non-linear relationships in large datasets (Zhou, 2012). Other learning algorithms may also be employed, reflecting the diversity and adaptability of ensemble methods (Zhou, 2012). The essence of this approach lies in leveraging the unique strengths of each base learner, which, when combined, contribute to a more accurate and robust prediction model. This methodology underscores the principle that a collective of models offering different perspectives on the data can achieve superior performance compared to any single model operating in isolation.

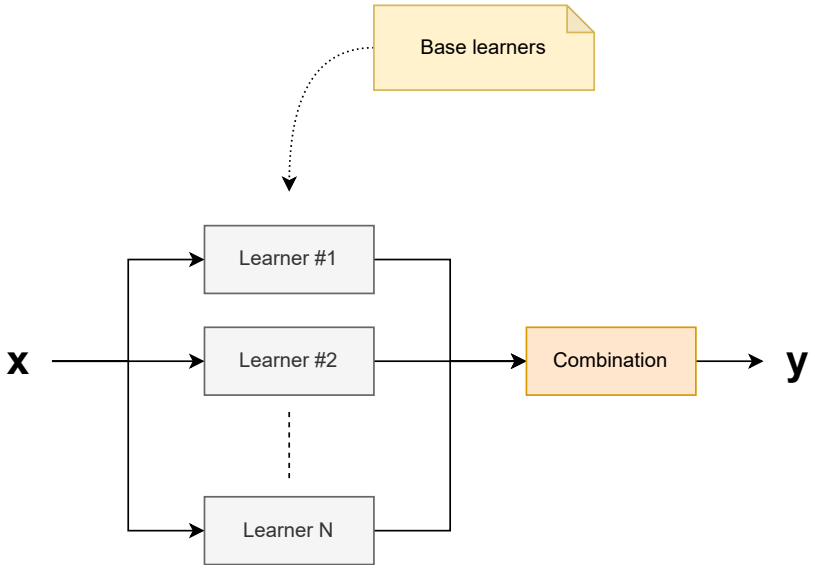

Figure 3.2: A common ensemble architecture

In the domain of software defect prediction (SDP), most strategies often rely on the use of a single classification algorithm to construct the predictive model (Marian et al., 2016, Goyal, 2022, Hernández-Molinos et al., 2023, Bhutamapuram, 2023, Suryawanshi and Kadam, 2024). These algorithms, such as the naïve Bayes classifier, decision trees, or logistic regression, operate on datasets that have already been labeled with instances of defects or non-defects. The naïve Bayes classifier is a simple probabilistic classifier based on applying Bayes' theorem with strong (naïve) independence assumptions between the features (Sayed, 2022). Decision trees, on the other hand, model decisions and their possible consequences as a tree-like structure, enabling straightforward classification based on feature values (Chopra and Khurana, 2023). Logistic regression is a statistical model used for binary classification that estimates the probabilities of binary outcomes based on one or more predictor variables (Kaur and Himanshu, 2023). It uses a logistic function to model the probability that a given input belongs to the default class,

often coded as '1'. This function outputs values between 0 and 1, which are interpreted as the likelihood of the input in the class labeled as '1' (Kaur and Himanshu, 2023).

Despite these single models' utility and widespread application for bug prediction, they have limitations. Each classifier has inherent biases and assumptions about the data, which can lead to weaknesses in predicting certain defects under specific circumstances (Zhou, 2012). For instance, the naïve Bayes classifier assumes feature independence, which may not hold in complex software systems (Sayed, 2022). Decision trees might suffer from overfitting, especially with very complex trees, and logistic regression can struggle with non-linear decision boundaries due to its linear nature (Chopra and Khurana, 2023). These limitations highlight the challenges in relying solely on a single classifier for defect prediction, as it may not adequately address the multifaceted nature of software defects across different contexts and datasets.

A significant advancement in SDP has been using several models to create ensemble or hybrid approaches. This strategy is predicated on the previously mentioned principle that combining the strengths of multiple classification techniques can substantially enhance the robustness and accuracy of defect prediction models (Zhou, 2012). Traditional single model approaches, while effective to a degree, are often hindered by inherent limitations such as bias towards particular data types or susceptibility to overfitting (Zhou, 2012). These limitations can lead to suboptimal prediction performance, especially in software defects' complex and varied landscape.

Ensemble approaches seek to mitigate these issues by leveraging the diversity among different classifiers. The ensemble method aims to achieve a more balanced and generalized prediction capability by aggregating the predictions from multiple models. This is because the errors from one classifier are likely compensated by the correct predictions from another, thus reducing the overall prediction error (Zhou, 2012, Ke et al., 2017).

The rationale behind ensembling models is rooted in the theory of the wisdom of crowds, where collective decisions tend to be more accurate than those made by any single member of a group (Susnjak et al., 2015). Consequently, the ensemble approach represents a significant advancement in the SDP domain, offering a more reliable and effective means of predicting software defects by overcoming the constraints associated with single classification techniques.

Several researchers have presented empirical evidence in the past years, indicating that ensemble bug prediction models offer superior classification accuracy compared to single models (Goyal, 2020, Goyal and Bhatia, 2022, Yang et al., 2022, J. Harikiran, 2023). Ensemble models can be categorized into two distinct groups based on the nature of the base learners involved: *homogeneous* and *heterogeneous* ensemble models (Zhou, 2012).

Homogeneous ensemble methods enhance the model's generalization ability by training identical base learners on diverse data segments, reducing the model's variance without significantly increasing its bias. Examples of such methods include bagging, boosting, and rotation forest, each with its unique operation mechanism. Heterogeneous ensembles diversify the base learners by using different algorithms for each. This diversity is crucial because it allows the ensemble to capture a broader range of patterns and relationships within the data, which might be missed if only a single type of algorithm were used. Once these diverse base learners are trained, their

predictions are aggregated to form a final decision. This aggregation can be done in several ways, but statistical integration and voting are two common strategies. Statistical integration involves combining the predictions based on statistical measures, such as averaging probabilities or using more complex models to weigh each learner's contribution. Conversely, voting is a simpler method where each base learner votes for a class, and the class with the majority votes is chosen as the final prediction (Zhou, 2012).

Heterogeneous ensemble models are particularly effective in SDP, as they mitigate the weaknesses of individual learners by balancing their biases and variances, leading to more robust and accurate defect prediction models (Zhou, 2012).

Ensemble prediction models can also be broadly classified into two categories: *linear* and *non-linear*, based on how they integrate the outputs from base learners. Linear ensemble methods employ a straightforward approach where the final output is derived from a linear combination of the predictions made by individual base learners. This combination can be a simple average, where each base learner's prediction is given equal importance, or a weighted average, where predictions are combined based on predetermined weights that reflect the reliability or performance of each base learner. The essence of this approach lies in its simplicity and interpretability, as it directly aggregates the predictions in a linear fashion (Zhou, 2012).

On the other hand, non-linear ensembles adopt a more complex strategy by using non-linear techniques to aggregate the predictions from base learners. Examples of such techniques include decision trees and support vector machines (SVMs). Unlike linear methods, non-linear ensembles can capture more complex patterns and interactions among the predictions of base learners. This is particularly useful in scenarios where the relationship between the input features and the target variable is not linear (Zhou, 2012). In the following sections, we briefly introduce each ensemble model we used in the experiments conducted for this dissertation.

3.2.2.1 Bagging

The term *Bagging* is derived from "**B**ootstrap **AGG**regat**ING**", a concept introduced by Breiman (1996). This methodological approach in ensemble learning combines two fundamental statistical techniques: bootstrap and aggregation. Bootstrap, a resampling method, involves randomly selecting a subset of data from the original dataset with replacement, allowing the creation of multiple different training datasets (Chatterjee and Ghosh, 2022). This process is crucial for generating diverse models from a single training process, addressing the variance component of the model's error.

Aggregation, conversely, refers to combining the predictions from multiple models to form a final prediction. This is typically done by averaging the predictions in regression tasks or by majority voting in classification tasks. The aggregation step aims to reduce the variance without increasing the bias, leveraging the diversity among the models generated through bootstrapping (Pinto et al., 2019).

By integrating bootstrap and aggregation, Bagging effectively enhances the stability and accuracy of machine learning algorithms, particularly for those sensitive to the variations in the training dataset (Boukir and Feng, 2021, Samantaray and Das, 2023). This ensemble method is

especially beneficial in reducing overfitting, thereby improving the predictive performance of models on unseen data.

The bagging process is shown in Algorithm 4. It starts with an original dataset $\mathbf{D}$ consisting of m samples. A base learning algorithm $\mathcal{L}$ is selected, along with a specified number of base learners T. For each base learner t, a new training set $\mathbf{D}_{\text{bs}}$ is generated through bootstrap sampling from the original dataset $\mathbf{D}$. Bootstrap sampling means randomly selecting samples from the dataset with replacements, allowing the same sample to be picked more than once. Each base learner h_t is then trained on its respective bootstrap sample. The final model $H(x)$ is an aggregation of these base learners, typically using majority voting for classification tasks (Zhou, 2012).

Algorithm 4 The Bagging algorithm – Adapted from Zhou (2012)

Input: Data set $D = \{(x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m)\}$
Input: Base learning algorithm $\mathcal{L}$
Input: Number of base learners T

procedure BAGGING
    for $t = 1$ to T **do**
        Generate bootstrap sample D_{bs} from D
        $h_t = \mathcal{L}(D, D_{bs})$ ▷ D_{bs} is the bootstrap distribution
    end for
    $H(x) = \arg\max_{y \in \mathcal{Y}} \sum_{t=1}^{T} I(h_t(x) = y)$ ▷ Majority voting
end procedure

An interesting statistical property of bootstrap sampling, as highlighted by Breiman (1996), is its relation to the *Poisson* distribution with $\lambda = 1$. This implies that for any given training sample in the original dataset, the probability of it being selected at least once in a bootstrap sample is approximately $1 - (1/e) \approx 0.632$, where e is the base of the natural logarithm. Consequently, on average, about 36.8% of the original training examples are not used in training a particular base learner. These unused samples, known as out-of-bag (OOB) samples, provide a unique opportunity to estimate the base learner's performance without the need for a separate validation set. This OOB error estimation is a valuable feature of Bagging, as it allows for an unbiased evaluation of the ensemble's generalization error without additional computational costs associated with cross-validation.

3.2.2.2 Gradient Boosting Decision Tree

Gradient Boosting Decision Tree (GBDT) is a sophisticated ensemble learning technique that integrates multiple decision trees to form a more robust model (Ke et al., 2017). Unlike models that train components in parallel, GBDT adopts a sequential approach where each decision tree is trained one after the other.

The core principle behind GBDT's training methodology involves fitting negative *gradients*, which essentially refers to minimizing residual errors during each iteration of the training process.

In the context of GBDT, these negative gradients are synonymous with the residual errors or the differences between the predicted and actual values from the previous iteration. This method

is a form of *gradient descent* in the function space, aiming to find the model that reduces the loss function as much as possible (Bishop and Bishop, 2024).

By iteratively fitting the negative gradients, GBDT effectively adjusts the model to address the shortcomings of the trees that were trained in previous iterations (Ke et al., 2017). This iterative correction of errors enables GBDT to progressively improve its predictions with each additional tree, leading to a highly accurate ensemble model.

In the context of Gradient Boosting Decision Tree (GBDT) algorithms, the primary computational cost is attributed to constructing decision trees, a fundamental component of the model (Ke et al., 2017). This process involves iteratively creating trees that minimize residual errors from previous iterations, thereby enhancing the model's predictive accuracy (Shailza Kanwar and Shrivastava, 2023). Within this iterative tree construction phase, the most time-intensive task is identifying the optimal split points for each node within a tree.

Split points are thresholds that partition the data into subsets based on feature values to maximize the homogeneity of the resulting subsets concerning the target variable (Zhu et al., 2013). This selection process is critical as it directly influences the quality of the decision trees and, by extension, the overall effectiveness of the GBDT model (Ke et al., 2017).

The determination of split points necessitates a comprehensive evaluation of all possible thresholds across all features, involving calculations of metrics such as information gain or *Gini impurity* for each potential split (Chickering et al., 2001, Ke et al., 2017). The *Gini impurity* is a metric used in decision tree algorithms to quantify the level of impurity or disorder in a dataset (Zhang and Gionis, 2023). This exhaustive search is computationally demanding, especially in scenarios involving large datasets with high dimensionality of features, as it requires scanning through all data instances to assess the efficacy of each possible split. Consequently, this step constitutes the major part of the computational efforts in the learning process of GBDT (Ke et al., 2017).

The histogram-based algorithm implements GBDT, aggregating continuous feature values into discrete bins (Algorithm 5). This binning process transforms continuous features into a finite number of intervals or "bins," effectively converting them into categorical variables. During the training phase, these bins are used to construct feature histograms, which are graphical representations of the distribution of data points across the binned feature values (Alsabti et al., 1998, Jin and Agrawal, 2003, Li et al., 2007, Ke et al., 2017).

According to Ke et al. (2017), this method significantly reduces the computational complexity of finding split points. Instead of evaluating every possible split across all feature values, the algorithm only needs to consider the bins themselves, which are substantially fewer in number. This reduction in computational tasks accelerates the training process without significantly compromising the model's accuracy.

The histogram-based approach offers a more efficient alternative to traditional split point identification methods in GBDT, making it particularly advantageous for handling datasets with large feature dimensions. By simplifying the feature space into discrete bins, this algorithm enhances the scalability and efficiency of gradient boosting models, facilitating faster training

Algorithm 5 GBDT Histogram-based Algorithm – Adapted from Ke et al. (2017)

```
Input: I: training data, d: max depth
Input: m: feature dimension
nodeSet ← {0}                                   ▷ tree nodes in current level
rowSet ← {{0, 1, 2, ...}}                       ▷ data indices in tree nodes
for i = 1 to d do
    for each node in nodeSet do
        usedRows ← rowSet[node]
        for k = 1 to m do
            H ← new Histogram()
            Build histogram
            for j in usedRows do
                bin ← I.f[k][j].bin
                H[bin].y ← H[bin].y + I.y[j]
                H[bin].n ← H[bin].n + 1
            end for
            Find the best split on histogram H
        end for
    end for
    Update rowSet and nodeSet according to the best split points.
end for
```

times and potentially broader applicability in various machine learning tasks (Li et al., 2007, Ke et al., 2017).

3.2.2.3 Random Forests

Algorithm 6 shows the pseudo-code of an implementation for random forests. Breiman (2001) proposed Random forests, which incorporate an additional layer of randomness beyond what is found in bagging techniques (Section 3.2.2.1). Unlike traditional bagging, where each tree in the ensemble is independently constructed from a bootstrap sample of the dataset without altering the tree construction process, random forests modify the data sampling and the decision-making process within the trees.

In conventional decision trees, the decision to split a node is based on identifying the best possible split across all available variables to optimize a specific criterion, such as information gain or *Gini impurity*, as mentioned earlier. However, random forests diverge from this approach by limiting the choice of variables at each split to a randomly selected subset of the predictors. This means that for each node, instead of considering all variables for the best split, only a subset is considered, and the best split within this subset is used (Liaw and Wiener, 2007).

3.2.2.4 Stacking Classifier

Stacking is an ensemble learning technique designed to enhance predictive performance by combining multiple models (Zhou, 2012). This method operates on a two-tiered structure. In the first tier, various models referred to as the first-level learners, are independently trained on the same dataset. These models can range from simple linear regressors to complex neural networks, each bringing its unique perspective or hypothesis about the data's underlying pattern (Gocheva-

Algorithm 6 Random Forests Algorithm – Adapted from Liaw and Wiener (2007)

Input: Original dataset D, number of trees n, number of variables to try at each split m_{try}, total number of predictors p
Output: Random forest model, OOB estimate of error rate
For classification and regression:
for $i = 1$ to n **do**
    Draw a bootstrap sample D_i from D
    Grow an unpruned tree T_i on D_i with the following modification:
        At each node:
            Randomly sample m_{try} predictors
            Choose the best split from these m_{try} variables
end for
Predict new data by aggregating the predictions of the n trees:
    For classification, use majority voting
    For regression, use average
Estimate error rate (OOB estimate):
for $i = 1$ to n **do**
    Predict the out-of-bag (OOB) data not in D_i using tree T_i
end for
Aggregate the OOB predictions for each data point
Calculate the error rate from these aggregated predictions
Return the OOB estimate of the error rate

Ilieva et al., 2022). The diversity among these first-level learners is crucial as it introduces different biases and variances into the ensemble, thereby enriching its overall predictive capacity (Zhou, 2012).

The main part of stacking lies in its second tier, where another model, known as the meta-learner or second-level learner, is trained. However, unlike traditional ensemble methods where simple rules like voting or averaging are used to combine predictions, the meta-learner in stacking is trained on a new dataset generated from the first-level learners' predictions (Zhou, 2012). This dataset comprises the outputs of the first-level models as features and the actual target values. The meta-learner's task is to discern the optimal way to combine these predictions to minimize the final prediction error. This approach allows stacking to effectively capture the strengths of each base model while mitigating their weaknesses, leading to a more accurate and robust predictive model than any of the individual learners could achieve on their own (Zhou, 2012).

The algorithmic representation of a generic stacking procedure is outlined in Algorithm 7. The first-level classifiers of stacking are trained on the original dataset, and their predictions form a new dataset for training a second-level learner, often called a meta-learner (Zian et al., 2021). A critical challenge in this process is avoiding overfitting, a condition where the model learns the training data too well, including its noise and outliers, leading to poor generalization on unseen data.

Overfitting is particularly risky in stacking when the same data used for training the first-level classifiers is also employed to create a new dataset for the second-level learner. This is because the second-level learner can inadvertently learn the specific predictions of the first-level

Algorithm 7 Stacking-based Ensemble Learning – Adapted from Ke et al. (2017)

Input: Data set $D = \{(x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m)\}$
Input: First-level learning algorithms $\mathcal{L}_1, \ldots, \mathcal{L}_T$
Input: Second-level learning algorithm $\mathcal{L}$
Process:
for $t = 1$ to T **do**
    Train a first-level learner by applying $\mathcal{L}_t$ on D
end for
Initialize $D' = \emptyset$
for $i = 1$ to m **do**
    for $t = 1$ to T **do**
        Predict $z_{it} = h_t(x_i)$ using learner h_t
    end for
    $D' = D' \cup \{((z_{i1}, \ldots, z_{iT}), y_i)\}$
end for
Train the second-level learner h' by applying $\mathcal{L}$ to D'
Output: $H(x) = h'(h_1(x), \ldots, h_T(x))$

classifiers, including their errors, rather than generalizing from the underlying data distribution (Ke et al., 2017).

It is recommended that the data used for training the first-level classifiers be separated from that used to generate the new dataset for the second-level learner to mitigate this risk. This can be achieved through cross-validation or leave-one-out procedures (Ke et al., 2017).

In cross-validation, the original dataset is partitioned into smaller sets, where some sets are used for training the first-level classifiers and others for validating their predictions, forming the new dataset for the second-level learner (Seraj et al., 2023).

The leave-one-out method is an extreme form of cross-validation where each instance, in turn, is left out of the training set and used as a single-instance validation set. These approaches ensure that the second-level learner is trained on data that it has not seen before, thereby reducing the likelihood of overfitting and enhancing the model's ability to generalize to new, unseen data (Ke et al., 2017).

3.2.2.5 Voting Classifier

Voting, within the context of ensemble learning methods, stands as a technique for aggregating the predictions of multiple models, mainly when dealing with nominal (categorical) outputs (Dogan and Birant, 2019). This method operates under the principle of collective decision-making, where each model in the ensemble, often referred to as a "learner," casts a "vote" for a particular class label. The final output class is then determined based on the majority or plurality of votes received from all the learners. This approach is fundamentally democratic, leveraging the diversity within the ensemble to arrive at a more robust decision than any individual learner could achieve on its own (Ke et al., 2017).

The popularity of voting as a combination method stems from its simplicity, interpretability, and effectiveness in enhancing the predictive performance of the ensemble (Muylle et al., 2022). It effectively mitigates the risk of overfitting by averaging biases and errors across the ensemble,

thereby improving the model's generalization ability (Merz, 1999, Ke et al., 2017, Muylle et al., 2022).

Voting can be implemented in various forms, including simple majority voting, where the class with the most votes wins, and weighted voting, where votes are weighted according to the confidence or performance of each learner, allowing for a more nuanced aggregation of predictions (Dogan and Birant, 2019). This method is particularly advantageous in scenarios where the decision boundary is not linearly separable, and the collective intelligence of the ensemble can navigate the complexity of the decision space more effectively than single models (Wardoyo et al., 2020).

The pseudo-code in Algorithm 8 assumes a simplified version of the voting process where classifiers are combined to decide based on majority voting. It reflects a general implementation similar to what one might find in the scikit-learn `VotingClassifier`[1] implementation.

The Voting Classifier algorithm starts by initializing with a list of classifiers and a specified type of voting, either *hard* or *soft*. The output from the classifier will be the decision made based on the combined predictions of the classifiers included.

The `Fit` function takes a training dataset, denoted as X (features) and y (labels), and trains each classifier in the list on this data. Each classifier independently learns from the same training dataset.

The `Predict` function initializes an empty list to store votes from each classifier. Each classifier then makes a prediction based on the input data X. For *hard* voting, the algorithm takes the *mode* of these predictions, i.e., the most frequently predicted label across all classifiers. For *soft* voting, it calculates the average probability assigned to each class by the classifiers. It selects the label with the highest average probability, thus making a decision based on a consensus of probabilities rather than labels.

3.3 Final Considerations

This chapter has provided a detailed exploration of the foundational concepts and relevant research in the fields of Search-Based Software Testing (SBST) and software defect prediction. By examining both single and ensemble prediction models, we have highlighted the critical role these models play in enhancing the effectiveness of automated test case generation. The review of related work has underscored the limitations of existing approaches, particularly in terms of their bug detection capabilities and the challenges posed by diverse software projects. These insights have reinforced the need for innovative strategies that leverage the strengths of ensemble prediction models to guide SBST. The knowledge gained from this background sets the stage for the development of the Ensemble Predictive Budget Allocation Strategy for Testing (EP-BAST) and the Ensemble Predictive Multi-Objective Sorting Algorithm (EPMOSA), which will be detailed in the subsequent chapters. This chapter has thus established a solid foundation for understanding the technical and theoretical underpinnings of the proposed research, paving the way for the empirical investigations to follow.

[1] https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.VotingClassifier.html

Algorithm 8 Voting Classifier

Input: $Classifiers = \{C_1, C_2, \ldots, C_n\}$, $VotingType$
Output: Combined classifier decision
function FIT(X, y)
    for each classifier C_i in $Classifiers$ **do**
        Train C_i on training data (X, y)
    end for
end function
function PREDICT(X)
    Initialize $votes$ to an empty list
    for each classifier C_i in $Classifiers$ **do**
        $prediction_i \leftarrow C_i$.predict($X$)
        Append $prediction_i$ to $votes$
    end for
    if $VotingType$ is "hard" **then**
        $final\_prediction \leftarrow$ mode of $votes$
    else ▷ soft voting
        Compute average probability across classifiers for X
        $final\_prediction \leftarrow$ label with highest average probability
    end if
    return $final\_prediction$
end function

CHAPTER

4

Methodology

In this chapter, we present the methodology adopted to assess the effectiveness and efficiency of EPMOSA. Our investigation is based on quantitative research (Di Penta and Tamburri, 2017), using an experimental strategy to evaluate the performance of EPMOSA relative to DynaMOSA, an established benchmark. In addition, we define standardized procedures for executing experiments and analyze their results based on statistical analysis, which informs our conclusions.

For this dissertation, we defined an experimental framework to support our research, facilitating result analysis. This experimental framework enables the manipulation of *independent variables*, such as approaches for test case generation and their parameters. Thus, it is possible to observe their impacts on *dependent variables*, mainly the *efficacy* and *efficiency* of test suites in identifying bugs. Also, our experimental framework provides a way to reduce bias, ensuring the integrity of our findings.

The following sections describe our experimental framework (Section 4.2), the rationale for selecting subject systems (Section 4.2.1), the data collection methods (Section 4.2.2), the benchmark and comparison criteria (Section 4.2.3), and we also give an introduction to the tools and algorithms used and developed to automatically generate test cases (Section 4.2.4). Also, we further elaborate on the data analysis strategies we used (Section 4.2.5), which helped ensure our results' validity and reliability.

This chapter describes the foundations of our methodology, which guides our research and serves as a template for future replication and extension efforts. Our study aspires to significantly contribute to software testing, particularly highlighting EPMOSA's contributions to the domain of SBST.

4.1 Research Design and Strategy

Our research centers on a comparative analysis of the proposed approach – EPMOSA – and DynaMOSA, a state-of-the-art approach for automatic test generation. The primary goal is to evaluate and compare these approaches based on their *efficiency* and *effectiveness* in generating test suites that are capable of detecting bugs within software systems.

As discussed in Section **??**, PreMOSA is the approach most closely related to EPMOSA, with both being pioneers in integrating bug defect prediction to guide SBST techniques. However, comparing our results directly with those from PreMOSA is not deemed justifiable. This is primarily because the authors of PreMOSA used a **theoretical** bug prediction model, and like EPMOSA, PreMOSA is still in its early stages of development. Consequently, we have chosen DynaMOSA as our benchmark and point of comparison.

To facilitate a comprehensive comparison between single and ensemble models in guiding SBST approaches, we have implemented several variations of DynaMOSA. These variations are specifically designed to follow the guidance of single prediction models during the time budget allocation steps. This allows us to assess the effectiveness of different predictive models in optimizing the allocation of testing resources within the SBST framework.

4.1.0.0.1 Independent Variables

Independent variables, by definition, are the factors manipulated or selected by the researcher to observe their impact on the *dependent variables*, thereby revealing possible relationships within the experimental setting (Leatham, 2012). Understanding the nature and scope of the independent variables is important, as they influence the validity and reliability of our experiment's outcomes. We defined two independent variables for our study:

1. **Automatic Test Case Generation Approach:** We can define the test case generation approach as an independent variable because we change it and observe its results. We consider two approaches: (1) DynaMOSA, a state-of-the-art approach, and (2) EPMOSA, the proposed approach. We aim to study the impact of these approaches on test cases by changing their workings. The selection of DynaMOSA as a benchmark was based on its prevalence in the literature, underlying methodology (e.g., search-based approaches), and availability for experimental use.

2. **Test Case Generation Parameters:** Once we have established which automatic test case generation approaches we aim to study, we can manipulate how they work, influencing the test case generation process. The different configurations of each approach were based on preliminary code analysis and, for DynaMOSA, studying the available documentation. Thus, we can explore the impact of each configuration on the effectiveness and efficiency of generated test cases. Our experiments focus on different configurations regarding the *time budget* allocated for the search step in each approach and the prediction model used to guide each one.

According to our investigation, introducing the Dynamic Many-Objective Sorting Algorithm (DynaMOSA) marked a significant advancement in automated test case generation, addressing

its intrinsic multi-objective nature with unprecedented efficiency and effectiveness. The seminal paper by Panichella et al. (2018) was published in 2018 and, up until the date of this writing, was cited by more than 200 researchers, achieving 3350 views on IEEEXplore[1]. Also, an open-source implementation of DynaMOSA is available in Evosuite. More details on our choice of benchmark can be found in Section 4.2.3.

4.1.0.0.2 Dependent Variables The dependent variable represents the consequences or outcomes when the independent variable changes (Leatham, 2012). It plays an essential role in measuring and evaluating hypotheses. This section aims to define the dependent variables: the *efficacy* and *efficiency* of the test cases generated in our experiments.

- **Effectiveness:** In our experiments, we quantify *effectiveness* by the number of bugs the generated test suite can identify. We use the Defects4J open-source dataset (Just et al., 2014) containing known bugs from several projects to ensure a consistent and objective analysis of the results from each test suite execution.

- **Efficiency:** We quantify efficiency by the time spent by each approach to generate a test suite. Specifically, the start time (`start_time_gen`) was recorded just before the beginning of the generation using the command `start_time_gen=$(date +%s)`, which captures the current time in seconds since the Unix epoch[2] (00:00:00 UTC on 1 January 1970). Similarly, immediately after the generation concluded, the end time (`end_time_gen`) was recorded using `end_time_gen=$(date +%s)`. The total execution time (`execution_time_gen`) was then computed by subtracting the start time from the end time, using the expression `execution_time_gen=$((end_time_gen - start_time_gen))`. This calculation yields the duration in seconds, providing a straightforward time metric.

Efficiency reflects the speed with which the different approaches can generate test suites and has implications for resource allocation in software projects. By reducing the time required to generate test suites, we can optimize the use of resources in software development. An efficient test suite generation approach enables developers to move quickly through the testing stages, allowing faster identification and resolution of bugs.

As mentioned previously, the configurations of each test case generation approach serve as independent variables. We mainly focus on the *time budget* allocated to test case generation and the bug prediction model used. These are crucial as they influence the search depth in DynaMOSA and EPMOSA algorithms and significantly impact both the effectiveness and efficiency of the test case generation – essentially determining the balance between the thoroughness of testing and time constraints.

By employing different bug prediction models, we vary the time budgets of each class being tested. By varying the time budget, we can explore its effect on the ability of the generated test

[1] https://ieeexplore.ieee.org/document/7840029
[2] https://en.wikipedia.org/wiki/Unix_time

suites to uncover bugs. Thus, we understand how the strategy to define testing time constraints affects the quality and utility of the automatically generated test cases.

Our analysis is based on the assumption that using ensemble bug prediction models to target more bug-prone areas can lead to a more strategic allocation of resources in automated testing environments compared to single bug prediction models. This approach optimizes the time budget by focusing testing efforts where they are most needed, potentially enhancing the efficiency and effectiveness of the testing process. More details about how we propose to optimize the time budget allocation for SBST can be found in Chapter 5.

4.2 Experimental Framework

We created an experimental framework to assess EPMOSA under different configurations. This framework outlines the theoretical basis and guiding principles that shape the design and execution of our experiments. Establishing a well-defined experimental framework ensures that our study relies on a solid rationale and clearly defined scope and objectives. This section describes our procedures for conducting experiments and analyzing EPMOSA's performance.

Although we define clear objectives for our research (Section 1.3), those converge to a core objective: determining how well EPMOSA, guided by ensemble bug prediction models, can generate test suites capable of identifying bugs within real-world software projects. The framework outlines the procedures we followed to generate and evaluate test suites for Defects4j (Just et al., 2014) projects known to have bugs.

4.2.1 Selection of Experimental Subjects

We selected the subject systems for our experiments from Defects4j (Just et al., 2014), a known and extensively studied bug dataset within the software testing community. Defects4j provides a collection of real-world bugs from open-source projects, which facilitates the assessment of the proposed approach's bug detection capabilities. Using projects from Defects4j means that our experiments are rooted in practical, real-world scenarios, thereby improving the relevance and applicability of the research outcomes.

We chose version 2.1.0 of the Defects4J dataset, which encompasses a curated collection of 835 bugs (plus 29 deprecated bugs) from seventeen real-world open-source Java projects. Developers have verified and fixed these bugs, ensuring a reliable basis for developing and assessing novel software testing approaches.

We selected a subset of four projects from Defects4J: Commons-Lang, Commons-Codec, Commons-Compress, Joda-Time (Table4.1). Due to the characteristics of these projects, we considered them sufficient to provide a diverse and substantial dataset of real-world bugs for analysis. The number of studied projects is also sufficient to evaluate EPMOSA across multiple projects and types of defects, strengthening the generality and robustness of our results.

We employed a strategic approach to select the subject systems based on *representativeness* and *diversity*, ensuring that the chosen projects offer broad insights into the nature of software defects. This approach was critical, not as a limitation of our study's scope, but as a deliberate methodological choice to optimize the depth and quality of our analysis.

Project Name	Number of Active Bugs	Active Bug Ids	Deprecated Bug Ids
Commons-Lang	64	1, 3-65	1
Commons-Codec	18	1-18	None
Commons-Compress	47	1-47	None
Joda-Time	26	1-20, 22-27	21

Table 4.1: Subject Systems

The rationale behind not experimenting with all of Defects4J's projects lies in our objective to maintain a manageable and focused investigation that allows for an in-depth analysis of the results. By choosing a subset of projects that exemplifies a wide range of sizes, complexities, and application domains, we ensure that our findings are not skewed toward the idiosyncrasies of a specific project or a narrow set of bug types. Thus, we can draw generalized conclusions.

It is important to observe that the projects we discarded from our study may also offer valuable insights in their own right. However, our criteria prioritized those best meeting our research goals through their representativeness and diversity. Through the chosen subject systems, we aim to contribute an understanding of software bugs that go beyond the specifics of individual projects, thereby offering valuable contributions to the field of SBST.

For each project system, Defects4j provides two versions of each bug, a *buggy version* containing the bug and a *fixed version* where the bug is fixed. A patch highlighting the code changes that fixed the bug accompanies the versions provided. Thus, locating the bug within the codebase is easier. Additionally, Defects4J provides a testing framework that supports, among other operations, test case execution and automatic management of flaky tests, which improves the reliability of the generated test suites.

Defects4J is a widely recognized dataset in automated unit test generation, automated program repair, fault localization, and test case prioritization. Its extensive usage across various research studies makes it an ideal benchmark, allowing for direct comparison with existing work. The dataset has been actively maintained, with a significant addition of 401 bugs from 11 open-source projects to the pre-existing collection of 438 bugs from 6 open-source projects, enhancing its relevance and utility for current research endeavors.

In contrast, other datasets like Bugs.jar (Saha et al., 2018) and Bears benchmark (Madeiral et al., 2019) have not been extensively used in automated test generation studies despite containing many actual bugs from multiple open-source Java projects. This underutilization can be attributed to their lack of support for addressing flaky test cases and evaluating test suites for bug detection capabilities (Perera et al., 2023a). Defects4J, on the other hand, provides a robust test execution framework that facilitates these tasks, making it a more suitable choice for the research at hand. From the seventeen projects in Defects4j, we selected four: commons-lang, commons-codec, commons-compress, joda-time.

In the subsequent sections, we provide an overview of each subject system and the grounds for including them in our experiments. For each project, we discuss how it demonstrates the diversity of characteristics that are essential to our research. This includes factors such as

their size, complexity, and the particular types of bugs they contain, showing how they offer a comprehensive understanding of the different defects found in the Defects4J dataset.

4.2.1.1 Commons-Lang

The Apache Commons Lang project[3] is a part of the Apache Commons project, which consists of reusable Java components maintained by the Apache Software Foundation. Commons Lang provides a range of utility functions not found in the standard Java library. Its goal is to complement the standard Java JDK utility classes, primarily Java.lang[4], with additional features to reduce the amount of boilerplate and repetitive code developers must write. Table 4.2 shows the number of commits and the number of lines of code we considered in each buggy version of the Commons-Lang for our experiments. The bug ID *2b* is a deprecated bug. That is why it is marked in red and ignored in our investigations.

Using Apache Commons Lang in our experiments' bug prediction and test case generation phases provides some advantages. Its popularity establishes it as one of the most used Java libraries, leading to experimental results that can be more relevant to developers. Also, the project is ranked among the ten most popular Java artifacts in the MVNRepository [5], which is a widely used platform for finding, reporting, and discussing Maven artifacts and their dependencies [6].

Further, the library offers several functionalities, from text manipulation to advanced Java object reflection. This allows EPMOSA to be tested with different code structures. Experimenting with such a complex library used in production environments gives us results that can bring real-world software development insights, increasing the practical usefulness of the research in this dissertation. Also, as an open-source project, Apache Commons Lang provides complete access to its source code, issue tracker, and version history. This transparency helps build an experimental protocol that other researchers can reproduce and validate.

4.2.1.2 Commons-Codec

The Apache Commons Codec project is part of the Apache Commons project. It consists of Java libraries that implement utility functions and reusable components. The Commons Codec library implements encoding and decoding algorithms, offering consistent, highly usable, well-documented interfaces for several encoding and decoding schemes. Some key components of the Commons Codec library are (1) Base64 and Base32 Encoding/Decoding schemes that can convert binary data into ASCII characters, making them suitable for transmission over text-based protocols such as HTTP or SMTP. (2) Hex Encoding/Decoding schemes are used to convert binary data into its hexadecimal representation and vice versa. It is often used for debugging or transferring binary data in an easily read and understood format. (3) URL Encoding/Decoding scheme used to encode and decode data for safe transmission over the Internet by escaping characters that may interfere with URLs. (4) Phonetic Encoding uses algorithms like Soundex, Metaphone, and Double Metaphone to encode words into a phonetic form representing their

[3]https://commons.apache.org/proper/commons-lang/

[4]https://docs.oracle.com/javase/8/docs/api/java/lang/package-summary.html

[5]The MVNRepository site ranks its artifacts based on their usage and popularity within the Java and JVM communities.

[6]https://mvnrepository.com/popular

1b to 35b			36b to 65b		
ID	**Total Commits**	**Total Lines of Code**	**ID**	**Total Commits**	**Total Lines of Code**
1b	3573	247714	36b	2288	184816
2b	0	0	37b	2279	187132
3b	3556	245766	38b	2274	187044
4b	3540	245548	39b	2261	186378
5b	3477	243804	40b	2134	192478
6b	3465	242912	41b	2117	191452
7b	3458	242812	42b	1899	221372
8b	3396	233144	43b	1896	221344
9b	3385	233448	44b	1890	219914
10b	3384	233444	45b	1871	219616
11b	3322	229022	46b	1855	212764
12b	3321	228962	47b	1788	216400
13b	3281	228394	48b	1764	212726
14b	3270	223566	49b	1739	210854
15b	3257	221890	50b	1689	206168
16b	3152	222370	51b	1686	201832
17b	3014	209462	52b	1684	201822
18b	3011	209454	53b	1656	201292
19b	2973	209512	54b	1634	200154
20b	2971	209466	55b	1614	201082
21b	2684	205572	56b	1553	199742
22b	2641	205568	57b	1548	199800
23b	2634	207580	58b	1540	199512
24b	2623	207202	59b	1537	199240
25b	2610	207126	60b	1523	198180
26b	2570	203032	61b	1522	198164
27b	2533	202310	62b	1521	198120
28b	2467	196588	63b	1512	197080
29b	2463	196428	64b	1467	194776
30b	2420	192950	65b	1426	193140
31b	2410	192376			
32b	2367	188550			
33b	2365	188552			
34b	2362	188576			
35b	2328	187470			

Table 4.2: Size of each buggy version considered in the Commons-Lang experiments

Versions 1 to 9			Versions 10 to 18		
ID	**Total Commits**	**Total Lines of Code**	**ID**	**Total Commits**	**Total Lines of Code**
1b	418	32912	10b	741	49256
2b	466	35960	11b	943	76240
3b	562	41292	12b	981	78028
4b	596	42670	13b	1337	87130
5b	614	43250	14b	1348	86952
6b	624	44644	15b	1438	94580
7b	627	44714	16b	1448	94566
8b	632	44796	17b	1624	100972
9b	662	49528	18b	1629	101048

Table 4.3: Size of each buggy version considered in the Commons-Codec experiments

sound. These encoding/decoding schemes are useful for search applications and tasks requiring matching similar-sounding words despite different spellings. (5) Utility methods for computing message digests (hashes) using algorithms like MD5 and SHA-1. While not primarily a cryptography library, these utilities are often used to create checksums or fingerprints of data.

These components for Commons Codec provide a toolkit for handling various encoding and decoding needs, focusing on reliability, performance, and ease of use. Commons Codec is widely used in Java applications that require encoding and decoding capabilities, especially when dealing with text processing, data interchange, and storing or transmitting binary data in a text-readable format. The library's appeal lies in its simplicity and ability to handle many commonly used encoding and decoding schemes that are too complex to implement. It helps developers avoid the boilerplate and error-prone code associated with these encoding operations, promoting code reuse and maintaining consistency across projects.

Table 4.3 shows the number of commits and the number of lines of code for each buggy version of the Commons-Codec we considered in our experiments.

Choosing the Apache Commons Codec project as a subject for our experiments offers some benefits similar to those of the Commons Lang project but with unique aspects due to its focus on encoding and decoding. The library implements a variety of coding and decoding algorithms, each with its intricacies and edge cases. Encoding and decoding operations are often prone to bugs due to handling edge cases, character compatibility, and binary data manipulation. This complexity provides a rich testing ground for exploring the effectiveness of bug prediction models and automated test generation approaches. In addition, it is a well-known library used by many developers, taking the top spot in the MVNRepositories ranking in the Base64 Libraries category.

4.2.1.3 Commons-Compress

The Apache Commons Compress project is a library for the Java programming environment. It facilitates compression and archive formats-related tasks and offers a set of APIs that enable developers to read and write compressed files and archives easily. This library simplifies archive manipulation features in Java applications, allowing for easy management of compression

formats. It eliminates the requirement for developers to know complex compression algorithms or undertake hands-on, low-level file-handling coding tasks.

The Commons Compress library supports several archive formats. This support enables Java applications to create, extract, and manipulate archives. The available formats include the widely recognized ZIP format and TAR, commonly found on UNIX and Linux systems; archive format AR, the compression-oriented ARJ format; CPIO; the high-compression 7z format developed by 7-Zip; and JAR, which aggregates Java class files along with metadata and resources.

In addition to archive format handling, the Commons Compress library excels in data compression and decompression across a spectrum of algorithms. It supports GZIP, a compression format in Unix/Linux systems; BZIP2, known for its high compression efficiency; XZ and LZMA, both known for their impressive compression ratios; Snappy, which prioritizes speed in compression and decompression; Deflate, used in ZIP and GZIP formats; Zstandard (Zstd), a fast and high-ratio compression algorithm developed by Facebook; and LZ4, distinguished by its rapid compression and decompression speeds. A significant feature of the library is its streaming API for specific formats, designed to facilitate the processing of large files or streams without necessitating their complete storage in memory.

The library architecture focuses on ease of use, abstracting the complexity inherent to compression and archive operations. This design philosophy ensures the library remains accessible to developers, including those unfamiliar with compression algorithms.

Table 4.4 shows the number of commits and the number of lines of code for each buggy version of the Commons-Compress we considered in our experiments.

4.2.1.4 Joda-Time

The Joda-Time project is a Java library that provides an option for the Java date and time classes. Before the introduction of the Java.time package in Java 8 (also known as JSR-310, which Joda-Time influenced), the standard Java date and time classes (found in Java.util.Date and java.util.Calendar) were often criticized for being poorly designed, difficult to understand and use for many common date and time operations. Joda-Time addressed these issues by offering a more intuitive and comprehensive solution for handling dates and times.

Despite its popularity and the significant improvements it offered over the standard Java date and time API, the introduction of Java.time package in Java 8 has led to Joda-Time being recommended primarily for legacy projects or when the Java.time is unavailable. The Java.time package incorporates the lessons learned from Joda-Time, providing a comprehensive date-time API as part of the standard Java library. It is generally recommended that new projects use Java.time package. However, Joda-Time remains an integral part of the Java ecosystem's history, and maybe it is still used in legacy projects that have not migrated to Java 8 or newer. Joda-Time was widely used in Java applications before the introduction of Java.Time. Many applications and libraries still depend on Joda-Time because they target older Java versions or have not migrated to Java.time for compatibility reasons. Understanding common bugs in such libraries can help maintain and improve legacy systems.

Choosing Joda-Time as a subject in our investigations can be particularly insightful for

Versions 1 to 23			Versions 24 to 47		
ID	**Total Commits**	**Total Lines of Code**	**ID**	**Total Commits**	**Total Lines of Code**
1b	266	35270	24	1448	179224
2b	346	41272	25	1450	179304
3b	476	44786	26	1517	182104
4b	477	44814	27	1527	183178
5b	547	48696	28	1528	183022
6b	552	49106	29	1715	193286
7b	660	58146	30	1724	194710
8b	671	58522	31	1726	194724
9b	900	100022	32	1736	195554
10b	949	103376	33	1738	195610
11b	959	104892	34	1753	196364
12b	970	105042	35	1791	199450
13b	974	105264	36	1862	244432
14b	976	105328	37	1872	246206
15b	1061	109464	38	1873	245526
16b	1068	109920	39	1881	248368
17b	1078	110716	40	1914	248168
18b	1091	111084	41	2010	254376
19b	1215	124982	42	2071	268574
20b	1248	126108	43	2252	301314
21b	1387	140614	44	2276	303770
22b	1442	147734	45	2281	303800
23b	1443	179058	46	2299	307022
-	-	-	47	2463	311828

Table 4.4: Size of each buggy version considered in the Commons-Compress experiments

Versions 1 to 13			Versions 14 to 27		
ID	**Total Commits**	**Total Lines of Code**	**ID**	**Total Commits**	**Total Lines of Code**
1	1721	341868	15	1574	333870
2	1720	342770	16	1572	333190
3	1719	341764	17	1557	331660
4	1718	341396	18	1547	331176
5	1705	340434	19	1544	331092
6	1668	338860	20	1543	331026
7	1659	337816	21	0	0
8	1648	337304	22	1493	664716
9	1647	337222	23	1492	664556
10	1643	336854	24	1489	664156
11	1632	336138	25	1479	663504
12	1598	335550	26	1478	663406
13	1587	335112	27	1464	660672
14	1576	334086	-	-	-

Table 4.5: Size of each buggy version considered in the Joda-Time experiments

several reasons, despite the advent of Java.time in Java 8, which has superseded Joda-Time for many new projects. Timekeeping is inherently complex, dealing with issues like time zones, leap years, daylight saving changes, and more. Joda-Time's attempt to provide a comprehensive solution to these problems offers a rich domain for identifying and predicting potential bugs, especially those related to edge cases and rare conditions.

Joda-Time introduced many design improvements over the original Java date and time API, including immutability and a more intuitive interface. Analyzing bugs and test cases in Joda-Time can provide insights into how its design influences error rates and what patterns lead to more robust code.

Given the critical role of date and time operations in performance-sensitive applications, exploring how automated test generation can uncover performance regressions or inefficiencies in Joda-Time implementations can yield valuable strategies for optimizing time-related code.

Joda-Time provides a moderately complex, manageable, and simple real-world codebase for researchers and practitioners new to bug prediction and automated testing. It is an excellent case study for developing, applying, and refining techniques in a practical context.

As a well-known and stable library, Joda-Time can be a benchmark for evaluating the effectiveness of bug prediction algorithms and automatic test case generation tools. The insights gained can be compared across different domains or libraries to assess the generalizability and efficiency of these techniques.

Table 4.5 shows the total number of commits and the number of lines of code for each buggy version of the Joda-Time we considered in our experiments.

4.2.2 Data Collection Methods

For each bug prediction model used in our experiments, it was necessary to develop datasets on which they could be developed. We created four datasets for prediction model development, one for each subject system. As cross-project defect prediction involves several factors beyond our research's scope, we chose to create models specific for each project (Goel and Gupta, 2020).

To gather the software history metrics necessary for each dataset, we employed automated tools and custom scripts designed to integrate with Git. The metrics collected are as follows:

- **Number of Lines of Code (LOC) per Class in Each Commit:** We developed a custom script to calculate each class's LOC within the subject systems' buggy versions at every commit. This metric provides insights into the code's growth or reduction over time, offering a measure of complexity and identifying potential risk areas.
- **Number of Commits per File:** By executing specific Git commands, we tracked the total number of commits made to each file. This metric helps identify frequently modified files, which may be more susceptible to bugs.
- **Number of Added Lines of Code for Each File by Commit:** Utilizing the `git diff` command, we captured the number of lines added to each file throughout its commit history. This metric is indicative of development activity and the evolution of the code.
- **Number of Deleted Lines of Code for Each File by Commit:** Similarly, we used the `git diff` command to measure the number of lines removed from each file over its commit history, providing insight into code simplification or refactoring efforts.
- **Number of Authors:** We calculated the number of unique authors contributing to each file using Git's log features. This metric assesses the breadth of developer engagement and can indicate files with higher collaboration levels and bug proneness.
- **Number of Commits:** We summarized each file's total number of commits until the buggy version to account for the overall development activity.
- **Age:** This metric calculates the time from the first commit to the last in the dataset for each file, offering an understanding of the file's longevity and maturity within the project.

These metrics are critical for understanding the historical and developmental context of the software systems being studied, providing a quantitative foundation for our experiments.

We automated the data collection as much as possible to minimize human error and ensure consistency across multiple experimental runs. Automation also made it easier to replicate the experiments under different conditions and configurations. This data collection approach is critical for evaluating the impact of EPMOSA on the quality and efficiency of the generated test suites. Figure 4.1 shows an overview of the data collection flow we adopted in this dissertation.

The Algorithm 9 shows the pseudo-code for the algorithm developed to obtain the history features for each buggy version of the subject systems. Size metrics like the number of authors,

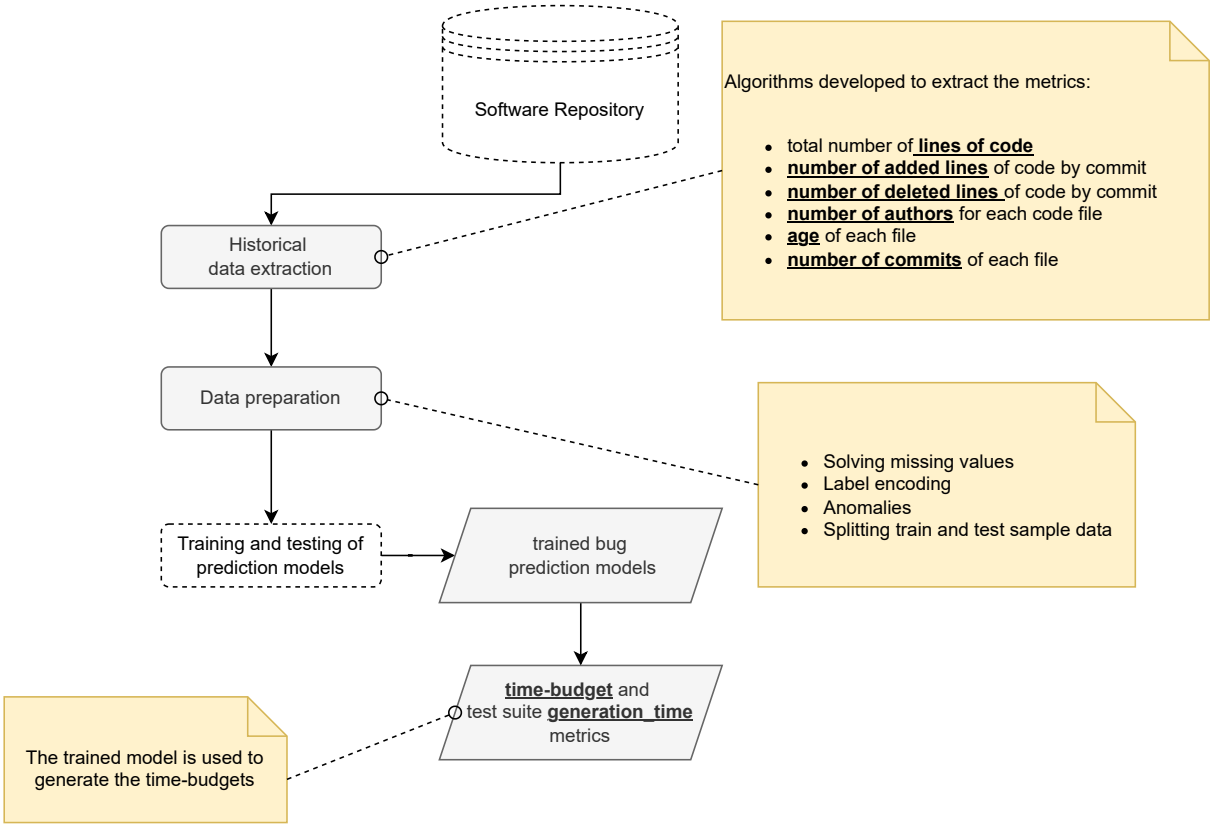

Figure 4.1: Data collection flow for the development of the bug prediction datasets used in the experiments

changes, and age of software history features provide unique insights into a file's characteristics and development history within a software system (Marinescu and Marinescu, 2019).

The number of authors metric highlights the collaborative nature of software development, revealing the breadth of knowledge and variety of coding styles that influence the file's code. A file with contributions from several authors might indicate a thorough review process, potentially leading to higher-quality code (Marinescu and Marinescu, 2019). However, this could also increase the complexity and variability in coding practices, affecting the code's consistency and maintainability. By examining this metric, we can investigate the connection between collaboration and software quality, especially regarding the introduction and resolution of bugs.

Calculating the age of the file from the time it was last committed to the present provides insights into the maturity and stability of the software component (Challet and Valverde, 2008). Older files may have undergone more extensive testing and debugging, potentially indicating reliability. However, they might also be more prone to legacy issues or compatibility challenges with newer technologies. The age metric helps in understanding the lifecycle of a code file and its evolution over time, offering clues about its maintenance, updates, and relevance in the current codebase (Challet and Valverde, 2008).

The number of distinct changes (Commits) can also reflect the stability of a file. A higher number of commits might indicate active development or frequent modifications, which could be due to feature enhancements, bug fixes, or code refactoring (Trautsch et al., 2023). While active development can lead to improvements and adaptation to new requirements, it might also introduce new bugs or lead to regression in existing functionalities (Tang et al., 2015). This

metric is essential for assessing the dynamic nature of the file's development history and its impact on software quality.

The question of whether size code metrics are better than complexity code metrics for software engineering research purposes has been explored in several studies, each offering insights into different aspects (Peitek et al., 2021, Chowdhury et al., 2022, Trautsch et al., 2023). Size metrics, such as lines of code, have been widely recognized for their correlation with many other standard metrics and ability to estimate software maintenance effort effectively (Jangra et al., 2022, Chowdhury et al., 2022). Studies have shown that keeping methods under a specific size can significantly decrease future maintenance efforts (Chowdhury et al., 2022). This suggests that size metrics provide a straightforward and effective measure for guiding development practices to reduce maintenance burdens (Moh, 2019, Abd, 2017). Furthermore, the critical role of size in determining software cost, schedule, and effort has been emphasized, highlighting its importance in accurate project estimation and management (Sathesh and Hamdan, 2022, AL-Saleem and Hammo, 2022, Kumar and Jayaram, 2022).

On the other hand, complexity metrics, which include cyclomatic complexity and coupling, have been used to predict software quality aspects such as maintainability, bug proneness, and robustness (Chowdhury and Zulkernine, 2011, Masmali et al., 2021). However, the relationship between complexity metrics and software testing efforts shows only a moderate correlation, indicating that complexity metrics alone may not be sufficient for estimating testing efforts (Yamashita et al., 2016). Moreover, the predictive power of complexity metrics for identifying vulnerable code locations has been questioned (Medeiros et al., 2020, Dam et al., 2021), with studies finding that measurement of complexity metrics, which are collected during code execution, may offer better indicators of vulnerability than static complexity metrics (Chowdhury et al., 2022).

Interestingly, some research suggests that the effectiveness of both size and complexity metrics may be influenced by their correlation with the size of the code artifact (Shah and Morisio, 2013, Yamashita et al., 2016, Gil and Lalouche, 2017a). It has been argued that the validity of many metrics can be explained by their correlation to size, indicating that size may be the "unique" valid metric (Chowdhury et al., 2022). Additionally, the practical applicability of complexity metrics has been compared, with cognitive functional size identified as a potentially more suitable metric for real-world use (Gil and Lalouche, 2017b). In conclusion, while both size and complexity metrics have their applications and benefits, size metrics offer a more direct and generally applicable measure for estimating testing efforts and guiding development practices (Shin and Williams, 2011, De Silva et al., 2012).

4.2.3 Benchmark and General Comparison Criteria

The overall objective of EPMOSA is to use an ensemble of different bug prediction models to enhance the effectiveness of automated test suites in finding bugs. In this section, we discuss the benchmark and comparison strategies we used in our investigation and explain the rationale behind these decisions.

Conventional test case generation SBST approaches often focus on a single objective or

Algorithm 9 Get History Features from Git Repository

Input:
    graph: The graph data structure containing commit and file information
    repo_path: The path to the git repository
    branch: The branch from which to extract commit history
Output:
    history_features: A list of historical features including the total number of authors, age, and the total number of changes
procedure GETHISTORYFEATURES
    $repo \leftarrow$ project repository path
    $head \leftarrow$ git head reference for the selected branch
    $commits \leftarrow$ list of commits from the repository
    $history\_features \leftarrow$ empty list
    for each commit in commits **do**
        $files \leftarrow$ list of files changed in the commit
        $authors\_set \leftarrow$ empty set
        $commit\_list \leftarrow$ empty list
        $previous\_commit\_datetime \leftarrow$ None
        for each file in files **do**
            Add authors that changed the file to $authors\_set$
            Add commit to $commit\_list$
            if $previous\_commit\_datetime \neq$ None **then**
                $time\_difference \leftarrow current\_commit\_datetime - previous\_commit\_datetime$
                $age \leftarrow time\_difference.total\_seconds()/60$
            else
                $age \leftarrow 0$
            end if
            $previous\_commit\_datetime \leftarrow current\_commit\_datetime$
        end for
        $number\_of\_authors \leftarrow$ size of $authors\_set$
        $number\_of\_changes \leftarrow$ length of $commit\_list$
        Add $\{number\_of\_authors, age, number\_of\_changes\}$ to $history\_features$
    end for
    Save $history\_features$ to file
    return $history\_features$
end procedure

combine several objectives into one composite fitness function, facing scalability challenges (Lemieux et al., 2023, Neelofar et al., 2023). These traditional approaches struggle to address the complex and multifaceted requirements of modern software testing, such as achieving high coverage across multiple dimensions, including branches, statements, and mutations. In this context, DynaMOSA stands out by explicitly tackling the inherent multi-objective nature of test case generation (Panichella et al., 2018).

DynaMOSA employs a many-objective optimization technique, considering each test target as distinct objectives. This strategy allows DynaMOSA to prioritize and adapt test targets based on their significance. Test targets critical for fulfilling specific coverage criteria – e.g., statement, branch, or mutation coverage – are considered more significant. Consequently, DynaMOSA facilitates the generation of test suites that effectively achieve high code coverage.

When comparing DynaMOSA to other standard SBST test case generation approaches, it is clear that DynaMOSA can improve the quality of the automatically generated test suites by offering higher test coverage (Panichella et al., 2018). Also, DynaMOSA outperforms traditional SBST test case generation approaches when it comes to solving multi-objective problems, thanks to its effective navigation through complex code.

The main strength of DynaMOSA lies in its dynamic prioritization and selection of test targets. It continuously refines its focus on different aspects of the codebase based on ongoing coverage analysis and testing needs. This adaptive strategy improves testing efficiency by smartly allocating resources to areas of code needing more rigorous examination, thus accelerating the testing cycle without sacrificing testing depth or breadth. Panichella et al. (2018), Sales and de Santiago Júnior (2020), Li et al. (2022), Fieldsend et al. (2022) collectively highlight the critical role of many-objective optimization in SBST for test case generation and underscore the need to refine existing approaches to achieve better software testing results.

Our evaluation framework (explained in detail in Section 4.2.5.1) defines DynaMOSA (Dynamic Many-Objective Sorting Algorithm) as the benchmark for the experiments in this dissertation. We compare the proposed approach with DynaMOSA using two criteria: *effectiveness* and *efficiency*. We define effectiveness as the ability to discover software bugs, whereas efficiency is measured by the time required to create the test suites. In Figure 4.2, we show a mind map to illustrate how we differentiate between efficiency and efficacy. This mind map allows us to organize each concept's definitions, examples, and critical characteristics in a structured manner, facilitating a better understanding and comparison of our main high-level comparison criteria.

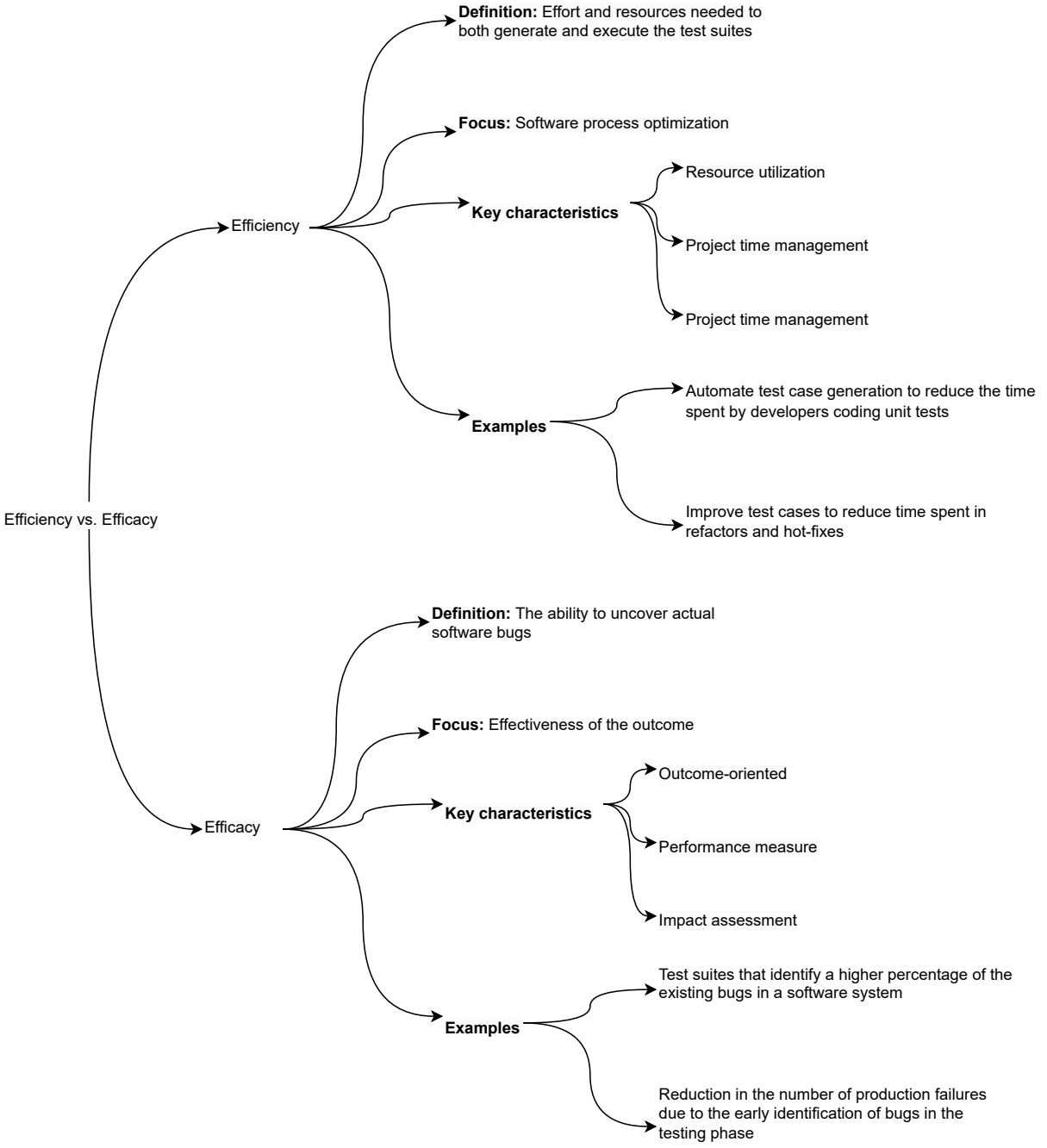

Figure 4.2: Efficiency vs. Efficacy Mindmap

In the context of our evaluation framework, effectiveness is the ability of the automatically generated test suite to identify bugs. This ability is crucial since it directly impacts software quality by revealing and allowing developers to avoid defects that could lead to failures. Ideally, a test suite should be able to detect the full range of bugs, including those that are difficult to detect, resulting in increased software reliability and stability. That is why effectiveness is an essential criterion for the test suite. Automatically generating test suites that excel at detecting actual bugs helps to improve the overall software development process, resulting in a product that meets its specifications and user requirements while alleviating the burden on testers and developers who often invest considerable time in coding effective test suites.

On the other hand, efficiency refers to the temporal effort dedicated to the testing process,

encompassing the time required to generate and execute test suites. Our evaluation framework defines efficiency as a measure of the test suite's resource economy, showing its ability to perform within acceptable time limitations. An efficient approach can quickly build and execute test suites, allowing for a shorter development cycle and earlier discovery of bugs. This can result in significant cost savings and resource optimization. This measure is crucial in large-scale software projects or when iterative development approaches, such as Agile, are used, as rapid testing cycles are critical to the project's success (Mohammadian and Javed, 2021).

By emphasizing these two comparison criteria, our evaluation framework seeks to provide a comprehensive picture of the performance of the automatic test case generation approaches under discussion. It tries to determine whether the studied approaches excel in uncovering most bugs and do so time-efficiently, ensuring that the software development process remains agile and cost-effective. This dual emphasis on efficacy and efficiency enables a fair evaluation of the proposed approach.

In Figure 4.3, we show an overview of the core sequence of activities conducted and defined in the methodology for this dissertation. This illustration provides insight into the use of several tools and frameworks, depicts the flow of data between them, and identifies the nature of resources employed – predominantly Perl scripts (.pl) from Defects4J for test suite generation and execution, along with executables (.jar) provided by EvoSuite, an external dependency of the Defects4J framework. Further details on each activity will be provided during the definition of the executed experiments.

4.2.4 Automatic Test Case Generation with SBST Approaches

The Reachability, Infection, and Propagation (RIP) model is a conceptual framework used to understand the conditions that must be met for a test case to detect a bug successfully (Morell, 1990, Offutt and Hayes, 1996, Ammann and Offutt, 2008). The RIP model outlines three stages a test case must go through to uncover a bug:

1. *Reachability:* The test case must reach the location in the code where the bug is. This means executing instructions that include the defective piece of code.

2. *Infection:* Once the bug is reached, the test case must cause the program's state to be incorrect or *infected*. The test case must trigger the bug, leading to an erroneous state within the program.

3. *Propagation:* The incorrect state must then propagate to the program's output or final state so that it can be observed. The bug might remain hidden without propagation despite causing an internal incorrect state.

Consider a scenario where a test case is generated and reaches a defective code segment (satisfying reachability), triggers the bug (satisfying infection), and the error propagates to the output (satisfying propagation). However, without a test oracle to judge the output, the bug's presence cannot be confirmed, making the test case ineffective in bug detection. The Listing 4.1 shows an example to illustrate the previous scenario, inspired by the kind of functionality one

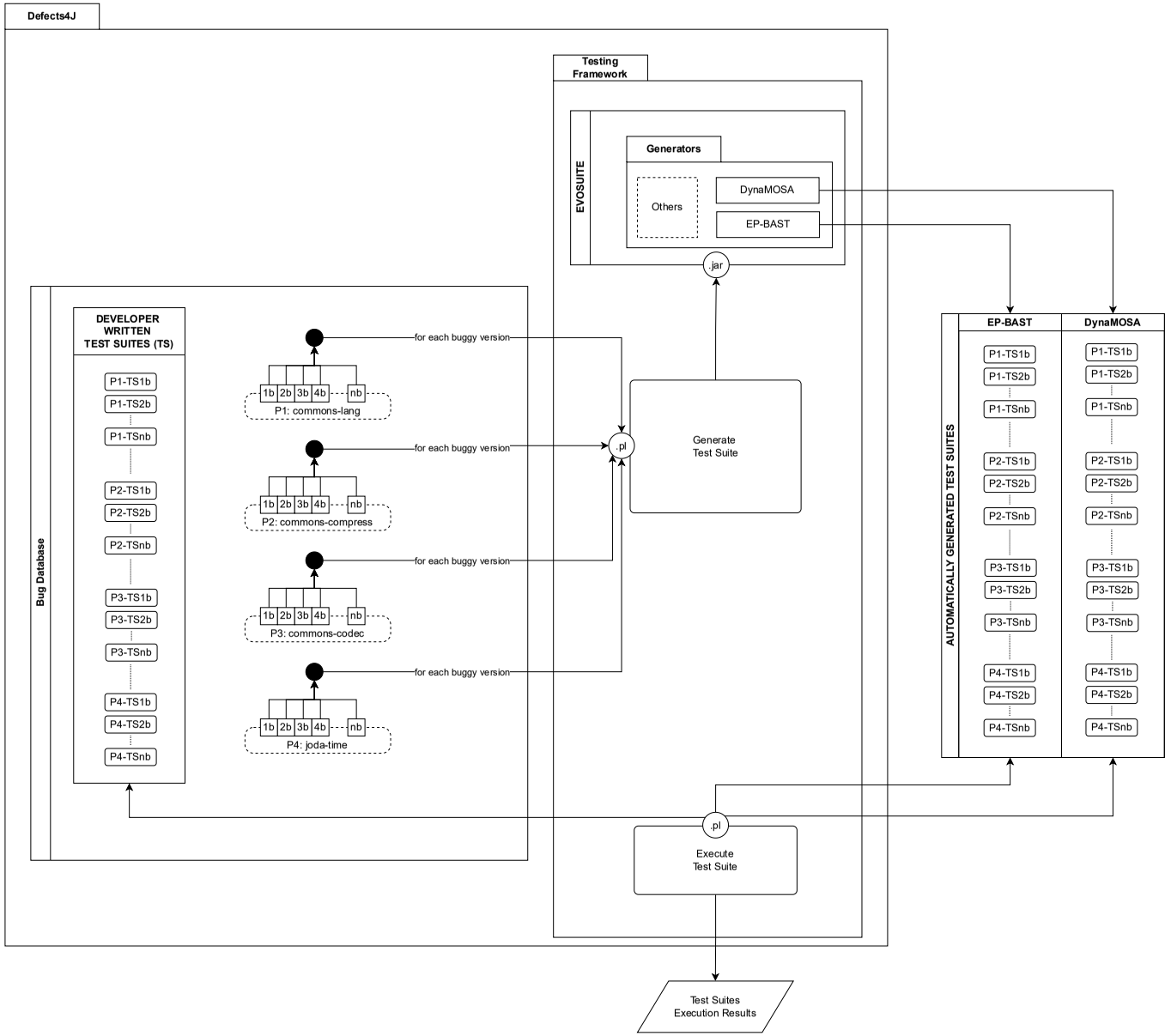

Figure 4.3: Overview of our methodology activities: Resources are represented by circles, with Perl scripts (.pl) for test generation/execution and EvoSuite executables (.jar) as Defects4J's external dependency. Buggy versions are denoted as <bug_number>b (e.g., "1b" for bug ID 1). "P" with a number refers to the project, and 'TS' to 'Test Suite'. For example, "P1-TS1b" is the test suite for project P1's bug 1.

might find in the Apache Commons Lang library (one of the subject systems of our experiments). The example focuses on string manipulation to illustrate the RIP model and the necessity of a test oracle.

Consider the java class in Listing 4.1 called `StringUtils` that contains a method `reverseString(Stringinput)`, which is supposed to reverse the input string parameter. However, this method tends to fail when given a **`null`** input, instead of safely returning a **`null`** value or throwing a controlled exception.

In Listing 4.2, a test case for the method `reverseString(Stringinput)` is shown. It is possible to observe that the test case successfully calls the `StringUtil.reverseString(null)` method, reaching the buggy code segment that does not handle **`null`** inputs (reachability). Also, the program's state becomes incorrect when attempting to instantiate a `StringBuilder` with a **`null`** input, leading to a `NullPointerException` (infection). Finally, the incorrect state (the exception) does indeed propagate to the output of the method if we were observing for exceptions (propagation).

```java
public class StringUtils {
    /**
     * Reverses the given string.
     *
     * @param input The string to be reversed.
     * @return The reversed string or throws a NullPointerException
     *         if the input is null.
     */
    public static String reverseString(String input) {
        // Bug: The method does not handle null inputs properly.
        return new StringBuilder(input).reverse().toString();
    }
}
```

Listing 4.1: StringUtils class implementation

```java
import static org.junit.Assert.*;
import org.junit.Test;

public class StringUtilsTest {

    @Test
    public void testReverseString() {
        String input = null;
        String result = StringUtils.reverseString(input);
        /**
        *  Missing Test Oracle: We do not verify the outcome against
        *  an expected result.
        */

    }
}
```

Listing 4.2: StringUtilsTest test case

In the previous example, although the test case satisfies the conditions of the RIP model, it lacks a test oracle. Consequently, the test case in Listing 4.2 cannot reveal bugs independently. This test case requires human intervention or additional information/tools to insert appropriate test oracles, which can assess the correctness of the program's output and confirm the detection of bugs.

A test oracle must be added to our test case to evaluate the bug detection. The test oracle will verify the outcome against an expected result or behavior, such as expecting a `NullPointerException` or checking for a specific safe handling behavior.

In the Listing 4.3, the test oracle is the expectation of a `NullPointerException` When the input is null. With the test oracle in place, the test can now effectively uncover the bug by verifying that the method's behavior aligns (or, in this case, does not align) with the expected outcome. In these code examples, we simplify real-world scenarios for illustrative purposes. However, they capture the essence of the RIP model and the crucial role of test oracles in automated testing.

```java
// Test Oracle: Expecting a NullPointerException.
@Test(expected = NullPointerException.class)
public void testReverseStringWithNullInput() {
    String input = null;
    StringUtils.reverseString(input);
}
```

Listing 4.3: StringUtilsTest test case with oracle

In our experiments, we generate test cases with Evosuite, a tool that automates the generation of test cases (Fraser and Arcuri, 2011). We chose Evosuite mainly because it implements the

DynaMOSA algorithm within its suite of search algorithms. This feature significantly enhances our ability to conduct comparisons and experiments efficiently and effectively against the chosen benchmark (Section 4.2.3).

When generating test cases, EvoSuite tries to maximize the coverage of the codebase by exploring as many execution paths as possible. This exhaustive approach ensures the evaluation of the program's behavior under various conditions. To validate the correctness of these behaviors, EvoSuite automatically generates assertions based on the program's output during the test case generation process.

It is important to note that EvoSuite generates assertions based on the program's current behavior, effectively assuming its current behavior is correct. If the program produces the correct output for a test input, the assertion will capture this correct behavior. However, in a scenario where the program produces incorrect output due to a bug, EvoSuite will still generate an assertion that accepts this incorrect output as correct because it has no external point of reference to judge the correctness of the output.

The assertions generated by Evosuite cannot identify current defects because they are based on the premise that the program's current behavior is correct. If defects in the program cause it to behave incorrectly, EvoSuite's assertions will codify this incorrect behavior as if it were correct. A simple example is given in Listing 4.4.

Supposing we have a simple method to add two numbers and that this method contains a bug; instead of summing up the numbers, it subtracts them. If EvoSuite tests this method with add(2, 2) and observes the output 0, it might generate an assertion like in Listing 4.5. This assertion reflects the observed (but incorrect) behavior, assuming it to be correct. It does not identify the defect; instead, it asserts that the result of `add(2,2)` should be 0, reinforcing the incorrect behavior.

```
1 public int add(int a, int b) {
2     // Defective implementation:
3     return a - b; // Should be a + b
4 }
```

Listing 4.4: Incorrect implementation of addition method

```
1 @Test
2 public void testAdd() {
3     assertEquals(0, add(2, 2));
4 }
```

Listing 4.5: Add method test case without oracle

For assertions to effectively identify bugs, they must be based on an external definition of correctness – an understanding of what the program is supposed to do, not just what it currently does. This could come from specifications, requirements, or other forms of documentation. Without this external reference, tools like EvoSuite can generate test cases that achieve high coverage and even include assertions. However, these assertions will only help identify defects if

the initial correctness of the behavior they are based on is verified by other means.

Test cases automatically generated by Evosuite are useful for regression testing, where the goal is to ensure that previously correct behavior has not changed. In our experiments, we consider only regression test cases, as most automatic test case generation tools do.

In this dissertation, we focus on improving the test suites' capability to identify bugs rather than automating the process of evaluating whether the test cases can reveal bugs, a problem known as oracle automation (Zhang et al., 2019, Kou et al., 2021). In the context of this research, SBST approaches like DynaMOSA or the one proposed in this dissertation are considered successful in detecting a bug if they can generate a test case that causes an internal error to manifest in the program's output. This means that the error in the code is propagated through the execution of the test case, making it observable outside the program by a fail in the test case. However, it is important to note that neither DynaMOSA nor the proposed approach in this dissertation can identify bugs *independently*. They require oracle automation, which is the mechanism that determines whether the output of a test case indicates the presence of a bug.

Without oracle automation, SBST approaches cannot confirm the existence of bugs; they can only generate test cases that might expose them. This limitation is not unique to the SBST approaches discussed and proposed in this dissertation. It is a known challenge faced by SBST for test case generation, as referenced by the literature (Zhang et al., 2019, Kou et al., 2021). The implication is that while SBST test case generation approaches can be guided to focus on areas of the code that are likely to contain defects, the actual identification of bugs still relies on having an effective oracle to interpret the test results.

4.2.5 Result Analysis Approach

We employ descriptive and inferential statistics to carefully analyze and understand the outcomes of our experiments, aiming for precision and rigor.

- **Descriptive Statistics:** Our initial analysis summarizes the bug identification rates observed across all experiments. We calculate the metrics mean, median, standard deviation, and variance. These statistics help capture the experimental outcomes' central tendency and dispersion. This analysis provides a comprehensive snapshot of how EPMOSA, guided by various ensemble prediction models, performs relative to DynaMOSA and its configurations when enhanced with single prediction models. By detailing these measures, we offer an overview of the behavior of these approaches in different configurations, highlighting how each type of prediction model influences the effectiveness of the generated test suites. This step not only elucidates the inherent performance characteristics of each approach but also sets the stage for deeper inferential analysis to assess the impact of these configurations on the overall success of the test case generation process.

- **Inferential Statistics:** Building on the insights gained from the descriptive analysis, inferential statistics allow us to conclude the broader implications of our findings. We employ the statistical tests ANOVA (von Eye and Wiedermann, 2023) and the Wilcoxon-Mann-Whitney U test (McKnight and Najab, 2010) to determine the statistical significance

of the observed effects. Specifically, these tests help assess the impact of the Ensemble Predictive Many-Objective Sorting Algorithm (EPMOSA) on the efficacy of the test suites generated.

4.2.5.1 Bug Detection Evaluation Protocol

Besides being a well-known bug database, Defects4J (Just et al., 2014) also provides a testing framework that defines a systematic approach to accessing and manipulating each project and bug in its repository. The Defects4J framework is primarily used to facilitate research in software testing, debugging, and repair. However, it does not independently generate test cases; instead, the Defects4J framework uses Evosuite (Fraser and Arcuri, 2011) to generate test cases.

The Defects4J organizes bugs in a database, with each bug linked to a particular commit in a version control system that either introduced or fixed the bug. Additionally, it includes developer-created test cases that expose the bug. A corresponding test suite is generated and subsequently executed for each buggy version of the subject systems considered in our study (Section 4.2.1). This way, we can evaluate the effectiveness of each generated test suite in accurately detecting bugs. As the Defects4J framework provides developer-written test cases that expose the bug of each studied version, we compare the outcomes of executing the generated test suites with those of the developer-written test cases. We determine that the generated test suite identified a bug if it fails for the same software components as the developer-written test cases. This process is illustrated in Figure 4.4.

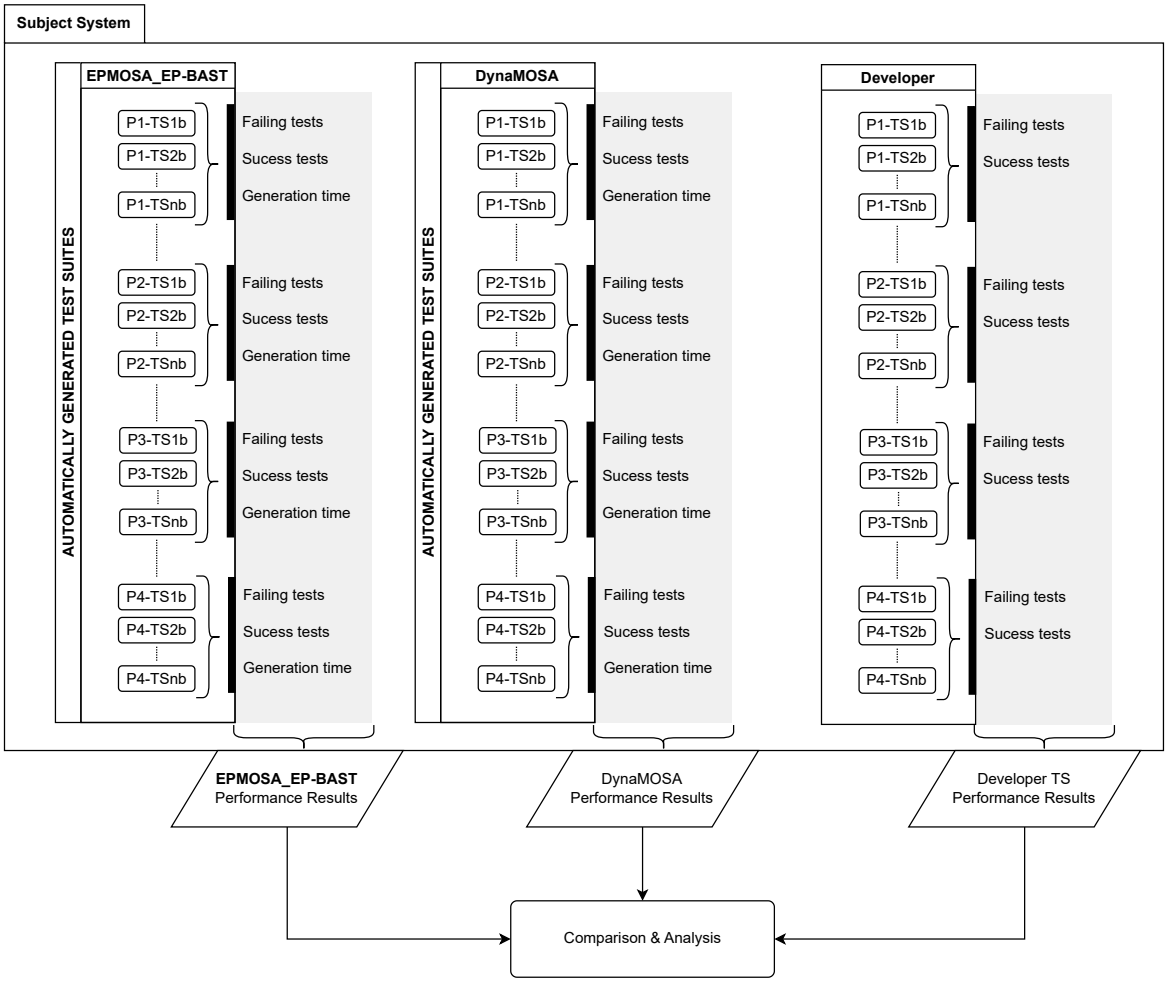

Figure 4.4: Evaluation Protocol

We directly compare the outcomes of executing test suites with EPMOSA and DynaMOSA. This direct comparison measures the effectiveness and efficiency of our test generation approach. It is based on the bug identification rates obtained in each experiment.

4.2.5.2 Performance Metrics

This section will elaborate on the specific metrics used to quantify efficacy and efficiency, explaining how each metric is calculated and why it is important. Understanding these metrics is essential for interpreting the results of our experiments.

4.2.5.2.1 Bug Identification Rate: The bug identification rate represents the proportion of known bugs successfully identified by the generated test suites. Specifically, this rate is calculated by dividing the number of bugs correctly identified by the testing suite by the total number of bugs for the subject system, often expressed as a percentage. A higher bug identification rate indicates a more effective test suite, as it demonstrates the suite's ability to catch a more significant number of existing defects, thereby reducing the risk of bugs slipping into production. Enhancing the bug identification rate is a primary goal in software testing, aiming to improve the accuracy and efficiency of test cases in uncovering flaws before software release.

4.2.5.2.2 Test Suite Generation Time: The test suite generation time refers to the duration of generating a set of test cases using the proposed approaches. This metric is essential to assess the computational effort and scalability of the SBST approach. It can be measured by recording the time from the initiation of the test generation process until the final test suite is produced.

Higher test suite generation times imply that generating test cases using a search-based algorithm is more computationally demanding and slower. This could be due to various factors, such as the complexity of the software being tested, the sophistication of the search algorithm used by the SBST approach, or the size and scope of the test suite being generated. High generation times can be a concern if the testing process becomes a bottleneck in the development cycle, especially in agile environments where quick feedback is essential.

Conversely, lower test suite generation times indicate that the test cases are being generated more quickly. This is generally advantageous as it allows for more rapid iteration and feedback in the testing cycle. It could result from efficient search algorithms in SBST approaches or test generation optimization. Lower generation times enhance the practicality of implementing search-based software testing in a continuous integration/continuous deployment (CI/CD) pipeline, as they minimize the delay between code submission and feedback on its correctness.

4.2.5.2.3 Test Suite Execution Time: The test suite execution time measures how long it takes for the generated test suite to execute all its test cases against the target software system. This metric provides insight into the runtime efficiency of the test suite and impacts the overall testing time budget in practical scenarios.

Higher test suite execution times indicate that the test cases take longer to run, which can be due to several reasons, such as the complexity or length of the test cases, performance issues with the software under test, or inefficient test design that does not prioritize execution speed. Extended execution times can lead to slower feedback loops in development cycles, which may hinder rapid iteration and can be particularly problematic in environments that rely on fast test cycles such as continuous integration.

Lower test suite execution times mean the test cases dash, which is beneficial for rapid testing cycles. This can allow for more frequent test execution, facilitating early defect detection and enabling faster development and release cycles. Efficient test execution is crucial in agile and DevOps practices, where quick turnaround times are essential for continuously delivering software updates.

4.2.5.2.4 ANOVA F-Value: The F-value in an ANOVA test measures how significant the differences are among group means in a dataset. It is a key component in understanding whether any differences between the means are statistically significant. The F-value is the ratio of two variances. It compares the variance between group means (how different the groups are from each other) to the variance within the groups (how much the data within each group varies from their group mean). Mathematically, it is calculated as the variance explained by the groups divided by the variance due to randomness or error within the groups as in Formula 4.1.

$$F = \frac{\text{Mean Square Between Groups (MSB)}}{\text{Mean Square Within Groups (MSW)}} \tag{4.1}$$

In Formula 4.1, the *Mean Square Between Groups (MSB)* is the variance estimate of the group means and reflects the variance of group means around the overall mean. It is calculated by dividing the sum of squares between the groups by the degrees of freedom between them. The *Mean Square Within Groups (MSW)* estimates the population variance based on the deviations of individual observations from their respective group means. It is calculated by dividing the sum of squares within the groups by the degrees of freedom.

ANOVA uses the F-value to test the null hypothesis that all group means are equal against the alternative hypothesis that at least one group mean is different. The significance of the F-value is determined by comparing it to a critical value from the F-distribution table at a chosen significance level (usually 0.05).

In this study, the between-group Variance is the variability in bug identification rate attributed to the different implementations of EPMOSA and DynaMOSA guided by ensemble and single prediction models, respectively. Within-group variance in the variability in bug identification rate within each approach experiment is due to individual differences. In this case, the ANOVA F-value helps determine if the type of defect prediction model used by each approach has significantly different effects on the bug identification rate of the generated test suites by comparing the variability between approaches to the variability within the group of configurations of each approach.

A high F-value indicates a significant variance among group means compared to the variance within the groups. It suggests that at least one group mean significantly differs from the others, implying that different bug prediction models affect the generated test suites differently. Low F-values suggest that any variance among group means is likely due to random fluctuations (i.e., the means of all groups are similar).

4.2.5.2.5 ANOVA p-value: The p-value in the context of an ANOVA test is the probability of observing an F-value as extreme as, or more extreme than, the F-value calculated from your data, assuming the null hypothesis is true. The null hypothesis in ANOVA typically states that all group means are equal. Low p-values (typically < 0.05) indicate strong evidence against the null hypothesis. This suggests a statistically significant difference between the group means, and you would reject the null hypothesis. A high p-value implies weak evidence against the null hypothesis. This means there is not enough evidence from the data to conclude that there is a difference between the group means, and we cannot reject the null hypothesis.

4.2.5.2.6 Mann-Whitney U-statistic: The Mann-Whitney U test is employed to test the hypothesis that two populations are the same against the alternative that one population tends to have larger values than the other. It is an alternative to the independent samples t-test but does not require the assumption of normal distributions.

The U-statistic value is calculated by combining all observations from both groups into a single dataset and ranking them from the smallest to the largest. Ties (equal values) receive a

rank equal to the average of the ranks they would have otherwise occupied. Then, the U statistic for each group is calculated using the Formula 4.2.

$$U = n_1 \times n_2 + \frac{2}{n(n+1)} - R \tag{4.2}$$

Where:

- n_1 and n_2 are the sample sizes of the two groups.
- n is the total number of observations.
- R is the sum of the ranks in the group.

Two U values are calculated, one for each group (U1 and U2). The smaller value of these two is often taken as the test statistic. A smaller U value indicates that most of the ranks in one group are smaller than those in the other, suggesting a difference between the groups. The U-statistic is compared against critical values from a table (or computed via approximation formulas) to determine if the observed difference between groups is statistically significant.

We use the Mann-Whitney U-statistic to compare the effectiveness of two automatic test case generation approaches. The bug identification rate of the generated test suites is ranked from lowest to highest regardless of which approach was used to generate them. If most of the lower ranks belong to one approach and most of the higher ranks to the other, the Mann-Whitney U test could indicate a statistically significant difference in effectiveness between test suites generated by EPMOSA and DynamMOSA or ensemble models and single prediction models.

The Mann-Whitney U test is a powerful tool for statistical analysis when dealing with non-normal distributions or ordinal data, ensuring robustness against violations of the normality assumption that underpins many parametric tests.

4.2.5.2.7 Mann-Whitney U p-value: The Mann-Whitney U test p-value is crucial in interpreting the test results. It provides a probability measure to help decide whether the observed difference between two independent samples is statistically significant. The p-value for the Mann-Whitney U test is determined based on the U-statistic computed from the data. The key steps involved in the calculation are:

1. Obtain the U-statistic: First, compute the U-statistic for each group, as explained previously. Choose the smaller of the two U values to use in subsequent calculations.

2. Determine the Distribution: For small sample sizes, the exact distribution of U under the null hypothesis (assuming no difference between the groups) can be used to determine the p-value. The U-statistic distribution approaches a normal distribution for larger samples due to the Central Limit Theorem.

3. Normal Approximation: When sample sizes are large enough (commonly recommended n1, n2 > 20), the U-statistic is approximately normally distributed with mean and variance given by Formulas 4.3 and 4.4, respectively. Where n1 and n2 are the sample sizes of the two groups, and n is the total number of observations.

4. Calculate Z-score: Convert the U-statistic to a Z-score using the Formula 4.5. This Z-score follows a standard normal distribution under the null hypothesis.

5. Compute the P-value: The p-value is calculated from the Z-score using the standard normal distribution. The calculation will depend on whether a one-tailed or two-tailed test is appropriate. A two-tailed test is commonly used, which looks at the probability of observing a Z-score as extreme or more extreme than the one calculated in either direction.

The result is statistically significant if the p-value ≤ 0.05.

$$\text{Mean} = \frac{2}{n_1 \times n_2} \tag{4.3}$$

$$\text{Variance} = \frac{n_1 \times n_2 \times (n_1 + n_2 + 1)}{12} \tag{4.4}$$

$$Z = \frac{U - \text{Mean}}{\sqrt{\text{Variance}}} \tag{4.5}$$

4.3 Final Considerations

This chapter has outlined the comprehensive research methodology employed to investigate the impact of ensemble bug prediction models on Search-Based Software Testing (SBST). We have detailed the research design, experimental framework, data collection methods, and the benchmarks used for comparison. The systematic approach ensures rigor and replicability in evaluating the proposed methodologies. By integrating multiple ensemble prediction models and comparing them against single prediction models, the study aims to provide robust empirical evidence of the effectiveness of the Ensemble Predictive Budget Allocation Strategy for Testing (EP-BAST) and the Ensemble Predictive Multi-Objective Sorting Algorithm (EPMOSA). The methodology in this chapter lays the groundwork for the experiments and analyses presented in the subsequent chapters, aiming to address the research objectives and validate the hypotheses of this dissertation. The careful planning and execution of these methods are crucial for deriving meaningful and generalizable insights into enhancing SBST with ensemble prediction models.

CHAPTER

5

Time Budget Allocation

Automation plays a crucial role in software testing due to the high cost and inefficiency associated with manual testing (Kumar and Mishra, 2016, Ama, 2014, Ram, 2006). Several Search-Based Software Testing (SBST) techniques have been proposed and effectively used to automatically produce test cases for large and complex software systems (Arcuri and Galeotti, 2021, Sahoo et al., 2021, Neelofar et al., 2023). The primary objective of unit testing is to create a suite of test cases that thoroughly evaluate the behavior of the class (or program) being tested, known as the Class Under Test (CUT). The efficacy of the generated test cases is assessed using predetermined adequacy criteria, such as branch coverage, statement coverage, mutation score, and input/output diversity (Panichella et al., 2015, Tonella, 2004a).

The term *test targets* (or coverage targets) denotes the specific software component that needs testing (Rechtberger et al., 2022). These targets can be different entities like classes or modules within a program, which are then transformed into mathematical functions that quantify a particular test case's proximity or *fitness* to the target. Essentially, these mathematical functions ascertain the proximity or effectiveness of a test case in covering the target aspect (Rechtberger et al., 2022).

A fitness function should minimize the distance between the test case and the target (Avdeenko and Serdyukov, 2021). The proximity of the test case to the target indicates the extent to which it covers the intended aspect of the software system.

Usually, fitness functions are defined using heuristics that evaluate the distance between the test case and the target. For instance, a combination of statement and branch distance heuristics can be used for branch and statement coverage targets (Arcuri, 2013, Bahaweres et al., 2017, Zheng and Lyu, 2023).

A statement heuristic evaluates the number of conditions or decisions (control dependencies) in the code that the test case must assess to reach the target branch, thereby offering a quantitative measure of the proximity of the test case to cover the target (Saha et al., 2022). On the contrary, the branch distance heuristic evaluates a test case's distance from satisfying a conditional

expression that dictates whether the test case execution follows a path leading to the target branch (Sahoo and Ray, 2020, Derakhshanfar et al., 2022).

Search algorithms use fitness functions during automatic test generation (Arcuri, 2013, Bahaweres et al., 2017, Zheng and Lyu, 2023). The primary objective of the search algorithm is to minimize a fitness function, aiming to identify test cases that offer optimal coverage and have higher probabilities of exposing faults in the CUT. In automatic test case generation, search algorithms are accompanied by a concept known as a `search budget` (Di Pompeo and Tucci, 2022). This term refers to the distribution of resources dedicated to testing, such as the number of iterations or the time allocated for the search algorithm to navigate the search space and produce test cases.

Within a straightforward test case generation method, referred to as a single-target approach, each target is assigned a portion of the total search budget (typically an equal amount of search time) (Scalabrino et al., 2021). Here, a search algorithm is executed to cover the target. In scenarios involving genetic algorithms, test cases (chromosomes) undergo iterative evolution through genetic operators like selection, crossover, and mutation. During each search process, a single fitness function is optimized, and the search halts upon reaching either a zero-fitness value (indicating the target achievement) or exhausting the search budget (Arcuri, 2013, Xu et al., 2018, Scalabrino et al., 2021). The final collection of test cases constituting the test suite is compiled by gathering generated test cases covering specific targets from various independent searches.

Nevertheless, it is essential to acknowledge that not all targets necessitate an identical search budget (Perera et al., 2023a). Due to their complexity, specific targets may demand additional time for test case generation. Moreover, some CUTs might encompass infeasible targets, resulting in a squander of testing endeavors. Regrettably, determining whether a target is infeasible or feasible yet challenging to cover through test cases remains difficult.

5.1 Formulating the Time Budget Optimization Problem

The definition of suitable strategies for time budget allocation in software testing is essential for optimizing resource utilization in software development. The primary objective of allocating time budgets is to enhance the bug detection capabilities of an automatically generated test suite while working within resource constraints (Perera et al., 2020).

In the context of software development, the testing phase aims to identify all bugs in the code to ensure a flawless release. However, due to the inherent nature of bugs, complete and accurate identification of errors before they cause failures is not always feasible. This limitation, combined with the impracticality of exhaustive testing, means that some bugs may go undetected even after testing.

The amount of time allocated to testing directly impacts the expected coverage. A larger time budget allows for a broader scope of testing, increasing coverage and thereby improving the chances of detecting more bugs. Nevertheless, it is crucial to recognize that despite generous time budgets and extensive coverage, achieving absolute bug detection is more of an aspirational

goal than a practical certainty in test suites.

Various strategies have been suggested to tackle the challenges of bug identification through testing activities. These strategies aim to optimize bug detection while adhering to time constraints in software development. Most proposed approaches rely on understanding the complexity of the code and the distribution of bugs in software systems.

Different structural properties of code components can influence the effectiveness of search-based techniques in achieving high coverage levels in automatic test case generation. Therefore, allocating a uniform search budget for all components is discouraged, and a component-specific search budget is preferred. However, determining the appropriate budget for each component is a non-trivial task.

Scalabrino et al. (2021) introduces a significant contribution to the field of SBST. Firstly, it presents the Budget Optimization for Testing (BOT) approach, designed to allocate a search budget to different classes under test adaptively. This approach is based on the understanding that code components possess varying structural properties, which influence the effectiveness of SBST techniques in achieving high code coverage levels. By employing BOT, the time budget allocation is tailored to the specific needs of each class, rather than using a one-size-fits-all fixed budget allocation approach, which is not optimal due to the diverse complexity and characteristics of code components. Secondly, the authors introduce BRANCHOS, an approach that predicts the branch coverage achieved in each class with a given search budget. BRANCHOS operates budget-awarely, providing time-efficient estimations of the branch coverage with minimal error. This predictive capability is crucial for the BOT approach to function effectively, as it relies on accurate predictions of coverage outcomes to make informed decisions about budget allocation. The experimental results detailed in the paper underscore the efficacy of both BOT and BRANCHOS. It is demonstrated that BRANCHOS can approximate branch coverage with a low error rate and that BOT can significantly enhance the coverage achieved by a test generation tool. Furthermore, the effectiveness of the generated tests is also improved, indicating that the adaptive search budget allocation strategy not only increases code coverage but also contributes to the generation of higher-quality test cases.

Other notable works in the field include the use of Genetic Algorithms (GAs) to evolve test suites towards maximizing code coverage, as seen in the foundational work by (Michael et al., 2001), which laid the groundwork for many SBST approaches. For example, (Lakhotia et al., 2007) introduced a formal study on applying Search-Based Software Engineering (SBSE) techniques to software testing, further establishing the relationship between code coverage and the effectiveness of bug detection. Moreover, the work by McMinn (2004) on Search-Based Software Test Data Generation provides a comprehensive survey of SBST techniques, highlighting the emphasis on code coverage as a primary objective for generating compelling test cases.

High code coverage alone is insufficient and does not guarantee high efficiency in detecting bugs. Test cases should be created only for buggy classes to reveal them to the developer. Perera et al. (2023a) research recognizes that achieving high code coverage does not necessarily

translate to high bug detection efficiency. The authors propose an approach that integrates defect prediction to direct the generation of test cases toward potentially buggy areas. Defect prediction to guide SBST techniques enables the test generation process to concentrate more on exploring tests in areas likely to contain defects rather than areas deemed non-defective.

Employing defect prediction to guide the automatic generation of test cases is appealing. Perera et al. (2023a) demonstrate promising empirical outcomes regarding test suite effectiveness using data from simulated single-model defect predictors to guide the test generation process with Evosuite, an advanced SBST tool. The good outcomes could be attributed to faulty code representing only a tiny fraction of the codebase. SBST techniques lacking guidance on the probable location of defective code tend to spend most of their time on non-defective regions.

The integration of defect prediction to guide SBST techniques is formulated to address the constraints of current SBST techniques that, despite achieving high code coverage, may not produce tests that efficiently uncover bugs. However, integrating defect prediction models into SBST encounters several constraints when relying on single-model defect predictors. A notable constraint is the potential inaccuracy of defect prediction, leading to the allocation of testing resources to non-defective areas of code, thereby diminishing the overall efficacy of the generated test suite. Single-model predictors may fail to capture the multifaceted nature of software defects, resulting in a heightened occurrence of false positives or negatives. This lack of precision can significantly impact the bug detection effectiveness of SBST, mainly if the defect predictor's recall is inadequate, indicating a prevalence of false negatives.

Furthermore, the dependence on a single-model approach may inadequately address the diversity of bugs or the intricacy of various software projects. The performance of defect predictors can vary considerably across diverse contexts and datasets, and a single model may not generalize effectively to all software types. Additionally, the approach may prove less effective when facing strict time constraints for test generation, as the defect prediction model's performance becomes crucial under such limitations. In scenarios where defect predictors with limited recall and precision are utilized, the strategy may not yield the desired enhancements in bug detection, especially if the SBST technique fails to manage potential false negatives efficiently.

We hypothesize that employing ensemble models instead of single models could potentially alleviate the shortcomings of single models. Ensemble models, which combine predictions from multiple defect predictors to improve overall prediction accuracy, make the time budget allocation function even more pertinent. Ensemble models, by their nature, can provide a more refined assessment of defect-proneness, which can then be translated into a more effective allocation of the time budget for SBST. The goal is to create a time budget allocation function that not only accommodates the strengths of individual defect predictors but also harnesses the collective insight provided by an ensemble, thereby maximizing the bug detection rate within the constraints of the allocated time budget. Our proposal is intended to culminate in developing such a function, which will significantly contribute to automated software testing, enabling SBST to utilize the full potential of ensemble defect prediction models.

5.2 Ensemble Predictive Budget Allocation for Software Testing (EP-BAST)

Our approach to time budget allocation has two main modules: (1) Ensemble Defect Predictor (DP) and (2) Ensemble Predictive Budget Allocation for Software Testing.

The defect predictor assigns a defect score to each class in the project, indicating the probability of it being defective. The vector s represents this output. We have chosen the class level as the level of granularity for the defect predictor, and we have selected a set of prediction models to compose the ensemble defect predictor module.

EP-BAST employs a range of models to define time budgets, including BME, GBT, RF, SC, and VC. Each model is used in separate experiments, ensuring that each test run features a distinct configuration of EP-BAST based on the specific prediction model employed. This approach allows us to evaluate the effectiveness of each ensemble model within the EP-BAST framework by isolating its impact on the bug detection rate.

EP-BAST builds upon the framework defined by Perera et al. (2020) for BADS but introduces a significant enhancement: it integrates defect scores derived from ensemble prediction models. This key differentiation not only extends the capabilities of BADS but also significantly enhances the precision and reliability of the defect predictions. By leveraging the collective intelligence of multiple models, EP-BAST provides a more robust and sophisticated approach to allocating testing resources based on predicted defect locations, setting it apart from the BADS approach. The EP-BAST is described in Algorithm 10.

Each model was executed in datasets especially created for the subject projects (Section 4.2.1). The datasets comprise six predictors (features) X, which corresponds to specific project metrics as discussed in Section 4.2.2 and a target variable y, which indicates if a class is buggy or not. In total, we have built four datasets, one for each of the studied projects. These datasets were used to create the models we used in our experiments.

Each dataset is split into training and testing sets, with 70% of the data allocated for training and 30% reserved for testing. This split is stratified based on the target variable to ensure that the proportion of the target classes is similar across training and testing sets. The random state for splitting is fixed at 42 to ensure reproducibility of the results. The dataset related configurations are the same for all experiments and models.

For model evaluation, a 10-fold cross-validation approach is used, where the dataset is split into ten subsets, and iteratively, each subset is used once as the test set. In contrast, the others form the training set. The model's performance is assessed using the metrics:

- Accuracy
- Weighted Precision
- Weighted Recall
- Weighted F1-Score

Algorithm 10 EP-BAST Time Budget Allocation Using Dynamic Prediction Model Execution

Input: Set of all classes C, where $N = |C|$
Prediction model M capable of generating defect scores
$T, t_{\text{min}}, T_{DP}$
Parameters for score integration e_a, e_b, e_c
Output: Time allocations $t = \{t_1, t_2, \ldots, t_N\}$
procedure ALLOCATETIMEBUDGET
    Execute prediction model M to obtain defect scores s
    $s = M.execute(C)$ ▷ Model execution to generate scores for each class
    Initialize total weight $W = 0$
    Initialize time allocations $t = \{0, 0, \ldots, 0\}$
    for each $c_i \in C$ **do**
        Calculate weight based on defect score:
        $w_i \leftarrow e_a + e_b \times \exp(e_c \times s[c_i])$
        $W \leftarrow W + w_i$
    end for
    for each $c_i \in C$ **do**
        Normalize weight:
        $w_i \leftarrow w_i / W$
        Calculate time allocation:
        $t_i \leftarrow w_i \times (T - N \times t_{\text{min}} - T_{DP}) + t_{\text{min}}$
        Update t for class c_i:
        $t[c_i] \leftarrow t_i$
    end for
    return t
end procedure

- ROC AUC Score, configured for multi-class setting using a one-vs-rest approach
- Negative Log-Loss

These metrics provide a comprehensive overview of the model's predictive performance, focusing not just on accuracy but also on the ability to handle class imbalance and the confidence of the predictions.

We built five EP-BAST configurations, each with a different bug prediction model (1) EP-BAST$_{\text{BME}}$, (2) EP-BAST$_{\text{GBT}}$, (3) EP-BAST$_{\text{RF}}$, (4) EP-BAST$_{\text{SC}}$, (5) EP-BAST$_{\text{VC}}$. The following details how we implemented and created each model and the corresponding performance metrics.

5.2.1 Defect Predictors

BME Model

In EP-BAST$_{\text{BME}}$, we configured EP-BAST with the prediction model $M \leftarrow$ BME. The `BaggingClassifier` algorithm from *scikit-learn* is employed, which internally uses a `KNeighborsClassifier` as the *base estimator*. While somewhat arbitrary, the selection of k-Nearest Neighbors (KNN) as the base estimator is also grounded in some beneficial aspects of KNN's behavior and characteristics, particularly in how it complements ensemble methods like bagging. KNN is a non-parametric model that is inherently adaptable to complex data structures due to its reliance on local proximity rather than global data distribution assumptions. This characteristic makes it particularly effective in capturing non-linear relationships that might be present in the data. Moreover, when integrated within a bagging framework, KNN benefits from reduced variance and improved generalization, as bagging effectively mitigate the impact of noise and outliers on the decision-making process. This ensemble approach not only leverages the simplicity and flexibility of KNN but also enhances its robustness and prediction accuracy, making it a good choice despite its initially arbitrary consideration.

The bagging ensemble is configured to use 50% of the samples and 50% of the features for building each base estimator to increase the diversity among the individual learners and potentially improve generalization capability. The performance metrics obtained from the cross-validation of BME the subject systems are summarized in Table 5.1.

Metric	commons-codec	commons-compress	commons-lang	joda-time
Precision	0.9010	0.9455	0.7803	0.8694
Recall	0.8981	0.9466	0.7826	0.8616
Accuracy	0.8981	0.9466	0.7826	0.8616
F1 Score	0.8773	0.9310	0.7744	0.8487
ROC AUC	0.8767	0.8880	0.8538	0.8792
Log Loss	-0.3038	-0.1992	-0.4805	-0.3787

Table 5.1: Summary of Cross-Validation Results Across Projects for the BME Model

These results demonstrate variable performance across different projects, reflecting how contextual factors, such as project-specific characteristics and data distributions, impact model

efficacy. The BME model exhibits particularly strong performance in commons-compress with the highest precision, recall, and accuracy, suggesting better model alignment or more homogeneous data characteristics in this project. In contrast, the commons-lang project shows relatively lower scores, indicating potential data variability or complexity challenges affecting model performance. The analysis emphasizes the importance of contextual adaptation and the potential calibration of predictive models to enhance their performance across varying datasets.

GBT Model

In EP-BAST$_{\text{GBT}}$, we configured EP-BAST with the prediction model $M \leftarrow$ GBT. For the GBT model, we used the `HistGradientBoostingClassifier` algorithm from Scikit-Learn's ensemble module.

The `HistGradientBoostingClassifier` is configured with 100 iterations (`max_iter=100`). This classifier is a decision tree-based ensemble method that uses a gradient boosting framework, which is particularly effective for large datasets and capable of handling missing data implicitly. The model's performance is evaluated using a 10-fold cross-validation approach.

The consolidated performance metrics obtained from the cross-validation of EP-BAST$_{\text{GBT}}$ across different projects are summarized in Table 5.2. This format allows for a direct comparison of metric performance across the projects.

Metric	commons-codec	commons-compress	commons-lang	joda-time
Precision	0.9224	0.9528	0.8385	0.9187
Recall	0.9261	0.9564	0.8399	0.9178
Accuracy	0.9261	0.9564	0.8399	0.9178
F1 Score	0.9210	0.9492	0.8377	0.9149
ROC AUC	0.9350	0.9297	0.9166	0.9502
Log Loss	-0.2037	-0.1301	-0.3501	-0.2185

Table 5.2: Summary of Cross-Validation Results Across Projects for the GBT Model

The table demonstrates variability in the predictive model's performance across different projects. Notably, the model shows particularly strong results in the commons-compress project with the highest precision, recall, accuracy, and F1 scores. In contrast, the commons-lang project exhibited the lowest values across these metrics, indicating potential challenges in model fit or the inherent complexity of the data. The high ROC AUC scores across all projects highlight the model's effective discrimination ability between classes. Moreover, the negative log loss values suggest that the model's probability estimates are well-calibrated across diverse scenarios.

RF Model

In EP-BAST$_{\text{RF}}$, we configured EP-BAST with the prediction model $M \leftarrow$ RF. For the RF model, we used the `RandomForestClassifier` of the `scikit-learn` library. This classifier was instantiated with default parameters and trained using a 10-fold cross-validation procedure.

The consolidated performance metrics obtained from the cross-validation of EP-BAST$_{\text{RF}}$ across different projects are summarized in Table 5.3.

Metric	commons-codec	commons-compress	commons-lang	joda-time
Precision	0.9158	0.9492	0.8370	0.9075
Recall	0.9196	0.9541	0.8379	0.9081
Accuracy	0.9196	0.9541	0.8379	0.9081
F1 Score	0.9133	0.9474	0.8370	0.9056
ROC AUC	0.9045	0.8943	0.9027	0.9119
Log Loss	-0.3994	-0.2646	-0.5212	-0.6044

Table 5.3: Summary of Cross-Validation Results Across Projects for the RF Model

SC Model

In EP-BAST$_{\text{SC}}$, we configured EP-BAST with the prediction model $M \leftarrow$ SC. For the SC model, we used the `StackingClassifier` from the `scikit-learn` library. This model uses a multi-level approach where the first level, referred to as `level0`, consists of multiple base models, and the second level called `level1`, uses a logistic regression model as the meta learner.

In `level0`, we deployed several prediction algorithms: (1) Logistic Regression with a maximum iteration limit set to 150, (2) Linear Discriminant Analysis, (3) k-Nearest Neighbors Classifier, (4) Decision Tree Classifier, (5) Support Vector Machine with probability estimates enabled, and a Gaussian Naïve Bayes. Each of these models was chosen for its unique approach to classification, aiming to capture different patterns in the data, which can be linear or non-linear. Also, these are among the models we used for experimenting with single models, thus enabling us to evaluate whether the joint use of single models in a stacking approach performs better than their execution separately. The prediction of these base models is used as input for the meta learner at `level1`.

The Logistic Regression model learner is a straightforward choice for combining the predictions from the base models due to its efficiency in handling linear combinations of features and its outputs, which provides a calibrated probability of class membership. It acts as a final decision maker that weights the inputs from various base models to improve prediction accuracy beyond what any base model could achieve.

The entire ensemble is cross-validated using a 10-fold strategy to ensure the model generalizes well over different subsets of the data. This method provides robust validation through extensive training and testing across multiple data splits. It leverages the diversity of the base classifiers to achieve a more reliable and potentially more accurate final prediction.

The consolidated performance metrics obtained from the cross-validation of EP-BAST$_{\text{SC}}$ across different projects are summarized in Table 5.4.

VC Model

In EP-BAST$_{\text{VC}}$, we configured EP-BAST with the prediction model $M \leftarrow$ VC. For the VC

Metric	commons-codec	commons-compress	commons-lang	joda-time
Precision	0.9049	0.9478	0.8048	0.9050
Recall	0.9105	0.9530	0.8070	0.9055
Accuracy	0.9105	0.9530	0.8070	0.9055
F1 Score	0.9005	0.9455	0.8047	0.9026
ROC AUC	0.8712	0.9004	0.8716	0.9095
Log Loss	-0.2565	-0.1489	-0.4271	-0.2798

Table 5.4: Summary of Cross-Validation Results Across Projects for the SC Model

model, we used the VotingClassifier from the scikit-learn library, designed to combine several individual classifiers' predictions to enhance the overall predictive performance, particularly for complex classification tasks.

Our VC model integrates six different types of classifiers: Logistic Regression, Linear Discriminant Analysis, K-Neighbors Classifier, Decision Tree Classifier, Support Vector Classifier (with probability estimates enabled), and Gaussian Naïve Bayes. Each classifier brings a unique approach to learning from the data, ranging from linear models like Logistic Regression and Linear Discriminant Analysis, which assume linear boundaries between classes, to more flexible models like Decision Trees and K-Neighbors that do not assume such linearities.

The `VotingClassifier` is set to use `soft` voting, which means that it predicts the class label based on the $argmax$ of the sums of the predicted probabilities provided by each component classifier. This approach is typically more flexible and powerful compared to `hard` voting, which merely counts the votes of each classifier in the ensemble for deciding on the class label. By aggregating the probabilistic estimates of each classifier, the ensemble can achieve a higher level of confidence in its predictions, leveraging the strengths of each underlying model.

The consolidated performance metrics obtained from the cross-validation of EP-BAST$_{SC}$ across different projects are summarized in Table 5.5.

Metric	commons-codec	commons-compress	commons-lang	joda-time
Precision	0.8800	0.9198	0.7858	0.8683
Recall	0.8666	0.9342	0.7878	0.8645
Accuracy	0.8666	0.9342	0.7878	0.8645
F1 Score	0.8149	0.9047	0.7861	0.8541
ROC AUC	0.8618	0.8956	0.8481	0.8742
Log Loss	-0.3191	-0.1716	-0.5235	-0.3711

Table 5.5: Summary of Cross-Validation Results Across Projects for the VC Model

5.3 Time Budget Allocation in the DynaMOSA Benchmark

We executed a series of experiments using DynaMOSA, which we enhanced with single bug prediction models to assess their efficacy in guiding Search-Based Software Testing (SBST)

approaches. Originally, DynaMOSA allocates a uniform predefined total time budget among all classes, for which it generated tests. This allocation does not consider the individual bug probability associated with each class.

For our research, we modify DynaMOSA to adapt its time budget allocation strategy based on the defect scores provided by single bug prediction models. This modification allows DynaMOSA to allocate more testing time to classes predicted to have a higher likelihood of defects, thus prioritizing potentially more problematic areas of the code.

Using DynaMOSA as a benchmark, our objective is to evaluate whether ensemble models outperform single models in improving SBST outcomes. By comparing DynaMOSA's performance when guided by single models against EPMOSA's performance with ensemble models, we aim to determine which approach provides a more effective strategy for allocating testing resources and identifying defects. This comparison is crucial as it will help establish whether integrating more complex predictive models into the SBST framework can significantly enhance bug detection capabilities within software systems.

In the following, we detail how we implemented and created each single model and the corresponding performance metrics.

5.3.1 Defect Predictors

CART Model

In DynaMOSA$_{\text{CART}}$, we configured DynaMOSA with the prediction model $M \leftarrow$ CART. We used the `DecisionTreeClassifier` algorithm from *scikit-learn* library. This classifier was configured with specific parameters to tailor its behavior during the training process. The `max_depth` parameter is set to 5, which limits the tree to a maximum of five levels. This restriction helps prevent the model from becoming overly complex and overfitting the data, where it performs well on training data but poorly on unseen data.

The `min_samples_split` parameter is set to 10, meaning that a node will only be split if it contains more than ten samples. This setting further helps avoid overfitting by ensuring that the splits are meaningful and based on sufficient samples, thereby providing more statistical significance.

Lastly, the criterion used is `entropy`, which is a way to measure the quality of a split. Entropy is a concept from information theory that measures the impurity or randomness in the data set. In the context of decision trees in the CART algorithm, it determines how a node should be split so that the classes in the resulting nodes are as pure as possible. The decision tree tries to maximize information gain by using entropy, effectively reducing uncertainty after each split.

These parameters together define a robust decision tree in handling different datasets by balancing underfitting and overfitting.

The consolidated performance metrics obtained from the cross-validation of CART across different projects are summarized in Table 5.6.

KNN Model

Metric	commons-codec	commons-compress	commons-lang	joda-time
Precision	0.8532	0.9341	0.7395	0.8843
Recall	0.8645	0.9422	0.7451	0.8826
Accuracy	0.8645	0.9422	0.7451	0.8826
F1 Score	0.8200	0.9261	0.7372	0.8759
ROC AUC	0.8304	0.8663	0.7846	0.8786
Log Loss	-0.3889	-0.2231	-0.5858	-0.3684

Table 5.6: Summary of Cross-Validation Results Across Projects for the CART Model

In DynaMOSA$_{\text{KNN}}$, we configured DynaMOSA with the prediction model $M \leftarrow$ KNN. We used the `KNeighborsClassifier` algorithm from *scikit-learn* library. The `n_neighbors` parameter is set to 5 for this classifier. This specifies that the classifier should consider the five nearest neighbors in the feature space when making a classification decision. This choice is based on the heuristic that similar instances tend to be clustered closely in the feature space, making the proximity of data points a strong predictor of their class labels.

The weights parameter is set to `distance`, which instructs the classifier to weigh the influence of each neighbor according to the inverse of their distance to the query point. This weighting approach prioritizes closer neighbors over those further away, assuming closer points are more likely to share a class with the query point. Such a scheme enhances the classifier's sensitivity to the local data structure, potentially improving accuracy by emphasizing the most relevant instances for each classification task.

Lastly, the algorithm parameter is set to `kd_tree`. The choice of a k-d tree algorithm allows the classifier to efficiently organize and query the data structure for the nearest neighbors. A k-d tree is a space-partitioning data structure, optimal for scenarios where the dimensionality and the dataset size allow for quick splitting of the space into nested hyperrectangles, each containing a portion of the dataset. This method significantly speeds up the search process for the nearest neighbors, particularly in medium-sized datasets where the balance between the number of samples and the number of features does not overly complicate the geometry of the data space.

These settings collectively enhance the `KNeighborsClassifier's` ability to perform classification tasks efficiently and effectively by focusing on both the proximity of data points and the computational aspects of searching in the feature space. This setup is especially suited to applications where the prediction accuracy is critical and dependent on the nuanced understanding of the local distributions of data points within the feature space.

The consolidated performance metrics obtained from the cross-validation of KNN across different projects are summarized in Table 5.7.

LDA Model

In DynaMOSA$_{\text{LDA}}$, we configured DynaMOSA with the prediction model $M \leftarrow$ LDA. We used the `LinearDiscriminantAnalysis` algorithm from *scikit-learn* library. LDA is a classifier with a linear decision boundary generated by fitting class conditional densities to the data and using Bayes' rule. It can also be used as a technique for dimensionality reduction.

Metric	commons-codec	commons-compress	commons-lang	joda-time
Precision	0.8972	0.9416	0.7570	0.8629
Recall	0.9029	0.9470	0.7587	0.8661
Accuracy	0.9029	0.9470	0.7587	0.8661
F1 Score	0.8983	0.9432	0.7575	0.8618
ROC AUC	0.8278	0.8033	0.8179	0.8590
Log Loss	-1.5275	-0.9471	-2.1174	-2.0223

Table 5.7: Summary of Cross-Validation Results Across Projects for the KNN Model

The `solver` parameter is set to `lsqr`, which stands for least squares solution. This solver is chosen when it is needed to compute the solution to a linear system. It is effective for calculating discriminant functions when the number of samples exceeds the number of features, especially in cases where shrinkage is applied.

The `shrinkage` parameter is set to `auto`. `Shrinkage` is a regularization technique used to improve the estimation of covariance matrices in cases where the number of training samples is small compared to the number of features. By setting it to `auto`, the method automatically determines the optimal amount of shrinkage, balancing the trade-off between bias and variance in the model. This is particularly useful in scenarios prone to overfitting, enhancing the model's ability to generalize from the training data to unseen data.

The consolidated performance metrics obtained from the cross-validation of LDA across different projects are summarized in Table 5.8.

Metric	commons-codec	commons-compress	commons-lang	joda-time
Precision	0.8755	0.8912	0.6043	0.8041
Recall	0.8543	0.9314	0.6412	0.8080
Accuracy	0.8543	0.9314	0.6412	0.8080
F1 Score	0.7871	0.9011	0.5225	0.7871
ROC AUC	0.6245	0.7787	0.6301	0.8245
Log Loss	-0.3982	-0.2215	-0.6230	-0.4313

Table 5.8: Summary of Cross-Validation Results Across Projects for the LDA Model

LR Model

In DynaMOSA$_{\text{LR}}$, we configured DynaMOSA with the prediction model $M \leftarrow$ LR. We used the `LogisticRegression` algorithm from *scikit-learn* library. Logistic Regression is a probabilistic, linear classifier well-suited for binary classification tasks. However, it can be extended to handle multi-class problems through techniques such as one-vs-rest (OvR) schemes. The model is fundamentally designed to estimate probabilities using a logistic function, an S-shaped curve that maps any real-valued number into the interval (0, 1), making it apt for probability estimation.

The `LogisticRegression` classifier was configured with specific parameters to ensure robustness and reproducibility of the results. The `max_iter` parameter was set to 1000

iterations. This configuration parameter specifies the maximum number of iterations for the solver to converge on the optimal coefficients. Setting a higher value for `max_iter` is crucial in complex scenarios where the decision surface is difficult to approximate, necessitating additional iterations to reach convergence.

Additionally, the `random_state` parameter was fixed at 42, a choice that ensures consistency and reproducibility in the results. This parameter acts as a seed to the pseudo-random number generator used by the logistic regression model to randomly initialize parameters. By setting this seed, the initial conditions for parameter estimation are standardized, thereby eliminating variability in the model outcomes attributable to stochastic processes during initialization. This is particularly essential in research contexts to facilitate the replication of the study by other researchers and to verify the findings reliably.

The consolidated performance metrics obtained from the cross-validation of LR across different projects are summarized in Table 5.9.

Metric	commons-codec	commons-compress	commons-lang	joda-time
Precision	0.8755	0.8910	0.7431	0.7615
Recall	0.8543	0.9314	0.6383	0.7768
Accuracy	0.8543	0.9314	0.6383	0.7768
F1 Score	0.7871	0.9005	0.5001	0.7608
ROC AUC	0.6153	0.7970	0.5518	0.6500
Log Loss	-0.4396	-0.2348	-0.6390	-0.6047

Table 5.9: Summary of Cross-Validation Results Across Projects for the RF Model

NB Model

In DynaMOSA$_{\text{NB}}$, we configured DynaMOSA with the prediction model $M \leftarrow$ NB. We used the `GaussianNB` algorithm from *scikit-learn* library. The GaussianNB algorithm is predicated on the Naïve Bayes principle, which operates under the assumption of conditional independence among features given the class label. This assumption, despite its simplicity, often leads to effective classification performance.

The `GaussianNB` implementation assumes explicitly that the likelihood of the features is *Gaussian* (normally distributed). When applied, the algorithm computes the *mean* and *variance* of each feature for each class, forming a part of the Gaussian distribution's parameters during the probability estimation phase. This method is particularly well-suited for continuous or nearly continuous feature values. It is commonly applied in various domains, including text classification and medical diagnosis, where feature distributions naturally align with Gaussian distributions.

The `GaussianNB` classifier was instantiated with all default parameters in our experiments. The default behavior of `GaussianNB` does not require the specification of parameters before model fitting, apart from the prior probabilities of the classes, which are, by default, assumed to be uniform if not explicitly provided. This default setting is particularly advantageous in experimental settings for several reasons:

- **Simplicity and Accessibility:** The lack of necessity to fine-tune parameters makes `GaussianNB` an accessible choice for baseline experiments, providing a straightforward implementation that can be readily understood and used by practitioners and researchers alike.
- **Empirical Robustness:** `GaussianNB` tends to work well out of the box without extensive calibration, especially in cases where the Gaussian assumption holds reasonably well. We considered this robustness suitable for our data characteristics, simplifying the model setup without compromising the integrity of the experimental results.
- **Reproducibility:** Employing the `GaussianNB` classifier with default parameters enhances the reproducibility of the research. Other researchers can easily replicate the study without adjusting for hyperparameter variability, thus directly comparing their results with ours under consistent conditions.

The consolidated performance metrics obtained from the cross-validation of NB across different projects are summarized in Table 5.10.

Metric	commons-codec	commons-compress	commons-lang	joda-time
Precision	0.7435	0.8895	0.6524	0.7363
Recall	0.8293	0.9186	0.4954	0.7578
Accuracy	0.8293	0.9186	0.4954	0.7578
F1 Score	0.7779	0.9015	0.4684	0.7304
ROC AUC	0.6146	0.7319	0.6406	0.7345
Log Loss	-0.5854	-0.4459	-1.3279	-0.8997

Table 5.10: Summary of Cross-Validation Results Across Projects for the NB Model

SVM Model

In DynaMOSA$_{\text{SVM}}$, we configured DynaMOSA with the prediction model $M \leftarrow$ SVM. We used the `SVC` algorithm from *scikit-learn* library. The Support Vector Machine (SVM) classifier is known for creating a maximal margin of separation between classes, which is particularly advantageous in high-dimensional spaces, where the separation surface can be defined with great precision. This classifier effectively handles linear and non-linearly separable data through the kernel trick, although the default setting employs a linear kernel.

The SVC was configured in our experiments with the parameter `probability=True`. This configuration is not typical of standard SVM implementations, which do not compute probabilistic estimates for class memberships by default due to the computational expense involved. Enabling the probability estimates involves internal cross-validation and using `Platt scaling` – a method of fitting a logistic regression model to the SVM's scores – calibrated to approximate class probabilities.

The choice of employing all default parameters, apart from enabling probabilities, was informed by the desire to maintain the classifier's intrinsic characteristics, such as the use of the

Radial Basis Function (RBF) kernel, which is the default in scikit-learn's SVC. The RBF kernel is versatile and effective for a wide range of classification tasks, making it a reliable choice when initial explorations do not strongly favor an alternative kernel.

The default setting of the penalty parameter `C` at 1.0 balances model complexity and the degree of *misclassifications* tolerated in the classifier's decision surface. This default value provides a starting point that is generally effective across various datasets and is particularly useful in exploratory phases of research where the primary goal is to establish a baseline understanding before engaging in parameter tuning.

The consolidated performance metrics obtained from the cross-validation of NB across different projects are summarized in Table 5.11.

Metric	commons-codec	commons-compress	commons-lang	joda-time
Precision	0.8755	0.9372	0.7695	0.7918
Recall	0.8543	0.9327	0.6398	0.7966
Accuracy	0.8543	0.9327	0.6398	0.7966
F1 Score	0.7871	0.9002	0.4993	0.7707
ROC AUC	0.4641	0.4967	0.6170	0.7559
Log Loss	-0.4155	-0.2467	-0.6293	-0.4750

Table 5.11: Summary of Cross-Validation Results Across Projects for the SVM Model

5.4 Final Considerations

n this chapter, we introduced the concept of time budget allocation in the context of Search-Based Software Testing (SBST) and presented the Ensemble Predictive Budget Allocation Strategy for Testing (EP-BAST). By formulating the time budget optimization problem and integrating ensemble prediction models, EP-BAST aims to enhance the efficiency and effectiveness of the testing process. The strategy intelligently allocates testing resources to code areas most likely to contain bugs, thereby optimizing the use of available time and improving bug detection rates. The development of EP-BAST marks a significant advancement in addressing the challenges of traditional SBST approaches, particularly in resource-constrained environments. The insights gained from this chapter provide a solid foundation for the empirical evaluation of EP-BAST and its implementation within the Ensemble Predictive Multi-Objective Sorting Algorithm (EP-MOSA). The subsequent chapter will detail the application and performance of these strategies, demonstrating their practical benefits in real-world software testing scenarios.

CHAPTER

6

Guiding Test Case Generation with Ensemble Predictive Models

This chapter presents the empirical evaluation of the Ensemble Predictive Multi-Objective Sorting Algorithm (EPMOSA), a novel approach designed to enhance Search-Based Software Testing (SBST) by guiding test case generation using ensemble predictive models. Building on the time budget allocation strategy introduced in the previous chapter, EPMOSA dynamically allocates testing efforts based on the predicted likelihood of defects. The chapter outlines the experimental settings, including the selection of subject systems, the configuration of predictive models, and the metrics used for evaluation. By comparing the performance of EPMOSA against traditional SBST approaches and single-model prediction strategies, we aim to demonstrate the effectiveness of ensemble models in improving the bug detection capabilities of automatically generated test suites. The results and analysis provided in this chapter offer valuable insights into the practical benefits and limitations of using ensemble prediction models to guide test case generation.

6.1 Ensemble Predictive Multi-Objective Sorting Algorithm - EPMOSA

We propose EPMOSA, an Ensemble Predictive Multi-Objective Sorting Algorithm, which enhances the well-established DynaMOSA algorithm by incorporating ensemble-based predictive capabilities through EP-BAST (Ensemble Prediction-Based Adaptive Software Testing). $\text{EPMOSA}_{\text{EP-BAST}}$ is specifically designed to optimize the generation of test suites by dynamically allocating resources based on predictions from an ensemble of defect prediction models.

EPMOSA is implemented within EVOSUITE, an automated software testing tool for Java programs. This implementation extends DynaMOSA, which was originally designed to prioritize uncovered goals for efficient test suite generation. EPMOSA goes further by integrating bug prediction models to forecast areas of the codebase most susceptible to defects, enhancing the

overall process.

Central to EPMOSA is the EP-BAST strategy, which allocates time budgets just before a class is selected for testing. This allocation is based on the ensemble-predicted probabilities of defectiveness, ensuring that testing efforts focus on high-risk areas, thereby improving efficiency and effectiveness.

By leveraging EP-BAST's capabilities, EPMOSA addresses multiple objectives in test suite generation, such as maximizing code coverage and fault detection while optimizing computational resource use. The predictive component of EP-BAST allows EPMOSA to allocate more time to classes with higher defect probabilities, increasing the likelihood of uncovering defects and significantly enhancing resource utilization.

PreMOSA (Perera et al., 2023a) excels at integrating defect prediction data with coverage information to prioritize likely buggy parts but relies on theoretical defect prediction models with a minimum precision of 75%. However, these theoretical models may not accurately reflect real-world software environments, leading to misallocation of testing resources and potential oversight of severe defects.

In this dissertation, we argue that no single defect prediction model is sufficient for automatic test case generation. We propose EPMOSA, a Multi-Objective Sorting Algorithm informed by ensemble bug prediction models. Unlike PreMOSA, which uses theoretical defect predictors, EPMOSA leverages multiple real defect prediction models, combining their strengths to improve accuracy and reliability.

The ensemble approach of EPMOSA mitigates the risks associated with relying on a single model, providing a more robust, adaptable, and context-sensitive tool for guiding test case generation.

EPMOSA enhances DynaMOSA's capabilities by focusing on components with a higher likelihood of defects, optimizing testing resources, and improving fault detection rates. This strategic focus is critical for maintaining high software quality.

The integration of EPMOSA with DynaMOSA involves several key modifications. Ensemble models provide a defect probability score for each class, which adjusts the priority of testing objectives during execution. This allows EPMOSA to adapt its testing focus based on evolving risk assessments.

Overall, by leveraging DynaMOSA and enhancing it with ensemble bug prediction models, EPMOSA represents a significant advancement in automatic test case generation. To our knowledge, it is the first effort to construct and use real ensemble bug prediction models to guide a test suite generation process.

6.2 Empirical Evaluation Strategy

The empirical evaluation consisted of three rounds of experiments structured as follows:

- **Round 1:** Testing $\text{EPMOSA}_{\text{EP-BAST}}$ on each subject system.
- **Round 2:** Testing DynaMOSA, enhanced with a single-model bug prediction, on each subject system.

- **Round 3:** Testing the standard implementation of DynaMOSA on each subject system.

In Round 1, we run experiments with five different configurations for $\text{EPMOSA}_{\text{EP-BAST}}$. Each configuration consists of changing the EP-BAST prediction model, directly impacting the time budget allocation. The EP-BAST was configured with the following bug prediction models:

1. Bagging Meta Estimator (BME)
2. Gradient Boosting Trees (GBT)
3. Random Forest (RF)
4. Stacking Classifier (SC)
5. Voting Classifier (VC)

Table 6.1 summarizes the experiments conducted in the first round. Each experiment consisted of five phases. In phase 1, the EP-BAST algorithm, configured with the BME model, was deployed. Specifically, $\text{EPMOSA}_{\text{EP-BAST}_{(\text{BME})}}$ was executed on all buggy versions of each subject system. For instance, for the buggy version #1 of the Commons-Lang project, we executed $\text{EPMOSA}_{\text{EP-BAST}_{(\text{BME})}}$ five times. Given that this project has 64 active buggy versions, $\text{EPMOSA}_{\text{EP-BAST}_{(\text{BME})}}$ was run a total of 320 times across all versions, resulting in the generation of 320 test suites.

In phase 2, the EP-BAST algorithm was configured with the GBT model, leading to the execution of $\text{EPMOSA}_{\text{EP-BAST}_{(\text{GBT})}}$ following the identical procedure used for $\text{EPMOSA}_{\text{EP-BAST}_{(\text{BME})}}$. Phases 3, 4, and 5 followed the same execution steps but with the EP-BAST algorithm configured with the RF, SC, and VC prediction models, respectively. Each phase adhered to the established protocol to ensure consistency across different configurations.

In the first round, 155 buggy versions were considered across all subject systems, and 3100 test suites were automatically generated.

Table 6.2 summarizes the experiments conducted in the second round. As in the first round, the experiments were conducted in phases. In the second round, we divided the experiments into six phases. In phase 1, we modified the DynaMOSA algorithm to behave as EPMOSA, but we accepted time budgets provided directly from single models. Specifically, $\text{DynaMOSA}_{+\text{CART}}$ was informed with time budgets generated by a CART bug prediction model and executed on all buggy versions of each subject system as in our experiments with EPMOSA. The same procedure was repeated for phases 2, 3, 4, 5, and 6 with KNN, LDA, LR, NB, and SVM, respectively. In phase 7, we repeat the experiments for each subject system using the standard DynaMOSA to generate the test suites. The results from this last phase serve as a baseline for comparing EPMOSA with a state-of-the-art approach and allow the comparison of DynaMOSA with its variants with single-model enhancement.

Table 6.3 summarizes the third round of experiments. In this round, we ran the standard DynaMOSA implementation for each buggy version across all subject systems. This process generated 775 test suites for a total of 155 bug versions.

Configuration	Subject System	Bug Versions	Runs/Test Suites Generated
$EPMOSA_{EP\text{-}BAST_{(GBT)}}$	Commons-Lang	64	320
	Commons-Codec	18	90
	Joda-Time	26	130
	Commons-Compress	47	235
$EPMOSA_{EP\text{-}BAST_{(RF)}}$	Commons-Lang	64	320
	Commons-Codec	18	90
	Joda-Time	26	130
	Commons-Compress	47	235
$EPMOSA_{EP\text{-}BAST_{(SC)}}$	Commons-Lang	64	320
	Commons-Codec	18	90
	Joda-Time	26	130
	Commons-Compress	47	235
$EPMOSA_{EP\text{-}BAST_{(VC)}}$	Commons-Lang	64	320
	Commons-Codec	18	90
	Joda-Time	26	130
	Commons-Compress	47	235
Total		**155/Configuration**	**3100**

Table 6.1: Overview of $EPMOSA_{EP\text{-}BAST}$ configurations and their respective test outcomes across the subject systems for the first round of experiments.

Configuration	Subject System	Bug Versions	Runs/Test Suites Generated
$DynaMOSA_{+CART}$	Commons-Lang	64	320
	Commons-Codec	18	90
	Joda-Time	26	130
	Commons-Compress	47	235
$DynaMOSA_{+KNN}$	Commons-Lang	64	320
	Commons-Codec	18	90
	Joda-Time	26	130
	Commons-Compress	47	235
$DynaMOSA_{+LDA}$	Commons-Lang	64	320
	Commons-Codec	18	90
	Joda-Time	26	130
	Commons-Compress	47	235
$DynaMOSA_{+LR}$	Commons-Lang	64	320
	Commons-Codec	18	90
	Joda-Time	26	130
	Commons-Compress	47	235
$DynaMOSA_{+NB}$	Commons-Lang	64	320
	Commons-Codec	18	90
	Joda-Time	26	130
	Commons-Compress	47	235
$DynaMOSA_{+SVM}$	Commons-Lang	64	320
	Commons-Codec	18	90
	Joda-Time	26	130
	Commons-Compress	47	235
Total		**155/Configuration**	**4650**

Table 6.2: Overview of DynaMOSA configurations (enhanced with single model bug prediction approaches) and their respective test outcomes across the subject systems for the second round of experiments.

Configuration	Subject System	Bug Versions	Runs/Test Suites Generated
$\text{DynaMOSA}_{\text{standard}}$	Commons-Lang	64	320
	Commons-Codec	18	90
	Joda-Time	26	130
	Commons-Compress	47	235
Total		**155/Configuration**	**775**

Table 6.3: Overview of DynaMOSA standard configuration and its test outcomes across the subject systems for the third round of experiments.

Our empirical evaluation strategy of segregating the experiments in three different rounds provided a controlled and systematic way of using $\text{EPMOSA}_{\text{EP-BAST}}$ and DynaMOSA for generating test suites encompassing all eleven bug prediction models selected for this study. Furthermore, this strategy helped analyze and compare how effective EPMOSA and DynaMOSA are and ensemble-model and single-model strategies to guide SBST test generation approaches.

The EP-BAST module was configured to use an exponential formula to define the time budgets based on the defect score of each class. This way,, EP-BAST can allocate time budgets that are more sensitive for buggy classes. The time budget can grow rapidly as the defect score increases, depending on the parameters ea, eb, and ec. These parameters shape the curve of the exponential function, which affects how the time budget increases as the likelihood of defects increases.

6.2.1 Experimental Settings

In this section, we outline the configurations, including parameter choices and specific experimental conditions, for each round of experiments. Each experiment was planned to control for potential variables that could impact the results. Ensuring consistency in our experimental setup allowed for an accurate comparison of the effects of different bug prediction models on the generated test suites, facilitating the drawing of meaningful conclusions from our analysis.

6.2.1.1 Time Budget Considerations

The EP-BAST strategy is a module of EPMOSA that calculates the time budget of each class being tested before the test generation process. The EP-BAST generates a file that lists all classes in the system under test, followed by their time budget.

After EP-BAST generates the file with classes and their time budgets, EPMOSA reads it during the test generation task. For each class, EPMOSA takes the time budget determined by EP-BAST and uses it as the stop criteria for its search of inputs and test case generation.

For our experiments and exploration of bug prediction guided SBST approaches, we do not set a time limit for the entire test generation process, instead the time budgets are determine by class during the generation process. We evaluate the efficiency of each studied approach in terms of the time taken to generate the whole test suite.

More details about how EP-BAST was implemented can be found in Chapter5.

6.2.1.2 Baseline Comparisons

The Defect4J offers a framework for test execution in each buggy version of the subject projects. Each subject project is accompanied by a default test suite with unit tests created manually by developers. These developer unit tests reveal the bugs existing in each version. These test suites are used as a baseline in executing the automatically generated test suites. When we execute an automatically generated test suite, we verify whether it fails for the same methods the developer-created unit tests fail. If the automatically generated test suites fail, it can detect the active buggy.

6.2.1.3 Prototype Implementation

We integrated the EPMOSA algorithm into the EvoSuite tool, which is widely recognized in search-based software testing (SBST). EvoSuite is an advanced framework that was created to generate JUnit test suites for Java projects automatically. This tool is notable for its ongoing maintenance and its shown efficacy in a diverse array of tasks. We chose EvoSuite version 1.1.0, obtained from the utility's GitHub repository in October 2020. The EvoSuite version with an EPMOSA prototype can be accessed at this location: https://anonymous.4open.science/r/EPMOSA.

6.3 Results

In this section, we present and discuss the results from our experiments, which are structured around the utilization of exponential formulas for time budget allocation in EP-BAST, as applied in different setups for EPMOSA in addition to modified versions of DynaMOSA with single-model predictors. We use $\text{EPMOSA}_{\text{EP-BAST}}$ to indicate all different setups of EPMOSA where EP-BAST uses different ensemble models to define time budget allocation values.

Our analysis explores the efficiency and efficacy of the proposed EPMOSA approach compared with each variation of DynaMOSA guided by single models.

EPMOSA Efficacy

> ***RQ1**: Does EPMOSA outperform other approaches in generating more effective test suites?*

This subsection evaluates the effectiveness of EPMOSA compared to the established DynaMOSA approach by analyzing their bug identification rates across various subject projects. The objective is to assess whether $\text{EPMOSA}_{\text{EP-BAST}}$, with its strategic use of EP-BAST to budget allocation, demonstrates superior bug identification capabilities compared to state-of-the-art approaches.

6.3.0.1 Commons-Lang

The Commons-Lang project comprises 64 buggy versions. Three experimental rounds were executed for each buggy version as detailed in Table 6.4. Each configuration of EPMOSA and DynaMOSA was tested in 64 separate experiments – one for each buggy version. Each

Round	Experiments	Configuration	Runs	Test Suites
1	64	$\text{EPMOSA}_{\text{EP-BAST}_{(\text{GBT})}}$	5	320
1	64	$\text{EPMOSA}_{\text{EP-BAST}_{(\text{RF})}}$	5	320
1	64	$\text{EPMOSA}_{\text{EP-BAST}_{(\text{SC})}}$	5	320
1	64	$\text{EPMOSA}_{\text{EP-BAST}_{(\text{VC})}}$	5	320
2	64	$\text{DynaMOSA}_{+\text{CART}}$	5	320
2	64	$\text{DynaMOSA}_{+\text{KNN}}$	5	320
2	64	$\text{DynaMOSA}_{+\text{LDA}}$	5	320
2	64	$\text{DynaMOSA}_{+\text{LR}}$	5	320
2	64	$\text{DynaMOSA}_{+\text{NB}}$	5	320
2	64	$\text{DynaMOSA}_{+\text{SVM}}$	5	320
3	64	$\text{DynaMOSA}_{\text{Standard}}$	5	320
Total				3520

Table 6.4: Detailed overview of the rounds of experiments for the Commons-Lang project

experiment was repeated five times to account for the stochastic nature of the algorithms used in test case generation. As a result, for each configuration, a total of 320 test suites were generated. This approach resulted in creating 3520 test suites for the Commons-Lang project.

Table 6.5 shows a detailed overview of the bug identification rates from our experiments, categorizing each set of results by the configuration variation based on the approach and bug prediction model employed. These bug identification rates measure the effectiveness of the test suites generated by EPMOSA and DynaMOSA, guided by various defect prediction models.

The data in Table 6.5 is meticulously organized to allow for straightforward comparison across different approach configurations, encompassing single and ensemble models. This organization facilitates a comprehensive evaluation of the relative performance of each model in identifying bugs within the Commons-Lang project. Additionally, Figure 6.1 visually depicts the bug identification rates, enhancing understanding by clearly illustrating the comparative effectiveness of each configuration.

In Table 6.5, the bug identification rates for DynaMOSA's configurations vary between 3.69% and 4.92%, with an average effectiveness rate of 4.21%. This suggests a moderate level of effectiveness. Conversely, EPMOSA's configurations display a significantly higher effectiveness range, with rates ranging from 4.62% to 7.08%. The standard deviation among the DynaMOSA's configurations is relatively low at 0.50%, indicating minimal bug identification rate variability. This consistency demonstrates that DynaMOSA, when employing single-model defect predictors such as CART, KNN, LDA, LR, NB, and SVM, achieves steady performance across various models, suggesting a robust approach to bug detection within these parameters.

The minimal range between the lowest and highest bug identification rates further highlights the consistency across single-model predictors, as evidenced by the proximity of the mean bug identification rate at 4.21% and the median at 4.15%. This tight alignment suggests a symmetric distribution of identification rates among the configurations with single models, indicating the absence of outliers and that most models demonstrate similar levels of effectiveness. Given the low variability and the symmetrical nature of the distribution, selecting a particular configuration of DynaMOSA might be influenced by factors beyond mere performance, such as operational

efficiency, ease of integration, and the specific characteristics of the bugs targeted. The relatively consistent performance across single models ensures each remains viable for the Commons-Lang project.

In contrast, all configurations of EPMOSA typically exhibit higher bug identification rates, ranging from 4.62% for the configuration $\text{EPMOSA}_{\text{EP-BAST}_{(\text{GBT})}}$ to 7.08% for the configuration $\text{EPMOSA}_{\text{EP-BAST}_{(\text{SC})}}$. This variation underscores that integrating multiple prediction models in an ensemble to guide an SBST approach can substantially improve the effectiveness of the generated test suites. Unlike DynaMOSA, which single models guide, EPMOSA shows a broader effectiveness range, suggesting variability that depends on the specific ensemble model employed. Notably, the highest rate achieved by EPMOSA, at 7.08% for $\text{EPMOSA}_{\text{EP-BAST}_{(\text{SC})}}$, significantly exceeds the top rate observed in experiments with DynaMOSA guided by single models. This points to the potential advantages of ensemble models in enhancing the robustness and accuracy of bug detection in software testing.

The variability in effectiveness observed for the EPMOSA configurations underscores the importance of carefully selecting or constructing ensemble models for the SBST process. Optimizing the design of these ensemble models can significantly enhance their ability to help the SBST approach generate test suites with higher bug identification rates.

Table 6.5: Commons-Lang: Bug Identification Rates

DynaMOSA with single-models	**Bug Identification Rate**
$\text{DynaMOSA}_{\text{+CART}}$	4.62%
$\text{DynaMOSA}_{\text{+KNN}}$	4.31%
$\text{DynaMOSA}_{\text{+LDA}}$	4.92%
$\text{DynaMOSA}_{\text{+LR}}$	3.69%
$\text{DynaMOSA}_{\text{+NB}}$	4.00%
$\text{DynaMOSA}_{\text{+SVM}}$	3.69%
EPMOSA with ensemble-models	**Bug Identification Rate**
$\text{EPMOSA}_{\text{EP-BAST}_{(\text{BME})}}$	5.85%
$\text{EPMOSA}_{\text{EP-BAST}_{(\text{GBT})}}$	4.62%
$\text{EPMOSA}_{\text{EP-BAST}_{(\text{RF})}}$	6.15%
$\text{EPMOSA}_{\text{EP-BAST}_{(\text{SC})}}$	7.08%
$\text{EPMOSA}_{\text{EP-BAST}_{(\text{VC})}}$	6.46%
DynaMOSA default impl.	**Bug Identification Rate**
$\text{DynaMOSA}_{\text{Default}}$	3.08%

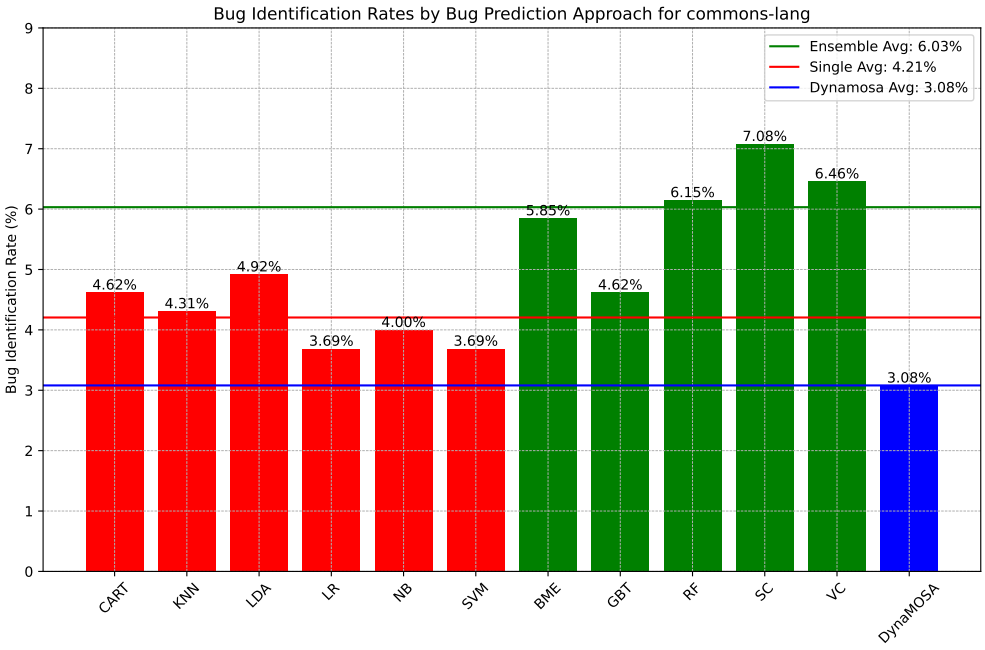

Figure 6.1: Bug identification rates by SBST according to its bug predictive approach for commons-lang

6.3.0.1.1 Result's Statistical Significance

A thorough statistical analysis of our experimental results is necessary to analyze the correlation between the class of prediction models used (ensemble or single models) to guide an SBST approach and the bug identification rate of the generated test suites. Initially, we conduct a performance distribution analysis to determine if any specific configuration of EPMOSA and DynaMOSA consistently yields superior bug detection rates.

The box plot in Figure 6.5 provides a comparison of the distribution of bug identification rates between EPMOSA and DynaMOSA's configurations. This visual representation allows for an immediate grasp of the differences in performance distribution between the two model types.

In addition, Table 6.6 summarizes the central tendency and variability of the identification rates. This data shows that the distribution of bug identification rates for single models is more compact, exhibiting less variability – a trend consistent with our earlier discussions. Conversely, there is a broader range in bug identification rates for ensemble models, with the median value notably higher than that observed for single models, as detailed in Table 6.6. It suggests ensemble models introduce more variability but achieve higher median effectiveness in bug identification, potentially offering a more robust solution in environments where maximizing bug detection is critical.

Model Class	Count	Mean	STD	Min	25%	50%	75%	Max
ensemble	5.0	6.032	0.911356	4.62	5.85	6.15	6.46	7.08
single	6.0	4.205	0.503379	3.69	3.76	4.155	4.54	4.92

Table 6.6: Summary statistics for the bug identification rate in the commons-lang project

Figure 6.2: Comparison of bug identification rates between SBST with different classes of prediction models applied to commons-Lang

These insights confirm that ensemble models generally lead SBST approaches to higher bug identification rates than single models, albeit with more significant variability. This variability suggests that by leveraging multiple modeling strategies, ensemble models potentially offer more robust guidance to SBST techniques, thereby leading to test suites with enhanced bug identification capabilities.

To further deepen our understanding, we can analyze the variance in performance within each group of configurations: (1) DynaMOSA's configurations, setup with single bug prediction models, and (2) EPMOSA's configuration, setup with ensemble prediction models. This analysis will allow us to determine whether any specific prediction model within each group consistently leads to bug detection rates at their respective ranges' higher or lower ends. By examining the performance variance, we can identify which models contribute most to the effectiveness of their respective SBST approaches and whether some models might consistently underperform or excel. This detailed examination helps refine the selection of prediction models for future SBST implementations, ensuring that the chosen models align closely with the desired outcomes of higher bug detection rates and optimal test suite effectiveness.

In Figure 6.3, we plot the histogram and density plot for each configuration group. It is possible to observe that the histogram shows a relatively even distribution with a slight

concentration around the lower and upper ends for configurations with single models. The density plot suggests a bimodal distribution, where the bug identification rates cluster around the lowest (3.69%) and the highest (4.92%) values.

For the configurations with ensemble models, the histogram in Figure 6.3 shows a more spread-out distribution, leaning towards higher identification rates. The density plot indicates a right-skewed distribution, with most models performing better than the single-model average and peaking around 6%.

With these results, we can conclude that SBST approaches guided by single models can exhibit less variability and have bug identification rates closely packed around the mean. This might suggest that single models like CART, KNN, and others used in our experiments offer similar performance when used to guide the test generation in SBST approaches. In our case, all the DynaMOSA's configurations showed similar results in terms of bug identification rate.

EPMOSA's configurations showed a broader range of bug identification rates, with higher peaks suggesting that combining different bug prediction models in an ensemble model can improve bug identification rates, though with increased variability.

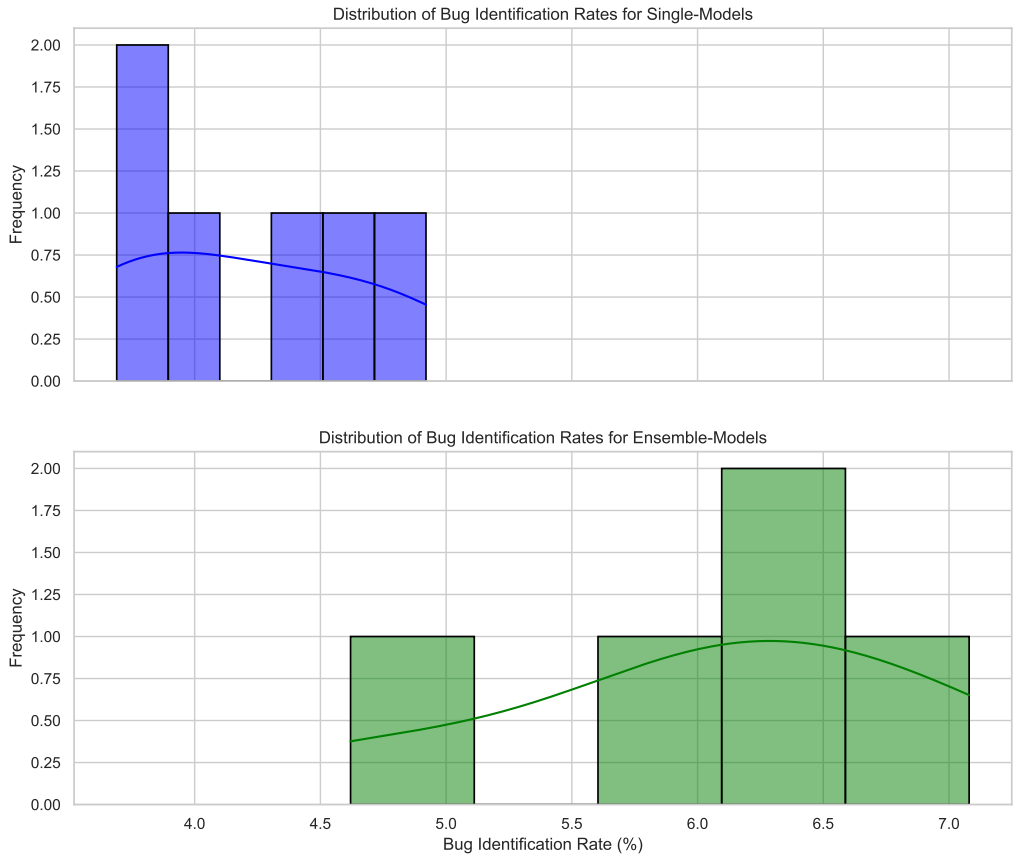

Figure 6.3: Distribution of bug identification rates

To further investigate the relationship between the bug prediction models and the bug identification rates of the test suites, we propose conducting a correlation analysis utilizing two

robust statistical methods:

- **ANOVA (Analysis of Variance):** This statistical tool is employed to determine if there are statistically significant differences in bug identification rates between test suites generated with EPMOSA's configurations and DynaMOSA's configurations. ANOVA is invaluable for comparing average values across multiple groups, assisting in identifying whether observed differences in bug identification rates are statistically significant.
- **Mann-Whitney U Test:** Suitable for comparing differences between two independent groups when the data distribution is non-normal, this non-parametric test assesses whether one group tends to have higher values than the other, providing insights into the relative effectiveness of the two model types.

The application of the ANOVA test will enable us to compare the mean bug identification rates across the different configuration groups to identify any statistically significant differences. This approach is particularly beneficial as it assesses differences based on group variances, offering a clear view of the impact of the choice between single and ensemble models on the effectiveness of SBST approaches. Through this analysis, we aim to ascertain the extent to which the type of predictive model influences the bug detection capabilities of SBST-generated test suites.

The results from the ANOVA test, as shown in Table 6.7, indicate a large F-value of 17.85. This substantial F-value demonstrates that the variability in bug identification rates between the configuration groups is significantly greater than the variability within each group. Such pronounced differences highlight the considerable impact that the choice of model type has on bug identification rates. With a p-value of less than 0.05, we can confidently assert that there is a statistically significant difference in bug identification rates between test suites generated using single and ensemble models to guide the SBST approaches.

This analysis not only enhances our understanding of the bug prediction model's influence on SBST performance but also underscores the strategic importance of model selection. It reveals that choosing between single and ensemble models can markedly impact the effectiveness of SBST approaches in identifying bugs. This insight is crucial for practitioners as it aids in making informed decisions regarding selecting tools and methodologies for their software testing processes, ultimately improving testing outcomes.

Project	**F-value**	**p-value**
commons-lang	17.835	0.002

Table 6.7: ANOVA results for the bug identification rates in the commons-lang project

Complementing our ANOVA findings, the Mann-Whitney U test offers additional insights by evaluating the differences between single and ensemble models within a non-parametric framework. This test is beneficial as it does not assume a normal distribution of data, providing a robust method for comparing the median values of the two groups. This analysis helps confirm

whether the observed differences in bug detection rates are consistent across different statistical methodologies, further validating the strategic importance of model selection in enhancing SBST for test case generation performance.

Project	U-statistic	p-value
commons-lang	1.5	0.0171

Table 6.8: Mann-Whitney U results for the bug identification rates in the commons-lang project

The low U-statistic from the Mann-Whitney U Test indicates that the majority of bug identification rates for one configuration group are consistently lower than those for the other (Table 6.8), highlighting a significant difference between single and ensemble models. A p-value of less than 0.05 further supports this substantial divergence. This outcome confirms that the differences in bug identification rates between the two model groups are statistically significant, even when evaluated using a non-parametric test.

These findings from the Mann-Whitney U Test are consistent with our previous ANOVA results, revealing a clear trend: ensemble models typically enable SBST approaches to generate test suites with higher bug identification rates than those guided by single models. This pattern supports the notion that ensemble bug prediction strategies, which incorporate multiple prediction models or techniques, generally enhance the effectiveness of bug identification in SBST approaches for test case generation.

While ensemble models often provide superior performance in SBST applications, it is crucial to consider their reliability and the specific context in which they are used, particularly in our study involving the Commons-Lang project. The choice between single and ensemble models should be made based on a comprehensive understanding of their strengths and limitations within the testing environment.

6.3.0.2 Commons-Codec

The Commons-Codec project includes 18 buggy versions. Three experimental rounds were conducted for each buggy version, as detailed in Table 6.9. The EPMOSA and DynaMOSA configurations were tested in 18 experiments, one for each buggy version. Each experiment was conducted five times to account for the stochastic nature of the algorithms used in test case generation. As a result, for each approach and configuration, a total of 90 test suites were generated. This process led to the creation of 990 test suites for the Commons-Codec project.

Table 6.10 offers a comprehensive overview of the bug identification rates from our experiments with EPMOSA and DynaMOSA within the Commons-Codec project. The DynaMOSA's configurations show an average bug identification rate of 37.22%. This indicates that, on average, single models enable DynaMOSA to generate test suites that successfully identify approximately 37% of bugs, demonstrating moderate effectiveness. In contrast, the experiments utilizing EPMOSA's configurations achieved a higher effectiveness, with an average bug identification rate of 43.11%. This suggests that integrating multiple modeling techniques within ensemble models significantly enhances the effectiveness of SBST approaches.

Round	Experiments	Approach/Configuration	Runs	Test Suites
1	18	$\text{EPMOSA}_{\text{EP-BAST}_{(GBT)}}$	5	90
1	18	$\text{EPMOSA}_{\text{EP-BAST}_{(RF)}}$	5	90
1	18	$\text{EPMOSA}_{\text{EP-BAST}_{(SC)}}$	5	90
1	18	$\text{EPMOSA}_{\text{EP-BAST}_{(VC)}}$	5	90
2	18	$\text{DynaMOSA}_{+\text{CART}}$	5	90
2	18	$\text{DynaMOSA}_{+\text{KNN}}$	5	90
2	18	$\text{DynaMOSA}_{+\text{LDA}}$	5	90
2	18	$\text{DynaMOSA}_{+\text{LR}}$	5	90
2	18	$\text{DynaMOSA}_{+\text{NB}}$	5	90
2	18	$\text{DynaMOSA}_{+\text{SVM}}$	5	90
3	18	$\text{DynaMOSA}_{\text{Standard}}$	5	90
Total				990

Table 6.9: Detailed overview of the rounds of experiments for the Commons-Codec project

The DynaMOSA configurations' results show a relatively low standard deviation of 2.96%, indicating consistent performance across the various single bug prediction models. However, the EPMOSA's configurations show a slightly higher variability, with a standard deviation of 3.55%. This increased variability among EPMOSA's configurations can be attributed to the diverse combinations of prediction models within the ensembles, which may react differently to specific data characteristics or bug patterns encountered in the Commons-Codec project. This variability highlights the complexity and adaptability of ensemble models, indicating that while they are potentially more effective at identifying bugs, they also bring a higher level of unpredictability in different testing scenarios.

Table 6.10: Commons-Codec: Identification Rates, Test Suites Execution and Generation Times

DynaMOSA with single-models	**Bug Identification Rate**
$\text{DynaMOSA}_{+\text{CART}}$	35.56%
$\text{DynaMOSA}_{+\text{KNN}}$	37.78%
$\text{DynaMOSA}_{+\text{LDA}}$	33.33%
$\text{DynaMOSA}_{+\text{LR}}$	35.56%
$\text{DynaMOSA}_{+\text{NB}}$	41.11%
$\text{DynaMOSA}_{+\text{SVM}}$	40.00%
EPMOSA with ensemble-models	**Bug Identification Rate**
$\text{EPMOSA}_{\text{EP-BAST}_{(BME)}}$	47.78%
$\text{EPMOSA}_{\text{EP-BAST}_{(GBT)}}$	41.11%
$\text{EPMOSA}_{\text{EP-BAST}_{(RF)}}$	42.22%
$\text{EPMOSA}_{\text{EP-BAST}_{(SC)}}$	45.56%
$\text{EPMOSA}_{\text{EP-BAST}_{(VC)}}$	38.89%
DynaMOSA default impl.	**Bug Identification Rate**
DynaMOSA	32.22%

The median bug identification rate for DynaMOSA's configurations is 36.67%, slightly below the mean. This suggests a distribution that leans toward lower identification rates, indicating a potential skew in the data. In contrast, the median for EPMOSA's configurations stands at 42.22%, also marginally lower than its mean, hinting at a slight skew toward higher effectiveness.

The performance range for DynaMOSA's configurations, with identification rates between 33.33% and 41.11%, shows a relatively narrow window of effectiveness. This contrasts with the range for EPMOSA's configurations, which spans from 38.89% to 47.78%. This comparison demonstrates that even the least effective ensemble model guiding EPMOSA exceeds the lowest effectiveness of single models used by DynaMOSA, and it highlights the significantly greater potential for peak performance offered by ensemble models.

These findings underscore the advantages of using ensemble models over single models, particularly regarding average effectiveness and potential maximum performance. However, as we have seen in the Commons-Lang experiments, they also reveal more significant variability among the ensemble configurations, suggesting that while they can lead to higher peaks, they may also introduce more fluctuation in effectiveness.

To complement the tabular data presented in Table 6.10, Figure 6.4 visually illustrates the bug identification rates. This graphical representation aims to enhance comprehension by clearly depicting the comparative effectiveness of each approach, thus facilitating a more intuitive understanding of the numerical data.

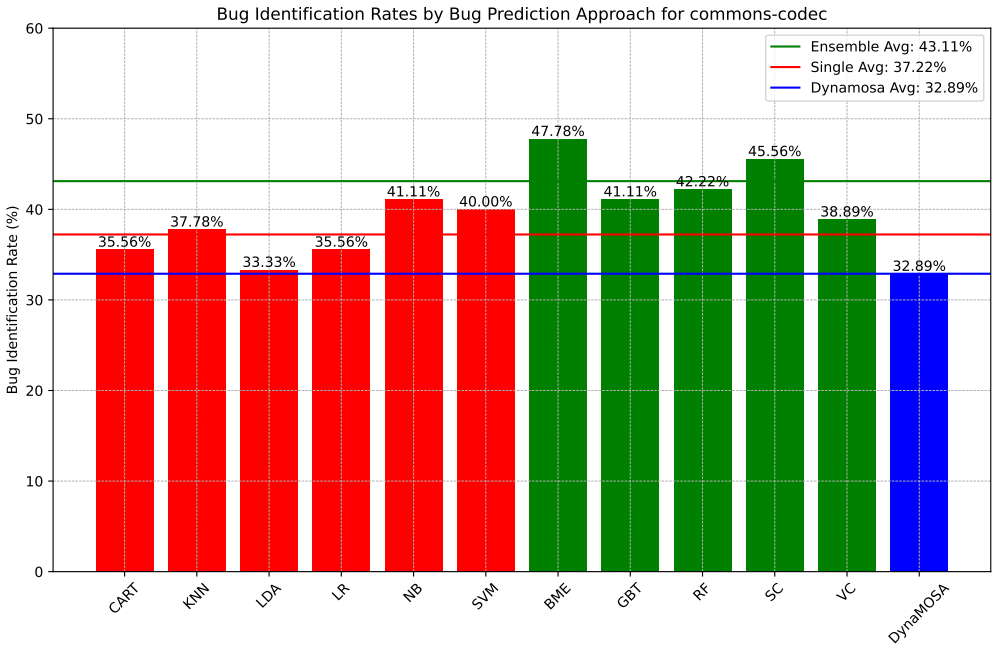

Figure 6.4: Bug identification rates by SBST according to its bug predictive approach for commons-codec

6.3.0.2.1 Result's Statistical Significance

We conduct a statistical analysis of our experimental results to assess the relationship between the EPMOSA and DynaMOSA configurations and the bug identification rate of the generated test suites. We begin this analysis by examining the distribution of bug identification rates in our experiments, aiming to determine if any specific bug prediction model consistently delivers superior bug identification rates.

The box plot in Figure 6.5 provides a visual comparison of the distribution of bug identification rates between EPMOSA and DynaMOSA's configurations. The data in Table 6.11

summarizes the central tendency and variability of the identification rates for the experiments conducted with the Commons-Codec subject system.

This analysis shows that the single models have a median (50th percentile) bug identification rate of around 36.67%, with an interquartile range (IQR) extending from approximately 35.56% to 39.45%. This IQR indicates that the performance of most single models is concentrated within this range. The overall spread from the minimum (33.33%) to the maximum (41.11%) bug identification rate is relatively narrow, suggesting moderate variability among these models.

Conversely, the ensemble models exhibit a higher median bug identification rate of 42.22%, with a broader IQR ranging from 41.11% to 45.56%. The total range spans from a minimum of 38.89% to a maximum of 47.78%, indicating higher variability and more significant potential for achieving superior performance compared to single models. This wider range and higher median suggest that ensemble models are generally more effective but show more variability in their performance outcomes. We have also seen this outcome in the results of Commons-Lang.

These findings indicate that EPMOSA configurations exhibit higher median bug identification rates and potentially achieve significantly higher rates than those seen with DynaMOSA's configurations. However, as observed in previous experiments with the Commons-Lang project, there is considerable variability in the effectiveness of ensemble models. This variability suggests that different ensemble configurations may impact the results of the SBST approach in diverse ways.

Our statistical analysis shows that DynaMOSA, when enhanced with single bug prediction models, tends to offer more consistent efficacy and less variability. This consistency makes single-model configurations reliable for environments where steady performance is valued. On the other hand, EPMOSA guided by ensemble models, though more variable, can achieve higher bug identification rates. This capability makes them more suitable for scenarios where maximizing bug detection is critical despite the associated risk of more significant fluctuations in effectiveness.

The contrast between the two approaches underscores a fundamental trade-off in SBST: the choice between reliability and peak performance. While single models provide a more predictable and stable output, ensemble models, with their capacity to integrate diverse predictive insights, offer the possibility of superior performance at the cost of increased variability. This distinction is crucial for practitioners to consider, as it directly influences the strategic deployment of SBST approaches based on specific project needs and risk tolerance levels.

Model Class	Count	Mean	STD	Min	25%	50%	75%	Max
ensemble	5.0	43.22	3.55	38.89	41.11	42.22	45.56	47.78
single	6.0	37.22	2.96	33.33	35.56	36.67	39.44	41.11

Table 6.11: Summary statistics for the bug identification rate in the commons-codec project

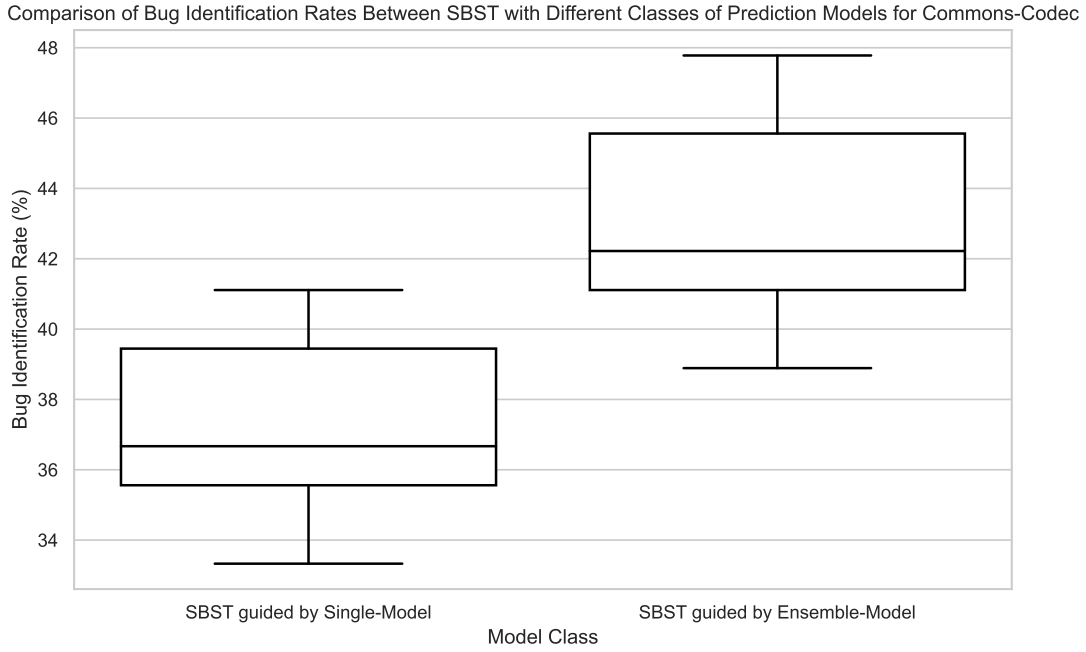

Figure 6.5: Comparison of bug identification rates between SBST with different classes of prediction models applied to commons-codec

We can deepen our analysis by examining the variance in performance within each group of configurations: (1) those with DynaMOSA guided by single bug prediction models, and (2) those with $\text{EPMOSA}_{\text{EP-BAST}}$ guided by ensemble bug prediction models. This examination will help determine if any specific prediction model consistently leads to bug identification rates at their respective ranges' higher or lower ends.

In Figure 6.6, we present a histogram and density plot illustrating the bug identification rates for single and ensemble models. The distribution for the DynaMOSA's configurations tends to cluster primarily around the mid-30% to low 40% range, with a notable peak at approximately 35-36%. This pattern suggests that most configurations of single models perform similarly, with minimal outliers. Noteworthy exceptions are the Naive Bayes and SVM models, which perform slightly better, as detailed in Table 6.10.

Conversely, the EPMOSA's configurations exhibit a broader spread in performance, ranging from just below 40% to nearly 48%. The histogram for these implementations shows multiple peaks, indicating that certain ensemble configurations, such as those involving specific combinations of prediction models, outperform others significantly. This variation underscores the impact of different ensemble strategies on the effectiveness of the SBST approach in different projects, demonstrating more significant variability and the potential for higher peak performance.

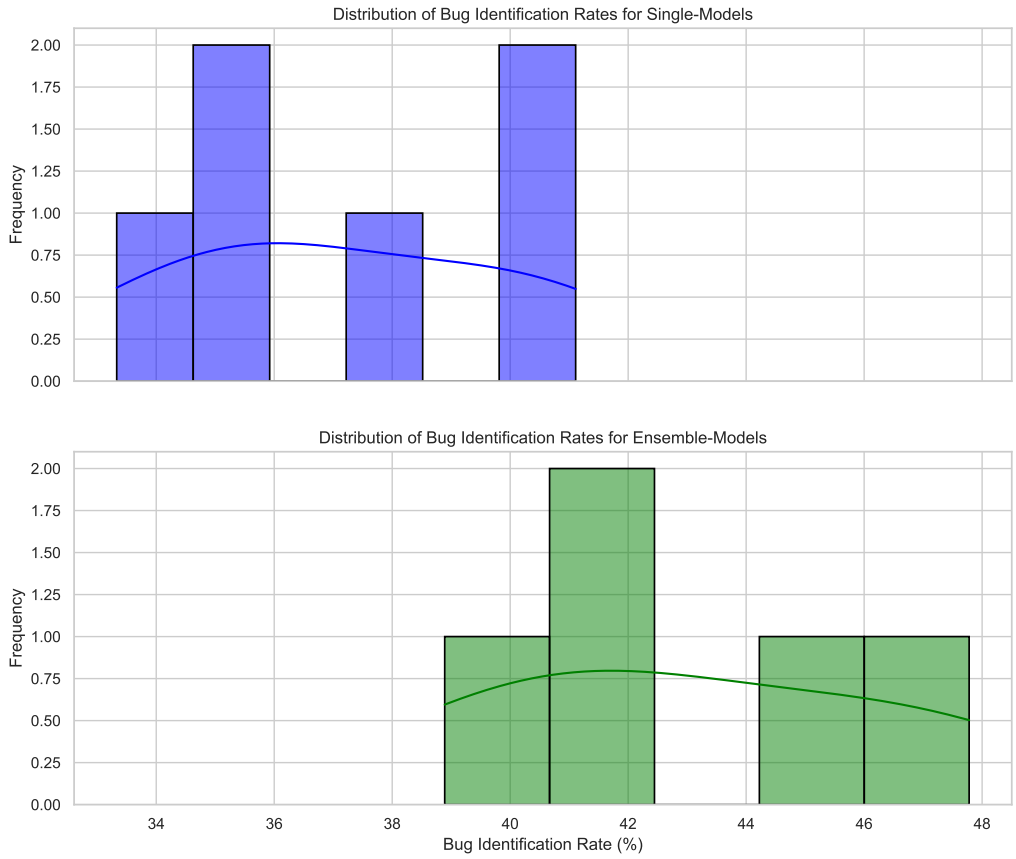

Figure 6.6: Comparison of bug identification rates between SBST with different classes of prediction models applied to commons-codec

The density plot in Figure 6.6 indicates that DynaMOSA configurations with single models display a peak around the mid-30s percent, reinforcing the histogram's suggestion of a prevalent performance level within this group. The relatively narrow curve implies less variability among the single models, indicating a consistent performance level across this group.

Conversely, the density plot for EPMOSA's configurations exhibits a broader and multi-peaked distribution. This indicates more significant variability in bug identification rates, with the wider base of the curve reflecting the potential for these models to achieve a spectrum of outcomes from moderate to high effectiveness in bug identification. Notably, the peak around 45-48% for the $\text{EPMOSA}_{\text{EP-BAST}_{(\text{BME})}}$ model, as referenced in Table 6.10, underscores its superior performance compared to other models.

To further examine the relationship between the class of bug prediction models employed (ensemble or single models) to guide SBST approaches and the bug identification rates of the generated test suites, we can extend our statistical analysis using ANOVA and the Mann-Whitney U Test, similar to our approach for the Commons-Lang project.

The results from the ANOVA test, as documented in Table 6.18, reveal a statistically significant difference with a p-value of approximately 0.0148. This confirms that the disparities in mean

bug identification rates between the single and ensemble model types are significant and not merely due to random variation. The observed F-value from this test also indicates a substantial variance between the groups, emphasizing a robust difference in performance that favors the ensemble models over the single models for guiding SBST approaches in the Commons-Codec project.

Project	F-value	p-value
commons-lang	9.03	0.0148

Table 6.12: ANOVA results for the bug identification rates in the commons-codec project

Complementing the ANOVA results, the Mann-Whitney U test further supports the superiority of ensemble models with a p-value of approximately 0.0277. This outcome affirms that the distribution of bug identification rates for ensemble models is significantly different and generally higher than for single models. The U statistic from this test illustrates the consistent ranking of one group above the other, reinforcing the effectiveness of ensemble models in practical applications.

Project	U-statistic	p-value
commons-lang	2.5	0.0277

Table 6.13: Mann-Whitney U results for the bug identification rates in the commons-codec project

These statistical tests highlight the superior effectiveness of ensemble models over single models in guiding SBST approaches and quantify the degree of this advantage within the Commons-Codec project. The consistency observed across various statistical evaluations – addressing means, distributions, and correlations – provides a comprehensive view of the advantages of ensemble strategies for SBST approaches. This robust statistical validation supports the strategic deployment of ensemble models to guide SBST test case generation techniques, particularly in scenarios where maximizing bug detection is crucial.

6.3.0.3 Commons-Compress

The Commons-Compress project includes 47 buggy versions as detailed in Table 6.14. Three experimental rounds were conducted for each buggy version. Each configuration of EPMOSA and DynaMOSA was tested in 47 separate experiments – one for each buggy version. Each experiment was repeated five times to account for the stochastic nature of the algorithms used in test case generation. As a result, for each approach and configuration, a total of 235 test suites were generated (47 experiments multiplied by five runs each). Consequently, this extensive testing effort created 2,585 test suites for the Commons-Compress project.

Table 6.15 provides an overview of the bug identification rates from experiments conducted with $\text{EPMOSA}_{\text{EP-BAST}}$ and DynaMOSA enhanced with single models for the Commons-Compress project. The results reveal significant bug identification rate differences between single and ensemble model configurations. DynaMOSA, when guided by single models, has an average bug identification rate of 3.83%, illustrating its relatively modest capability in generating test suites

Round	Experiments	Approach/Configuration	Runs	Test Suites
1	47	$\text{EPMOSA}_{\text{EP-BAST}_{(\text{GBT})}}$	5	235
1	47	$\text{EPMOSA}_{\text{EP-BAST}_{(\text{RF})}}$	5	235
1	47	$\text{EPMOSA}_{\text{EP-BAST}_{(\text{SC})}}$	5	235
1	47	$\text{EPMOSA}_{\text{EP-BAST}_{(\text{VC})}}$	5	235
2	47	$\text{DynaMOSA}_{+\text{CART}}$	5	235
2	47	$\text{DynaMOSA}_{+\text{KNN}}$	5	235
2	47	$\text{DynaMOSA}_{+\text{LDA}}$	5	235
2	47	$\text{DynaMOSA}_{+\text{LR}}$	5	235
2	47	$\text{DynaMOSA}_{+\text{NB}}$	5	235
2	47	$\text{DynaMOSA}_{+\text{SVM}}$	5	235
3	47	$\text{DynaMOSA}_{\text{Standard}}$	5	235
Total				2585

Table 6.14: Detailed overview of the rounds of experiments for the Commons-Compress project

that detect bugs. This is further underscored by a low standard deviation of 0.76%, indicating minimal variability and suggesting that single model configurations yield moderately consistent performance across various configurations.

Conversely, the EPMOSA configurations achieve a higher average bug identification rate of 5.96%, highlighting its enhanced effectiveness in identifying bugs. The identification rates among the ensemble models show more significant variability, with a standard deviation of 1.62%. This increased variability can be attributed to the diverse combinations of prediction models used within the ensemble, leading to various outcomes. This distinction emphasizes the potential of ensemble models to improve bug detection significantly, albeit with a higher degree of outcome variability.

The median bug identification rates further accentuate the distinctions between the two groups of configurations. For DynaMOSA guided by single models, the median rate is 3.62%, closely aligning with the average and suggesting a symmetric distribution of results. This consistency indicates that single-model approaches generally deliver stable performance. In contrast, for EPMOSA guided by ensemble models, the median rate is significantly higher at 6.38%, exceeding the average. This higher median suggests that, while some configurations may yield lower rates, the typical performance of ensemble models is notably robust.

Moreover, the range of identification rates further highlights the enhanced capabilities of ensemble models, which span from 3.40% to 7.66%. This compares to the much narrower range of 2.98% to 5.11% observed with single models, indicating that ensemble models offer a higher baseline performance and the potential for peak performance under optimal conditions.

These statistical insights have practical implications for model selection in SBST approaches, particularly for the Commons-Compress project. EPMOSA guided by ensemble models consistently shows superior effectiveness, with a higher average and the potential for exceptional performance in specific configurations. This analysis robustly supports ensemble models to guide SBST approaches in scenarios where maximizing bug identification is crucial, underscoring their strategic value in software testing and quality assurance. This preference for ensemble models reflects their ability to integrate diverse prediction strategies, enhancing the likelihood of

detecting a broader array of bugs and improving the quality of software testing outcomes.

Table 6.15: Commons-Compress: Identification Rates

DynaMOSA with single-models	**Bug Identification Rate**
$\text{DynaMOSA}_{+\text{CART}}$	3.83%
$\text{DynaMOSA}_{+\text{KNN}}$	3.40%
$\text{DynaMOSA}_{+\text{LDA}}$	2.98%
$\text{DynaMOSA}_{+\text{LR}}$	3.40%
$\text{DynaMOSA}_{+\text{NB}}$	4.26%
$\text{DynaMOSA}_{+\text{SVM}}$	5.11%
EPMOSA with ensemble-models	**Bug Identification Rate**
$\text{EPMOSA}_{\text{EP-BAST}_{(\text{BME})}}$	4.68%
$\text{EPMOSA}_{\text{EP-BAST}_{(\text{GBT})}}$	7.23%
$\text{EPMOSA}_{\text{EP-BAST}_{(\text{RF})}}$	7.66%
$\text{EPMOSA}_{\text{EP-BAST}_{(\text{SC})}}$	6.38%
$\text{EPMOSA}_{\text{EP-BAST}_{(\text{VC})}}$	6.38%
DynaMOSA default impl.	**Bug Identification Rate**
DynaMOSA	3.40%

Complementing the data in Table 6.15, Figure 6.7 provides a visual representation of the bug identification rates discussed. This graphical illustration enhances comprehension by clearly depicting the comparative effectiveness of each approach and respective configurations. The figure underscores the statistical differences and trends highlighted in the text by visually contrasting the performance of $\text{EPMOSA}_{\text{EP-BAST}}$ guided by ensemble models with DynaMOSA guided by single models.

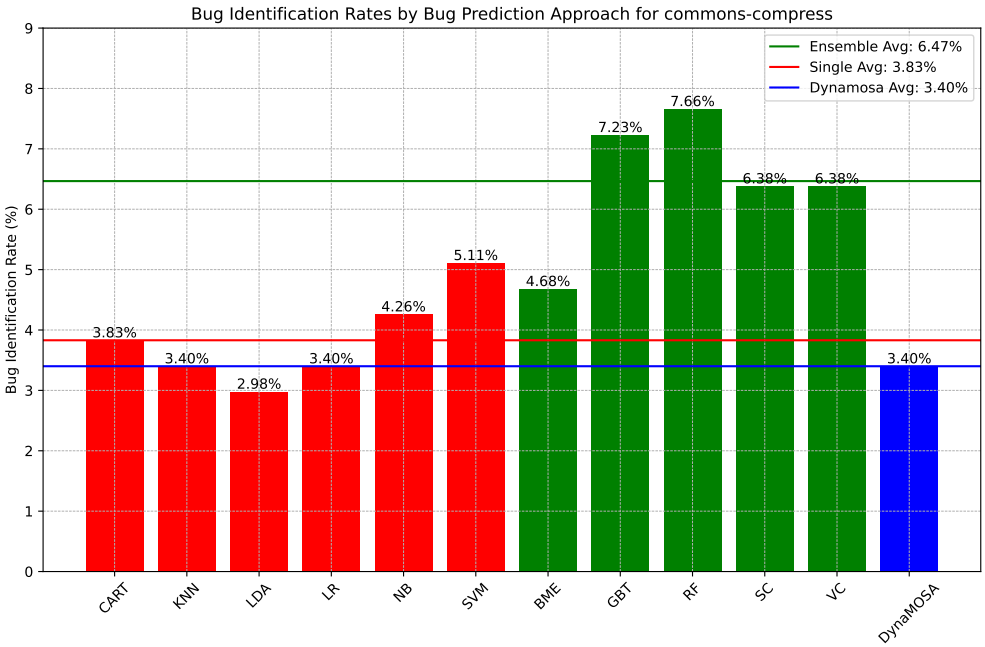

Figure 6.7: Bug Identification Rates by SBST according to its bug predictive approach commons-compress

6.3.0.3.1 Result's Statistical Significance

For the Commons-Compress project, we conducted the same statistical tests performed for the Commons-Lang and Commons-Codec projects. The results from these tests provide compelling evidence of significant performance differences between single and ensemble models, as indicated by the findings from the ANOVA and Mann-Whitney U tests. These results confirm the observed discrepancies in bug identification rates and emphasize the statistical significance of these differences, thereby reinforcing the superior efficacy of ensemble models in guiding SBST approaches.

The box-plot in Figure 6.8 illustrates the comparison of the distribution of bug identification rates between EPMOSA$_{(EP\text{-}BAST)}$ guided by ensemble models and DynaMOSA guided by single models. This visual representation, accompanied by the data in Table 6.16, provides a detailed summary of the central tendency and variability of the identification rates in the experiments conducted with the Commons-Compress project. This box plot effectively highlights the broader spread and generally higher identification rates achieved by the ensemble models compared to the single models' more consistent but lower rates, visually reinforcing the statistical analysis and providing a clear depiction of the performance landscape within this specific project context.

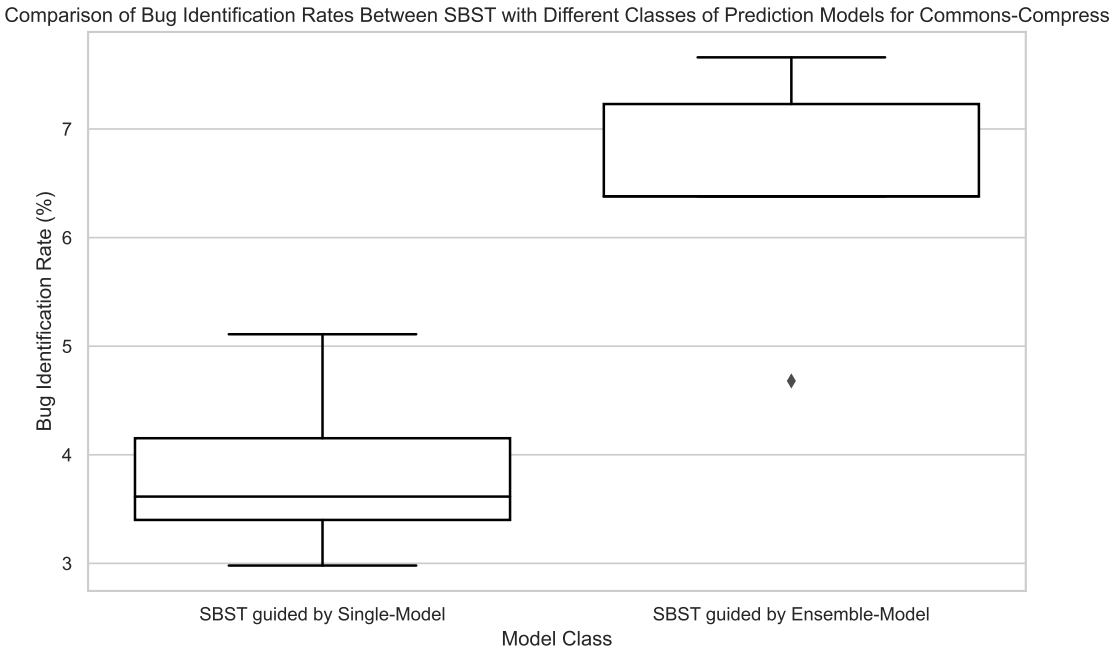

Figure 6.8: Comparison of bug identification rates between SBST with different classes of prediction models applied to commons-compress

It is possible to observe that the average bug identification rate for EPMOSA$_{(EP\text{-}BAST)}$ guided by ensemble models is 6.466%, which suggests SBST guided by ensemble models perform better than SBST guided by single models in identifying bugs.

The standard deviation of 1.1417% among the ensemble models used in EPMOSA's configurations indicates a higher variability in bug identification rates than in single models. This increased spread suggests a broader diversity in performance within the ensemble model group, reflecting the varied effectiveness of different ensemble configurations.

Model Class	Count	Mean	STD	Min	25%	50%	75%	Max
ensemble	5.0	6.466	1.1417	4.68	6.38	6.38	7.23	7.66
single	6.0	3.83	0.7636	2.98	3.40	3.615	4.15	5.11

Table 6.16: Summary statistics for the bug identification rate in the commons-compress project

Interestingly, the lowest bug identification rate recorded for the EPMOSA's configurations using ensemble models is 4.68%, surpassing the highest rate observed with single models. This highlights the enhanced baseline performance of ensemble models over single models.

Regarding quartile distribution, the first quartile shows that EPMOSA achieves a bug identification rate of 6.38% or lower, coinciding with the median rate. This median indicates that half of the ensemble models tested in the $\text{EPMOSA}_{\text{(EP-BAST)}}$ experiment perform at or above this rate, demonstrating a central tendency towards higher effectiveness within this group.

Moving to the third quartile, the bug identification rates for experiments with $\text{EPMOSA}_{\text{(EP-BAST)}}$ guided by ensemble models reach up to 7.23%, indicating a skew towards higher performance among the upper quartile of models.

The peak performance observed in these experiments is a bug identification rate of 7.66% among the $\text{EPMOSA}_{\text{(EP-BAST)}}$ ensemble models, showcasing their potential for high effectiveness in the Commons-Compress project. This upper range demonstrates the significant capability of ensemble models to achieve superior bug detection rates, further validating their strategic value in scenarios where maximizing bug identification is crucial.

The average bug identification rate among the DynaMOSA configurations enhanced with single models is 3.83%, significantly lower than that observed for the ensemble models. The minimum rate among these configurations is 2.98%, which is the lowest recorded across the ensemble and single model configurations.

In the distribution of results, the first quartile for DynaMOSA with single models indicates a bug identification rate of 3.40% or lower. The median rate is 3.615%, showing that half of the single models yield identification rates below this level. In the third quartile, we observe that 75% of the single models achieve a bug identification rate of 4.15% or lower, illustrating only modest performance gains in the upper quartile. The highest rate recorded among the single models is 5.11%, which falls below the lowest rate observed for the ensemble models.

This analysis highlights that ensemble models have significantly outperformed single models in guiding SBST for automatic test case generation. This is evidenced by higher mean and maximum values in the experiments conducted with $\text{EPMOSA}_{\text{(EP-BAST)}}$. However, similar to the findings from experiments with Commons-Lang and Commons-Codec, $\text{EPMOSA}_{\text{(EP-BAST)}}$ guided by ensemble models exhibit more significant variability in bug identification rates, as reflected by a higher standard deviation. This variability suggests that while some ensemble prediction models are highly effective for the Commons-Compress project, others perform closer to the average identification rate observed with single models in DynaMOSA configurations. Its range of effectiveness emphasizes the need to carefully select and customize ensemble models to maximize their performance in specific testing environments.

The comparative analysis between ensemble and single models regarding bug identification rates underscores the superior performance of ensemble models in the context of the Commons-Compress project. However, it also highlights the need to manage the more significant variability observed among them. The higher standard deviation and broader range in performance metrics indicate that while some ensemble models achieve exceptional effectiveness, others may not consistently deliver the same level of performance. This variability is a critical factor in decision-making processes where the choice of model type can significantly influence the success of bug identification efforts.

To deepen our understanding of performance variance within each group of approaches, we can analyze the results from EPMOSA$_{(EP\text{-}BAST)}$ guided by ensemble models and DynaMOSA guided by single models. As with the previous analyses conducted for the Commons-Lang and Commons-Codec projects, this examination will help us determine if specific prediction models within each group consistently lead to bug detection rates at the higher or lower ends of their respective ranges.

In Figure 6.9, we present histograms and density plots for the experimental results with EPMOSA$_{(EP\text{-}BAST)}$ guided by ensemble models and DynaMOSA guided by single models. These visual representations provide a clear understanding of the distribution of bug identification rates for each group, illustrating each approach's variability and central tendencies. This graphical analysis reinforces the textual analysis and offers a more nuanced view of how different models perform within each framework, aiding stakeholders in making informed choices about which models to deploy in specific testing scenarios.

In Figure 6.9, we can observe that the bug identification rates for DynaMOSA guided by single models are clustered more closely together, with the majority of rates falling between approximately 3% and 4%. This clustering indicates a narrower range of variation in bug identification effectiveness, suggesting that single models deliver a consistently moderate level of performance.

In contrast, the results from experiments with EPMOSA$_{(EP\text{-}BAST)}$ guided by ensemble models demonstrate a broader range of identification rates, primarily stretching from about 4.5% to 7.5%. The histogram in Figure 6.9 displays a more dispersed distribution, with notable peaks at intermediate rates around 6-7%. This pattern suggests that specific ensemble models significantly outperform others, indicating variability in effectiveness across different ensemble configurations.

The accompanying density plots over the histogram further elucidate these observations. The density for DynaMOSA guided by single models is peaked and narrow, centered around 3.5% to 4%, which confirms the tight clustering evident in the histogram. This peak reinforces that while DynaMOSA, guided by single models, achieves consistent results, its effectiveness in bug identification remains moderately limited.

These visual representations help to underscore the distinct performance characteristics of the two approaches, with single models showing limited but consistent effectiveness and ensemble models demonstrating a wider range of potential outcomes, from moderate to high effectiveness. This differentiation is crucial for understanding the strategic deployment of these models in

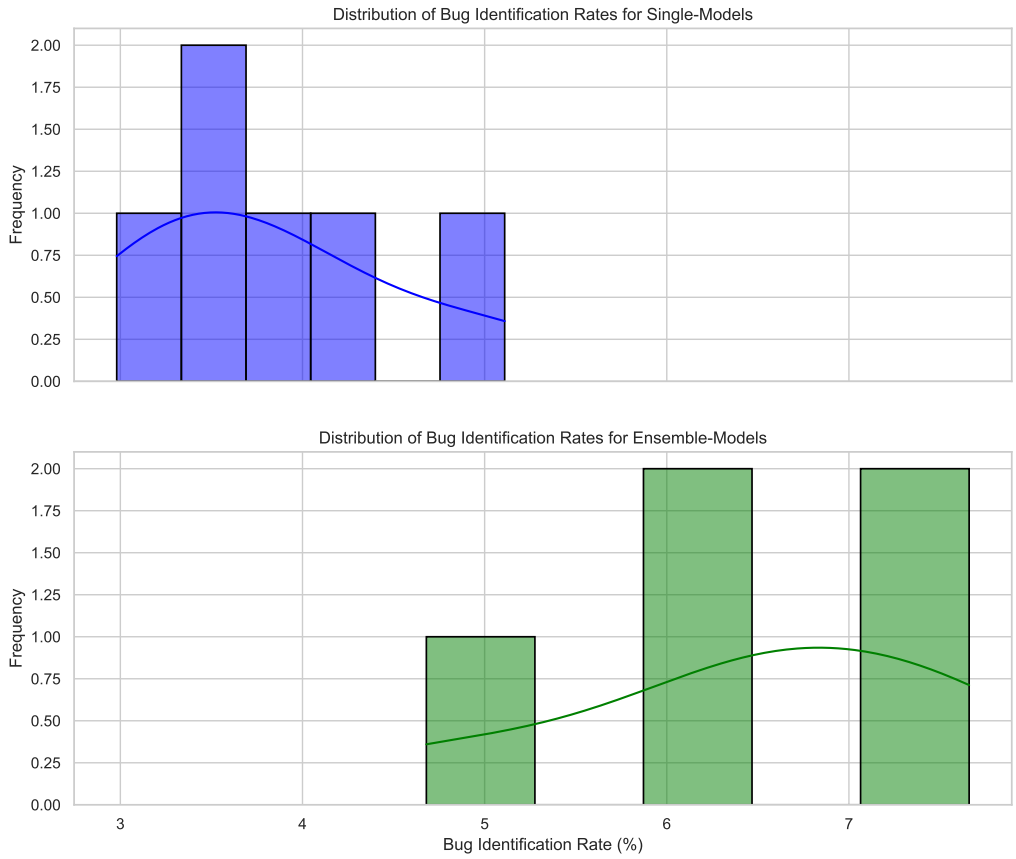

Figure 6.9: Comparison of bug identification rates between SBST with different classes of prediction models applied to commons-compress

SBST, particularly in scenarios where the balance between consistency and peak performance must be carefully managed.

The density curve for $EPMOSA_{(EP\text{-}BAST)}$ guided by ensemble models is broader and flatter, with multiple peaks indicating variability in effectiveness. The right-skewness of this plot suggests that while most ensemble models perform better than the average single models used by DynaMOSA, a few also achieve exceptionally high bug identification rates (above 7%). This distribution indicates that ensemble models not only generally surpass the performance of single models but also have the potential to achieve significantly higher effectiveness.

For the Commons-Compress project, our experiments reveal that while DynaMOSA guided by single models offers more consistent performance, it does so generally at lower effectiveness levels. In contrast, $EPMOSA_{(EP\text{-}BAST)}$ guided by ensemble models exhibits more significant variability in their effectiveness but can achieve substantially higher rates. However, this variability has the potential for lower performance in some configurations.

Consistent with our analysis of the Commons-Lang and Commons-Codec projects, the choice between employing single or ensemble models to guide SBST techniques depends on the preference for either consistency or the potential for higher performance with some risk of lower effectiveness.

Statistical validation of these differences is provided by the ANOVA test results shown in Table 6.17, which recorded an F-value of 8.49. This significant F-value indicates a substantial variance between the means of bug identification rates from single models and ensemble models in guiding SBST approaches, with a corresponding p-value of 0.0155, well below the threshold of 0.05. This confirms that the observed differences in mean bug identification rates are statistically significant and not due to random variation, thereby statistically substantiating the superior average performance of $EPMOSA_{(EP\text{-}BAST)}$ guided by ensemble models.

Project	**F-value**	**p-value**
commons-lang	8.49	0.0155

Table 6.17: ANOVA results for the bug identification rates in the commons-compress project

Complementing the ANOVA, the Mann-Whitney U test provides further analysis of these differences by comparing the distributions of the two groups without assuming normality. As shown in Table 6.18, a U-statistic of 5.0 and a p-value of 0.0435 confirm that the rankings of bug identification rates between single and ensemble models are significantly different. This result from the non-parametric test aligns with the findings from the ANOVA, offering robust evidence that ensemble models consistently outperform single models in terms of bug detection capabilities for the Commons-Compress project.

The consistency between these statistical tests underscores the reliability of the conclusion that ensemble models not only provide higher average effectiveness but also demonstrate a statistically significant advantage over single models in enhancing the bug detection capabilities of automatically generated test suites. This comprehensive analysis solidifies the strategic importance of choosing ensemble models for guiding SBST approaches, especially in scenarios

where maximizing bug identification is crucial.

Project	U-statistic	p-value
commons-lang	2.5	0.0277

Table 6.18: Mann-Whitney U results for the bug identification rates in the commons-compress project

The combined results from these statistical tests provide a holistic view that the superior performance of ensemble models is not merely an anomaly due to sampling or experimental setup but a consistent, reproducible outcome. The statistical robustness furnished by these findings is instrumental in justifying the use of ensemble models in practical applications, particularly in contexts where detecting a higher percentage of bugs is crucial to the success of software testing and quality assurance efforts.

6.3.0.4 Joda-Time

The Joda-Time project includes 26 buggy versions as part of the Defects4J dataset. Detailed in Table 6.19, three experimental rounds were conducted for each buggy version. For each configuration of EPMOSA and DynaMOSA, 26 experiments were executed – one for each buggy version. Each experiment was run five times due to the stochastic nature of the algorithms used in test case generation. As a result, each approach or configuration produced a total of 130 test suites. Consequently, these efforts created 1,300 test suites for the Joda-Time project.

Round	Experiments	Approach/Configuration	Runs	Test Suites
1	26	$\text{EPMOSA}_{\text{EP-BAST}_{(\text{GBT})}}$	5	130
1	26	$\text{EPMOSA}_{\text{EP-BAST}_{(\text{RF})}}$	5	130
1	26	$\text{EPMOSA}_{\text{EP-BAST}_{(\text{SC})}}$	5	130
1	26	$\text{EPMOSA}_{\text{EP-BAST}_{(\text{VC})}}$	5	130
2	26	$\text{DynaMOSA}_{+\text{CART}}$	5	130
2	26	$\text{DynaMOSA}_{+\text{KNN}}$	5	130
2	26	$\text{DynaMOSA}_{+\text{LDA}}$	5	130
2	26	$\text{DynaMOSA}_{+\text{LR}}$	5	130
2	26	$\text{DynaMOSA}_{+\text{NB}}$	5	130
2	26	$\text{DynaMOSA}_{+\text{SVM}}$	5	130
Total				1300

Table 6.19: Detailed overview of the rounds of experiments for the Joda-Time project

In analyzing the bug identification rates from the Joda-Time project as presented in Table 6.19, it is evident that ensemble models demonstrate a higher mean bug identification rate of 32.82%. This suggests a slight but notable advantage in effectiveness compared to single models, which exhibit a mean rate of 31.03%. This superiority in average performance indicates that integrating multiple algorithms or strategies within ensemble models may enhance SBST bug detection capabilities. Furthermore, the median bug identification rates support this assessment, with $\text{EPMOSA}_{(\text{EP-BAST})}$ and ensemble models achieving a median of 35%, significantly higher than the 31.93% observed for single models. This comparison shows that more than half of the ensemble models outperform the median rate of single models, highlighting the robustness of ensemble strategies in practical testing scenarios.

These findings suggest ensemble models offer higher average bug identification rates and exhibit greater consistency and peak performance. This makes them potentially more reliable and effective within the context of the Joda-Time project. The data strongly supports the strategic use of ensemble models in scenarios where superior bug detection is crucial, thereby enhancing software development's overall quality assurance process.

To further elucidate these findings, Figure 6.10 visually represents the same bug identification rates. This graphical depiction is crafted to enhance understanding by visually illustrating the comparative effectiveness of each approach, offering a clear visual aid to complement the quantitative data presented. This graphical approach helps to convey the performance disparities between single and ensemble models in a more intuitive and accessible manner, aiding stakeholders in making informed decisions about the methodologies employed in software testing.

Table 6.20: Joda-Time: Identification Rates

DynaMOSA with single-models	**Bug Identification Rate**
$\text{DynaMOSA}_{\text{+CART}}$	29.23%
$\text{DynaMOSA}_{\text{+KNN}}$	33.08%
$\text{DynaMOSA}_{\text{+LDA}}$	26.92%
$\text{DynaMOSA}_{\text{+LR}}$	33.08%
$\text{DynaMOSA}_{\text{+NB}}$	32.31%
$\text{DynaMOSA}_{\text{+SVM}}$	31.54%
EPMOSA with ensemble-models	**Bug Identification Rate**
$\text{EPMOSA}_{\text{EP-BAST}_{\text{(BME)}}}$	36.15%
$\text{EPMOSA}_{\text{EP-BAST}_{\text{(GBT)}}}$	35.38%
$\text{EPMOSA}_{\text{EP-BAST}_{\text{(RF)}}}$	36.15%
$\text{EPMOSA}_{\text{EP-BAST}_{\text{(SC)}}}$	33.85%
$\text{EPMOSA}_{\text{EP-BAST}_{\text{(VC)}}}$	34.62%
DynaMOSA default impl.	**Bug Identification Rate**
DynaMOSA	20.77%

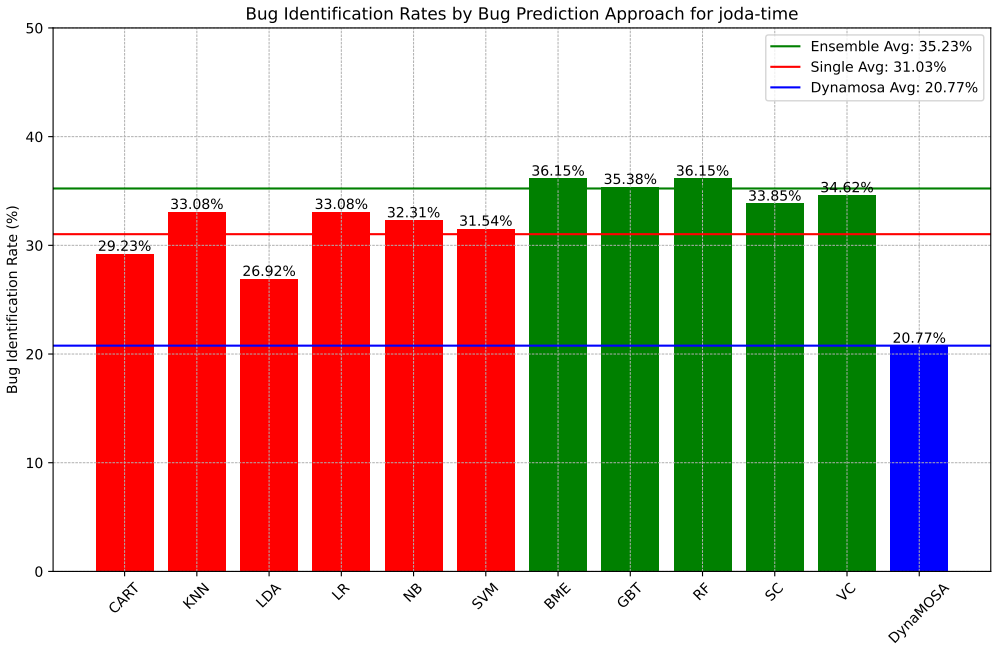

Figure 6.10: Bug Identification Rates by SBST according to its bug predictive approach for joda-time

6.3.0.4.1 Result's Statistical Significance

The box plot in Figure 6.11 illustrates the comparison of the distribution of bug identification rates from our experiments involving $\text{EPMOSA}_{\text{EP-BAST}}$ guided by ensemble models and DynaMOSA guided by single models. This visual representation is further supported by data from Table 6.21, which provides a detailed summary of the central tendency and variability of the identification rates observed in the experiments with the Joda-Time project.

This box plot delineates the range and concentration of bug identification rates for each approach, highlighting the differences in performance between single and ensemble model configurations. The box plot effectively complements the statistical summary by presenting the spread and central values of the identification rates, giving a more nuanced understanding of how each model type performs in practical testing scenarios.

This visual and statistical approach helps assess the relative effectiveness of the different modeling strategies used in SBST, offering valuable insights into their operational strengths and weaknesses within the specific context of the Joda-Time project.

Table 6.21 provides descriptive statistics for the bug identification rates using two classes of bug prediction models– ensemble and single – that were used to guide $\text{EPMOSA}_{\text{EP-BAST}}$ and DynaMOSA in generating test cases for the Joda-Time project. Relatively consistent performance is evident for the ensemble models, with an average bug identification rate of 35.27%. This consistency is further highlighted by a low standard deviation of 1.01%, indicating that the identification rates are closely clustered around the mean. The minimum rate observed among the ensemble models is 33.85%, with the maximum rate reaching only slightly higher at 36.15%. This narrow range, coupled with a median rate of 35.58% that slightly exceeds the mean, suggests a tight distribution of outcomes without extreme deviations.

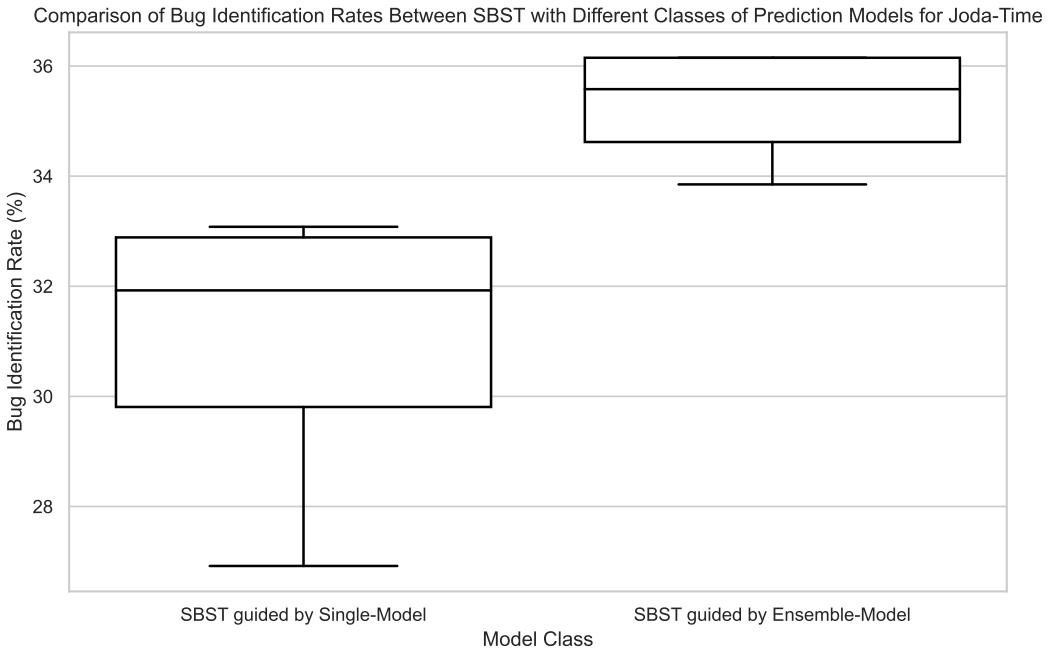

Figure 6.11: Comparison of bug identification rates between SBST with different classes of prediction models applied to Joda-Time

Conversely, the results using single models with DynaMOSA exhibit a broader spread. The average bug identification rate for single models is significantly lower at 31.02%. This lower average is accentuated by a higher standard deviation of 2.46%, which points to greater variability in performance among the single models. The minimum rate dips to 26.92% (specifically for LDA), indicating underperformance in some cases. In contrast, the maximum rate is capped at 33.08% (notably for LR and KNN), suggesting that single models rarely achieve the effectiveness levels observed with ensemble models. The median bug identification rate of 31.92% closely aligns with the average, indicating a distribution that is either normal or slightly skewed but with a range that highlights more pronounced variability in performance.

The overall analysis for Joda-Time reveals that ensemble models achieve higher bug identification rates on average and maintain a more consistent performance than single models. This consistency and reduced variability in ensemble models suggest that they are more robust and reliable for identifying bugs in the Joda-Time project. On the other hand, the greater variability and occasionally much lower effectiveness of single models could make them less reliable. However, they might still offer value under certain conditions where variability is less of a concern. The decision on which model class to deploy should consider these aspects of performance to optimize bug identification efficiency and reliability in real-world applications.

Model Class	Count	Mean	STD	Min	25%	50%	75%	Max
ensemble	5.0	35.27	1.01	33.85	34.62	35.58	36.15	36.15
single	6.0	31.02	2.46	26.92	29.80	31.92	32.88	33.08

Table 6.21: Summary statistics for the bug identification rate in the Joda-Time project

As noted, there was considerable variance in some rounds of experiments, with certain configurations of $\text{EPMOSA}_{\text{EP-BAST}}$ exhibiting significant variability in performance. To further assess and understand this variance, we can analyze the histogram and density plot in Figure 6.12.

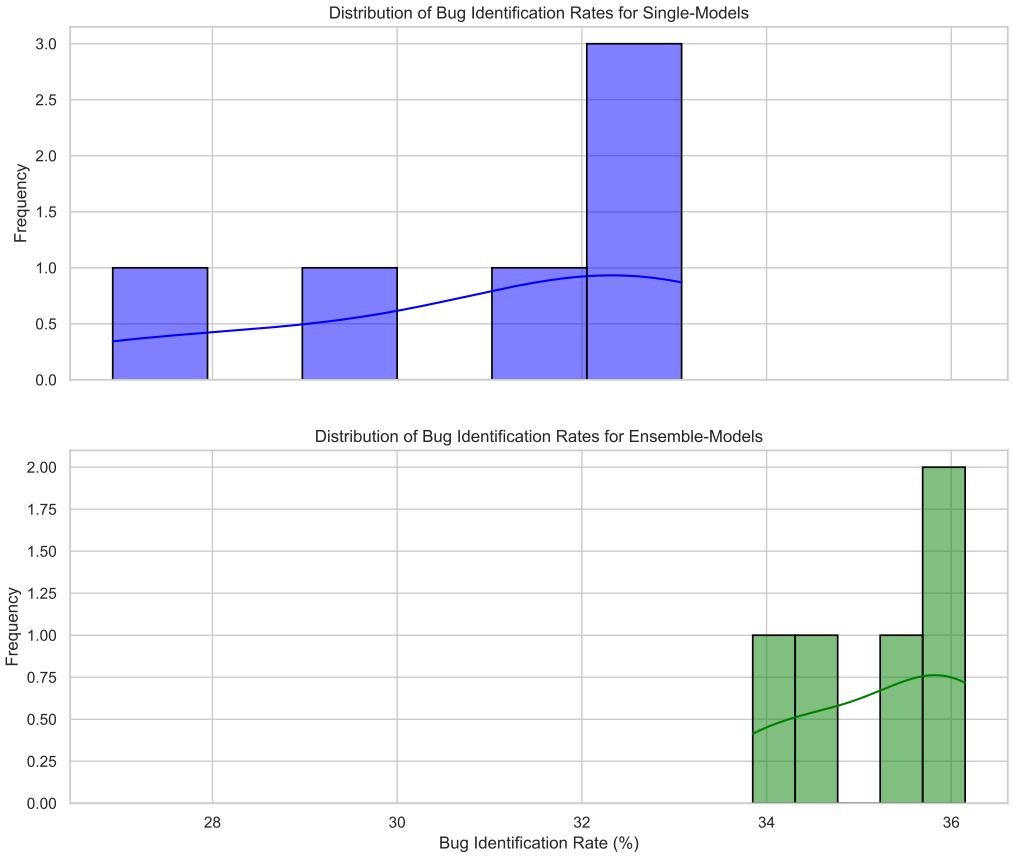

Figure 6.12: Comparison of bug identification rates between SBST with different classes of prediction models applied to Joda-Time

From the histograms and density plots displayed in Figure 6.12, it is clear that there is a significant variance in bug identification rates between single and ensemble model setups. Single models, which include various configurations of DynaMOSA, typically exhibit a broader range of identification rates, indicating a wide variation in effectiveness. In contrast, ensemble models, represented by various EPMOSA configurations, tend to show a tighter clustering of rates, usually at higher levels.

The density plot for single models (Figure 6.12) reveals a broad spread, indicating diverse performance levels across different configurations. This plot features multiple peaks, suggesting that various single models perform optimally at distinct levels, underscoring the variability in their effectiveness. Notably, the peaks around the mid-20s to low 30% ranges correspond to models like LDA, with more effective models such as KNN and LR achieving slightly higher rates. This spread implies that the effectiveness of single models can be highly dependent on the nature of the data they are applied to and the specific implementation details of each prediction

algorithm. The broader base of the density curve for single models also suggests a higher likelihood of encountering moderate to low bug identification rates when compared to ensemble models.

Conversely, the density plot for ensemble models is markedly more concentrated, with a higher peak, indicating less variability and a higher consistency around elevated identification rates. This focused clustering around the 35-36% range highlights the effectiveness of ensemble methods like RF and BME, which consistently achieve higher bug identification rates. The sharp peak and narrower spread of the ensemble models' density plots demonstrate that these configurations are generally more reliable and offer superior performance across various testing scenarios.

Overall, these visual analyses from the histograms and density plots reinforce the strategic advantage of utilizing ensemble models over single models in settings where maximizing bug detection is crucial. Ensemble models' consistent higher performance and reliability make them preferable for enhancing the effectiveness of SBST approaches.

We can confirm the statistical significance of our observations by performing the ANOVA and Mann-Whitney U test, as we did for the results in Commons-Lang, Commons-Compress, and Commons-Codec.

The ANOVA test results are shown in Table 6.22. An F-value of 12.803 is relatively high, indicating a strong variance between the means of the tested groups (single and ensemble). A higher F-value generally suggests that at least one of the group means is significantly different from the others. In the context of our results, this reinforces that the performance (as measured by bug identification rates) of at least one model group (like single models vs. ensemble models) is significantly different from the others.

Since the obtained p-value is less than the commonly used significance level of 0.05, we can assume that the differences in bug identification rates between the model groups are statistically significant and not due to random chance.

Project	**F-value**	**p-value**
Joda-Time	12.803	0.0059

Table 6.22: ANOVA results for the bug identification rates in the Joda-Time project

Complementing this, the Mann-Whitney U test (Table 6.23), which is a non-parametric test that does not require the assumption of a normal distribution, was utilized to assess differences between the distributions of the two model groups. This test provided a p-value of 0.0056, which, while not falling under the traditional threshold of statistical significance ($p < 0.05$), suggests a marginal significance that indicates potential differences in how the models perform across their range. This result implies that the ensemble models might exhibit greater variability in performance, potentially achieving higher bug identification rates under certain conditions but not consistently across all scenarios.

U-statistic of 0, although rare, can indeed occur under certain conditions in data analysis, especially in scenarios involving very small sample sizes or where there is a stark difference

Project	F-value	p-value
Joda-Time	0.0	0.0056

Table 6.23: Mann-Whitney U test results for the bug identification rates in the Joda-Time project

between the characteristics or outcomes of the two groups being compared.

When one group consistently scores higher or lower than the other on the test measure, achieving a U-statistic of 0 is possible. This situation often arises when there is no overlap between the distributions of the two groups. For example, in the context of the bug identification rates in the Joda-Time project, the ensemble models consistently outperformed the single models across all measures. This can lead to such an extreme result.

In practical terms, this result points to a large effect size, meaning the superiority of ensemble models over single models in the Joda-Time project is statistically significant and practically meaningful.

6.3.1 EPMOSA Efficiency

***RQ2**: Is EPMOSA efficient compared to the state-of-the-art generation approach DynaMOSA?*

This section evaluates the efficiency of the Ensemble Predictive Many-Objective Sorting Algorithm (EPMOSA) compared to DynaMOSA by examining their test generation and execution times across various subject projects.

Analyzing the average execution and generation times of these approaches is essential for assessing their efficiency in the testing process. Execution time measures how long the tests are to run, while generation time focuses on the duration required to create these tests. These metrics are crucial indicators of the algorithms' operational efficiency.

However, it is also important to consider how these times relate to each method's effectiveness in detecting bugs. An algorithm that generates tests quicker but fails to detect a significant number of bugs may not be as desirable as one that takes longer but achieves higher bug detection rates. Thus, while efficiency in terms of time is valuable, it should not be evaluated in isolation; the balance between time efficiency and testing effectiveness is key.

By comparing these aspects of EPMOSA and DynaMOSA, we can determine which algorithm offers the best trade-off between speed and efficacy. This provides insights into their overall practicality for software testing environments. This comprehensive analysis helps inform decisions on selecting appropriate testing tools based on specific project needs and constraints.

6.3.1.1 Commons-Lang

6.3.1.1.1 Generation Time Analysis

The generation times for single models configured with DynaMOSA variants, as shown in Table 6.24, span a range that underscores differences in algorithm efficiency. DynaMOSA$_{+\text{SVM}}$, for instance, has the shortest generation time at 80.18 seconds, suggesting that despite the complexities associated with Support Vector Machines, this configuration generates test cases

efficiently. Conversely, $_{+KNN}$ records the longest generation time at 92.23 seconds, which can likely be attributed to the computational demands of the K-Nearest Neighbors algorithm, involving iterative calculations of distances between data points. Other configurations, including CART, LDA, LR, and NB, display generation times that fall within these extremes, indicating a balance between computational requirements and operational efficiency.

Ensemble-model configurations, on the other hand, generally exhibit shorter generation times compared to most single-model setups, with the notable exception of the DynaMOSA default implementation, which is higher at 94.86 seconds. The ensemble models, such as those involving EPMOSAEP$_{\text{EP-BAST}}$ variants, have generation times ranging from 76.18 to 80.00 seconds. These shorter times suggest effective optimization strategies within ensemble models, where the synergistic effects of multiple algorithms expedite the test case generation process. For example, the EPMOSA$_{\text{EP-BAST}_{(SC)}}$ configuration demonstrates one of the shorter generation times among ensemble models, illustrating how well-designed ensemble strategies can efficiently handle the complexities of integrating multiple testing methods.

This analysis reveals that while ensemble models might demand more execution time, they compensate by generating test cases faster than their single-model counterparts. This efficiency is particularly valuable in dynamic testing environments, where the speed of test generation can significantly influence the overall development and testing cycle. Despite their complexity, the quicker generation times of ensemble models underscore their potential to enhance coverage and fault detection capabilities without substantially extending the testing process.

In conclusion, the generation time analysis highlights the importance of choosing the right SBST model based on specific project requirements. If rapid test generation is a priority, ensemble models offer considerable advantages, making them preferable in scenarios where testing efficiency is as critical as the thoroughness and quality of the testing outcomes. This balance between time efficiency and depth of testing is vital for optimizing SBST strategies to meet diverse project demands effectively.

Table 6.24: Commons-Lang: Test suites execution and generation times

DynaMOSA with single-models	**Avg. Execution Time (s)**	**Avg. Generation Time (s)**
DynaMOSA$_{+CART}$	4.76	86.68
DynaMOSA$_{+KNN}$	5.29	92.23
DynaMOSA$_{+LDA}$	5.15	91.31
DynaMOSA$_{+LR}$	4.95	87.84
DynaMOSA$_{+NB}$	5.18	84.13
DynaMOSA$_{+SVM}$	5.76	80.18
EPMOSA with ensemble-models	**Avg. Execution Time (s)**	**Avg. Generation Time (s)**
EPMOSA$_{\text{EP-BAST}_{(BME)}}$	7.41	80.00
EPMOSA$_{\text{EP-BAST}_{(GBT)}}$	6.54	77.41
EPMOSA$_{\text{EP-BAST}_{(RF)}}$	7.20	77.07
EPMOSA$_{\text{EP-BAST}_{(SC)}}$	6.40	76.18
EPMOSA$_{\text{EP-BAST}_{(VC)}}$	9.81	77.00
DynaMOSA default impl.	**Avg. Execution Time (s)**	**Avg. Generation Time (s)**
DynaMOSA	2.86	94.86

6.3.1.1.2 Execution Time Analysis

The execution time analysis for the Search-Based Software Testing (SBST) configurations used in the commons-lang project provides significant insights into the efficiency of single-model configurations of DynaMOSA compared to the ensemble-model configurations of EPMOSA. This analysis evaluates how different algorithms and their combinations affect the speed of test case execution.

As shown in Table 6.24, the execution times for single-model configurations vary, with $\text{DynaMOSA}_{+\text{SVM}}$ recording the longest time at 5.76 seconds and $\text{DynaMOSA}_{+\text{CART}}$ exhibiting the shortest at 4.76 seconds. The extended execution time for $\text{DynaMOSA}_{+\text{SVM}}$ might be attributed to the complexity and computational demands of the Support Vector Machine algorithm, which, despite offering the potential for high accuracy, tends to be more resource-intensive. Conversely, as implemented in $\text{DynaMOSA}_{+\text{CART}}$, the CART algorithm appears more efficient in execution, suggesting a streamlined process that possibly sacrifices some analytical depth for increased speed.

Contrarily, ensemble models, which integrate multiple strategies to enhance testing efficacy, generally show longer execution times. These times range from 6.40 seconds for $\text{EPMOSA}_{\text{EP-BAST}_{(\text{SC})}}$, the fastest among the ensemble models, to 9.81 seconds for $\text{EPMOSA}_{\text{EP-BAST}_{(\text{VC})}}$, the slowest. This range indicates the computational overhead inherent in ensemble approaches, where the combination of multiple test generation strategies, while potentially improving test coverage and fault detection, necessitates additional processing power and time, as evidenced by these extended execution periods.

The comparative analysis between single and ensemble models indicates that while ensemble models are more robust in their approach, they require more execution time. This suggests that the choice between using a single or an ensemble model should consider the specific testing requirements: if quicker test execution is crucial, single models might be preferable; however, if comprehensive testing that potentially offers deeper insights and improved fault detection is more important, the additional time required by ensemble models could be justified.

This examination of execution times provides a nuanced view of the trade-offs in selecting SBST configurations. It also highlights the importance of tailoring testing strategy choices to align with project-specific goals and constraints, effectively balancing execution efficiency with the thoroughness provided by more complex systems.

6.3.1.2 Commons-Codec

6.3.1.2.1 Generation Time Analysis

In the single-model configurations, the generation times differ, as outlined in Table 6.25, revealing the inherent efficiency of each algorithm. For example, $\text{DynaMOSA}_{+\text{KNN}}$ displays a relatively quick generation time of 80.82 seconds, indicating its capability to rapidly produce test cases, which is particularly advantageous in continuous integration settings where speed is crucial. On the other hand, $\text{DynaMOSA}_{+\text{SVM}}$ and $\text{DynaMOSA}_{+\text{NB}}$ record generation times of 85.83 seconds and 84.10 seconds, respectively, suggesting that these models may involve more complex computations or more substantial data processing, which extends their test case

generation time.

In comparison, the ensemble-model configurations generally show generation times closely aligned with those of the single models, yet slight variations reflect the computational compromises of their more intricate strategies. Notably, $\text{EPMOSA}_{\text{EP-BAST}_{(SC)}}$ achieves an impressive generation time of 66.30 seconds, making it significantly faster than all other models examined. This performance suggests that this particular ensemble configuration effectively utilizes its combined algorithms to optimize test case generation. Other ensemble models, like $\text{EPMOSA}_{\text{EP-BAST}_{(RF)}}$ and $\text{EPMOSA}_{\text{EP-BAST}_{(VC)}}$, with generation times of 88.00 seconds and 79.53 seconds, respectively, demonstrate that while integrating multiple strategies, they maintain a reasonable balance between complexity and speed.

This analysis underscores that the decision between single and ensemble models should consider not only the execution time but also the speed of test case generation. Although ensemble models typically require more time to execute, they do not consistently take longer to generate test cases and can sometimes surpass single models in generation efficiency. This nuanced understanding is crucial for teams needing to balance the urgency of generating tests and the depth of test coverage.

Overall, the generation time analysis emphasizes the capability of both single and ensemble SBST models to produce test cases within acceptable time frames. Project teams must consider these factors and other project-specific needs to select the most suitable SBST approach, ensuring the testing processes are well-integrated with the development cycles and project timelines. Achieving this balance between speed and thoroughness in test generation is essential for optimizing software testing strategies to meet diverse project requirements efficiently.

Table 6.25: Commons-Codec: Test suites execution and generation times

DynaMOSA with single-models	**Avg. Execution Time (s)**	**Avg. Generation Time (s)**
$\text{DynaMOSA}_{\text{+CART}}$	6.27	85.15
$\text{DynaMOSA}_{\text{+KNN}}$	4.57	80.82
$\text{DynaMOSA}_{\text{+LDA}}$	7.18	83.62
$\text{DynaMOSA}_{\text{+LR}}$	7.10	80.71
$\text{DynaMOSA}_{\text{+NB}}$	8.70	84.10
$\text{DynaMOSA}_{\text{+SVM}}$	9.06	85.83
EPMOSA with ensemble-models	**Avg. Execution Time (s)**	**Avg. Generation Time (s)**
$\text{EPMOSA}_{\text{EP-BAST}_{(BME)}}$	8.91	83.35
$\text{EPMOSA}_{\text{EP-BAST}_{(GBT)}}$	8.82	80.75
$\text{EPMOSA}_{\text{EP-BAST}_{(RF)}}$	9.12	88.00
$\text{EPMOSA}_{\text{EP-BAST}_{(SC)}}$	8.74	66.30
$\text{EPMOSA}_{\text{EP-BAST}_{(VC)}}$	8.48	79.53
DynaMOSA default impl.	**Avg. Execution Time (s)**	**Avg. Generation Time (s)**
DynaMOSA	2.42	84.48

6.3.1.2.2 Execution Time Analysis

In the single-model configurations, as outlined in Table 6.25, execution times vary, with $\text{DynaMOSA}_{\text{+KNN}}$ displaying the shortest time at 4.57 seconds. This indicates a relatively efficient execution process for this model, which may be advantageous in scenarios requiring

rapid feedback, such as continuous integration or agile development environments. Conversely, DynaMOSA$_{+SVM}$ shows the longest execution time at 9.06 seconds, suggesting that the computational complexity of the support vector machine algorithm renders it the most resource-intensive among the single-model variants. The execution times for other models like DynaMOSA$_{+CART}$, DynaMOSA$_{+LDA}$, DynaMOSA$_{+LR}$, and DynaMOSA$_{+NB}$ fall between these extremes, which highlights the varied algorithmic efficiency and computational requirements of these approaches.

In contrast, ensemble-model configurations generally exhibit longer execution times than most single models, ranging from 8.48 seconds to 9.12 seconds. This extended execution time suggests that integrating multiple algorithms or features to enhance testing capabilities introduces a higher computational overhead. Among these, EPMOSA$_{\text{EP-BAST}_{(RF)}}$ records the longest execution time at 9.12 seconds, likely reflecting the added complexity of incorporating random forests in an ensemble setting. EPMOSA$_{\text{EP-BAST}_{(SC)}}$ has a somewhat lower execution time at 8.74 seconds, but this still surpasses the times of all single models except for the two slowest.

This comparison between single and ensemble models about execution times suggests that while ensemble models offer robust testing strategies by integrating multiple methods, they also demand more processing time. This trade-off between execution efficiency and the depth of testing provided by ensemble methods is critical for project managers and developers to consider when choosing an appropriate SBST strategy. Decisions should be based on specific project requirements. If quicker test execution is essential, single models might be more suitable; however, if more comprehensive test coverage and fault detection are priorities, the additional execution time required by ensemble models could be a worthwhile investment.

Overall, this analysis underscores the need for a balanced approach to selecting SBST configurations, considering both the speed and thoroughness of testing, to effectively align with project goals and constraints.

6.3.1.3 Commons-Compress

6.3.1.3.1 Generation Time Analysis

In the single-model configurations of DynaMOSA, generation times display considerable variation, as detailed in Table 6.26. This variation reflects the intrinsic computational complexities associated with each algorithm. For example, DynaMOSA$_{+KNN}$, with the longest generation time at 105.98 seconds, illustrates that the K-Nearest Neighbors algorithm may take longer due to its iterative process of finding nearest neighbors within the dataset. Conversely, DynaMOSA$_{+SVM}$, with the shortest generation time among single models at 95.52 seconds, indicates more efficient handling of data and generation of test cases despite the typically complex nature of Support Vector Machines.

In contrast, ensemble models generally demonstrate improved efficiency in test case generation. For instance, EPMOSA$_{\text{EP-BAST}_{(BME)}}$ boasts a generation time of 80.68 seconds, making it the most efficient model among those evaluated, significantly outperforming the single-model variants. This efficiency likely results from the ensemble approach, which combines multiple testing strategies to enhance the test generation process by leveraging synergies between different algorithms. Other ensemble configurations, such as EPMOSA$_{\text{EP-BAST}_{(GBT)}}$, EPMOSA$_{\text{EP-BAST}_{(RF)}}$,

$EPMOSA_{EP\text{-}BAST_{(SC)}}$, and $EPMOSA_{EP\text{-}BAST_{(VC)}}$, also exhibit generation times that are competitive with, if not superior to, most single-model configurations.

The findings indicate that while ensemble models tend to have longer execution times, as seen in separate data analyses, they compensate by generating test cases more rapidly than their single-model counterparts. This efficiency in generation time is particularly crucial in dynamic testing environments where the quick delivery of test cases can significantly influence the overall development and testing cycle. Therefore, the choice between single or ensemble models should be guided by specific testing requirements. If faster test generation is a priority, ensemble models might be more advantageous despite their longer execution times.

The generation time data underscores the trade-offs in choosing between single and ensemble SBST models. Project teams must weigh these factors alongside other project-specific requirements to select the most suitable SBST approach, ensuring that testing strategies are well-aligned with project timelines and the desired thoroughness of testing. Achieving this balance is essential for optimizing software testing strategies to effectively meet the diverse demands of different projects.

Table 6.26: Commons-compress: Identification rates, test suites execution and generation times

DynaMOSA with single-models	**Avg. Execution Time (s)**	**Avg. Generation Time (s)**
$DynaMOSA_{+CART}$	5.67	99.25
$DynaMOSA_{+KNN}$	5.50	105.98
$DynaMOSA_{+LDA}$	6.50	101.72
$DynaMOSA_{+LR}$	6.19	101.68
$DynaMOSA_{+NB}$	6.64	98.95
$DynaMOSA_{+SVM}$	6.95	95.52
EPMOSA with ensemble-models	**Avg. Execution Time (s)**	**Avg. Generation Time (s)**
$EPMOSA_{EP\text{-}BAST_{(BME)}}$	7.41	80.68
$EPMOSA_{EP\text{-}BAST_{(GBT)}}$	7.76	92.55
$EPMOSA_{EP\text{-}BAST_{(RF)}}$	8.20	84.87
$EPMOSA_{EP\text{-}BAST_{(SC)}}$	7.55	82.33
$EPMOSA_{EP\text{-}BAST_{(VC)}}$	9.17	85.64
DynaMOSA default impl.	**Avg. Execution Time (s)**	**Avg. Generation Time (s)**
DynaMOSA	2.43	98.98

6.3.1.3.2 Execution Time Analysis

The execution time analysis for the "Commons-compress" project offers crucial insights into the operational performance of DynaMOSA's single-model configurations and EPMOSA's ensemble-model configurations. This metric is vital for evaluating the computational efficiency of each testing model, highlighting the duration each model requires to execute test cases.

In the single-model configurations, as detailed in Table 6.26, the execution times are relatively moderate but exhibit variability that likely reflects the computational demands of each algorithm. For example, $DynaMOSA_{+LDA}$ and $DynaMOSA_{+LR}$ display execution times of 6.50 and 6.19 seconds, respectively, suggesting a higher computational requirement potentially due to the algorithmic complexity inherent in Linear Discriminant Analysis and Logistic Regression. Interestingly, $DynaMOSA_{+KNN}$ executes relatively quickly at 5.50 seconds despite its longer

generation time, showcasing a notable distinction where a model may take longer to generate but executes more swiftly.

Ensemble models, on the other hand, generally show longer execution times compared to single models, with times ranging from 7.41 to 9.17 seconds. This suggests that integrating multiple algorithms or features to enhance test coverage and fault detection increases execution time. $\text{EPMOSA}_{\text{EP-BAST}_{(VC)}}$, with the highest execution time at 9.17 seconds, likely reflects the complexity and computational overhead required to integrate various components in an ensemble setting. Conversely, $\text{EPMOSA}_{\text{EP-BAST}_{(BME)}}$ represents a more optimized ensemble model, demonstrating a comparatively lower execution time of 7.41 seconds among the ensemble options, yet still slower than any single model.

This comparative analysis between single and ensemble models regarding execution times indicates that while ensemble models provide enhanced robustness in improving test accuracy and coverage, this comes at the cost of increased execution time. This trade-off is critical for developers and project managers when selecting an appropriate testing strategy. If minimizing test execution time is a priority, single models might be preferable. However, if achieving a comprehensive depth of testing and thorough fault detection is more crucial, the additional time required by ensemble models may be justified.

Overall, understanding these execution time dynamics is essential for making informed decisions about the selection of SBST models, balancing between execution efficiency and the thoroughness of testing provided by more complex systems. Aligning these choices with project-specific goals and constraints is critical to optimizing software testing strategies to effectively meet the diverse demands of various projects.

6.3.1.4 Joda-Time

6.3.1.4.1 Generation Time Analysis

In the single-model configurations, generation times exhibit significant variability, as shown in Table 6.27. This range highlights the computational complexities inherent in each algorithm. For instance, $\text{DynaMOSA}_{\text{+KNN}}$ and $\text{DynaMOSA}_{\text{+LR}}$ display some longer generation times, at 119.26 and 130.87 seconds, respectively. The extended duration for $\text{DynaMOSA}_{\text{+KNN}}$ can be attributed to its intensive distance calculations, while $\text{DynaMOSA}_{\text{+LR}}$'s lengthier time is likely due to its iterative process required to converge on the best model fit. Conversely, $\text{DynaMOSA}_{\text{+SVM}}$, known for complex calculations related to margins and support vectors, shows a relatively efficient generation time of 107.41 seconds, suggesting an effective optimization within its SVM implementation for test case generation.

Among the ensemble models, generation times generally indicate a higher efficiency, with most models completing the generation quicker than the slower single models. $\text{EPMOSA}_{\text{EP-BAST}_{(RF)}}$ and $\text{EPMOSA}_{\text{EP-BAST}_{(VC)}}$ feature some of the shortest generation times at 98.06 and 97.13 seconds, respectively. This efficiency implies that integrating multiple strategies enhances the depth and comprehensiveness of the testing process and does so efficiently without significantly extending the test case generation phase. However, $\text{EPMOSA}_{\text{EP-BAST}_{(GBT)}}$ is an exception, with a considerably longer generation time of 190.96 seconds, likely due to a complex ensemble configuration

that may not be as optimized as other models.

The default implementation of DynaMOSA also demonstrates notable efficiency with a generation time of 105.08 seconds, placing it well within the range of more intricate single and ensemble models. This indicates that the baseline DynaMOSA configuration strikes a good balance between efficiency and thoroughness, making it a robust choice for scenarios that require both speed and comprehensive testing.

This analysis emphasizes the importance of considering execution and generation times when selecting appropriate SBST models for a project. While ensemble models often show a capacity for quicker generation, exceptions like $\text{EPMOSA}_{\text{EP-BAST}_{(\text{GBT})}}$ highlight that this is not always the case. Project teams must balance these factors and other considerations, such as test coverage and fault detection capabilities, to choose the most suitable testing strategy. Achieving this balance is crucial for optimizing software testing strategies to meet diverse project demands effectively.

Table 6.27: Joda-time: Identification rates, test suites execution and generation times

DynaMOSA with single-models	**Avg. Execution Time (s)**	**Avg. Generation Time (s)**
$\text{DynaMOSA}_{\text{+CART}}$	20.06	118.73
$\text{DynaMOSA}_{\text{+KNN}}$	10.26	119.26
$\text{DynaMOSA}_{\text{+LDA}}$	11.63	124.00
$\text{DynaMOSA}_{\text{+LR}}$	15.90	130.87
$\text{DynaMOSA}_{\text{+NB}}$	11.67	121.89
$\text{DynaMOSA}_{\text{+SVM}}$	12.00	107.41
EPMOSA with ensemble-models	**Avg. Execution Time (s)**	**Avg. Generation Time (s)**
$\text{EPMOSA}_{\text{EP-BAST}_{(\text{BME})}}$	14.26	107.00
$\text{EPMOSA}_{\text{EP-BAST}_{(\text{GBT})}}$	13.59	190.96
$\text{EPMOSA}_{\text{EP-BAST}_{(\text{RF})}}$	14.44	98.06
$\text{EPMOSA}_{\text{EP-BAST}_{(\text{SC})}}$	17.96	95.62
$\text{EPMOSA}_{\text{EP-BAST}_{(\text{VC})}}$	13.50	97.13
DynaMOSA default impl.	**Avg. Execution Time (s)**	**Avg. Generation Time (s)**
DynaMOSA	5.54	105.08

6.3.1.4.2 Execution Time Analysis

In the single-model configurations of DynaMOSA, the execution times, as reported in Table 6.27, show significant variability. This variability suggests diverse computational demands across the various algorithms. $\text{DynaMOSA}_{\text{+CART}}$, for example, has the longest execution time at 20.06 seconds, potentially indicating a complex decision-tree-building process that requires substantial computational resources. On the other hand, $\text{DynaMOSA}_{\text{+KNN}}$, despite its reputation for computational intensity due to extensive distance calculations among data points, registers a surprisingly lower execution time of 10.26 seconds. This might reflect optimizations within the testing framework or specific data characteristics that simplify computations.

Other single models, such as $\text{DynaMOSA}_{\text{+LDA}}$, $\text{DynaMOSA}_{\text{+LR}}$, $\text{DynaMOSA}_{\text{+NB}}$, and $\text{DynaMOSA}_{\text{+SVM}}$, exhibit varied execution times ranging from 11.63 to 15.90 seconds. These variations likely mirror each algorithm's mathematical and data-handling complexities, including matrix operations in LDA and LR, probabilistic calculations in NB, and margin optimization in

SVM. Despite these complexities, the execution times are moderately paced, showcasing the efficient implementation of these algorithms.

Ensemble models under the EPMOSA banner generally display more consistent execution times, though they are slightly higher on average than some quicker single models. This consistency could be due to the synergistic integration of multiple strategies, which stabilizes execution time despite the increased computational overhead. EPMOSA configurations, such as $\text{EPMOSA}_{\text{EP-BAST}_{(VC)}}$ and $\text{EPMOSA}_{\text{EP-BAST}_{(GBT)}}$, with execution times of 13.50 and 13.59 seconds, respectively, are competitive with single-model times. This performance indicates that while ensemble models incorporate more complex interactions among different algorithms, their execution is optimized to ensure that this complexity does not excessively prolong the execution phase.

Notably, the default implementation of DynaMOSA showcases the shortest execution time of all configurations at 5.54 seconds, highlighting the base algorithm's efficiency without the additional complexities introduced by specialized techniques.

This analysis highlights the trade-offs between single and ensemble SBST models based on execution times. While single models like $\text{DynaMOSA}_{\text{+KNN}}$ may offer quicker execution, they might not provide the comprehensive testing depth achieved by the slightly slower but more thorough ensemble models. Project teams must weigh these considerations against specific project needs to select the most suitable SBST approach, ensuring that the testing strategies align well with project timelines and effectively meet the software's quality demands. This careful balancing act is crucial for optimizing software testing strategies to meet diverse project requirements efficiently.

6.3.2 EPMOSA versus DynaMOSA standard implementation

The standard implementation of DynaMOSA consistently shows the fastest execution times across all evaluated projects, significantly outperforming both the ensemble and single predictive models of EPMOSA and other DynaMOSA variants. This consistent performance highlights DynaMOSA's potential for greater efficiency in executing test suites. While DynaMOSA generally offers lower or comparable generation times to EPMOSA, it does not uniformly surpass the generation speeds of its single predictive model variants. This discrepancy indicates that rapid execution does not necessarily correlate with equally fast test suite generation for DynaMOSA.

These observations provide a nuanced understanding of the trade-offs involved in Search-Based Software Testing (SBST) approaches that utilize actual defect predictors. They underscore DynaMOSA's efficiency, which could be especially beneficial in continuous integration environments where rapid feedback is crucial for maintaining development velocity. The data also present a promising research opportunity: investigating the mechanisms behind DynaMOSA's quick execution times and further exploring ways to enhance these efficiencies.

Although EPMOSA exhibits slower execution times, it performs better bug detection than the original DynaMOSA implementation. This finding is significant as it shows that combining SBST approaches with ensemble bug prediction models can achieve the same or improved

performance compared to more advanced individual techniques. This suggests that integrating ensemble modeling techniques with SBST can enhance bug detection effectiveness without compromising overall testing speed.

These comparative insights emphasize the importance of selecting an SBST model that aligns with specific project requirements. If execution speed is paramount, DynaMOSA might be the preferred choice. However, in scenarios where bugs have a considerable impact, investing in an approach like EPMOSA, which potentially offers higher bug detection rates, could be more advantageous, even if it entails slower execution times.

This analysis not only aids in choosing the most suitable SBST approach but also steers the development of more sophisticated methods to balance speed, efficiency, and effectiveness in SBST. By understanding and adjusting these parameters, project teams can optimize their software testing strategies to meet their projects' unique demands, ensuring both rapid development cycles and high-quality software outputs.

6.4 Threats to Validity

This section examines the potential risks to the validity of the research conducted to accomplish this thesis's primary objective. The validity threats examined in this thesis include construct validity, internal validity, conclusion validity, and external validity.

6.4.1 Construct Validity

In the context of defect prediction, the metrics *recall* and *precision* play crucial roles in assessing the performance of defect predictors. Recall, also known as sensitivity, gauges the proportion of actual buggy entities the predictor correctly identifies. This metric is important because it highlights the model's capacity to minimize false negativesâinstances where a buggy entity is mistakenly classified as non-buggy. Achieving high recall is beneficial as it ensures that fewer bugs go unnoticed.

Precision, on the other hand, measures the accuracy of the predictions by indicating the proportion of entities identified as buggy that are genuinely buggy. It assesses the model's ability to minimize false positivesâcases where a non-buggy entity is wrongly flagged as buggy. In defect prediction, striving for a balance between recall and precision is crucial because high precision means testing resources are not spent on investigating false alarms.

Highly imbalanced datasets can distort both recall and precision. This is a typical situation in defect datasets, where non-buggy entities far outnumber buggy entities. This imbalance can adversely affect the performance metrics, complicating the accurate evaluation of a defect predictorâs effectiveness.

We also employ additional metrics such as Accuracy, F1-Score, ROC AUC, and Log Loss to overcome these challenges and offer a more thorough evaluation. These metrics provide a broader view of the model's performance across different aspects, helping to ensure a more balanced and comprehensive assessment.

6.4.2 Internal Validity

6.4.2.1 Bug Dataset

Internal validity refers to the extent to which our study can demonstrate a cause-and-effect relationship, free from the influence of confounding variables. In this context, one of the possible limitations of our study lies in achieving high internal validity due to the nature of the defined experimental framework, which can be characterized by *limited control* (Ayala et al., 2022).

This limitation originates from using a retrospective repository, Defects4J, as the primary bug dataset. Retrospective repositories contain historical data on software defects, which were not collected or controlled directly by us but are instead pre-existing. As a result, our experimental framework lacks the *hallmark* of control, as identified by Ayala et al. (2022), which is crucial for establishing a robust cause-and-effect relationship.

The hallmark of control refers to our ability to manipulate variables and conditions within the experiment to isolate the effects of the test case generation approaches (independent variable) on the effectiveness and efficiency (dependent variable) of the generated test suites. By relying on a retrospective dataset, the researchers are unable to exert this level of control, thus threatening the internal validity of their findings.

We rely on Defects4J for the accuracy of the datasets we created in our experiments. The Defects4J is a well-regarded repository containing a collection of bugs from real-world Java projects, used extensively in software testing research for evaluating the effectiveness of various testing techniques. However, this reliance introduces potential biases inherent to the dataset. These biases can stem from various factors, such as the selection of projects included in the benchmark, the types of bugs it contains, or the programming practices prevalent in the projects from which the bugs were sourced. Since we do not control the Defects4J collection process, we cannot mitigate these biases directly within our experiments.

It would be necessary to mitigate this threat by using *prospective* repositories. This would allow us to achieve maximum control over the bug dataset biases, creating datasets with a forward-looking approach where bug data collection and curation processes are explicitly designed to minimize biases relevant to our research objectives.

Prospective repositories would allow us to tailor the dataset characteristics, such as the diversity of software projects, bug types, and coding practices, to ensure a more balanced and representative dataset. This approach contrasts with retrospective datasets like Defects4J, where the data is collected from existing sources with inherent biases. However, creating or using prospective repositories has challenges, including significant time and resource investments to curate such datasets.

Currently, most available bug datasets are retrospective repositories (Just et al., 2014, Saha et al., 2018, Madeiral et al., 2019). Ayala et al. (2022) reports a significant challenge related to bug datasets: the scarcity of research using prospective repositories, with only one out of 254 studies opting for this approach.

6.4.2.2 Use of Genetic Algorithms

In Search-Based Software Testing (SBST), genetic algorithms play a crucial role by iteratively evolving test cases to optimize certain objectives, such as maximizing code coverage or bug detection. However, a characteristic feature of genetic algorithms is their inherent randomness in operations like selection, crossover, and mutation. This randomness can lead to variability in the outcomes of test generation runs, making it challenging to consistently assess the effectiveness of SBST approaches.

To address the threat of randomness in genetic algorithms, we conducted multiple test generation runsâfive times. While we acknowledge that five runs are not extensive and fall short of the minimum of 20 runs recommended in the literature, our hardware limitations constrained our ability to perform more. However, the adopted five repetitions still helps mitigate the influence of randomness by providing a broader dataset from which statistical measures can be derived. The performance of EPMOSA is then evaluated based on the mean and median of these runs, representing the central tendencies of the results, offering a more stable and reliable basis for comparison. We are working towards conducting additional experiments with a minimum of 20 runs to adhere to the recommended standards.

Additionally, statistical tests were used to determine the significance of the observed differences in performance. These tests can help identify whether variations in performance metrics are due to the inherent efficacy of the SBST approach or merely a result of the stochastic nature of genetic algorithms. By adopting this methodology, we can ensure a more accurate and fair comparison of EPMOSA, DynaMOSA, and corresponding configurations, thereby enhancing the reliability of conclusions drawn from our experimental evaluations.

6.4.3 External Validity

6.4.3.1 Generalization to Other Software

Our research on EPMOSA focuses on a range of well-known open-source Java projects. However, the results may not apply to other programming languages, software domains, or proprietary systems with different characteristics or complexity. For that, the research strategically selected projects from different domains to test the effectiveness and efficiency of EPMOSA across different software architectures and use cases.

Extensive testing and validation procedures were employed to ensure the reliability of the results and their potential applicability to other Java-based projects beyond those specifically studied. Comparing EPMOSA's performance with traditional SBST techniques and other defect prediction models within the same projects provided a baseline for understanding the new method's performance relative to existing methods.

Statistical analyses were also used to evaluate the significance of the results across different projects, ensuring that the improvements observed with EPMOSA were statistically significant and not due to random variations within the project sample. This approach strengthens the argument for the potential generalization of the findings.

6.4.3.2 Diversity of Defect Predictors

The research uses real defect predictors, but their performance and effectiveness may vary depending on the quality of data and the specific machine learning models used. This variability can affect the accuracy and effectiveness of the EPMOSA approach in guiding Search-Based Software Testing (SBST).

To mitigate this threat, we carefully selected defect predictors validated in previous research and known for their reliability and robustness. Cross-validation techniques were employed to ensure consistent performance across different datasets, dividing the data into subsets and training and testing the models on one subset while testing on another. This helps identify any overfitting or biases in the models and ensures that the predictors are not tailored to specific datasets.

Ensemble bug prediction models were also used, combining multiple machine learning models to make a single predictive decision. Ensemble methods are known for improving prediction performance by reducing variance and bias, which are common issues in single models. By aggregating predictions from various models, ensemble methods enhance the overall reliability and robustness of the predictions, mitigating the threat posed by the variability of individual defect predictors.

6.4.4 Conclusion Validity

To ensure the robustness and reliability of our experimental results' conclusions, we used a comprehensive suite of statistical tests. These tests are designed to mitigate any threats to conclusion validity, which refers to the degree to which conclusions we draw about relationships in our data are reasonable.

The Mann-Whitney U-Test is a non-parametric test used to compare differences between two independent groups when the dependent variable is ordinal or continuous but not normally distributed. The two-way ANOVA test assesses the effect of two independent variables on a dependent variable and can identify interactions between them. These statistical methods provide a rigorous framework for analyzing experimental data, ensuring that the conclusions drawn are statistically sound and reflect true effects rather than artifacts of the data or methodology.

6.5 Final Considerations

The empirical evaluation conducted in this chapter provides strong evidence supporting the use of ensemble predictive models to guide Search-Based Software Testing (SBST). The results show that EPMOSA significantly outperforms traditional SBST approaches and single-model prediction strategies in terms of bug detection rates and testing efficiency. By intelligently focusing testing efforts on the most defect-prone areas of the code, EPMOSA optimizes resource allocation and enhances the overall effectiveness of the testing process. These findings highlight the practical advantages of integrating ensemble models into SBST and underscore the potential for further research and development in this area. The insights gained from this evaluation pave the way for future work aimed at refining and extending the application of ensemble prediction

models in software testing, ultimately contributing to more reliable and robust software systems.

CHAPTER

7

Conclusions and Future Work

Our research aims to develop and validate the effectiveness and efficiency of ensemble bug prediction models to guide approaches to automatic test case generation. Ensemble models can help to predict the most defect-prone areas of the code, allowing testing efforts to be targeted to the areas most likely to have the greatest impact. Our research demonstrates that it is possible to generate more effective test cases by incorporating ensemble bug prediction models into the test case generation process.

The study's results indicate that using bug prediction to direct the generation of test cases can improve resource allocation in software testing activities. If one targets areas that are more likely to contain bugs, testing efforts can be concentrated more effectively, thereby optimizing both time and computational resources.

Through the use of several open-source Java projects, we carried out an empirical validation. When compared to traditional SBST implementations that do not make use of bug prediction models or its variations with single-model defect predictors, the results demonstrate that the proposed SBST approach – EPMOSA – and time budget allocation strategy EP-BAST have the potential to enhance the capabilities of bug detection techniques significantly.

The Ensemble Predictive Multi-Objective Sorting Algorithm (EPMOSA) is a novel automatic test case generation approach that uses ensemble bug prediction models. To validate this methodology's impact on improving the quality of software testing processes, it was subjected to rigorous testing across a variety of software projects and our findings demonstrate its effectiveness and efficiency.

The bug identification rate of EPMOSA is the primary metric used to evaluate the effectiveness of the generated test suites. This rate directly correlates with the test suites' capability to identify actual bugs in the software. Based on the empirical findings, it is evident that EPMOSA consistently achieves higher rates of bug identification compared to traditional methods and single prediction models. For example, in the Joda-Time project, EPMOSA demonstrated a mean bug identification rate of 35.23%, higher than the 31.03% achieved by single prediction

model setups. This enhancement in bug detection capability is a proof of the effectiveness of EPMOSA, as it leverages the combined strengths of various prediction models to concentrate testing efforts on the areas of the code that are the most problematic.

One reason for EPMOSA's success is its sophisticated ensemble approach. EPMOSA incorporates multiple predictive models to predict defect-prone areas with a higher degree of precision. Ensemble models aggregate predictions to reduce false negatives and positives, ensuring a more comprehensive coverage of bug-prone zones. On the other hand, single models have biases or limitations in identifying all potential bugs.

Regarding effectiveness, EPMOSA stands out due to the strategic utilization of computational resources that it takes advantage of. The algorithm prioritizes the generation of test cases in areas predicted to be more prone to bugs, allowing it to allocate time and effort in the most efficient manner possible. This minimizes the amount of wasted computational effort on less important parts of the code and maximizes the impact of the testing process. When it comes to large-scale software projects, where testing time and computational power are important, it is crucial to use the available resources efficiently.

EPMOSA's capability to adjust the search depth adaptively based on the predicted risk levels of various code sections is another demonstration of its applicability. Through this dynamic allocation of resources, higher-risk areas are guaranteed to undergo more intensive testing, which increases the likelihood of discovering subtle or complex bugs that may have been missed by either simpler or less intensive testing.

An additional component of our findings is a comparison of EPMOSA with other approaches, such as DynaMOSA. There is a preference for EPMOSA in terms of performance metrics, which highlights its capacity to manage and exploit ensemble predictions effectively. These results were validated through statistical tests such as analysis of variance (ANOVA) and Mann-Whitney U. These tests confirmed that the differences in performance are statistically significant and are not the result of random variations in the data.

Histograms and density plots are two examples of visual aids that clearly illustrate how EPMOSA distributes its bug identification rates across a variety of test scenarios and projects. These visualizations help us better understand EPMOSA's consistency and reliability under various conditions and configurations and also provide insights into the proposed approach's robustness and adaptability.

EPMOSA is an innovative advance in integrating defect prediction models with Search-Based Software Testing (SBST) approaches. It represents a significant leap forward in developing practical software testing methodologies. This novel approach is the first to use real defect predictors in guiding SBST techniques across well-known open-source systems. As a result, it establishes a new benchmark regarding the efficiency and applicability of automated testing techniques.

The significance of EPMOSA lies in its use of ensemble defect prediction models, a novel approach that uses collective insights from multiple predictive algorithms to identify the components of the code most susceptible to vulnerabilities. EPMOSA can strategically concentrate the

SBST process on regions prone to defects and is likely to produce the most significant impact on bug detection due to this action. This targeted approach optimizes testing resources and enhances the effectiveness of testing efforts, a method not previously realized in SBST applications with real defect predictors.

The use of EPMOSA across various well-known open-source projects demonstrates the robustness and adaptability of the software in application to real-world scenarios. This study's results provide empirical evidence demonstrating its superiority to conventional SBST methods that do not make use of defect prediction. It is possible to ensure that EPMOSA's strategies are grounded in practical, actionable data by employing real defect predictors rather than theoretical or simulated models. This helps to reduce the gap between theoretical research and practical application significantly. Because of the direct application's ability to improve the relevance and utility of the testing process, it is a pioneering solution for developers looking to improve the reliability and efficiency of their software testing protocols.

Moreover, the implementation of EPMOSA as a channel for applying real defect predictors in SBST techniques addresses challenges that have been present in the industry for a long time. These challenges include the efficient allocation of testing resources and the requirement for focused testing in areas that are prone to defects. By validating this approach in widely used open-source systems, EPMOSA demonstrates that it is effective and establishes a precedent for future research and development in SBST. This encourages further innovations and refinements in this essential area of software engineering.

EPMOSA's pioneering application of real defect predictors to guide SBST techniques represents a transformative development in software testing. It provides a compelling and proven solution that improves bug detection capabilities and efficiency and promises substantial improvements in software quality and reliability for developers worldwide. This breakthrough paves the way for more intelligent and resource-efficient testing strategies directly aligned with software development challenges' realities. It also establishes a new standard for integrating defect prediction with SBST.

Although we are working towards conducting additional experiments with a minimum of 20 runs to adhere to the recommended standards, the text was maintained as presented to the doctoral evaluation committee.

7.0.1 Recommendations for Future Work

Given the nature of our research on EPMOSA and the integration of ensemble defect prediction models with search-based software testing (SBST), future work can focus on several areas to extend the utility and applicability of our findings. Our recommendations for future work aim to expand the empirical base, refine the methodology, and explore new applications and improvements.

Expanding the scope of test environments and configurations could provide a deeper understanding of EPMOSA's adaptability. Incorporating a more diverse range of software projects, including commercial and proprietary software in addition to open-source projects, would validate EPMOSA's robustness under different development paradigms. This expansion could also

include testing in real-time systems or safety-critical applications where defect prediction and resolution are crucial.

Secondly, enhancing the ensemble models used in EPMOSA could increase the precision of defect predictions. Future work could explore integrating more advanced machine learning techniques, such as deep learning and reinforcement learning, which might uncover patterns in bug occurrence that are not detectable with current prediction models. Additionally, incorporating feedback loops that allow the system to learn and adapt from past testing cycles could improve the effectiveness of the predictions over time.

Furthermore, a comparative analysis of EPMOSA with other emerging technologies in software testing, like Artificial Intelligence (AI) powered dynamic testing tools, could be valuable. Such studies would help position EPMOSA within the broader landscape of automated software testing and might lead to hybrid approaches that leverage the strengths of various methodologies.

Another area of interest could be the development of a user-friendly interface or a development toolkit that integrates EPMOSA seamlessly into common development environments. This would facilitate its adoption by practitioners and could lead to broader industry acceptance and practical feedback, which in turn could drive further refinements.

Lastly, conducting longitudinal studies on the long-term impacts of implementing EPMOSA in regular software development cycles could provide insights into software quality, developer productivity, and overall project costs. This would involve tracking projects over several years to gather data on the reliability and maintenance costs associated with software tested using EPMOSA, comparing these to projects using traditional SBST approaches.

By pursuing these avenues, future research can build on the strong foundation laid by our study, further proving the value of EPMOSA in improving software testing processes and enhancing the quality and reliability of software systems globally. These efforts will strengthen the theoretical underpinnings of EPMOSA and enhance its practical applications, making it a cornerstone of modern software testing techniques.

Bibliography

Economic perspectives in test automation: balancing automated and manual testing with opportunity cost, ACM, 2006.

A search-based approach for cost-effective software test automation decision support and an industrial case study, IEEE Computer Society, 2014.

Search based software engineering. *Lecture Notes in Computer Science*, 2016. DisponÃvel em http://dx.doi.org/10.1007/978-3-319-47106-8

Correlations of software code metrics: an empirical study, p. 255–266. 2017.

Evaluating and comparing size, complexity and coupling metrics as web applications vulnerabilities predictors. *International Journal of Information Technology and Computer Science*, v. 11, n. 7, p. 35–42, 2019.

ADAM, S. P.; ALEXANDROPOULOS, S.-A. N.; PARDALOS, P. M.; VRAHATIS, M. N. *No free lunch theorem: A review* Cham: Springer International Publishing, p. 57–82, 2019. DisponÃvel em https://doi.org/10.1007/978-3-030-12767-1_5

AFAN, H. A.; ZEB, A.; DIN, F.; FAYAZ, M.; MEHMOOD, G.; ZAMLI, K. Z. A systematic literature review on robust swarm intelligence algorithms in search-based software engineering. *Complexity*, v. 2023, p. 4577581, 2023.

AL-SALEEM, A. L. .; HAMMO, A. Y. Software size estimation: A survey. *Technium: Romanian Journal of Applied Sciences and Technology*, v. 4, n. 9, p. 62â70, 2022. DisponÃvel em https://techniumscience.com/index.php/technium/article/view/7251

ALAM, T.; QAMAR, S.; DIXIT, A.; BENAIDA, M. Genetic algorithm: Reviews, implementations, and applications. *International Journal of Engineering Pedagogy (iJEP)*, v. 10, n. 6, p. 57, 2020.
DisponÃvel em http://dx.doi.org/10.3991/ijep.v10i6.14567

ALI, S.; SAMAD, A.; MAHDIN, H. B.; KAZMI, R.; IBRAHIM, R.; BAHARUM, Z. Multiobjective test case prioritization using test case effectiveness: Multicriteria scoring method. *Scientific Programming*, v. 2021, 2021.

ALMOMANY, A.; AYYAD, W. R.; JARRAH, A. Optimized implementation of an improved knn classification algorithm using intel fpga platform: Covid-19 case study. *Journal of King Saud University - Computer and Information Sciences*, v. 34, n. 6, Part B, p. 3815–3827, 2022.
DisponÃvel em https://www.sciencedirect.com/science/article/pii/S1319157822001239

ALSABTI, K.; RANKA, S.; SINGH, V. Clouds: a decision tree classifier for large datasets. In: *Proceedings of the Fourth International Conference on Knowledge Discovery and Data Mining*, KDD'98, AAAI Press, 1998, p. 2â8 (*KDD'98*,).

ALTAY, A.; CINAR, D. *Fuzzy decision trees* Cham: Springer International Publishing, p. 221–261, 2016.
DisponÃvel em https://doi.org/10.1007/978-3-319-39014-7_13

AMMANN, P.; OFFUTT, J. *Introduction to software testing*. 1 ed. USA: Cambridge University Press, 2008.

AN, L.; KHOMH, F. An empirical study of crash-inducing commits in mozilla firefox. In: *Proceedings of the 11th International Conference on Predictive Models and Data Analytics in Software Engineering*, New York, NY, USA: Association for Computing Machinery, 2015 (*PROMISE 2015*, v.").
DisponÃvel em https://doi.org/10.1145/2810146.2810152

ANDRADE, L.; MACHADO, P. D. L.; ANDRADE, W. Can operational profile coverage explain postârelease bug detection? *Software Testing*, v. 30, 2020.

ARCURI, A. It really does matter how you normalize the branch distance in search-based software testing. *Software Testing, Verification and Reliability*, v. 23, n. 2, p. 119–147, 2013.
DisponÃvel em https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.457

ARCURI, A.; GALEOTTI, J. P. Enhancing search-based testing with testability transformations for existing apis. *ACM Trans. Softw. Eng. Methodol.*, v. 31, n. 1, 2021.
DisponÃvel em https://doi.org/10.1145/3477271

AVDEENKO, T.; SERDYUKOV, K. Genetic algorithm fitness function formulation for test data generation with maximum statement coverage. In: TAN, Y.; SHI, Y., eds. *Advances in Swarm Intelligence*, Cham: Springer International Publishing, 2021, p. 379–389.

AYALA, C.; TURHAN, B.; FRANCH, X.; JURISTO, N. Use and misuse of the term âexperimentâ in mining software repositories research. *IEEE Transactions on Software Engineering*, v. 48, n. 11, p. 4229–4248, 2022.

BA, K.; CHARTERS, S. Guidelines for performing systematic literature reviews in software engineering. v. 2, 2007.

BABU, S.; SINGH, R. Impact of inheritance and reusability on design quality of object oriented system. In: *2022 22nd International Conference on Computational Science and Its Applications (ICCSA)*, 2022, p. 35–40.

BAHAWERES, R. B.; ZAWAWI, K.; KHAIRANI, D.; HAKIEM, N. Analysis of statement branch and loop coverage in software testing with genetic algorithm. In: *2017 4th International Conference on Electrical Engineering, Computer Science and Informatics (EECSI)*, 2017, p. 1–6.

BANSAL, J. C. Particle swarm optimization. *Studies in Computational Intelligence*, p. 11â23, 2018.
DisponÃvel em http://dx.doi.org/10.1007/978-3-319-91341-4_2

BARANI, M.; LABICHE, Y.; ROLLET, A. On factors that impact the relationship between code coverage and test suite effectiveness: a survey. In: *2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2023, p. 381–388.

BASSEUR, M.; GOEFFON, A. Climbing combinatorial fitness landscapes. *Applied Soft Computing*, v. 30, p. 688â704, 2015.
DisponÃvel em http://dx.doi.org/10.1016/j.asoc.2015.01.047

BERTSIMAS, D.; KING, A. Logistic Regression: From Art to Science. *Statistical Science*, v. 32, n. 3, p. 367 – 384, 2017.
DisponÃvel em https://doi.org/10.1214/16-STS602

BEZBARUAH, A.; PRATAP, B.; HAKE, S. B. Automation of tests and comparative analysis between manual and automated testing. In: *2020 IEEE Students Conference on Engineering Systems (SCES)*, 2020, p. 1–5.

BHUTAMAPURAM, U. S. Some investigations of machine learning models for software defects. In: *2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, 2023, p. 259–263.

BISHOP, C. M.; BISHOP, H. *Gradient descent* Cham: Springer International Publishing, p. 209–232, 2024.
DisponÃvel em https://doi.org/10.1007/978-3-031-45468-4_7

BO ZHANG; CHEN WANG Automatic generation of test data for path testing by adaptive genetic simulated annealing algorithm. In: *2011 IEEE International Conference on Computer Science and Automation Engineering*, 2011, p. 38–42.

BOGNÁR, L.; FAUSZT, T. Factors and conditions that affect the goodness of machine learning models for predicting the success of learning. *Computers and Education: Artificial Intelligence*, v. 3, p. 100100, 2022.
DisponÃvel em https://www.sciencedirect.com/science/article/pii/S2666920X22000558

BOUKIR, S.; FENG, W. Boundary bagging to address training data issues in ensemble classification. In: *2020 25th International Conference on Pattern Recognition (ICPR)*, 2021, p. 9975–9981.

BREIMAN, L. Bagging predictors. *Machine Learning*, v. 24, n. 2, p. 123–140, 1996.

BREIMAN, L. Random forests. *Machine Learning*, v. 45, n. 1, p. 5â32, 2001.
DisponÃvel em https://doi.org/10.1023/a:1010933404324

BREIMAN, L.; FRIEDMAN, J.; STONE, C.; OLSHEN, R. *Classification and regression trees*. Taylor & Francis, 1984.
DisponÃvel em https://books.google.com.br/books?id=JwQx-WOmSyQC

CAMBRIDGE, U. Cambridge university study states software bugs cost economy $312 billion per year. 2013.
DisponÃvel em https://www.prweb.com/releases/2013/1/prweb10298185.htm

DE CAMPOS, J. C. M. Search-based unit test generation for evolving software, 2017.
DisponÃvel em https://etheses.whiterose.ac.uk/20271/

CANTORO, R.; CETRULO, E.; SANCHEZ, E.; REORDA, M. S.; VOZA, A. Automated test program reordering for efficient sbst. In: *2017 32nd Conference on Design of Circuits and Integrated Systems (DCIS)*, 2017, p. 1–6.

CHALLET, D.; VALVERDE, S. Fat tails, long memory, maturity and ageing in open-source software projects. IEEE, 2008.

CHATTERJEE, A.; GHOSH, S. A review of bootstrap methods in ranked set sampling. *Ranked Set Sampling Models and Methods*, p. 171â189, 2022.
DisponÃvel em http://dx.doi.org/10.4018/978-1-7998-7556-7.ch008

CHEN, J.; CHEN, H.; GUO, Y.; ZHOU, M.; HUANG, R.; MAO, C. A novel test case generation approach for adaptive random testing of object-oriented software using

k-means clustering technique. *IEEE Transactions on Emerging Topics in Computational Intelligence*, v. 6, n. 4, p. 969–981, 2022a.

CHEN, J.; YIN, Y.; CAI, S.; GENG, Y.; HUANG, L. An improved test case generation method based on test requirements for testing software component. In: *2022 IEEE 22nd International Conference on Software Quality, Reliability, and Security Companion (QRS-C)*, 2022b, p. 209–218.

CHICKERING, D.; MEEK, C.; ROUNTHWAITE, R. Efficient determination of dynamic split points in a decision tree. In: *Proceedings 2001 IEEE International Conference on Data Mining*, 2001, p. 91–98.

CHOPRA, D.; KHURANA, R. Decision trees. *Introduction to Machine Learning with Python*, p. 74â82, 2023.
DisponÃvel em http://dx.doi.org/10.2174/9789815124422123010007

CHOWDHURY, I.; ZULKERNINE, M. Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities. *Journal of Systems Architecture*, v. 57, n. 3, p. 294â313, 2011.
DisponÃvel em https://www.sciencedirect.com/science/article/abs/pii/S1383762110000615

CHOWDHURY, S. A.; UDDIN, G.; HOLMES, R. An empirical study on maintainable method size in java. MSR '22, New York, NY, USA: Association for Computing Machinery, 2022, p. 252â264 (*MSR '22*,).
DisponÃvel em https://doi.org/10.1145/3524842.3527975

COLE SMITH, J.; COCHRAN, J. J.; GENDREAU, M.; POTVIN, J. Tabu search. *Wiley Encyclopedia of Operations Research and Management Science*, p. 1â12, 2011.
DisponÃvel em http://dx.doi.org/10.1002/9780470400531.eorms1045.pub2

CROARKEN, M. The beginnings of the manchester computer phenomenon: People and influences. *IEEE Ann. Hist. Comput.*, v. 15, n. 3, p. 9â16, 1993.
DisponÃvel em https://doi.org/10.1109/85.222835

CROSBY, P. B. *Quality is free: The art of making quality certain*. New York: McGraw-Hill, 1979.
DisponÃvel em https://archive.org/details/qualityisfree00cros

DAM, H. K.; TRAN, T.; PHAM, T.; NG, S. W.; GRUNDY, J.; GHOSE, A. Automatic feature learning for predicting vulnerable software components. *IEEE Transactions on Software Engineering*, v. 47, n. 1, p. 67–85, 2021.

DE SILVA, D. I.; KODAGODA, N.; PERERA, H. Applicability of three complexity metrics. In: *International Conference on Advances in ICT for Emerging Regions (ICTer2012)*, 2012, p. 82–88.

DEMARCO, F.; XUAN, J.; LE BERRE, D.; MONPERRUS, M. Automatic repair of buggy if conditions and missing preconditions with smt. In: *Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis*, New York, NY, USA: Association for Computing Machinery, 2014, p. 30Â39 (*CSTVA 2014*, v.").
DisponÃvel em https://doi.org/10.1145/2593735.2593740

DERAKHSHANFAR, P.; DEVROEY, X.; ZAIDMAN, A. Basic block coverage for search-based unit testing and crash reproduction. *Empirical Software Engineering*, v. 27, n. 7, p. 192, 2022.

DI NUCCI, D.; PANICHELLA, A.; ZAIDMAN, A.; DE LUCIA, A. A test case prioritization genetic algorithm guided by the hypervolume indicator. *IEEE Transactions on Software Engineering*, v. 46, n. 6, p. 674–696, 2020.

DI PENTA, M.; TAMBURRI, D. A. Combining quantitative and qualitative studies in empirical software engineering research. In: *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, 2017, p. 499–500.

DI POMPEO, D.; TUCCI, M. Search budget in multi-objective refactoring optimization: a model-based empirical study. In: *2022 48th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 2022, p. 406–413.

DIJKSTRA, E. W. *Chapter i: Notes on structured programming* GBR: Academic Press Ltd., p. 1â82, 1972.

DOGAN, A.; BIRANT, D. A weighted majority voting ensemble approach for classification. *2019 4th International Conference on Computer Science and Engineering (UBMK)*, p. 1–6, 2019.

DORIGO, M.; DI CARO, G. Ant colony optimization: a new meta-heuristic. In: *Proceedings of the 1999 Congress on Evolutionary Computation-CEC99 (Cat. No. 99TH8406)*, 1999, p. 1470–1477 Vol. 2.

DOWSON, M. The ariane 5 software failure. *SIGSOFT Softw. Eng. Notes*, v. 22, n. 2, p. 84, 1997.
DisponÃvel em https://doi.org/10.1145/251880.251992

DUAN, Z.; LI, Y.; MA, P.; GOU, X.; YANG, S. A multi-layer fault triggering framework based on evolutionary strategy guided symbolic execution for automated test case generation. In: *2022 IEEE 22nd International Conference on Software Quality, Reliability, and Security Companion (QRS-C)*, 2022, p. 255–262.

DUQUE, J.; MOREIRA, J. J.; COSTA, J. Data mining to support decision-making—a research approach. In: NAGAR, A. K.; SINGH JAT, D.; MISHRA, D. K.; JOSHI, A., eds. *Intelligent Sustainable Systems*, Singapore: Springer Nature Singapore, 2023, p. 553–563.

DUVIVIER, D.; PREUX, P.; TALBI, E. G. Climbing up np-hard hills. In: VOIGT, H.-M.; EBELING, W.; RECHENBERG, I.; SCHWEFEL, H.-P., eds. *Parallel Problem Solving from Nature — PPSN IV*, Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, p. 574–583.

EROKHIN, S. D.; BORISENKO, B. B.; MARTISHIN, I. D.; FADEEV, A. S. The dataset features selection for detecting and classifying network attacks. In: *2022 Systems of Signal Synchronization, Generating and Processing in Telecommunications (SYNCHROINFO)*, 2022, p. 1–8.

VON EYE, A.; WIEDERMANN, W. *Analysis of variance (anova)* Cambridge University Press, p. 124â161, 2023.

FAN, S.; YAO, N.; WAN, L.; MA, B.; ZHANG, Y. An evolutionary generation method of test data for multiple paths based on coverage balance. *IEEE Access*, v. 9, p. 86759–86772, 2021.

FIELDSEND, J. E.; CHUGH, T.; ALLMENDINGER, R.; MIETTINEN, K. A visualizable test problem generator for many-objective optimization. *IEEE Transactions on Evolutionary Computation*, v. 26, n. 1, p. 1–11, 2022.

FOSTER, J. A. Evolutionary computation. *Nature Reviews Genetics*, v. 2, n. 6, p. 428–436, 2001.
DisponÃvel em https://doi.org/10.1038/35076523

FRASER, G.; ARCURI, A. Evosuite: automatic test suite generation for object-oriented software. In: *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, New York, NY, USA: Association for Computing Machinery, 2011, p. 416â419 (*ESEC/FSE '11*,).
DisponÃvel em https://doi.org/10.1145/2025113.2025179

FRASER, G.; ARCURI, A. Whole test suite generation. *IEEE Transactions on Software Engineering*, v. 39, n. 2, p. 276–291, 2013.

GAO, Q.; XIONG, Y.; MI, Y.; ZHANG, L.; YANG, W.; ZHOU, Z.; XIE, B.; MEI, H. Safe memory-leak fixing for c programs. In: *Proceedings of the 37th International Conference on Software Engineering*, IEEE Press, 2015, p. 459Â470 (*ICSE 2015*, v.'1').

GAO, Q.; ZHANG, H.; WANG, J.; XIONG, Y.; ZHANG, L.; MEI, H. Fixing recurring crash bugs via analyzing q a sites (t). In: *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015, p. 307–318.

GARAI, G. Application of genetic algorithm in numerous scientific fields. *Genetic Algorithms*, 2022.
DisponÃvel em http://dx.doi.org/10.5772/intechopen.105740

GIL, Y.; LALOUCHE, G. On the correlation between size and metric validity. *Empirical Software Engineering*, v. 22, n. 5, p. 2585–2611, 2017a.

GIL, Y.; LALOUCHE, G. On the correlation between size and metric validity. *Empirical Software Engineering*, v. 22, n. 5, p. 2585â2611, 2017b.
DisponÃvel em http://dx.doi.org/10.1007/s10664-017-9513-5

GOCHEVA-ILIEVA, S.; KULINA, H.; YORDANOVA, A. Stacking machine learning models using factor analysis to predict the output laser power. In: *2022 International Conference on Electrical, Computer, Communications and Mechatronics Engineering (ICECCME)*, 2022, p. 1–6.

GOEL, L.; GUPTA, S. *Cross projects defect prediction modeling* Cham: Springer International Publishing, p. 1–21, 2020.
DisponÃvel em https://doi.org/10.1007/978-3-030-25797-2_1

GOLDBERG, D. E.; DEB, K. A comparative analysis of selection schemes used in genetic algorithms. *Foundations of Genetic Algorithms*, p. 69â93, 1991.
DisponÃvel em http://dx.doi.org/10.1016/b978-0-08-050684-5.50008-2

GONG, D.; SU, X.; WANG, T.; MA, P.; YU, W. State dependency probabilistic model for fault localization. *Information and Software Technology*, v. 57, p. 430 – 445, 2015.
DisponÃvel em http://www.sciencedirect.com/science/article/pii/S0950584914001396

GONG, Y.; LIU, G.; XUE, Y.; LI, R.; MENG, L. A survey on dataset quality in machine learning. *Information and Software Technology*, v. 162, p. 107268, 2023.
DisponÃvel em https://www.sciencedirect.com/science/article/pii/S0950584923001222

GOYAL, S. Heterogeneous stacked ensemble classifier for software defect prediction. In: *2020 Sixth International Conference on Parallel, Distributed and Grid Computing (PDGC)*, 2020, p. 126–130.

GOYAL, S. Effective software defect prediction using support vector machines (svms). *International Journal of System Assurance Engineering and Management*, v. 13, n. 2, p. 681–696, 2022.

GOYAL, S.; BHATIA, P. K. Heterogeneous stacked ensemble classifier for software defect prediction. *Multimedia Tools and Applications*, v. 81, n. 26, p. 37033–37055, 2022.

GUI, J.; CAO, Y.; QI, H.; LI, K.; YE, J.; LIU, C.; XU, X. Fast knn search in weighted hamming space with multiple tables. *IEEE Transactions on Image Processing*, v. 30, p. 3985–3994, 2021.

GUPTA, M.; BATHLA, D. Comparative study of software testing technique using manually and automated way. *International Journal of Scientific Research in Science and Technology*, p. 384–394, 2022.

GUPTA, S.; CHOPRA, S.; ARORA, M. Implementation of efficient test case optimization technique using meta-heuristic algorithm. In: *2021 9th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions) (ICRITO)*, 2021, p. 1–4.

HARALDSSON, S. O.; WOODWARD, J. R.; BROWNLEE, A. E. I.; SIGGEIRSDOTTIR, K. Fixing bugs in your sleep: How genetic improvement became an overnight success. In: *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, New York, NY, USA: Association for Computing Machinery, 2017, p. 1513Â1520 (*GECCO 2017*, v.").
DisponÃvel em https://doi.org/10.1145/3067695.3082517

HARMAN, M.; JONES, B. F. Search-based software engineering. *Information and Software Technology*, v. 43, n. 14, p. 833–839, 2001.
DisponÃvel em https://www.sciencedirect.com/science/article/pii/S0950584901001896

HARMAN, M.; MANSOURI, S. A.; ZHANG, Y. Search-based software engineering: Trends, techniques and applications. *ACM Comput. Surv.*, v. 45, n. 1, 2012.
DisponÃvel em https://doi.org/10.1145/2379776.2379787

HENAUX, V.; GOEFFON, A.; SAUBION, F. Evolution of deterministic hill-climbers. In: *2020 IEEE 32nd International Conference on Tools with Artificial Intelligence (ICTAI)*, 2020, p. 564–571.

HENDERSON, D.; JACOBSON, S. H.; JOHNSON, A. W. *The theory and practice of simulated annealing* Boston, MA: Springer US, p. 287–319, 2003a.

HENDERSON, D.; JACOBSON, S. H.; JOHNSON, A. W. *The theory and practice of simulated annealing* Boston, MA: Springer US, p. 287–319, 2003b.
DisponÃvel em https://doi.org/10.1007/0-306-48056-5_10

HERNÁNDEZ-MOLINOS, M. J.; SÁNCHEZ-GARCÍA, A. J.; BARRIENTOS-MARTÍNEZ, R. E.; PÉREZ-ARRIAGA, J. C.; OCHARÁN-HERNÁNDEZ, J. O. Software defect prediction with bayesian approaches. *Mathematics*, v. 11, n. 11, 2023.
DisponÃvel em https://www.mdpi.com/2227-7390/11/11/2524

HERNANDO, L.; MENDIBURU, A.; LOZANO, J. A. Hill-climbing algorithm: Let's go for a walk before finding the optimum. In: *2018 IEEE Congress on Evolutionary Computation (CEC)*, 2018, p. 1–7.

HERZIG, K.; JUST, S.; ZELLER, A. The impact of tangled code changes on defect prediction models. *Empirical Software Engineering*, v. 21, 2015.

HONG, S.; LEE, B.; KWAK, T.; JEON, Y.; KO, B.; KIM, Y.; KIM, M. Mutation-based fault localization for real-world multilingual programs. In: *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, IEEE Press, 2015, p. 464Â475 (*ASE 2015*, v.").
DisponÃvel em https://doi.org/10.1109/ASE.2015.14

HOSSAIN, S. B.; DWYER, M. B.; ELBAUM, S.; NGUYEN-TUONG, A. Measuring and mitigating gaps in structural testing. In: *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, p. 1712–1723.

HUA, J.; ZHANG, M.; WANG, K.; KHURSHID, S. Towards practical program repair with on-demand candidate generation. In: *Proceedings of the 40th International Conference on Software Engineering*, New York, NY, USA: Association for Computing Machinery, 2018, p. 12Â23 (*ICSE 2018*, v.").
DisponÃvel em https://doi.org/10.1145/3180155.3180245

HUANG, R.; SUN, W.; XU, Y.; CHEN, H.; TOWEY, D.; XIA, X. A survey on adaptive random testing. *IEEE Transactions on Software Engineering*, v. 47, n. 10, p. 2052–2083, 2021.

IBRAHIMZADA, A. R.; VARLI, Y.; TEKINOGLU, D.; JABBARVAND, R. Perfect is the enemy of test oracle. In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, New York, NY, USA: Association for Computing Machinery, 2022, p. 70â81 (*ESEC/FSE 2022*,).
DisponÃvel em https://doi.org/10.1145/3540250.3549086

IHSAN AQUIL, M. A. Predicting software defects using machine learning techniques. *International Journal of Advanced Trends in Computer Science and Engineering*, v. 9, n. 4, p. 6609â6616, 2020.
DisponÃvel em http://dx.doi.org/10.30534/ijatcse/2020/352942020

INOZEMTSEVA, L. Understanding the Software Fault Introduction Process. 2015, p. 843–846.

J. HARIKIRAN, B. SAI CHANDANA, B. S. B. R. T. S. R. Software defect prediction based ensemble approach. *Computer Systems Science and Engineering*, v. 45, n. 3,

p. 2313–2331, 2023.
DisponÃvel em http://www.techscience.com/csse/v45n3/50727

JAAFAR, F.; GUÃ©HÃ©NEUC, Y.-G.; HAMEL, S.; KHOMH, F.; ZULKERNINE, M. Evaluating the impact of design pattern and anti-pattern dependencies on changes and faults. *Empirical Software Engineering*, 2015.

JANGRA, R.; SANGWAN, O. P.; NANDAL, D. A novel approach for software effort estimation using optimized ck metrics. In: *2022 Fifth International Conference on Computational Intelligence and Communication Technologies (CCICT)*, 2022, p. 505–513.

JIN, R.; AGRAWAL, G. *Communication and memory efficient parallel decision tree construction* p. 119–129, 2003.
DisponÃvel em https://epubs.siam.org/doi/abs/10.1137/1.9781611972733.11

JONES, B.; STHAMER, H.-H.; EYRES, D. Automatic structural testing using genetic algorithms. *Software Engineering Journal*, v. 11, n. 5, p. 299, 1996.
DisponÃvel em http://dx.doi.org/10.1049/sej.1996.0040

JUST, R.; JALALI, D.; ERNST, M. D. Defects4j: a database of existing faults to enable controlled testing studies for java programs. In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, New York, NY, USA: Association for Computing Machinery, 2014, p. 437â440 (*ISSTA 2014*,).
DisponÃvel em https://doi.org/10.1145/2610384.2628055

KALRA, V.; KASHYAP, I.; KAUR, H. Effect of distance measures on k-nearest neighbour classifier. In: *2022 Second International Conference on Computer Science, Engineering and Applications (ICCSEA)*, 2022, p. 1–7.

KASIKCI, B.; SCHUBERT, B.; PEREIRA, C.; POKAM, G.; CANDEA, G. Failure sketching: A technique for automated root cause diagnosis of in-production failures. In: *Proceedings of the 25th Symposium on Operating Systems Principles*, New York, NY, USA: Association for Computing Machinery, 2015, p. 344Â360 (*SOSP 2015*, v.").
DisponÃvel em https://doi.org/10.1145/2815400.2815412

KAUR, N.; HIMANSHU Logistic regression: A basic approach. In: JOSHI, A.; MAHMUD, M.; RAGEL, R. G., eds. *Information and Communication Technology for Competitive Strategies (ICTCS 2022)*, Singapore: Springer Nature Singapore, 2023, p. 481–488.

KE, G.; MENG, Q.; FINLEY, T.; WANG, T.; CHEN, W.; MA, W.; YE, Q.; LIU, T.-Y. Lightgbm: a highly efficient gradient boosting decision tree. In: *Proceedings of the 31st*

International Conference on Neural Information Processing Systems, NIPS'17, Red Hook, NY, USA: Curran Associates Inc., 2017, p. 3149â3157 (*NIPS'17*,).

KE, Y.; STOLEE, K. T.; GOUES, C. L.; BRUN, Y. Repairing programs with semantic code search (t). In: *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015, p. 295–306.

KHATIWADA, S.; TUSHEV, M.; MAHMOUD, A. Just enough semantics: An information theoretic approach for ir-based software bug localization. *Information and Software Technology*, v. 93, p. 45 – 57, 2018.
DisponÃvel em http://www.sciencedirect.com/science/article/pii/S0950584916302269

KIM, D.; NAM, J.; SONG, J.; KIM, S. Automatic patch generation learned from human-written patches. In: *Proceedings of the 2013 International Conference on Software Engineering*, IEEE Press, 2013, p. 802Â811 (*ICSE 2013*, v.").

KIM, S.; WHITEHEAD, E. J. How long did it take to fix bugs? In: *Proceedings of the 2006 International Workshop on Mining Software Repositories*, New York, NY, USA: Association for Computing Machinery, 2006, p. 173Â174 (*MSR 2006*, v.").
DisponÃvel em https://doi.org/10.1145/1137983.1138027

KITCHENHAM, B.; PEARL BRERETON, O.; BUDGEN, D.; TURNER, M.; BAILEY, J.; LINKMAN, S. Systematic literature reviews in software engineering - a systematic literature review. *Inf. Softw. Technol.*, v. 51, n. 1, p. 7Â15, 2009.
DisponÃvel em https://doi.org/10.1016/j.infsof.2008.09.009

KIZZA, J. M. *Software issues: Risks and liabilities* Cham: Springer International Publishing, p. 151–178, 2023.
DisponÃvel em https://doi.org/10.1007/978-3-031-31906-8_7

KOCHHAR, P. S.; LO, D.; LAWALL, J.; NAGAPPAN, N. Code coverage and postrelease defects: A large-scale study on open source projects. *IEEE Transactions on Reliability*, v. 66, p. 1213–1228, 2017.

KOREL, B. Automated software test data generation. *IEEE Transactions on Software Engineering*, v. 16, n. 8, p. 870–879, 1990.

KOREL, B. Dynamic method for software test data generation. *Software Testing, Verification and Reliability*, v. 2, n. 4, p. 203â213, 1992.
DisponÃvel em http://dx.doi.org/10.1002/stvr.4370020405

KOU, Y.; JEONG, H.; LEE, E. Image-based bug oracle automation for bug report reproduction using wt detection. In: *2021 IEEE International Conference on Software Engineering and Artificial Intelligence (SEAI)*, 2021, p. 43–47.

KOZLOV, D.; MYASNIKOV, V. Ensemble method for reinforcement learning algorithms based on hierarchy. In: *2023 IX International Conference on Information Technology and Nanotechnology (ITNT)*, 2023, p. 1–5.

KRAMER, O. *K-nearest neighbors* Berlin, Heidelberg: Springer Berlin Heidelberg, p. 13–23, 2013.
DisponÃvel em https://doi.org/10.1007/978-3-642-38652-7_2

KUHN, M.; JOHNSON, K. *Applied predictive modeling*. SpringerLink : Bücher. Springer New York, 2013.
DisponÃvel em https://books.google.com.br/books?id=xYRDAAAAQBAJ

KUMAR, D.; MISHRA, K. The impacts of test automation on software's cost, quality and time to market. *Procedia Computer Science*, v. 79, p. 8–15, proceedings of International Conference on Communication, Computing and Virtualization (ICCCV) 2016, 2016.
DisponÃvel em https://www.sciencedirect.com/science/article/pii/S1877050916001277

KUMAR, T. M. K.; JAYARAM, M. A. Software effort estimation using hard limiting techniques with special reference to small size technical analytical projects. In: *2022 Fourth International Conference on Emerging Research in Electronics, Computer Science and Technology (ICERECT)*, 2022, p. 1–8.

KUN, Z.; NANA, Z.; SHI, Y.; XU, W. Within-project and cross-project software defect prediction based on improved transfer naive bayes algorithm. *Computers, Materials & Continua*, v. 63, n. 2, p. 891–910, 2020.
DisponÃvel em http://www.techscience.com/cmc/v63n2/38550

KUSHWAHA, D. S.; MISRA, A. K. Software test effort estimation. *SIGSOFT Softw. Eng. Notes*, v. 33, n. 3, 2008.
DisponÃvel em https://doi.org/10.1145/1360602.1361211

LAKHOTIA, K.; HARMAN, M.; MCMINN, P. A multi-objective approach to search-based test data generation. In: *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, GECCO '07, New York, NY, USA: Association for Computing Machinery, 2007, p. 1098â1105 (*GECCO '07*,).
DisponÃvel em https://doi.org/10.1145/1276958.1277175

LE GOUES, C.; FORREST, S.; WEIMER, W. The case for software evolution. In: *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, FoSER '10, New York, NY, USA: Association for Computing Machinery, 2010, p. 205â210 (*FoSER '10*,).
DisponÃvel em https://doi.org/10.1145/1882362.1882406

LE GOUES, C.; NGUYEN, T.; FORREST, S.; WEIMER, W. Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, v. 38, n. 1, p. 54–72, 2012.

LEATHAM, K. R. Problems identifying independent and dependent variables. *School Science and Mathematics*, v. 112, n. 4, p. 349–358, 2012.
DisponÃvel em https://consensus.app/papers/problems-identifying-independent-dependent-variables-leatham/0ea4a17465035f0881d5855db14eb783/?utm_source=chatgpt

LEESATAPORNWONGSA, T.; LUKMAN, J. F.; LU, S.; GUNAWI, H. S. Taxdc: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems. *SIGPLAN Not.*, v. 51, n. 4, p. 517Â530, 2016.
DisponÃvel em https://doi.org/10.1145/2954679.2872374

LELOUDAS, P. *The importance of software testing* Berkeley, CA: Apress, p. 1–4, 2023.
DisponÃvel em https://doi.org/10.1007/978-1-4842-9514-4_1

LEMIEUX, C.; INALA, J. P.; LAHIRI, S. K.; SEN, S. Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models. In: *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, p. 919–931.

LENGLER, J. A general dichotomy of evolutionary algorithms on monotone functions. *IEEE Transactions on Evolutionary Computation*, v. 24, n. 6, p. 995–1009, 2020.

LI, D.; WONG, W. E.; PAN, S.; KOH, L.-S.; LI, S.; CHAU, M. Automatic test case generation using many-objective search and principal component analysis. *IEEE Access*, v. 10, p. 85518–85529, 2022.

LI, F.; PAXSON, V. A large-scale empirical study of security patches. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, New York, NY, USA: Association for Computing Machinery, 2017, p. 2201Â2215 (*CCS 2017*, v.").
DisponÃvel em https://doi.org/10.1145/3133956.3134072

LI, J.; LIN, S.; YU, K.; GUO, G. Quantum k-nearest neighbor classification algorithm based on hamming distance. *Quantum Information Processing*, v. 21, n. 1, p. 18, 2021.

LI, N.; OFFUTT, J. Test oracle strategies for model-based testing. *IEEE Transactions on Software Engineering*, v. 43, n. 4, p. 372–395, 2017.

LI, P.; BURGES, C. J. C.; WU, Q. Mcrank: learning to rank using multiple classification and gradient boosting. In: *Proceedings of the 20th International Conference on Neural Information Processing Systems*, NIPS'07, Red Hook, NY, USA: Curran Associates Inc., 2007, p. 897â904 (*NIPS'07*,).

LIAW, A.; WIENER, M. C. Classification and regression by randomforest. *R news*, v. 2, n. 3, 2007.
DisponÃvel em https://journal.r-project.org/articles/RN-2002-022/RN-2002-022.pdf

LIU, C. Research on software test data generation based on particle swarm optimization algorithm. In: *2021 5th International Conference on Trends in Electronics and Informatics (ICOEI)*, 2021, p. 1375–1378.

LONG, F.; RINARD, M. Staged program repair with condition synthesis. In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, New York, NY, USA: Association for Computing Machinery, 2015, p. 166Â178 (*ESEC/FSE 2015*, v.").
DisponÃvel em https://doi.org/10.1145/2786805.2786811

LUBIS, A. R.; PRAYUDANI, S.; AL-KHOWARIZMI; LASE, Y. Y.; FATMI, Y. Similarity normalized euclidean distance on knn method to classify image of skin cancer. In: *2021 4th International Seminar on Research of Information Technology and Intelligent Systems (ISRITI)*, 2021, p. 68–73.

M. ALMUFTI, S.; AHMAD SHABAN, A.; ARIF ALI, Z.; ISMAEL ALI, R.; A. DELA FUENTE, J. Overview of metaheuristic algorithms. *Polaris Global Journal of Scholarly Research and Trends*, v. 2, n. 2, p. 10â32, 2023.
DisponÃvel em https://pgjsrt.com/pgjsrt/index.php/qaj/article/view/144

MACHADO, P.; SAMPAIO, A. *Automatic test-case generation* Berlin, Heidelberg: Springer Berlin Heidelberg, p. 59–103, 2010.
DisponÃvel em https://doi.org/10.1007/978-3-642-14335-9_3

MADEIRAL, F.; URLI, S.; MAIA, M.; MONPERRUS, M. Bears: An extensible java bug benchmark for automatic program repair studies. In: *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Los Alamitos, CA, USA: IEEE Computer Society, 2019, p. 468–478.
DisponÃvel em https://doi.ieeecomputersociety.org/10.1109/SANER.2019.8667991

MARIAN, Z.; MIRCEA, I.-G.; CZIBULA, I.-G.; CZIBULA, G. A novel approach for software defect prediction using fuzzy decision trees. In: *2016 18th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, 2016, p. 240–247.

MARINESCU, C.; MARINESCU, R. Towards a more accurate assessment of file authorship. In: *2019 23rd International Conference on System Theory, Control and Computing (ICSTCC)*, 2019, p. 273–278.

MARTINEZ, M.; MONPERRUS, M. Mining software repair models for reasoning on the search space of automated program fixing. *Empirical Softw. Eng.*, v. 20, n. 1, p. 176Â205, 2015.
DisponÃvel em https://doi.org/10.1007/s10664-013-9282-8

MASMALI, O.; BADREDDIN, O.; KHANDOKER, R. Metrics to measure code complexity based on software design: Practical evaluation. In: ARAI, K., ed. *Advances in Information and Communication*, Cham: Springer International Publishing, 2021, p. 142–157.

MAZZONETTO, A.; FRANTZ, R. Z.; ROOS-FRANTZ, F.; MOLINA-JIMENEZ, C.; SAWICKI, S. A systematic mapping study of search-based software engineering for enterprise application integration. *International Journal of Software Engineering and Knowledge Engineering*, v. 32, n. 02, p. 163–191, 2022.

MCKNIGHT, P. E.; NAJAB, J. Mannâwhitney u test. *The Corsini Encyclopedia of Psychology*, p. 1â1, 2010.
DisponÃvel em http://dx.doi.org/10.1002/9780470479216.corpsy0524

MCMINN, P. Search-based software test data generation: a survey. *Software Testing, Verification and Reliability*, v. 14, n. 2, p. 105–156, 2004.
DisponÃvel em https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.294

MCMINN, P. Search-based software testing: Past, present and future. In: *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, 2011, p. 153–163.

MECHTAEV, S.; NGUYEN, M.-D.; NOLLER, Y.; GRUNSKE, L.; ROYCHOUDHURY, A. Semantic program repair using a reference implementation. In: *Proceedings of the 40th International Conference on Software Engineering*, New York, NY, USA: Association for Computing Machinery, 2018, p. 129Â139 (*ICSE 2018*, v.").
DisponÃvel em https://doi.org/10.1145/3180155.3180247

MECHTAEV, S.; YI, J.; ROYCHOUDHURY, A. Directfix: Looking for simple program repairs. In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, 2015, p. 448–458.

MECHTAEV, S.; YI, J.; ROYCHOUDHURY, A. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In: *Proceedings of the 38th International Conference on Software Engineering*, New York, NY, USA: Association for Computing Machinery, 2016, p. 691Â701 (*ICSE 2016*, v.").
DisponÃvel em https://doi.org/10.1145/2884781.2884807

MEDEIROS, N.; IVAKI, N.; COSTA, P.; VIEIRA, M. Vulnerable code detection using software metrics and machine learning. *IEEE Access*, v. 8, p. 219174–219198, 2020.

MENDONÇA, M. O.; NETTO, S. L.; DINIZ, P. S.; THEODORIDIS, S. Chapter 13 - machine learning: Review and trends. In: DINIZ, P. S., ed. *Signal Processing and Machine Learning Theory*, Academic Press, p. 869–959, 2024.
DisponÃvel em https://www.sciencedirect.com/science/article/pii/B9780323917728000193

MERZ, C. Using correspondence analysis to combine classifiers. *Machine Learning*, v. 36, p. 33–58, 1999.

MICHAEL, C.; MCGRAW, G.; SCHATZ, M. Generating software test data by evolution. *IEEE Transactions on Software Engineering*, v. 27, n. 12, p. 1085–1110, 2001.

MILLER, W.; SPOONER, D. Automatic generation of floating-point test data. *IEEE Transactions on Software Engineering*, v. SE-2, n. 3, p. 223–226, 1976.

MISHRA, D. B.; MISHRA, R.; DAS, K. N.; ACHARYA, A. A. Test case generation and optimization for critical path testing using genetic algorithm. In: BANSAL, J. C.; DAS, K. N.; NAGAR, A.; DEEP, K.; OJHA, A. K., eds. *Soft Computing for Problem Solving*, Singapore: Springer Singapore, 2019, p. 67–80.

MKAOUER, M. W.; KESSENTINI, M. Model transformation using multiobjective optimization. *Advances in Computers*, p. 161â202, 2014.
DisponÃvel em http://dx.doi.org/10.1016/b978-0-12-420232-0.00004-0

MOHAMMADIAN, M.; JAVED, Z. Intelligent evaluation of test suites for developing efficient and reliable software. *International Journal of Parallel, Emergent and Distributed Systems*, v. 36, n. 2, p. 159–188, 2021.

MOHAMMED AARIF, K. O.; SIVAKUMAR, P.; CAFFIYAR, M. Y.; HASHIM, B. A. M.; HASHIM, C. M.; RAHMAN, C. A. Nature-inspired optimization algorithms: Past to present. *Nature-Inspired Optimization Methodologies in Biomedical and Healthcare*, p. 1â32, 2022.
DisponÃvel em http://dx.doi.org/10.1007/978-3-031-17544-2_1

MOHANTY, R.; RAVI, V. Machine learning techniques to predict software defect. *Artificial Intelligence*, p. 1473â1487, 2017.
DisponÃvel em http://dx.doi.org/10.4018/978-1-5225-1759-7.ch059

MONDAL, M.; ROY, C. K.; SCHNEIDER, K. A. Bug propagation through code cloning: An empirical study. In: *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2017, p. 227–237.

MORELL, L. A theory of fault-based testing. *IEEE Transactions on Software Engineering*, v. 16, n. 8, p. 844–857, 1990.

MUJTABA, S.; PETERSEN, K.; FELDT, R.; MATTSSON, M. Software product line variability: A systematic mapping study. 2008.

MÜLLER, S. C.; FRITZ, T. Using (bio)metrics to predict code quality online. In: *Proceedings of the 38th International Conference on Software Engineering*, New York, NY, USA: Association for Computing Machinery, 2016, p. 452Â463 (*ICSE 2016*, v."). DisponÃvel em https://doi.org/10.1145/2884781.2884803

MUYLLE, K.; CORNU, P.; COOLS, W.; BARBÃ©, K.; BUYL, R.; LAERE, S. V. Optimization of performance by combining most sensitive and specific models in data science results in majority voting ensemble. *Studies in health technology and informatics*, v. 294, p. 435–439, 2022.

NAYROLLES, M.; HAMOU-LHADJ, A.; TAHAR, S.; LARSSON, A. Jcharming: A bug reproduction approach using crash traces and directed model checking. In: *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2015, p. 101–110.

NEELOFAR, N.; SMITH-MILES, K.; MUÃ±$oz, M.\ A.$; $Aleti, A.\ Instancespaceanalysisofsearch basedsoftwaretesting.$ *IEEE Transactions on Software Engineering*, $v.\ 49, n.\ 4, p.\ 2642--2660, 2023.$

NGUYEN, H. D. T.; QI, D.; ROYCHOUDHURY, A.; CHANDRA, S. Semfix: Program repair via semantic analysis. In: *35th International Conference on Software Engineering (ICSE)*, 2013, p. 772–781.

NURMELA, K.; OSTERGARD, P. Tabu search in coding theory. In: *Proceedings of 1995 IEEE International Symposium on Information Theory*, 1995, p. 346–.

OFFUTT, A. J.; HAYES, J. H. A semantic model of program faults. *SIGSOFT Softw. Eng. Notes*, v. 21, n. 3, p. 195â200, 1996.
DisponÃvel em https://doi.org/10.1145/226295.226317

O'REGAN, G. *Grace murray hopper* London: Springer London, p. 201–204, 2013.
DisponÃvel em https://doi.org/10.1007/978-1-4471-5340-5_43

OZKAYA, I. A watershed moment for search-based software engineering. *IEEE Software*, v. 38, n. 4, p. 3–6, 2021.

PANDEY, S. K.; MISHRA, R. B.; TRIPHATHI, A. K. Software bug prediction prototype using bayesian network classifier: A comprehensive model. *Procedia Computer Science*, v. 132, p. 1412–1421, international Conference on Computational Intelligence and Data Science, 2018.

DisponÃvel em https://www.sciencedirect.com/science/article/pii/S1877050918308032

PANICHELLA, A.; KIFETEW, F. M.; TONELLA, P. Reformulating branch coverage as a many-objective optimization problem. In: *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, 2015, p. 1–10.

PANICHELLA, A.; KIFETEW, F. M.; TONELLA, P. Incremental control dependency frontier exploration for many-criteria test case generation. In: COLANZI, T. E.; MCMINN, P., eds. *Search-Based Software Engineering*, Cham: Springer International Publishing, 2018, p. 309–324.

PANIGRAHI, S. S.; JENA, A. K. *Test scenarios generation and optimization of object-oriented models using meta-heuristic algorithms* Singapore: Springer Nature Singapore, p. 45–74, 2023.
DisponÃvel em https://doi.org/10.1007/978-981-99-1482-1_3

PANWAR, D.; TOMAR, P.; HARSH, H.; SIDDIQUE, M. H. Improved meta-heuristic technique for test case prioritization. In: PANT, M.; RAY, K.; SHARMA, T. K.; RAWAT, S.; BANDYOPADHYAY, A., eds. *Soft Computing: Theories and Applications*, Singapore: Springer Singapore, 2018, p. 647–664.

PAPADHOPULLI, I.; MEÇE, E. A Fitness Function for Search-Based Testing of Java Classes, which is based on the States Reached by the Object under Test. *International Journal of Computer Science and Software Engineering*, v. 5, n. 6, p. 11, 2016.

PARGAS, R. P.; HARROLD, M. J.; PECK, R. R. Test-data generation using genetic algorithms. *Software Testing, Verification and Reliability*, v. 9, n. 4, p. 263–282, 1999.
DisponÃvel em https://onlinelibrary.wiley.com/doi/abs/10.1002/%28SICI%291099-1689%28199912%299%3A4%3C263%3A%3AAID-STVR190%3E3.0.CO%3B2-Y

PATERSON, D.; CAMPOS, J.; ABREU, R.; KAPFHAMMER, G. M.; FRASER, G.; MCMINN, P. An empirical study on the use of defect prediction for test case prioritization. *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, p. 346–357, 2019.

PEITEK, N.; APEL, S.; PARNIN, C.; BRECHMANN, A.; SIEGMUND, J. Program comprehension and code complexity metrics: An fmri study. In: *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, p. 524–536.

PERERA, A.; ALETI, A.; BÃ¶HME, M.; TURHAN, B. Defect Prediction Guided Search-Based Software Testing, p. 13. 2020.

PERERA, A.; ALETI, A.; TURHAN, B.; BÃ¶HME, M. An experimental assessment of using theoretical defect predictors to guide search-based software testing. *IEEE Transactions on Software Engineering*, v. 49, n. 1, p. 131–146, 2023a.

PERERA, A.; ALETI, A.; TURHAN, B.; BÃ¶HME, M. An experimental assessment of using theoretical defect predictors to guide search-based software testing. *IEEE Transactions on Software Engineering*, v. 49, n. 1, p. 131–146, 2023b.

PERERA, A.; TURHAN, B.; ALETI, A.; BÖHME, M. On the impact of lower recall and precision in defect prediction for guiding search-based software testing. *ACM Trans. Softw. Eng. Methodol.*, just Accepted, 2024.
DisponÃvel em https://doi.org/10.1145/3655022

PETERSEN, K.; FELDT, R.; MUJTABA, S.; MATTSSON, M. Systematic mapping studies in software engineering. In: *Proceedings of the 12th International Conference on Evaluation and Assessment in Software Engineering*, Swindon, GBR: BCS Learning Development Ltd., 2008, p. 68Â77 (*EASEÂ08*, v.").

PINTO, J.; DE AZEVEDO, C. R.; OLIVEIRA, R.; VON STOSCH, M. A bootstrap-aggregated hybrid semi-parametric modeling framework for bioprocess development. *Bioprocess and Biosystems Engineering*, v. 42, n. 11, p. 1853–1865, 2019.

PRASETYO, S. Y.; ZAIN NABIILAH, G.; NABILA IZDIHAR, Z.; PRABOWO, A. S.; ASH SHIDDIQI, H. K-nearest neighbor algorithm for heart disease detection: A comparative evaluation of minkowski and manhattan distances. In: *2023 6th International Seminar on Research of Information Technology and Intelligent Systems (ISRITI)*, 2023, p. 237–241.

PUTRO DIRGANTORO, G.; SOELEMAN, M. A.; SUPRIYANTO, C. Smoothing weight distance to solve euclidean distance measurement problems in k-nearest neighbor algorithm. In: *2021 IEEE 5th International Conference on Information Technology, Information Systems and Electrical Engineering (ICITISEE)*, 2021, p. 294–298.

QI, Z.; LONG, F.; ACHOUR, S.; RINARD, M. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In: *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, New York, NY, USA: Association for Computing Machinery, 2015, p. 24Â36 (*ISSTA 2015*, v.").
DisponÃvel em https://doi.org/10.1145/2771783.2771791

QIU, Z.; LI, X.; QIN, C.; CHEN, S. Automating test case generation based on symbolic execution for railway transportation control software. In: *2023 IEEE 5th International Conference on Civil Aviation Safety and Information Technology (ICCASIT)*, 2023, p. 1414–1418.

QUINLAN, J. R. Induction of decision trees. *Machine Learning*, v. 1, n. 1, p. 81–106, 1986.

RAJAGOPAL, M.; SIVASAKTHIVEL, R.; LOGANATHAN, K.; SARRIS, L. E. An automated path-focused test case generation with dynamic parameterization using adaptive genetic algorithm (aga) for structural program testing. *Information*, v. 14, n. 3, p. 166, 2023.
DisponÃvel em http://dx.doi.org/10.3390/info14030166

RAO, D. N.; SRINATH, M. V.; HIRANMANI BALA, P. Reliable code coverage technique in software testing. In: *2013 International Conference on Pattern Recognition, Informatics and Mobile Engineering*, 2013, p. 157–163.

RAY, B.; HELLENDOORN, V.; GODHANE, S.; TU, Z.; BACCHELLI, A.; DEVANBU, P. On the ÂnaturalnessÂ of buggy code. In: *Proceedings of the 38th International Conference on Software Engineering*, New York, NY, USA: Association for Computing Machinery, 2016, p. 428Â439 (*ICSE 2016*, v.").
DisponÃvel em https://doi.org/10.1145/2884781.2884848

RAZZAGHI, P.; ABBASI, K.; GHASEMI, J. B. Chapter 3 - multivariate pattern recognition by machine learning methods. In: GHASEMI, J. B., ed. *Machine Learning and Pattern Recognition Methods in Chemistry from Multivariate and Data Driven Modeling*, Elsevier, p. 47–72, 2023.
DisponÃvel em https://www.sciencedirect.com/science/article/pii/B9780323904087000022

RECHTBERGER, V.; BURES, M.; AHMED, B. S. Overview of test coverage criteria for test case generation from finite state machines modelled as directed graphs. In: *2022 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2022, p. 207–214.

REN, W. Study on software defects prediction model based on machine learning. In: ZHANG, K., ed. *International Conference on Mathematics, Modeling, and Computer Science (MMCS2022)*, International Society for Optics and Photonics, SPIE, 2023, p. 126250P.
DisponÃvel em https://doi.org/10.1117/12.2670396

RIBAGIN, S.; LYUBENOVA, V. *Metaheuristic algorithms: Theory and applications* Cham: Springer International Publishing, p. 385–419, 2021.
DisponÃvel em https://doi.org/10.1007/978-3-030-72284-5_18

ROHDE, L. Pressure grows on eds over u.k. government it system. 2004.
DisponÃvel em https://www.computerworld.com/article/1707769/pressure-grows-on-eds-over-u-k-government-it-system.html

SAHA, R.; LYU, Y.; LAM, W.; YOSHIDA, H.; PRASAD, M. Bugs.jar: A large-scale, diverse dataset of real-world java bugs. In: *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, 2018, p. 10–13.

SAHA, R. K.; KHURSHID, S.; PERRY, D. E. An empirical study of long lived bugs. In: *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, 2014, p. 144–153.

SAHA, R. K.; LYU, Y.; YOSHIDA, H.; PRASAD, M. R. Elixir: Effective object-oriented program repair. In: *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2017, p. 648–659.

SAHA, S.; DOWNING, M.; BRENNAN, T.; BULTAN, T. Preach: a heuristic for probabilistic reachability to identify hard to reach statements. In: *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, New York, NY, USA: Association for Computing Machinery, 2022, p. 1706â1717 (*ICSE '22*,).
DisponÃvel em https://doi.org/10.1145/3510003.3510227

SAHOO, R. R.; RAY, M. Pso-based test case generation: A fitness function based on value combined branch distance. In: PATI, B.; PANIGRAHI, C. R.; BUYYA, R.; LI, K.-C., eds. *Advanced Computing and Intelligent Engineering*, Singapore: Springer Singapore, 2020, p. 589–598.

SAHOO, R. R.; RAY, M.; NAYAK, G. Test case generation based on search-based testing. In: MISHRA, D.; BUYYA, R.; MOHAPATRA, P.; PATNAIK, S., eds. *Intelligent and Cloud Computing*, Singapore: Springer Singapore, 2021, p. 309–317.

SALES, C. P.; DE SANTIAGO JÚNIOR, V. A. Investigating multi and many-objective metaheuristics to support software integration testing. In: *Proceedings of the 5th Brazilian Symposium on Systematic and Automated Software Testing*, SAST '20, New York, NY, USA: Association for Computing Machinery, 2020, p. 1â10 (*SAST '20*,).
DisponÃvel em https://doi.org/10.1145/3425174.3425175

SAMANTARAY, R.; DAS, H. Performance analysis of machine learning algorithms using bagging ensemble technique for software fault prediction. In: *2023 6th International Conference on Information Systems and Computer Networks (ISCON)*, 2023, p. 1–7.

SATHESH, A.; HAMDAN, Y. B. Analysis of software sizing and project estimation prediction by machine learning classification. *Journal of Ubiquitous Computing and Communication Technologies*, 2022.
DisponÃvel em https://irojournals.com/jucct/article/view/3/4/6

SAYED, A. H. *Naïve bayes classifier* Cambridge University Press, p. 2341â2356, 2022.

SCALABRINO, S.; GRANO, G.; DI NUCCI, D.; OLIVETO, R.; DE LUCIA, A. Search-based testing of procedural programs: Iterative single-target or multi-target approach? *Search Based Software Engineering*, p. 64â79, 2016.
DisponÃvel em http://dx.doi.org/10.1007/978-3-319-47106-8_5

SCALABRINO, S.; MASTROPAOLO, A.; BAVOTA, G.; OLIVETO, R. An adaptive search budget allocation approach for search-based test case generation. *ACM Trans. Softw. Eng. Methodol.*, v. 30, n. 3, 2021.
DisponÃvel em https://doi.org/10.1145/3446199

SCHMIDT, R. F. Chapter 15 - software verification and validation practice. In: SCHMIDT, R. F., ed. *Software Engineering*, Boston: Morgan Kaufmann, p. 263–273, 2013.
DisponÃvel em https://www.sciencedirect.com/science/article/pii/B978012407768300015X

SCHÖLKOPF, B.; SMOLA, A.; MÜLLER, K.-R. Kernel principal component analysis. In: GERSTNER, W.; GERMOND, A.; HASLER, M.; NICOUD, J.-D., eds. *Artificial Neural Networks — ICANN'97*, Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, p. 583–588.

SCHÖLKOPF, B.; SMOLA, A.; MÜLLER, K.-R. Nonlinear component analysis as a kernel eigenvalue problem. *Neural Computation*, v. 10, n. 5, p. 1299–1319, 1998.
DisponÃvel em https://doi.org/10.1162/089976698300017467

SEMUJJU, S. D.; HUANG, H.; LIU, F.; XIANG, Y.; HAO, Z. Search-based software test data generation for path coverage based on a feedback-directed mechanism. *Complex System Modeling and Simulation*, v. 3, n. 1, p. 12–31, 2023.

SERAJ, A.; MOHAMMADI-KHANAPOSHTANI, M.; DANESHFAR, R.; NASERI, M.; ESMAEILI, M.; BAGHBAN, A.; HABIBZADEH, S.; ESLAMIAN, S. Chapter 5 - cross-validation. In: ESLAMIAN, S.; ESLAMIAN, F., eds. *Handbook of Hydroinformatics*, Elsevier, p. 89–105, 2023.
DisponÃvel em https://www.sciencedirect.com/science/article/pii/B978012821285100021X

SHAH, S. M. A.; MORISIO, M. Complexity metrics significance for defects: An empirical view. In: LU, W.; CAI, G.; LIU, W.; XING, W., eds. *Proceedings of the 2012 International Conference on Information Technology and Software Engineering*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, p. 29–37.

SHAILZA KANWAR, L. K. A.; SHRIVASTAVA, V. Efficient random forest algorithm for multi-objective optimization in software defect prediction. *IETE Journal of Research*, v. 0, n. 0, p. 1–13, 2023.
DisponÃvel em https://doi.org/10.1080/03772063.2023.2205377

SHIN, Y.; WILLIAMS, L. An initial study on the use of execution complexity metrics as indicators of software vulnerabilities. In: *Proceedings of the 7th International Workshop on Software Engineering for Secure Systems*, SESS '11, New York, NY, USA: Association for Computing Machinery, 2011, p. 1â7 (*SESS '11*,).
DisponÃvel em https://doi.org/10.1145/1988630.1988632

SON, L.; PRITAM, N.; KHARI, M.; KUMAR, R.; PHUONG, P.; PHAM, T. Empirical study of software defect prediction: A systematic mapping. *Symmetry*, v. 11, p. 212, 2019.

SONG, L.; LU, S. Performance diagnosis for inefficient loops. In: *Proceedings of the 39th International Conference on Software Engineering*, IEEE Press, 2017, p. 370–380 (*ICSE 2017*, v.").
DisponÃvel em https://doi.org/10.1109/ICSE.2017.41

SONG, Q.; JIA, Z.; SHEPPERD, M.; YING, S.; LIU, J. A general software defect-proneness prediction framework. *IEEE Transactions on Software Engineering*, v. 37, n. 3, p. 356–370, 2011.

SOTO, M.; THUNG, F.; WONG, C.-P.; LE GOUES, C.; LO, D. A deeper look into bug fixes: Patterns, replacements, deletions, and additions. In: *Proceedings of the 13th International Conference on Mining Software Repositories*, New York, NY, USA: Association for Computing Machinery, 2016, p. 512Â515 (*MSR 2016*, v.").
DisponÃvel em https://doi.org/10.1145/2901739.2903495

STOLTZFUS, A. Mutation, randomness, and evolution. 2021.
DisponÃvel em http://dx.doi.org/10.1093/oso/9780198844457.001.0001

SURYAWANSHI, R.; KADAM, A. Software defect prediction by logistic regression with gradient descent cost computation. In: *2024 International Conference on Emerging Smart Computing and Informatics (ESCI)*, 2024, p. 1–5.

SUSNJAK, T.; KERRY, D.; BARCZAK, A.; REYES, N.; GAL, Y. Wisdom of crowds: An empirical study of ensemble-based feature selection strategies. In: PFAHRINGER, B.; RENZ, J., eds. *AI 2015: Advances in Artificial Intelligence*, Cham: Springer International Publishing, 2015, p. 526–538.

TAN, M.; TAN, L.; DARA, S.; MAYEUX, C. Online defect prediction for imbalanced data. In: *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, IEEE Press, 2015, p. 99Â108 (*ICSE 2015*, v.").

TAN, S. H.; ROYCHOUDHURY, A. Relifix: Automated repair of software regressions. In: *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, IEEE Press, 2015, p. 471Â482 (*ICSE 2015*, v.").

TANG, X.; WANG, S.; MAO, K. Will this bug-fixing change break regression testing? In: *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2015, p. 1–10.

TANTITHAMTHAVORN, C.; MCINTOSH, S.; HASSAN, A. E.; IHARA, A.; MATSUMOTO, K. The impact of mislabelling on the performance and interpretation of defect prediction models. In: *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, IEEE Press, 2015, p. 812Â823 (*ICSE 2015*, v.").

TAYLOR, R.; LI, X. Software-based branch predication for amd gpus. *SIGARCH Comput. Archit. News*, v. 38, n. 4, p. 66â72, 2011.
DisponÃvel em https://doi.org/10.1145/1926367.1926379

TONELLA, P. Evolutionary testing of classes. In: *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '04, New York, NY, USA: Association for Computing Machinery, 2004a, p. 119â128 (*ISSTA '04*,).
DisponÃvel em https://doi.org/10.1145/1007512.1007528

TONELLA, P. Evolutionary testing of classes. In: *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '04, New York, NY, USA: Association for Computing Machinery, 2004b, p. 119â128 (*ISSTA '04*,).
DisponÃvel em https://doi.org/10.1145/1007512.1007528

TRAUTSCH, A.; ERBEL, J.; HERBOLD, S.; GRABOWSKI, J. What really changes when developers intend to improve their source code: a commit-level study of static metric value and static analysis warning changes. *Empirical Software Engineering*, v. 28, n. 2, p. 30, 2023.

TRENDOWICZ, A.; JEFFERY, R. Classification and regression trees. *Software Project Effort Estimation*, p. 295â304, 2013.
DisponÃvel em http://dx.doi.org/10.1007/978-3-319-03629-8_10

TSAI, C.-W.; CHIANG, M.-C. Tabu search. *Handbook of Metaheuristic Algorithms*, p. 95â109, 2023.
DisponÃvel em http://dx.doi.org/10.1016/b978-0-44-319108-4.00019-8

TUKEY, J. W. The teaching of concrete mathematics. *The American Mathematical Monthly*, v. 65, n. 1, p. 1–9, 1958.
DisponÃvel em http://www.jstor.org/stable/2310294

URLI, S.; YU, Z.; SEINTURIER, L.; MONPERRUS, M. How to design a program repair bot? insights from the repairnator project. In: *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, New York, NY,

USA: Association for Computing Machinery, 2018, p. 95Â104 (*ICSE-SEIP 2018*, v.”). DisponÃvel em https://doi.org/10.1145/3183519.3183540

VENGADESWARAN, S.; GEETHA, K. Symbolic execution â an efficient approach for test case generation. In: *2013 International Conference on Recent Trends in Information Technology (ICRTIT)*, 2013, p. 575–581.

WAPPLER, S.; LAMMERMANN, F. Using evolutionary algorithms for the unit testing of object-oriented software. In: *Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation*, GECCO ’05, New York, NY, USA: Association for Computing Machinery, 2005, p. 1053â1060 (*GECCO ’05*,).
DisponÃvel em https://doi.org/10.1145/1068009.1068187

WAPPLER, S.; WEGENER, J. Evolutionary unit testing of object-oriented software using strongly-typed genetic programming. In: *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*, GECCO ’06, New York, NY, USA: Association for Computing Machinery, 2006, p. 1925â1932 (*GECCO ’06*,).
DisponÃvel em https://doi.org/10.1145/1143997.1144317

WARDOYO, R.; MUSDHOLIFAH, A.; PRADIPTA, G. A.; SANJAYA, I. N. H. Weighted majority voting by statistical performance analysis on ensemble multiclassifier. *2020 Fifth International Conference on Informatics and Computing (ICIC)*, p. 1–8, 2020.

WEGENER, J.; BARESEL, A.; STHAMER, H. Evolutionary test environment for automatic structural testing. *Information and Software Technology*, v. 43, n. 14, p. 841â854, 2001.
DisponÃvel em http://dx.doi.org/10.1016/s0950-5849(01)00190-2

WEYUKER, E. J.; BELL, R. M.; OSTRAND, T. J. We’re finding most of the bugs, but what are we missing? In: *2010 Third International Conference on Software Testing, Verification and Validation*, 2010, p. 313–322.

XANTHAKIS, S.; ELLIS, C.; SKOURLAS, C.; LE GALL, A.; KATSIKAS, S.; KARAPOULIOS, K. Application of genetic algorithms to software testing. In: *Proceedings of the 5th International Conference on Software Engineering and Applications*, 1992, p. 625–636.

XIA, Y.; SUN, W.; YAN, M.; XU, L.; YANG, D. An adaptive partition-based approach for adaptive random testing on real programs. In: *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2023, p. 668–672.

XIAO, Y.; KEUNG, J.; MI, Q.; BENNIN, K. E. Improving bug localization with an enhanced convolutional neural network. In: *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, 2017, p. 338–347.

XIN, Q.; REISS, S. P. Leveraging syntax-related code for automated program repair. In: *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2017, p. 660–670.

XU, X.; JIAO, L.; ZHU, Z. Boosting search based software testing by using ensemble methods. In: *2018 IEEE Congress on Evolutionary Computation (CEC)*, 2018, p. 1–10.

XU, Z.; LIU, J.; LUO, X.; YANG, Z.; ZHANG, Y.; YUAN, P.; TANG, Y.; ZHANG, T. Software defect prediction based on kernel pca and weighted extreme learning machine. *Information and Software Technology*, v. 106, p. 182 – 200, 2019.
DisponÃvel em http://www.sciencedirect.com/science/article/pii/S0950584918302088

XUAN, J.; MARTINEZ, M.; DEMARCO, F.; CLÃ©MENT, M.; MARCOTE, S. L.; DURIEUX, T.; LE BERRE, D.; MONPERRUS, M. Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Transactions on Software Engineering*, v. 43, n. 1, p. 34–55, 2017.

YAMASHITA, K.; HUANG, C.; NAGAPPAN, M.; KAMEI, Y.; MOCKUS, A.; HASSAN, A. E.; UBAYASHI, N. Thresholds for size and complexity metrics: A case study from the perspective of defect density. In: *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, 2016, p. 191–201.

YANG, X.; LO, D.; XIA, X.; ZHANG, Y.; SUN, J. Deep learning for just-in-time defect prediction. In: *2015 IEEE International Conference on Software Quality, Reliability and Security*, 2015, p. 17–26.

YANG, X.; TANG, K.; YAO, X. A learning-to-rank approach to software defect prediction. *IEEE Transactions on Reliability*, v. 64, n. 1, p. 234–246, 2015.

YANG, Z.; JIN, C.; ZHANG, Y.; WANG, J.; YUAN, B.; LI, H. Software defect prediction: An ensemble learning approach. *Journal of Physics: Conference Series*, v. 2171, n. 1, p. 012008, 2022.
DisponÃvel em http://dx.doi.org/10.1088/1742-6596/2171/1/012008

ZABINSKY, Z. B. Random search algorithms. *Wiley Encyclopedia of Operations Research and Management Science*, 2011.
DisponÃvel em http://dx.doi.org/10.1002/9780470400531.eorms0704

ZAKARI, A.; LEE, S. P.; ALAM, K. A.; AHMAD, R. Software fault localisation: a systematic mapping study. *IET Software*, v. 13, n. 1, p. 60–74, 2019.

ZAMPETTI, F.; SCALABRINO, S.; OLIVETO, R.; CANFORA, G.; DI PENTA, M. How open source projects use static code analysis tools in continuous integration pipelines. In: *Proceedings of the 14th International Conference on Mining Software Repositories*, IEEE Press, 2017, p. 334Â344 (*MSR 2017*, v.").
DisponÃvel em https://doi.org/10.1109/MSR.2017.2

ZHAN, L. Optimal model of software testing path selection based on genetic algorithm and its evolutionary solution. *Wireless Communications and Mobile Computing*, v. 2022, p. 1â9, 2022.
DisponÃvel em http://dx.doi.org/10.1155/2022/7601096

ZHANG, G.; GIONIS, A. Regularized impurity reduction: accurate decision trees with complexity guarantees. *Data Mining and Knowledge Discovery*, v. 37, n. 1, p. 434–475, 2023.

ZHANG, R.; WANG, Y.-W.; ZHANG, M.-Z. Automatic test oracle based on probabilistic neural networks. In: PATNAIK, S.; JAIN, V., eds. *Recent Developments in Intelligent Computing, Communication and Devices*, Singapore: Springer Singapore, 2019, p. 437–445.

ZHANG, W.; LIU, Y.; LV, H. A step-size adaptive hill-climbing algorithm for local search. In: *2021 4th International Conference on Intelligent Robotics and Control Engineering (IRCE)*, 2021, p. 1–4.

ZHAO, H.; LAI, Z.; LEUNG, H.; ZHANG, X. *A gentle introduction to feature learning* Cham: Springer International Publishing, p. 1–12, 2020.
DisponÃvel em https://doi.org/10.1007/978-3-030-40794-0_1

ZHAO, J.; RONG, Y.; GUO, Y.; HE, Y.; CHEN, H. Understanding programs by exploiting (fuzzing) test cases. *Findings of the Association for Computational Linguistics (ACL)*, 2023.
DisponÃvel em https://par.nsf.gov/biblio/10435994

ZHENG, Z.; LYU, Y. Stsearch: State tracing-based search heuristics for rtl validation. In: *2023 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2023, p. 1–6.

ZHONG, H.; SU, Z. An empirical study on real bug fixes. In: *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, IEEE Press, 2015, p. 913Â923 (*ICSE 2015*, v.").

ZHOU, B.; NEAMTIU, I.; GUPTA, R. A cross-platform analysis of bugs and bug-fixing in open source projects: Desktop vs. android vs. ios. In: *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering*, New

York, NY, USA: Association for Computing Machinery, 2015 (*EASE 2015*, v.").
DisponÃvel em https://doi.org/10.1145/2745802.2745808

ZHOU, Z.; ZHOU, Y.; FANG, C.; CHEN, Z.; TANG, Y. Selectively combining multiple coverage goals in search-based unit test generation. In: *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, New York, NY, USA: Association for Computing Machinery, 2023 (*ASE '22*,).
DisponÃvel em https://doi.org/10.1145/3551349.3556902

ZHOU, Z.-H. *Ensemble methods: Foundations and algorithms*. 1st ed. Chapman & Hall/CRC, 2012.

ZHU, M.; SHEN, D.; YU, G.; KOU, Y.; NIE, T. Computing the split points for learning decision tree in mapreduce. In: MENG, W.; FENG, L.; BRESSAN, S.; WINIWARTER, W.; SONG, W., eds. *Database Systems for Advanced Applications*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, p. 339–353.

ZHU, Z.; JIAO, L. Improving search-based software testing by constraint-based genetic operators. In: *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '19, New York, NY, USA: Association for Computing Machinery, 2019, p. 1435â1442 (*GECCO '19*,).
DisponÃvel em https://doi.org/10.1145/3321707.3321720

ZIAN, S.; KAREEM, S. A.; VARATHAN, K. D. An empirical evaluation of stacked ensembles with different meta-learners in imbalanced classification. *IEEE Access*, v. 9, p. 87434–87452, 2021.

ZIVKOVIC, T.; ZIVKOVIC, M. Comparative analysis of techniques for testing object oriented programs. In: *2020 Zooming Innovation in Consumer Technologies Conference (ZINC)*, 2020, p. 270–275.

APPENDIX

A

Selected Papers - Systematic Mapping

Table A.1: Papers selected in the systemac review

Bug Localization	Hong et al. (2015)
	Kasikci et al. (2015)
	Nayrolles et al. (2015)
	Xiao et al. (2017)
	Song and Lu (2017)
	(Gong et al., 2015)
	Khatiwada et al. (2018)
Bug Prediction	Yang et al. (2015)
	Herzig et al. (2015)
	An and Khomh (2015)
	Müller and Fritz (2016)
	Tan et al. (2015)
	Xu et al. (2019)
Automatic Repair & Automatic Patch Generation	Qi et al. (2015)
	Mechtaev et al. (2015)
	Ke et al. (2015)
	Tan and Roychoudhury (2015)
	Gao et al. (2015)
	Gao et al. (2015)
	Mechtaev et al. (2016)
	Xuan et al. (2017)
	Saha et al. (2017)
	Haraldsson et al. (2017)
	Xin and Reiss (2017)
	Hua et al. (2018)
	Urli et al. (2018)
	Mechtaev et al. (2018)
Analysis of Bugs & Bug Fixes	Zhou et al. (2015)
	Jaafar et al. (2015)
	Zhong and Su (2015)
	Tantithamthavorn et al. (2015)
	Soto et al. (2016)
	Ray et al. (2016)
	Leesatapornwongsa et al. (2016)
	Li and Paxson (2017)
	Mondal et al. (2017)
	Zampetti et al. (2017)